Advanced Configuration and Power Interface Specification Revision 2.0 Errata

Compaq Computer Corporation
Intel Corporation
Microsoft Corporation
Phoenix Technologies Ltd.
Toshiba Corporation

Errata document revision 1.5
April 13, 2001

Revision	Change Description	Affected Sections
1.5	Clarified Power / Sleep button override action will cause system to enter soft-off state. Override action will not cause the system to reset.	1.5
	Updated “SRAT” DESCRIPTION_HEADER signature reference.	5.2.5
	Replaced ASL Data Type section with a new section that clarifies ASL Data Type conversions.	16.2.2
1.4	Corrected Figure 5-1 location of system description tables. Removed redundant description of finding the RSDP on IA-PC systems – added reference to other sections.	5.1
	Corrected FADT Boot Architecture Flags Reserved field bit offset from 3 to 2.	5.2.8.3
	Clarified _INI object evaluation – OSPM evaluates _SB._INI	6.5.1
	Corrected ElseTerm definition. Changed CMOS RegionSpaceKeyword to SystemCMOS to avoid collisions with existing ASL.	16.1.3
	Corrected description of Mutex object.	16.2.3.3.1.14
	Changed ASL CopyTerm to CopyObjectTerm to avoid collision with existing ASL.	16.2.3.4.2.8
1.3	Corrected location of Firmware ACPI Control Structure may exist anywhere in the system’s memory address map.	5.2.9
	Corrected description of the Local APIC Address Override Structure.	5.2.10.11
	Corrected Local SAPIC Structure’s ACPI Processor ID field length from two bytes to one byte to enable a correct comparison with processor term’s ProcessorID field. Rearranged field ordering to more closely match the Local APIC Structure.	5.2.10.13
	Corrected _SCP reference section.	5.6.5
	Corrected TermArg and NameTerm to reference DataObject rather than DataRefObject. Added NameString to TermArg. Added missing DDBHandle and ObjectReference to ASL type definitions.	16.1.3
	Corrected Load and Unload operator descriptions – does not apply to Differentiated Definition Block	16.2.3.4.1.7, 16.2.3.4.1.17
	Corrected table reference.	16.2.3.4.2.37
1.2	Clarified that OSPM is only required to write non-zero values of FADT fields PSTATE_CNT and CST_CNT to the SMI Command Port. Corrected PM1_CNT_LEN value is ≥ 2.	5.2.8
	Changed ASL type conversion function names to avoid collision with existing ASL (Buff >ToBuffer, DecStr>ToDecimalString, HexStr>ToHexString, Int>ToInteger, String>ToString).	16.1.3, 16.2.3.4.2, 16.2.3.4.2.4, 16.2.3.4.2.10, 16.2.3.4.2.16, 16.2.3.4.2.19, 16.2.3.4.2.44
1.1	Clarified hardware interfaces may be defined as Functional Fixed Hardware **only when directed by the CPU manufacturer** as proprietary OS support is required that must be coordinated with the OS vendor.	4.1.1

	Clarified Definition Block support expanding from 32-bit to 64-bit integers.	5.2.10, 5.2.10.1, 5.2.10.2
	Local SAPIC Structure length corrected to 8 from 10 bytes.	5.2.10.13
	Updated DSDT DefinitionBlock example compliance revision.	5.5
	End value correction of event values for status bits in GPE0_BLK.	5.6.2.2
	Corrected Defined Generic Object and Control Method section references.	5.6.5
	Corrected Generic Register Descriptor Definition to include GAS reserved field.	6.4.3.7
	Corrected memory term's type field from TranslationType to Type	16.1.3
	Corrected Switch ACPI 1.0 translation	16.2.3.4.1.16
1.0	Initial errata document for ACPI 2.0.	
	Re-inserted mistakenly deleted sentence fragment.	5.2
	FADT SCI_INT field - clarified to be the SCI interrupts's Global System Interrupt number when no 8259 exists in the system.	5.2.8
	Incorrect reference to Processor declaration section.	5.2.10.5
	Local APIC Address Override Structure length field corrected.	5.2.10.11
	I/O SAPIC Strucure - length field corrected, Global System Interrupt Base and I/O SAPIC Address field descriptions expanded/clarified.	5.2.10.12
	Local SAPIC Structure flags length corrected to 4 from 2. Other offsets adjusted accordingly. Incorrect reference to Processor declaration section.	5.2.10.13
	_CS4 critical thermal trip point renamed to _HOT	12.4, 12.5
	Corrected Embedded Controller method name - removed trailing numbers	14.2
	LNOT(Logical Not) evaluation result correction.	16.2.3.4.2.26
	ASL macro for fixed I/O port descriptor listed incorrectly in previous section.	16.2.4.5, 16.2.4.6
	AML Root-Path only encoding for NamePath was missing as was NullName	17.2.1

1.5 Power and Sleep Buttons

OSPM provides a new appliance interface to consumers. In particular, it provides for a sleep button that is a "soft" button that does *not* turn the machine physically off but signals the OS to put the machine in a soft off or sleeping state. ACPI defines two types of these "soft" buttons: one for putting the machine to sleep and one for putting the machine in soft off.

This gives the OEM two different ways to implement machines: A one-button model or a two-button model. The one-button model has a single button that can be used as a power button or a sleep button as determined by user settings. The two-button model has an easily accessible sleep button and a separate power button. In either model, an override feature that forces the machine to the soft-off state without OSPM interaction is also needed to deal with various rare, but problematic, situations.

Deleted: or resets it

Deleted: consent

2.3 Device Power State Definitions

Device power states are states of particular devices; as such, they are generally not visible to the user. For example, some devices may be in the Off state even though the system as a whole is in the Working state.

Device states apply to any device on any bus. They are generally defined in terms of four principal criteria:

- **Power consumption.** How much power the device uses.
- **Device context.** How much of the context of the device is retained by the hardware. The OS is responsible for restoring any lost device context (this may be done by resetting the device).
- **Device driver.** What the device driver must do to restore the device to full on.
- **Restore time.** How long it takes to restore the device to full on.

The device power states are defined below, although very generically. Many devices do not have all four power states defined. Devices may be capable of several different low-power modes, but if there is no user-perceptible difference between the modes only the lowest power mode will be used. The Device Class Power Management Specifications, included in Appendix A of this specification, describe which of these power states are defined for a given type (class) of device and define the specific details of each power state for that device class. For a list of the available *Device Class Power Management Specifications*, see "Appendix A: Device Class Specifications."

D3 Off

Power has been fully removed from the device. The device context is lost when this state is entered, so the OS software will reinitialize the device when powering it back on. Since device context and power are lost, devices in this state do not decode their address lines. Devices in this state have the longest restore times. All classes of devices define this state.

D2

The meaning of the D2 Device State is defined by each device class. Many device classes may not define D2. In general, D2 is expected to save more power and preserve less device context than D1 or D0. Buses in D2 may cause the device to lose some context (for example, by reducing power on the bus, thus forcing the device to turn off some of its functions).

D1

The meaning of the D1 Device State is defined by each device class. Many device classes may not define D1. In general, D1 is expected to save less power and preserve more device context than D2.

D0 Fully-On

This state is assumed to be the highest level of power consumption. The device is completely active and responsive, and is expected to remember all relevant context continuously.

Table 2-2 Summary of Device Power States

Device State	Power Consumption	Device Context Retained	Driver Restoration
D0 - Fully-On	As needed for operation	All	None
D1	D0>D1>D2>D3	>D2	<D2
D2	D0>D1>D2>D3	<D1	>D1
D3 - Off	0	None	Full initialization and load

Note: Devices often have different power modes within a given state. Devices can use these modes as long as they can automatically transparently switch between these modes from the software, without violating the rules for the current D*x* state the device is in. Low-power modes that adversely affect performance (in other words, low speed modes) or that are not transparent to software cannot be done automatically in hardware; the device driver must issue commands to use these modes.

4.1.1 Functional Fixed Hardware

ACPI defines the fixed hardware low-level interfaces as a means to convey to the system OEM the minimum interfaces necessary to achieve a level of capability and quality for motherboard configuration and system power management. Additionally, the definition of these interfaces, as well as others defined in this specification, conveys to OS Vendors (OSVs) developing ACPI-compatible operating systems, the necessary interfaces that operating systems must manipulate to provide robust support for system configuration and power management.

While the definition of low-level hardware interfaces defined by ACPI 1.0 afforded OSPM implementations a certain level of stability, controls for existing and emerging diverse CPU architectures cannot be accommodated by this model as they can require a sequence of hardware manipulations intermixed with native CPU instructions to provide the ACPI-defined interface function. In this case, an ACPI-defined fixed hardware interface can be functionally implemented by the CPU manufacturer through an equivalent combination of both hardware and software and is defined by ACPI 2.0 as Functional Fixed Hardware.

In IA-32-based systems, functional fixed hardware can be accommodated in an OS independent manner by using System Management Mode (SMM) based system firmware. Unfortunately, the nature of SMM-based code makes this type of OS independent implementation difficult if not impossible to debug. As such, this implementation approach is **not** recommended. In some cases, Functional Fixed Hardware implementations may require coordination with other OS components. As such, an OS independent implementation may not be viable.

OS-specific implementations of functional fixed hardware can be implemented using technical information supplied by the CPU manufacturer. The downside of this approach is that functional fixed hardware support must be developed for each OS. In some cases, the CPU manufacturer may provide a software component providing this support. In other cases support for the functional fixed hardware may be developed directly by the OS vendor.

In ACPI 2.0, the hardware register definition has been expanded to allow registers to exist in address spaces other than the System I/O address space. This is accomplished through the specification of an address space ID in the register definition (see section 5.2.3.1, “Generic Address Structure,” for more information). **When specifically directed by the CPU manufacturer,** the system firmware may define an interface as functional fixed hardware by supplying a special address space identifier, ***FfixedHW (0x7F)***, in the address space ID field for register definitions. It is emphasized that functional fixed hardware definitions may be declared in the ACPI system firmware **only as indicated by the CPU Manufacturer** for specific interfaces as the use of functional fixed hardware requires specific coordination with the OS vendor.

Deleted: To
Formatted
Deleted: , the system firmware supplies
Formatted
Deleted: affected
Deleted: .

Only certain ACPI-defined interfaces may be implemented using functional fixed hardware and only when the interfaces are common across machine designs for example, systems sharing a common CPU architecture that does not support fixed hardware implementation of an ACPI-defined interface. OEMs are cautioned *not* to anticipate that functional fixed hardware support will be provided by OSPM differently on a system-by-system basis. The use of functional fixed hardware carries with it a reliance on OS specific software that must be considered. OEMs should consult OS vendors to ensure that specific functional fixed hardware interfaces are supported by specific operating systems.

5.1 Overview of the System Description Table Architecture

The Root System Description Pointer (RSDP) structure is located in the system's memory address space and is setup by the BIOS. This structure contains the address of the Root System Description Table (RSDT), which references other description tables that provide data to OSPM, supplying it with knowledge of the base system's implementation and configuration (see Figure 5-1).

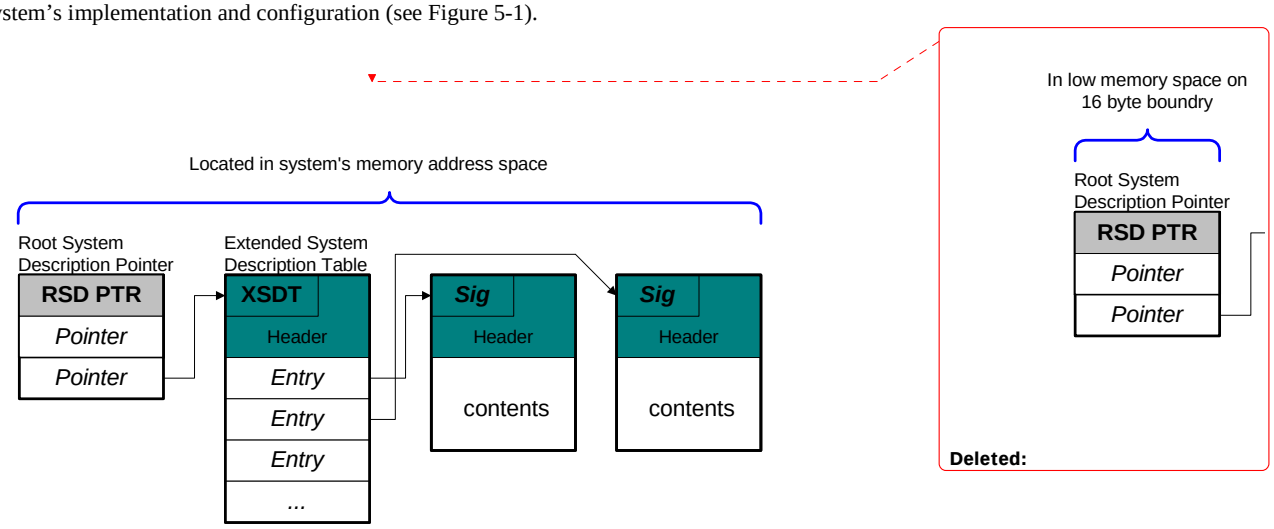

Figure 5-1 Root System Description Pointer and Table

All system description tables start with identical headers. The primary purpose of the system description tables is to define for OSPM various industry-standard implementation details. Such definitions enable various portions of these implementations to be flexible in hardware requirements and design, yet still provide OSPM with the knowledge it needs to control hardware directly.

The Root System Description Table (RSDT) points to other tables in memory. Always the first table, it points to the Fixed ACPI Description table (FADT). The data within this table includes various fixed-length entries that describe the fixed ACPI features of the hardware. The FADT table always refers to the Differentiated System Description Table (DSDT), which contains information and descriptions for various system features. The relationship between these tables is shown in Figure 5-2.

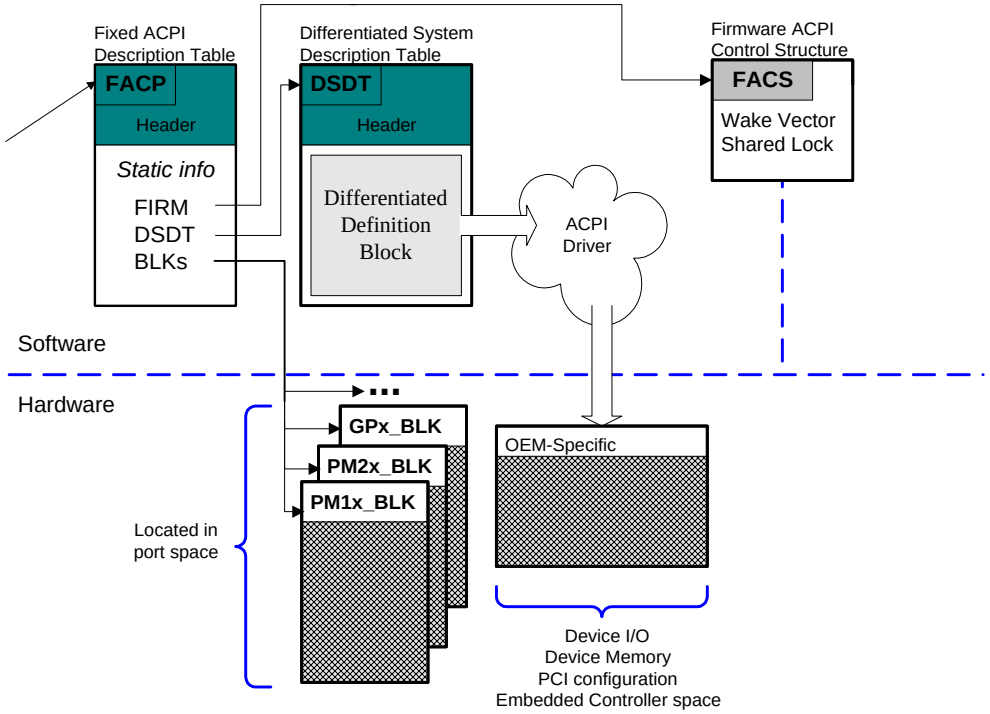

Figure 5-2 Description Table Structures

OSPM finds the RSDP structure as described in section 5.2.4.1 (“Finding the RSDP on IA-PC Systems”) or section 5.2.4.2 (“Finding the RSDP on EFI Enabled Systems”). When OSPM locates the structure, it looks at the physical address for the Root System Description Table. The Root System Description Table starts with the signature “RSDT” and contains one or more physical pointers to other system description tables that provide various information about the system. As shown in Figure 5-1, there is always a physical address in the Root System Description Table for the Fixed ACPI Description table (FADT).

Deleted: searches the following physical ranges on 16-byte boundaries for a RSDP structure. This structure is located by searching the areas listed below for a valid signature and checksum match: ¶
<#>The first 1 KB of the Extended BIOS Data Area (EBDA). For EISA or MCA systems, the EBDA can be found in the two-byte location 40:0Eh on the BIOS data area.¶
<#>In the BIOS read-only memory space between 0E0000h and 0FFFFFh.¶

5.2.5 System Description Table Header

All system description tables begin with the structure shown in Table 5-4. The Signature field determines the content of the system description table. System description table signatures defined by this specification are listed in Table 5-5.

Table 5-4 DESCRIPTION_HEADER Fields

Field	Byte Length	Byte Offset	Description
Signature	4	0	The ASCII string representation of the table identifier. Notice that if OSPM finds a signature in a table that is not listed in Table 5-5, OSPM ignores the entire table (it is not loaded into ACPI namespace); OSPM ignores the table even though the values in the Length and Checksum fields are correct.
Length	4	4	The length of the table, in bytes, including the header, starting from offset 0. This field is used to record the size of the entire table.
Revision	1	8	The revision of the structure corresponding to the signature field for this table. Larger revision numbers are backward compatible to lower revision numbers with the same signature.
Checksum	1	9	The entire table, including the checksum field, must add to zero to be considered valid.
OEMID	6	10	An OEM-supplied string that identifies the OEM.
OEM Table ID	8	16	An OEM-supplied string that the OEM uses to identify the particular data table. This field is particularly useful when defining a definition block to distinguish definition block functions. The OEM assigns each dissimilar table a new OEM Table ID.
OEM Revision	4	24	An OEM-supplied revision number. Larger numbers are assumed to be newer revisions.
Creator ID	4	28	Vendor ID of utility that created the table. For tables containing Definition Blocks, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For tables containing Definition Blocks, this is the revision for the ASL Compiler.

For OEMs, good design practices will ensure consistency when assigning OEMID and OEM Table ID fields in any table. The intent of these fields is to allow for a binary control system that support services can use. Because many support functions can be automated, it is useful when a tool can programmatically determine which table release is a compatible and more recent revision of a prior table on the same OEMID and OEM Table ID.

Table 5-5 contains the system description table signatures defined by this specification. These system description tables may be defined by ACPI or reserved by ACPI and declared by other industry specifications. This allows OS and platform specific tables to be defined and pointed to by the RSDT/XSDT as needed. For tables defined by other industry specifications, the ACPI specification acts as gatekeeper to avoid collisions in table signatures. Table signatures will be reserved by the ACPI promoters and posted independently of this specification on the ACPI Web site between specification revisions with the goal of avoiding collisions.

Table 5-5 DESCRIPTION_HEADER Signatures

Signature	Description	Reference
“APIC”	Multiple APIC Description Table	Section 5.2.10.4, “Multiple APIC Description Table”
“BOOT”	Simple Boot Flag Table	Microsoft Simple Boot Flag Specification http://www.microsoft.com/HWDEV/desinit/simp_bios.htm
“DBGP”	Debug Port Table	Microsoft Debug Port Specification http://www.microsoft.com/hwdev/newPC/debugspec.htm
“DSDT”	Differentiated System Description Table	Section 5.2.10.1, “Differentiated System Description Table”
“ECDT”	Embedded Controller Boot Resources Table	Section 5.2.13, “Embedded Controller Boot Resources Table”
“ETDT”	Event Timer Description Table	IA-PC Multimedia Timers Specification
”FACP”	Fixed ACPI Description Table (FADT)	Section 5.2.8, “Fixed ACPI Description Table”
“FACS”	Firmware ACPI Control Structure	Section 5.2.9, “Firmware ACPI Control Structure”
“OEMx”	OEM Specific Information Tables	OEM Specific tables. All table signatures starting with “OEM” are reserved for OEM use.
“PSDT”	Persistent System Description Table	Section 5.2.10.3, “Persistent System Description Table”
“RSDT”	Root System Description Table	Section 5.2.6, “Root System Description Table”
“SBST”	Smart Battery Specification Table	Section 5.2 12, “Smart Battery Table”
“SLIT”	System Locality Information Table	http://devresource.hp.com/devresource/Docs/TechPapers/IA64/slit.pdf
“SPCR”	Serial Port Console Redirection Table	Microsoft Serial Port Console Redirection Table http://www.microsoft.com/hwdev/download/SerialPortRedir.zip
“SRAT”	Static Resource Affinity Table	Interim processor-memory proximity table http://www.microsoft.com/HWDEV/design/SRAT.htm
“SSDT”	Secondary System Description Table	Section 5.2.10.2, “Secondary System Description Table”
“SPMI”	Server Platform Management Interface Table	http://devresource.hp.com/devresource/Docs/TechPapers/IA64/hpspmi.pdf
“XSDT”	Extended System Description Table	Section 5.2.7, “Extended System Description Table”

Compaq/Intel/Microsoft/Phoenix/Toshiba

5.2.8 Fixed ACPI Description Table (FADT)

The Fixed ACPI Description Table (FADT) defines various fixed hardware ACPI information vital to an ACPI-compatible OS, such as the base address for the following hardware registers blocks: PM1a_EVT_BLK, PM1b_EVT_BLK, PM1a_CNT_BLK, PM1b_CNT_BLK, PM2_CNT_BLK, PM_TMR_BLK, GPE0_BLK, and GPE1_BLK.

The FADT also has a pointer to the DSDT that contains the Differentiated Definition Block, which in turn provides variable information to an ACPI-compatible OS concerning the base system design.

All fields in the FADT that provide hardware addresses provide processor-relative physical addresses.

Table 5-8 Fixed ACPI Description Table (FADT) Format

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	'FACP'. Signature for the Fixed ACPI Description Table.
Length	4	4	Length, in bytes, of the entire FADT.
Revision	1	8	3
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	For the FADT, the table ID is the manufacture model ID. This field must match the OEM Table ID in the RSDT.
OEM Revision	4	24	OEM revision of FADT for supplied OEM Table ID.
Creator ID	4	28	Vendor ID of utility that created the table. For tables containing Definition Blocks, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For tables containing Definition Blocks, this is the revision for the ASL Compiler.
FIRMWARE_CTRL	4	36	Physical memory address (0-4 GB) of the FACS, where OSPM and Firmware exchange control information. See section 5.2.6, "Root System Description Table," for a description of the FACS.
DSDT	4	40	Physical memory address (0-4 GB) of the DSDT.
Reserved	1	44	ACPI 1.0 defined this offset as a field named INT_MODEL, which has been eliminated in ACPI 2.0.as operating systems to date have had no use for this field. New systems should set this field to zero but field values of one are also allowed to maintain compatibility with ACPI 1.0.

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
Preferred_PM_Profile	1	45	This field is set by the OEM to convey the preferred power management profile to OSPM. OSPM can use this field to set default power management policy parameters during OS installation. Field Values: 0–Unspecified 1–Desktop 2–Mobile 3–Workstation 4–Enterprise Server 5–SOHO Server 6–Appliance PC >6–Reserved
SCI_INT	2	46	System vector the SCI interrupt is wired to in 8259 mode. On systems that do not contain the 8259, this field contains the Global System interrupt number of the SCI interrupt. OSPM is required to treat the ACPI SCI interrupt as a sharable, level, active low interrupt.
SMI_CMD	4	48	System port address of the SMI Command Port. During ACPI OS initialization, OSPM can determine that the ACPI hardware registers are owned by SMI (by way of the SCI_EN bit), in which case the ACPI OS issues the ACPI_ENABLE command to the SMI_CMD port. The SCI_EN bit effectively tracks the ownership of the ACPI hardware registers. OSPM issues commands to the SMI_CMD port synchronously from the boot processor. This field is reserved and must be zero on system that does not support System Management mode.
ACPI_ENABLE	1	52	The value to write to SMI_CMD to disable SMI ownership of the ACPI hardware registers. The last action SMI does to relinquish ownership is to set the SCI_EN bit. During the OS initialization process, OSPM will synchronously wait for the transfer of SMI ownership to complete, so the ACPI system releases SMI ownership as quickly as possible. This field is reserved and must be zero on systems that do not support Legacy Mode.

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
ACPI_DISABLE	1	53	The value to write to SMI_CMD to re-enable SMI ownership of the ACPI hardware registers. This can only be done when ownership was originally acquired from SMI by OSPM using ACPI_ENABLE. An OS can hand ownership back to SMI by relinquishing use to the ACPI hardware registers, masking off all SCI interrupts, clearing the SCI_EN bit and then writing ACPI_DISABLE to the SMI_CMD port from the boot processor. This field is reserved and must be zero on systems that do not support Legacy Mode.
S4BIOS_REQ	1	54	The value to write to SMI_CMD to enter the S4BIOS state. The S4BIOS state provides an alternate way to enter the S4 state where the firmware saves and restores the memory context. A value of zero in S4BIOS_F indicates S4BIOS_REQ is not supported. (See Table 5-12.)
PSTATE_CNT	1	55	If non-zero, this field contains the value OSPM writes to the SMI_CMD register to assume processor performance state control responsibility.
PM1a_EVT_BLK	4	56	System port address of the PM1a Event Register Block. See section 4.7.3.1, “PM1 Event Grouping,” for a hardware description layout of this register block. This is a required field. This field is superseded in ACPI 2.0 by the X_PM1a_EVT_BLK field.
PM1b_EVT_BLK	4	60	System port address of the PM1b Event Register Block. See section 4.7.3.1, “PM1 Event Grouping,” for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. This field is superseded in ACPI 2.0 by the X_PM1b_EVT_BLK field.
PM1a_CNT_BLK	4	64	System port address of the PM1a Control Register Block. See section 4.7.3.2, “PM1 Control Grouping,” for a hardware description layout of this register block. This is a required field. This field is superseded in ACPI 2.0 by the X_PM1a_CNT_BLK field.
PM1b_CNT_BLK	4	68	System port address of the PM1b Control Register Block. See section 4.7.3.2, “PM1 Control Grouping,” for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. This field is superseded in ACPI 2.0 by the X_PM1b_CNT_BLK field.
PM2_CNT_BLK	4	72	System port address of the PM2 Control Register Block. See section 4.7.3.4, “PM2 Control (PM2_CNT),” for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. This field is superseded in ACPI 2.0 by the X_PM2_CNT_BLK field.

Deleted: T

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
PM_TMR_BLK	4	76	System port address of the Power Management Timer Control Register Block. See section 4.7.3.3, "Power Management Timer (PM_TMR)," for a hardware description layout of this register block. This is a required field. This field is superseded in ACPI 2.0 by the X_PM_TMR_BLK field.
GPE0_BLK	4	80	System port address of General-Purpose Event 0 Register Block. See section 5.2.8, "Fixed ACPI Description Table," for a hardware description of this register block. This is an optional field; if this register block is not supported, this field contains zero. This field is superseded in ACPI 2.0 by the X_GPE0_BLK field.
GPE1_BLK	4	84	System port address of General-Purpose Event 1 Register Block. See section 5.2.8, "Fixed ACPI Description Table," for a hardware description of this register block. This is an optional field; if this register block is not supported, this field contains zero. This field is superseded in ACPI 2.0 by the X_GPE1_BLK field.
PM1_EVT_LEN	1	88	Number of bytes decoded by PM1a_EVT_BLK and, if supported, PM1b_ EVT_BLK. This value is ≥ 4.
PM1_CNT_LEN	1	89	Number of bytes decoded by PM1a_CNT_BLK and, if supported, PM1b_CNT_BLK. This value is ≥ 2.
PM2_CNT_LEN	1	90	Number of bytes decoded by PM2_CNT_BLK. Support for the PM2 register block is optional. If supported, this value is ≥ 1. If not supported, this field contains zero.
PM_TMR_LEN	1	91	Number of bytes decoded by PM_TMR_BLK. This field's value must be 4.
GPE0_BLK_LEN	1	92	Number of bytes decoded by GPE0_BLK. The value is a non-negative multiple of 2.
GPE1_BLK_LEN	1	93	Number of bytes decoded by GPE1_BLK. The value is a non-negative multiple of 2.
GPE1_BASE	1	94	Offset within the ACPI general-purpose event model where GPE1 based events start.
CST_CNT	1	95	If non-zero, this field contains the value OSPM writes to the SMI_CMD register to indicate OS support for the _CST object and C States Changed notification.
P_LVL2_LAT	2	96	The worst-case hardware latency, in microseconds, to enter and exit a C2 state. A value > 100 indicates the system does not support a C2 state.
P_LVL3_LAT	2	98	The worst-case hardware latency, in microseconds, to enter and exit a C3 state. A value > 1000 indicates the system does not support a C3 state.

Deleted: 1

Deleted: T

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
FLUSH_SIZE	2	100	If WBINVD=0, the value of this field is the number of flush strides that need to be read (using cacheable addresses) to completely flush dirty lines from any processor's memory caches. Notice that the value in FLUSH_STRIDE is typically the smallest cache line width on any of the processor's caches (for more information, see the FLUSH_STRIDE field definition). If the system does not support a method for flushing the processor's caches, then FLUSH_SIZE and WBINVD are set to zero. Notice that this method of flushing the processor caches has limitations, and WBINVD=1 is the preferred way to flush the processors caches. This value is typically at least 2 times the cache size. The maximum allowed value for FLUSH_SIZE multiplied by FLUSH_STRIDE is 2 MB for a typical maximum supported cache size of 1 MB. Larger cache sizes are supported using WBINVD=1. This value is ignored if WBINVD=1. This field is maintained for ACPI 1.0 processor compatibility on existing systems. Processors in new ACPI 2.0-compatible systems are required to support the WBINVD function and indicate this to OSPM by setting the WBINVD field = 1.
FLUSH_STRIDE	2	102	If WBINVD=0, the value of this field is the cache line width, in bytes, of the processor's memory caches. This value is typically the smallest cache line width on any of the processor's caches. For more information, see the description of the FLUSH_SIZE field. This value is ignored if WBINVD=1. This field is maintained for ACPI 1.0 processor compatibility on existing systems. Processors in new ACPI 2.0-compatible systems are required to support the WBINVD function and indicate this to OSPM by setting the WBINVD field = 1.
DUTY_OFFSET	1	104	The zero-based index of where the processor's duty cycle setting is within the processor's P_CNT register.

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
DUTY_WIDTH	1	105	The bit width of the processor's duty cycle setting value in the P_CNT register. Each processor's duty cycle setting allows the software to select a nominal processor frequency below its absolute frequency as defined by: THTL_EN = 1 BF * DC/(2^{DUTY_WIDTH}) Where: BF–Base frequency DC–Duty cycle setting When THTL_EN is 0, the processor runs at its absolute BF. A DUTY_WIDTH value of 0 indicates that processor duty cycle is not supported and the processor continuously runs at its base frequency.
DAY_ALRM	1	106	The RTC CMOS RAM index to the day-of-month alarm value. If this field contains a zero, then the RTC day of the month alarm feature is not supported. If this field has a non-zero value, then this field contains an index into RTC RAM space that OSPM can use to program the day of the month alarm. See section 4.7.2.4, "Real Time Clock Alarm," for a description of how the hardware works.
MON_ALRM	1	107	The RTC CMOS RAM index to the month of year alarm value. If this field contains a zero, then the RTC month of the year alarm feature is not supported. If this field has a non-zero value, then this field contains an index into RTC RAM space that OSPM can use to program the month of the year alarm. If this feature is supported, then the DAY_ALRM feature must be supported also.
CENTURY	1	108	The RTC CMOS RAM index to the century of data value (hundred and thousand year decimals). If this field contains a zero, then the RTC centenary feature is not supported. If this field has a non-zero value, then this field contains an index into RTC RAM space that OSPM can use to program the centenary field.
IAPC_BOOT_ARCH	2	109	IA-PC Boot Architecture Flags. See Table 5-10 for a description of this field.
Reserved	1	111	Must be 0.
Flags	4	112	Fixed feature flags. See Table 5-9 for a description of this field.

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
RESET_REG	12	116	The address of the reset register represented in Generic Address Structure format (See section 4.7.3.6, "Reset Register," for a description of the reset mechanism.) **Note**: Only System I/O space, System Memory space and PCI Configuration space (bus #0) are valid for values for Address_Space_ID. Also, Register_Bit_Width must be 8 and Register_Bit_Offset must be 0.
RESET_VALUE	1	128	Indicates the value to write to the RESET_REG port to reset the system. (See section 4.7.3.6, "Reset Register," for a description of the reset mechanism.)
Reserved	3	129	Must be 0.
X_FIRMWARE_CTRL	8	132	64bit physical address of the FACS.
X_DSDT	8	140	64bit physical address of the DSDT.
X_PM1a_EVT_BLK	12	148	Extended address of the PM1a Event Register Block, represented in Generic Address Structure format. See section 4.7.3.1, "PM1 Event Grouping," for a hardware description layout of this register block. This is a required field.
X_PM1b_EVT_BLK	12	160	Extended address of the PM1b Event Register Block, represented in Generic Address Structure format. See section 4.7.3.1, "PM1 Event Grouping," for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero.
X_PM1a_CNT_BLK	12	172	Extended address of the PM1a Control Register Block, represented in Generic Address Structure format. See section 4.7.3.2, "PM1 Control Grouping," for a hardware description layout of this register block. This is a required field.
X_PM1b_CNT_BLK	12	184	Extended address of the PM1b Control Register Block, represented in Generic Address Structure format. See section 4.7.3.2, "PM1 Control Grouping," for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero.
X_PM2_CNT_BLK	12	196	Extended address of the Power Management 2 Control Register Block, represented in Generic Address Structure format. See section 4.7.3.4, "PM2 Control (PM2_CNT)," for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero.

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
X_PM_TMR_BLK	12	208	Extended address of the Power Management Timer Control Register Block, represented in Generic Address Structure format. See section 4.7.3.3, "Power Management Timer (PM_TMR)," for a hardware description layout of this register block. This is a required field.
X_GPE0_BLK	12	220	Extended address of the General-Purpose Event 0 Register Block, represented in Generic Address Structure format. See section 5.2.8, "Fixed ACPI Description Table," for a hardware description of this register block. This is an optional field; if this register block is not supported, this field contains zero.
X_GPE1_BLK	12	232	Extended address of the General-Purpose Event 1 Register Block, represented in Generic Address Structure format. See section 5.2.8, "Fixed ACPI Description Table," for a hardware description of this register block. This is an optional field; if this register block is not supported, this field contains zero.

5.2.8.3 IA-PC Boot Architecture Flags

This set of flags is used by an OS to guide the assumptions it can make in initializing hardware on IA-PC platforms. These flags are used by an OS at boot time (before the OS is capable of providing an operating environment suitable for parsing the ACPI namespace) to determine the code paths to take during boot. In IA-PC platforms with reduced legacy hardware, the OS can skip code paths for legacy devices if none are present. For example, if there are no ISA devices, an OS could skip code that assumes the presence of these devices and their associated resources. These flags are used independently of the ACPI namespace. The presence of other devices must be described in the ACPI namespace as specified in section 6, "Configuration."

These flags pertain only to IA-PC platforms. On other system architectures, the entire field should be set to 0.

Table 5-10 Fixed ACPI Description Table Boot Architecture Flags

BOOT_ARCH	Bit length	Bit offset	Description
LEGACY_DEVICES	1	0	If set, indicates that the motherboard supports user-visible devices on the LPC or ISA bus. User-visible devices are devices that have end-user accessible connectors (for example, LPT port), or devices for which the OS must load a device driver so that an end-user application can use a device. If clear, the OS may assume there are no such devices and that all devices in the system can be detected exclusively via industry standard device enumeration mechanisms (including the ACPI namespace).
8042	1	1	If set, indicates that the motherboard contains support for a port 60 and 64 based keyboard controller, usually implemented as an 8042 or equivalent micro-controller.
Reserved	14	2	Must be 0.

Deleted: 3

5.2.9 Firmware ACPI Control Structure (FACS)

The Firmware ACPI Control Structure (FACS) is a structure in read/write memory that the BIOS reserves for ACPI usage. This structure is passed to an ACPI-compatible OS using the FADT. For more information about the FADT FIRMWARE_CTRL field, see section 5.2.8, “Fixed ACPI Description Table (FADT).”

The BIOS aligns the FACS on a 64-byte boundary anywhere within the system’s memory address space. The memory where the FACS structure resides must not be reported as system AddressRangeMemory in the system address map. For example, the E820 address map reporting interface would report the region as AddressRangeReserved. For more information about system address map reporting interfaces, see section 15, “System Address Map Interfaces.”

Deleted: 0-4G

5.2.10 Definition Blocks

A Definition Block consists of data in AML format (see section 5.4 “Definition Block Encoding”) and contains information about hardware implementation details in the form of AML objects that contain data, AML code, or other AML objects. The top-level organization of this information after a definition block is loaded is name-tagged in a hierarchical namespace.

OSPM “loads” or “unloads” an entire definition block as a logical unit. OSPM will load a definition block either as a result of executing the AML **Load()** or **LoadTable()** operator or encountering a table definition during initialization. During initialization, OSPM loads the Differentiated System Description Table (DSDT), which contains the Differentiated Definition Block, using the DSDT pointer retrieved from the FADT. OSPM will load other definition blocks during initialization as a result of encountering Secondary System Description Table (SSDT) definitions in the RSDT/XSDT. The DSDT and SSDT are described in the following sections.

As mentioned, the AML **Load()** and **LoadTable()** operators make it possible for a Definition Block to load other Definition Blocks, either statically or dynamically, where they in turn can either define new system attributes or, in some cases, build on prior definitions. Although this gives the hardware the ability to vary widely in implementation, it also confines it to reasonable boundaries. In some cases, the Definition Block format can describe only specific and well-understood variances. In other cases, it permits implementations to be expressible only by means of a specified set of “built in” operators. For example, the Definition Block has built in operators for I/O space.

In theory, it might be possible to define something like PCI configuration space in a Definition Block by building it from I/O space, but that is not the goal of the definition block. Such a space is usually defined as a “built in” operator.

Some AML operators perform simple functions, and others encompass complex functions. The power of the Definition block comes from its ability to allow these operations to be glued together in numerous ways, to provide functionality to OSPM.

The AML operators defined in this specification are intended to allow many useful hardware designs to be easily expressed, not to allow all hardware designs to be expressed.

Note: To accommodate addressing beyond 32 bits, the integer type is expanded to 64 bits in ACPI 2.0, see section 16.2.2, “ASL Data Types”. Existing ACPI definition block implementations may contain an inherent assumption of a 32-bit integer width. Therefore, to maintain backwards compatibility, OSPM uses the Revision field, in the header portion of system description tables containing Definition Blocks, to determine whether integers declared within the Definition Block are to be evaluated as 32-bit or 64-bit values. A Revision field value greater than or equal to 2 signifies that integers declared within the Definition Block are to be evaluated as 64-bit values. The ASL writer specifies the value for the Definition Block table header’s Revision field via the ASL DefinitionBlockTerm’s *ComplianceRevision* field. See section 16.2.3.1, “Definition Block Term”, for more information. It is the responsibility of the ASL writer to ensure the Definition Block’s compatibility with the corresponding integer width when setting the *ComplianceRevision* field.

Formatted

Formatted

5.2.10.1 Differentiated System Description Table (DSDT)

The Differentiated System Description Table (DSDT) is part of the system fixed description. The DSDT is comprised of a system description table header followed by data in Definition Block format. This Definition Block is like all other Definition Blocks, with the exception that it cannot be unloaded. See section 5.2.10, “Definition Blocks,” for a description of Definition Blocks.

Table 5-13a Differentiated System Description Table Fields (DSDT)

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	‘DSDT.’ Signature for the Differentiated System Description Table.
Length	4	4	Length, in bytes, of the entire DSDT (including the header).
Revision	1	8	2
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	The manufacture model ID.
OEM Revision	4	24	OEM revision of DSDT for supplied OEM Table ID.
Creator ID	4	28	Vendor ID for the ASL Compiler.
Creator Revision	4	32	Revision number of the ASL Compiler.
Definition Block	*n*	36	*n* bytes of AML code (see section 5.4, “Definition Block Encoding”)

Formatted

5.2.10.2 Secondary System Description Table (SSDT)

Secondary System Description Tables (SSDT) are a continuation of the DSDT. The SSDT is comprised of a system description table header followed by data in Definition Block format. There can be multiple SSDTs present. OSPM first loads the DSDT and then loads each SSDT. This allows the OEM to provide

Deleted: After the

Deleted: is loaded,

Deleted: is loaded

the base support in one table and add smaller system options in other tables. For example, the OEM might put dynamic object definitions into a secondary table such that the firmware can construct the dynamic information at boot without needing to edit the static DSDT. A SSDT can only rely on the DSDT being loaded prior to it.

Table 5-13b Secondary System Description Table Fields (SSDT)

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	'SSDT.' Signature for the Secondary System Description Table.
Length	4	4	Length, in bytes, of the entire SSDT (including the header).
Revision	1	8	2
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	The manufacture model ID.
OEM Revision	4	24	OEM revision of DSDT for supplied OEM Table ID.
Creator ID	4	28	Vendor ID for the ASL Compiler.
Creator Revision	4	32	Revision number of the ASL Compiler.
Definition Block	*n*	36	*n* bytes of AML code (see section 5.4 , "Definition Block Encoding")

Formatted

5.2.10.5 Processor Local APIC

When using the APIC interrupt model, each processor in the system is required to have a Processor Local APIC record and an ACPI Processor object. OSPM does not expect the information provided in this table to be updated if the processor information changes during the lifespan of an OS boot. While in the sleeping state, processors are not allowed to be added, removed, nor can their APIC ID or Flags change. When a processor is not present, the Processor Local APIC information is either not reported or flagged as disabled.

Table 5-17 Processor Local APIC Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	0–Processor Local APIC structure
Length	1	1	8
ACPI Processor ID	1	2	The ProcessorId for which this processor is listed in the ACPI Processor declaration operator. For a definition of the Processor operator, see section 16.2.3.3.1.17, "Processor (Declare Processor)."
APIC ID	1	3	The processor's local APIC ID.
Flags	4	4	Local APIC flags. See Table 5-18 for a description of this field.

Deleted: 6

5.2.10.11 Local APIC Address Override Structure

This optional structure supports 64-bit systems by providing an override of the physical address of the local APIC in the MADT'stable header, which is defined as a 32-bit field.

If defined, OSPM must use the address specified in this structure for all local APICs (and local SAPICs), rather than the address contained in the MADT's table header. Only one Local APIC Address Override Structure may be defined.

Table 5-24 Local APIC Address Override Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	5–Local APIC Address Override Structure
Length	1	1	12
Reserved	2	2	Reserved (must be set to zero)
Local APIC Address	8	4	Physical address of Local APIC

Deleted: s

Deleted: using the Generic Address Structure

Deleted: 6

5.2.10.12 I/O SAPIC Structure

The I/O SAPIC structure is very similar to the I/O APIC structure. If both I/O APIC and I/O SAPIC structures exist for a specific APIC ID, the information in the I/O SAPIC structure must be used.

The I/O SAPIC structure uses the I/O_APIC_ID field as defined in the I/O APIC table. The Vector_Base field remains unchanged but has been moved. The I/O APIC address has been deleted. A new address and reserved field have been added.

Table 5-25 I/O SAPIC Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	6–I/O SAPIC Structure
Length	1	1	16
I/O APIC ID	1	2	I/O SAPIC ID
Reserved	1	3	Reserved (must be zero)
Global System Interrupt Base	4	4	The global system interrupt number where this I/O SAPIC's interrupt inputs start. The number of interrupt inputs is determined by the I/O SAPIC's *Max Redir Entry* register.
I/O SAPIC Address	8	8	The 64-bit physical address to access this I/O SAPIC. Each I/O SAPIC resides at a unique address.

Deleted: 20

Deleted: Global System Interrupt Base

Deleted: Physical address for I/O SAPIC

If defined, OSPM must use the information contained in the I/O SAPIC structure instead of the information from the I/O APIC structure.

If both I/O APIC and an I/O SAPIC structures exist in an MADT, the OEM/BIOS writer must prevent "mixing" I/O APIC and I/O SAPIC addresses. This is done by ensuring that there are at least as many I/O SAPIC structures as I/O APIC structures and that every I/O APIC structure has a corresponding I/O SAPIC structure (same APIC ID).

5.2.10.13 Local SAPIC Structure

The Processor local SAPIC structure is very similar to the processor local APIC structure. When using the SAPIC interrupt model, each processor in the system is required to have a Processor Local SAPIC record and an ACPI Processor object. OSPM does not expect the information provided in this table to be updated if the processor information changes during the lifespan of an OS boot. While in the sleeping state, processors are not allowed to be added, removed, nor can their SAPIC ID or Flags change. When a processor is not present, the Processor Local SAPIC information is either not reported or flagged as disabled.

Table 5-26 Processor Local SAPIC Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	7–Processor Local SAPIC structure
Length	1	1	12
ACPI Processor ID	1	2	The Processor Id listed in the processor object. For a definition of the Processor object, see section 16.2.3.3.1.17 "Processor (Declare Processor)."
Local SAPIC ID	1	3	The processor's local SAPIC ID
Local SAPIC EID	1	4	The processor's local SAPIC EID
Reserved	3	5	Reserved (must be set to zero)
Flags	4	8	Local SAPIC flags. See Table 5-18 for a description of this field.

Deleted: 8
Deleted: 2
Deleted: 6
Deleted: Flags
Deleted: 2
Deleted: 4
Deleted: Local SAPIC flags. See Table 5-18 for a description of this field.
Deleted: 6
Deleted: Local SAPIC EID
Deleted: 1
Deleted: 7
Deleted: The processor's local SAPIC EID

5.6.2.2 General-Purpose Event Handling

When OSPM receives a general-purpose event, it either passes control to an ACPI-aware driver, or uses an OEM-supplied control method to handle the event. An OEM can implement up to 128 general-purpose event inputs in hardware per GPE block, each as either a level or edge event. It is also possible to implement a single 256-pin block as long as it's the only block defined in the system.

An example of a general-purpose event is specified in section 4, "ACPI Hardware Specification," where EC_STS and EC_EN bits are defined to enable OSPM to communicate with an ACPI-aware embedded controller device driver. The EC_STS bit is set when either an interface in the embedded controller space has generated an interrupt or the embedded controller interface needs servicing. Notice that if a platform uses an embedded controller in the ACPI environment, then the embedded controller's SCI output must be directly and exclusively tied to a single GPE input bit.

Hardware can cascade other general-purpose events from a bit in the GPEx_BLK through status and enable bits in Operational Regions (I/O space, memory space, PCI configuration space, or embedded controller space). For more information, see the specification of the General-Purpose Event Blocks (GPEx_BLK) in section 4.7.4.1, "General-Purpose Event Register Blocks."

OSPM manages the bits in the GPEx blocks directly, although the source to those events is not directly known and is connected into the system by control methods. When OSPM receives a general-purpose event (the event is from a GPEx_BLK STS bit), OSPM does the following:

1. Disables the interrupt source (GPEx_BLK EN bit).
2. If an edge event, clears the status bit.

3. Performs one of the following:
   - Dispatches to an ACPI-aware device driver.
   - Queues the matching control method for execution.
   - Manages a wake event using device _PRW objects.
4. If a level event, clears the status bit.
5. Enables the interrupt source.

The OEM AML code can perform OEM-specific functions custom to each event the particular platform might generate by executing a control method that matches the event. For GPE events, OSPM will execute the control method of the name _GPE.*_TXX* where *XX* is the hex value format of the event that needs to be handled and *T* indicates the event handling type (*T* must be either 'E' for an *edge* event or 'L' for a *level* event). The event values for status bits in GPE0_BLK start at zero (_*T*00) and end at the (GPE0_BLK_LEN / 2) - 1. The event values for status bits in GPE1_BLK start at GPE1_BASE and end at GPE1_BASE + (GPE1_BLK_LEN / 2) - 1. GPE0_BLK_LEN, GPE1_BASE, and GPE1_BLK_LEN are all defined in the FADT.

For OSPM to manage the bits in the GPEx_BLK blocks directly:

- Enable bits must be read/write.
- Status bits must be latching.
- Status bits must be read/clear, and cleared by writing a "1" to the status bit.

5.5 Using the ACPI Control Method Source Language

OEMs and BIOS vendors write definition blocks using the ACPI Control Method Source language (ASL) and use a translator to produce the byte stream encoding described in section 5.4. For example, the ASL statements that produce the example byte stream shown in that earlier section are shown in the following ASL example. For a full specification of the ASL statements, see section 16, "ACPI Source Language Reference."

```
// ASL Example
DefinitionBlock (
    "forbook.aml",     // Output Filename
    "DSDT",            // Signature
    0x02,              // DSDT Compliance Revision
    "OEM",             // OEMID
    "forbook",         // TABLE ID
    0x1000             // OEM Revision
    )
{   // start of definition block
    OperationRegion(\GIO, SystemIO, 0x125, 0x1)
    Field(\GIO, ByteAcc, NoLock, Preserve)   {
           CT01,   1,
    }

    Scope(\_SB){   // start of scope
        Device(PCI0)   {   // start of device
            PowerResource(FET0, 0, 0) {       // start of pwr
                Method(_ON){
                    Store (Ones, CT01)        // assert power
                    Sleep (30)                // wait 30ms
                }
                Method(_OFF) {
                    Store (Zero, CT01)        // assert reset#
                }
                Method(_STA) {
                    Return (CT01)
                }
            } // end of pwr
        } // end of device
    } // end of scope
} // end of definition block
```

Deleted: 1

5.6.5 Defined Generic Objects and Control Methods

The following table lists all of the generic object and control methods defined in this specification and provides a reference to the defining section of the specification.

Table 5-43 Defined Generic Object and Control Methods

Object	Description	Reference
_Acx	Thermal Zone object that returns active cooling policy threshold values in tenths of degrees Kelvin.	12.3.1
_ADR	Device object that evaluates to a device's address on its parent bus. For the display output device, this object returns a unique ID. (B.5.1, "_ADR - Return the Unique ID for this Device.")	6.1.1
_AL*x*	Thermal zone object containing a list of cooling device objects.	12.3.2
_ALN	Resource data type reserved field name	16.2.4
_ASI	Resource data type reserved field name	16.2.4.16

Table 5-43 Defined Generic Object and Control Methods *(continued)*

Object	Description	Reference
_BAS	Resource data type reserved field name	16.2.4
_BBN	PCI bus number setup by the BIOS	6.5.5
_BCL	Returns a buffer of bytes indicating list of brightness control levels supported.	B.5.2
_BCM	Sets the brightness level of the built-in display output device.	B.5.3
_BDN	Correlates a docking station between ACPI and legacy interfaces.	6.5.3
_BFS	Control method executed immediately following a wake event.	7.3.1
_BIF	Control Method Battery information object	11.2.2.1
_BM	Resource data type reserved field name	16.2.4
_BST	Control Method Battery status object	11.2.2.2
_BTP	Sets Control Method Battery trip point	11.2.2.3
_CID	Device identification object that evaluates to a device's Plug and Play Compatible ID list.	6.1.2
_CRS	Device configuration object that specifies a device's *current* resource settings, or a control method that generates such an object.	6.2.1
_CRT	Thermal zone object that returns critical trip point in tenths of degrees Kelvin.	12.3.3
_CST	Processor power state declaration object	8.3.2
_DCK	Indicates that the device is a docking station.	6.5.2
_DCS	Returns the status of the display output device.	B.5.5
_DDC	Returns the EDID for the display output device	B.5.4
_DDN	Object that associates a logical software name (for example, COM1) with a device.	6.1.3
_DEC	Resource data type reserved field name	16.2.4
_DGS	Control method used to query the state of the output device.	B.5.6
_DIS	Device configuration control method that disables a device.	6.2.2
_DMA	Object that specifies a device's *current* resources for DMA transactions.	6.2.3
_DOD	Control method used to enumerate devices attached to the display adapter.	B.4.2
_DOS	Control method used to enable/disable display output switching.	B.4.1
_DSS	Control method used to set display device state.	B.5.7
_Exx	Control method executed as a result of a general-purpose event.	5.6.2.2, 5.6.2.2.3
_EC	Control Method used to define the offset address and Query value of an SMB-HC defined within an embedded controller device.	13.12
_EDL	Device removal object that returns a packaged list of devices that are dependent on a device.	6.3.1
_EJx	Device insertion/removal control method that ejects a device.	6.3.3

Deleted: 5.3

Table 5-43 Defined Generic Object and Control Methods *(continued)*

Object	Description	Reference
_EJD	Device removal object that evaluates to the name of a device object upon which a device is dependent. Whenever the named device is ejected, the dependent device must receive an ejection notification.	6.3.2
_FDE	Object that indicates the presence or absence of floppy disks.	10.9.1
_FDI	Object that returns floppy drive information.	10.9.2
_FDM	Control method that changes the mode of floppy drives.	10.9.3
_FIX	Object used to provide correlation between the fixed hardware register blocks defined in the FADT and the devices that implement these fixed hardware registers.	6.2.4
_GL	OS-defined Global Lock mutex object	5.7.1
_ GLK	Indicates the need to acquire the Global Lock, must be acquired when accessing the device.	6.5.7
_GPD	Control method that returns which VGA device will be posted at boot	B.4.4
_GPE	1. General-Purpose Events root name space 2. Object that returns the SCI interrupt within the GPx_STS register that is connected to the EC.	5.3.1 13.11
_GRA	Resource data type reserved field name.	16.2.4
_GTF	IDE device control method to get the Advanced Technology Attachement (ATA) task file needed to re-initialize the drive to bootup defaults.	10.8.1
_GTM	IDE device control method to get the IDE controller timing information.	10.8.2
_GTS	Control method executed just prior to setting the sleep enable (SLP_EN) bit.	7.3.3
_HE	Resource data type reserved field name	16.2.4
_HID	Device identification object that evaluates to a device's Plug and Play Hardware ID.	6.1.4
_HPP	An object that specifies the Cache-line size, Latency timer, SERR enable, and PERR enable values to be used when configuring a PCI device inserted into a hot-plug slot or initial configuration of a PCI device at system boot.	6.2.5
_INI	Device initialization method that performs device specific initialization.	6.5.1
_INT	Resource data type reserved field name	16.2.4
_IRC	Power management object that signifies the device has a significant inrush current draw.	7.2.11
_Lxx	Control method executed as a result of a general-purpose event.	5.6.2.2, 5.6.2.2.3
_LCK	Device insertion/removal control method that locks or unlocks a device.	6.3.4
_LEN	Resource data type reserved field name	16.2.4
_LID	Object that returns the status of the Lid on a mobile system.	10.3.1
_LL	Resource data type reserved field name	16.2.4

Deleted: 4

Deleted: 5.3

Table 5-43 Defined Generic Object and Control Methods *(continued)*

Object	Description	Reference
_MAF	Resource data type reserved field name	16.2.4
_MAT	Object evaluates to a buffer of MADT APIC Structure entries.	6.2.6
_MAX	Resource data type reserved field name	16.2.4
_MEM	Resource data type reserved field name	16.2.4
_MIF	Resource data type reserved field name	16.2.4
_MIN	Resource data type reserved field name	16.2.4
_MSG	System indicator control that indicates messages are waiting.	10.1.2
_OFF	Power resource object that sets the resource off.	7.1.2
_ON	Power resource object that sets the resource on.	7.1.3
_OS	Object that evaluates to a string that identifies the operating system.	5.7.2
_PCL	Power source object that contains a list of devices powered by a power source.	11.3.2
_PCT	Processor performance control object	8.3.3.1
_PIC	Control method that conveys interrupt model in use to the system firmware.	5.8.1
_PPC	Control method used to determine number of performance states currently supported by the platform.	8.3.3.3
_PR	ACPI 1.0 Processor Namespace	5.3.1
_PR0	Power management object that evaluates to the device's power requirements in the D0 device state (device fully on).	7.2.6
_PR1	Power management object that evaluates to the device's power requirements in the D1 device state. Only devices that can achieve the defined D1 device state according to its given device class would supply this level.	7.2.7
_PR2	Power management object that evaluates to the device's power requirements in the D2 device state. Only devices that can achieve the defined D2 device state according to its given device class would supply this level.	7.2.8
_PRS	Device configuration object that specifies a device's *possible* resource settings, or a control method that generates such an object.	6.2.7
_PRT	An object that specifies the PCI interrupt Routing Table.	6.2.8
_PRW	Power management object that evaluates to the device's power requirements in order to wake the system from a system sleeping state.	7.2.9
_PS0	Power management control method that puts the device in the D0 device state. (device fully on).	7.2.1
_PS1	Power management control method that puts the device in the D1 device state.	7.2.2
_PS2	Power management control method that puts the device in the D2 device state.	7.2.3

Table 5-43 Defined Generic Object and Control Methods *(continued)*

Object	Description	Reference
_PS3	Power management control method that puts the device in the D3 device state (device off).	7.2.4
_PSC	Power management object that evaluates to the device's current power state.	7.2.5
_PSL	Thermal zone object that returns list of passive cooling device objects.	12.3.4
_PSR	Power source object that returns present power source device.	11.3.1
_PSS	Object indicates the number of supported processor performance states.	8.3.3.2
_PSV	Thermal zone object that returns Passive trip point in tenths of degrees Kelvin.	12.3.5
_PSW	Power management control method that enables or disables the device's wake function.	7.2.10
_PTC	Object used to define a processor throttling control register.	8.3.1
_PTS	Control method used to prepare to sleep.	7.3.2
_PXM	Object used to describe proximity domains within a machine.	6.2.9
_Q*xx*	Embedded Controller Query control method	5.6.2.2.3
_RBO	Resource data type reserved field name	16.2.4
_RBW	Resource data type reserved field name	16.2.4
_REG	Notifies AML code of a change in the availability of an operation region.	6.5.4
_REV	Revision of the ACPI specification that OSPM implements.	5.7.3
_RMV	Device insertion/removal object that indicates that the given device is removable.	6.3.5
_RNG	Resource data type reserved field name	16.2.4
_ROM	Control method used to get a copy of the display devices' ROM data.	B.4.3
_RW	Resource data type reserved field name	16.2.4
_S0	Power management package that defines system _S0 state mode.	7.3.4.1
_S1	Power management package that defines system _S1 state mode.	7.3.4.2
_S2	Power management package that defines system _S2 state mode.	7.3.4.3
_S3	Power management package that defines system _S3 state mode.	7.3.4.4
_S4	Power management package that defines system _S4 state mode.	7.3.4.5
_S5	Power management package that defines system _S5 state mode.	7.3.4.6
_S1D	Highest D-state supported by the device in the S1 state.	7.2.12
_S2D	Highest D-state supported by the device in the S2 state.	7.2.13
_S3D	Highest D-state supported by the device in the S3 state.	7.2.14
_S4D	Highest D-state supported by the device in the S4 state.	7.2.15
_SB	System bus scope	5.3.1

Table 5-43 Defined Generic Object and Control Methods *(continued)*

Object	Description	Reference
_SBS	Smart Battery object that returns Smart Battery configuration.	11.1.2
_SCP	Thermal zone object that sets user cooling policy (Active or Passive).	12.3.7
_SEG	Bus identification object that evaluates to a bus's segment number.	6.5.6
_SHR	Resource data type reserved field name	16.4.2
_SI	System indicators scope	5.3.1
_SIZ	Resource data type reserved field name	16.4.2
_SPD	Control method used to update which video device will be posted at boot.	B.4.5
_SRS	Device configuration control method that sets a device's settings.	6.2.10
_SST	System indicator control method that indicates the system status.	10.1.1
_STA	1. Device insertion/removal control method that returns a device's status. 2. Power resource object that evaluates to the current on or off state of the Power Resource.	6.3.6 7.1.4
_STM	IDE device control method used to set the IDE controller transfer timings.	10.8.3
_STR	Object evaluates to a Unicode string to describe a device.	6.1.5
_SUN	Object that evaluates to the slot unique ID number for a slot.	6.1.6
_T_x	Reserved for use by the ASL compiler.	16.2.1.1
_TC1	Thermal zone object that contains thermal constant for Passive cooling.	12.3.7
_TC2	Thermal zone object that contains thermal constant for Passive cooling.	12.3.8
_TMP	Thermal zone object that returns current temperature in tenths of degrees Kelvin.	12.3.9
_TRA	Resource data type reserved field name	16.4.2
_TRS	Resource data type reserved field name	16.4.2
_TSP	Thermal zone object that contains thermal sampling period for Passive cooling.	12.3.10
_TTP	Resource data type reserved field name	16.4.2
_TYP	Resource data type reserved field name	16.4.2
_TZ	ACPI 1.0 thermal zone scope	5.3.1
_TZD	Object evaluates to a package of device names associated with a Thermal Zone.	12.3.11
_TZP	Thermal zone polling frequency in tenths of seconds.	12.3.12
_UID	Device identification object that specifies a device's unique persistent ID, or a control method that generates it.	6.1.7
_VPO	Returns 32-bit integer indicating the video post options.	B.4.6
_WAK	Power management control method run once system is awakened.	7.3.5

Deleted: 6

6.4.3.7 Generic Register Descriptor (Type 1, Large Item Name 0x2)

The generic register descriptor describes the location of a fixed width register within any of the ACPI-defined address spaces.

Table 6-32 Generic Register Descriptor Definition

Offset	Field Name, ASL Field Name	Definition
Byte 0	Generic register descriptor	Value = 10000010B (Type = 1, Large item name = 0x2)
Byte 1	Length, bits[7:0]	Value = 00001100B (12)
Byte 2	Length, bits[15:8]	Value = 00000000B (0)
Byte 3	Address Space ID, _ASI	The address space where the data structure or register exists. Defined values are: 0–System Memory 1–System I/O 2–PCI Configuration Space 3–Embedded Controller 4–SMBus 0x7F–Functional Fixed Hardware
Byte 4	Register Bit Width, _RBW	Indicates the register width in bits.
Byte 5	Register Bit Offset, _RBO	Indicates the offset to the start of the register in bits from the Register Address.

Deleted: 011

Deleted: 1

Table 6-32 Generic Register Descriptor Definition *(continued)*

Offset	Field Name, ASL Field Name	Definition
Byte 6	Reserved	Must be 0.
Byte 7	Register Address, _ADR bits[7:0]	Register Address
Byte 8	Register Address, _ADR bits[15:8]	
Byte 9	Register Address, _ADR bits[23:16]	
Byte 10	Register Address, _ADR bits[31:24]	
Byte 11	Register Address, _ADR bits[39:32]	
Byte 12	Register Address, _ADR bits[47:40]	
Byte 13	Register Address, _ADR bits[55:48]	
Byte 14	Register Address, _ADR bits[63:56]	

Deleted: 6

Deleted: 7

Deleted: 8

Deleted: 9

Deleted: 0

Deleted: 1

Deleted: 2

Deleted: 3

See section 16.2.4.16, “ASL Macro for Generic Register Descriptor,” for a description of the ASL macro that creates a Generic Register descriptor.

6.5.1 _INI (Init)

_INI is a device initialization object that performs device specific initialization. This control method is located under a device object and is run only when OSPM loads a description table. There are restrictions related to when this method is called and governing writing code for this method. The _INI method must only access Operation Regions that have been indicated to available as defined by the _REG method. The _REG method is described in section 6.5.4, “_REG (Region).” This control method is run before _ADR, _CID, _HID, _SUN, and _UID are run.

If the _STA method indicates that the device is present, OSPM will evaluate the __INI for the device (if the _INI method exists) and will examine each of the children of the device for _INI methods. If the _STA method indicates that the device is not present, OSPM will not run the _INI and will not examine the children of the device for _INI methods. If the device becomes present after the table has already been loaded, OSPM will not evaluate the _INI method, nor examine the children for _INI methods.

The _INI control method is generally used to switch devices out of a legacy operating mode. For example, BIOSes often configure CardBus controllers in a legacy mode to support legacy operating systems. Before enumerating the device with an ACPI operating system, the CardBus controllers must be initialized to CardBus mode. For such systems, the vendor can include an _INI control method under the CardBus controller to switch the device into CardBus mode.

In addition to device initialization, OSPM unconditionally evaluates an _INI object under the _SB namespace, if present, at the beginning of namespace initialization.

12.4 Thermal Zone Object Requirements

While not all thermal zone objects are required to be present in each thermal zone defined in the namespace, OSPM levies conditional requirements for the presence of specific thermal zone objects based on the definition of other related thermal zone objects. These requirements are outlined below:

- All thermal zones must contain the _TMP object.
- A thermal zone must define at least one trip point: _CRT, _HOT, _ACx, or _PSV.
- If _ACx is defined then an associated _ALx must be defined (e.g. defining _AC0 requires _AL0 also be defined).
- If _PSV is defined then either _PSL or _TZD must be defined. _PSL and _TZD may both be defined.
- If _PSL is defined then:

If a performance control register is defined (via either P_BLK or _PTC) for a processor defined in _PSL then _TC1, _TC2, and _TSP must be defined.
If a performance control register is not defined (via either P_BLK or _PTC) for a processor defined in _PSL then the processor must support processor performance states (in other words, the processor's processor object must include _PCT, _PSS, and _PPC).

- If _PSV is defined and _PSL is not defined (in other words, only _TZD is defined) then at least one device in the _TZD device list must support device performance states.
- _SCP is optional.
- _TZD is optional outside of the _PSV requirement outlined above.
- If _HOT is defined then the system must support the S4 sleeping state.

Deleted: CS4

Deleted: CS4

12.5 Thermal Zone Examples

12.5.1 Example: The Basic Thermal Zone

The following ASL describes a basic configuration where the entire system is treated as a single thermal zone. Cooling devices for this thermal zone consist of a processor and one single-speed fan. This is an example only.

Notice that this thermal zone object (TZ0) is defined in the _SB scope. Thermal zone objects should appear in the namespace under the portion of the system that comprises the thermal zone. For example, a thermal zone that is isolated to a docking station should be defined within the scope of the docking station device. Besides providing for a well-organized namespace, this configuration allows OSPM to dynamically adjust its thermal policy as devices are added or removed from the system.

```
Scope(\_SB) {
    Processor(
        CPU0,
        1,          // unique number for this processor
        0x110,      // system IO address of Pblk Registers
        0x06        // length in bytes of PBlk
        ) {}
```

```
Scope(\_SB.PCI0.ISA0) {
    Device(EC0) {
        Name(_HID, EISAID("PNP0C09"))     // ID for this EC
        // current resource description for this EC
        Name(_CRS,
                    ResourceTemplate() {
                        IO(Decode16,0x62,0x62,0,1)
                        IO(Decode16,0x66,0x66,0,1)
            })
        Name(_GPE, 0)      // GPE index for this EC

        // create EC's region and field for thermal support
        OperationRegion(EC0, EmbeddedControl, 0, 0xFF)
        Field(EC0, ByteAcc, Lock, Preserve) {
            MODE,   1,      // thermal policy (quiet/perform)
            FAN,    1,      // fan power (on/off)
            ,       6,      // reserved
            TMP,    8,      // current temp
            AC0,    8,      // active cooling temp (fan high)
            ,       8,      // reserved
            PSV,    8,      // passive cooling temp
            HOT,    8,      // critical S4 temp
            CRT,    8       // critical temp
            }

        // following is a method that OSPM will schedule after
        // it receives an SCI and queries the EC to receive value 7
        Method(_Q07) {
            Notify (\_SB.PCI0.ISA0.EC0.TZ0, 0x80)
        }   // end of Notify method

        // fan cooling on/off - engaged at AC0 temp
        PowerResource(PFAN, 0, 0) {
            Method(_STA) { Return (\_SB.PCI0.ISA0.EC0.FAN) }     // check power state
            Method(_ON) { Store (One, \_SB.PCI0.ISA0.EC0.FAN) }         // turn on fan
            Method(_OFF) { Store ( Zero, \_SB.PCI0.ISA0.EC0.FAN) }      // turn off fan
        }

        // Create FAN device object
        Device (FAN) {
            // Device ID for the FAN
            Name(_HID, EISAID("PNP0C0B"))
            // list power resource for the fan
            Name(_PR0, Package(){PFAN})
        }

        // create a thermal zone
        ThermalZone (TZ0) {
            Method(_TMP) { Return (\_SB.PCI0.ISA0.EC0.TMP )} // get current temp
            Method(_AC0) { Return (\_SB.PCI0.ISA0.EC0.AC0) } // fan high temp
            Name(_AL0, Package(){\_SB.PCI0.ISA0.EC0.FAN})    // fan is act cool dev
            Method(_PSV) { Return (\_SB.PCI0.ISA0.EC0.PSV) } // passive cooling temp
            Name(_PSL, Package (){\_SB.CPU0})                // passive cooling devices
            Method(_HOT) { Return (\_SB.PCI0.ISA0.EC0.HOT) } // get critical S4 temp
            Method(_CRT) { Return (\_SB.PCI0.ISA0.EC0.CRT) } // get critical temp
            Method(_SCP, 1) { Store (Arg1, \_SB.PCI0.ISA0.EC0.MODE) } // set cooling mode
            Name(_TC1, 4)                                    // bogus example constant
            Name(_TC2, 3)                                    // bogus example constant
            Name(_TSP, 150)                                  // passive sampling = 15 sec
            Name(_TZP, 0)                                    // polling not required
        } // end of TZ0

    }   // end of ECO
}   // end of \_SB.PCI0.ISA0 scope-

} // end of \_SB scope
```

Deleted: CS4

Deleted: CS4

Deleted: CS4

12.5.2 Example: Multiple-Speed Fans

The following ASL describes a thermal zone consisting of a processor and one dual-speed fan. As with the previous example, this thermal zone object (TZ0) is defined in the _SB scope and represents the entire system. This is an example only.

```
Scope(\_SB) {
    Processor(
        CPU0,
        1,          // unique number for this processor
        0x110,      // system IO address of Pblk Registers
        0x06        // length in bytes of PBlk
        ) {}


Scope(\_SB.PCI0.ISA0) {
    Device(EC0) {
        Name(_HID, EISAID("PNP0C09"))     // ID for this EC
        // current resource description for this EC
        Name(_CRS,
                    ResourceTemplate() {
                        IO(Decode16,0x62,0x62,0,1)
                        IO(Decode16,0x66,0x66,0,1)
            })
        Name(_GPE, 0)      // GPE index for this EC

        // create EC's region and field for thermal support
        OperationRegion(EC0, EmbeddedControl, 0, 0xFF)
        Field(EC0, ByteAcc, Lock, Preserve) {
            MODE,   1,      // thermal policy (quiet/perform)
            FAN0,   1,      // fan strength high/off
            FAN1,   1,      // fan strength low/off
            ,       5,      // reserved
            TMP,    8,      // current temp
            AC0,    8,      // active cooling temp (high)
            AC1,    8,      // active cooling temp (low)
            PSV,    8,      // passive cooling temp
            HOT,    8,      // critical S4 temp
            CRT,    8       // critical temp
        }

        // following is a method that OSPM will schedule after it
        // receives an SCI and queries the EC to receive value 7
        Method(_Q07) {
            Notify (\_SB.PCI0.ISA0.EC0.TZ0, 0x80)
        } end of Notify method

        // fan cooling mode high/off - engaged at AC0 temp
        PowerResource(FN10, 0, 0) {
            Method(_STA) { Return (\_SB.PCI0.ISA0.EC0.FAN0) }    // check power state
            Method(_ON) { Store (One, \_SB.PCI0.ISA0.EC0.FAN0) } // turn on fan at high
            Method(_OFF) { Store (Zero, \_SB.PCI0.ISA0.EC0.FAN0) }// turn off fan
        }

        // fan cooling mode low/off   - engaged at AC1 temp
        PowerResource(FN11, 0, 0) {
            Method(_STA) { Return (\_SB.PCI0.ISA0.EC0.FAN1) }    // check power state
            Method(_ON) { Store (One, \_SB.PCI0.ISA0.EC0.FAN1) } // turn on fan at low
            Method(_OFF) { Store (Zero, \_SB.PCI0.ISA0.EC0.FAN1) }// turn off fan
        }
```

Deleted: CS4

```
        // Following is a single fan with two speeds.  This is represented
        // by creating two logical fan devices.  When FN2 is turned on then
        // the fan is at a low speed.  When FN1 and FN2 are both on then
        // the fan is at high speed.
        //
        // Create FAN device object FN1
        Device (FN1) {
            // Device ID for the FAN
            Name(_HID, EISAID("PNP0C0B"))
            Name(_PR0, Package(){FN10, FN11})
        }

        // Create FAN device object FN2
        Device (FN2) {
            // Device ID for the FAN
            Name(_HID, EISAID("PNP0C0B"))
            Name(_PR0, Package(){FN10})
        }

        // create a thermal zone
        ThermalZone (TZ0) {
            Method(_TMP) { Return (\_SB.PCI0.ISA0.EC0.TMP )}  // get current temp
            Method(_AC0) { Return (\_SB.PCI0.ISA0.EC0.AC0) }  // fan high temp
            Method(_AC1) { Return (\_SB.PCI0.ISA0.EC0.AC1) }  // fan low temp
            Name(_AL0, Package() {\_SB.PCI0.ISA0.EC0.FN1})    // active cooling (high)
            Name(_AL1, Package() {\_SB.PCI0.ISA0.EC0.FN2})    // active cooling (low)
            Method(_PSV) { Return (\_SB.PCI0.ISA0.EC0.PSV) }  // passive cooling temp
            Name(_PSL, Package() {\_SB.CPU0})                 // passive cooling devices
            Method(_HOT) { Return (\_SB.PCI0.ISA0.EC0.HOT) } // get critical S4 temp
            Method(_CRT) { Return (\_SB.PCI0.ISA0.EC0.CRT) } // get crit. temp
            Method(_SCP, 1) { Store (Arg1, \_SB.PCI0.ISA0.EC0.MODE) } // set cooling mode
            Name(_TC1, 4)                                     // bogus example constant
            Name(_TC2, 3)                                     // bogus example constant
            Name(_TSP, 150)                                   // passive sampling = 15 sec
            Name(_TZP, 0)                                     // polling not required
        } // end of TZ0

    } // end of ECO
}   // end of \_SB.PCI0.ISA0 scope


} // end of \_SB scope
```

Deleted: CS4

Deleted: CS4

14.2 Declaring SMBus Host Controller Objects

EC-based SMBus 1.0-compatible HCs should be modeled in the ACPI namespace as described in section 13.12, “Defining an Embedded Controller SMBus Host Controller in ACPI Namespace.” An example definition is given below. Using the HID value “ACPI0001” identifies that this SMB-HC is implemented on an embedded controller using the standard SMBus register set defined in section 13.9, SMBus Host Controller Interface via Embedded Controller.”

```
Device (SMB0)
{
    Name(_HID, "ACPI0001")        // EC-based SMBus 1.0 compatible Host Controller
    Name(_EC, 0x2030)             // EC offset 0x20, query bit 0x30
      :
}
```

Deleted: 0

EC-based SMBus 2.0-compatible host controllers should be defined similarly in the name space as follows:

```
Device (SMB0)
{
    Name(_HID, "ACPI0005")        // EC-based SMBus 2.0 compatible Host Controller
    Name(_EC, 0x2030)             // EC offset 0x20, query bit 0x30
      :
}
```

Deleted: 1

Non–EC-based SMB-HCs should be modeled in a manner similar to the EC-based SMBus HC. An example definition is given below. These devices use a vendor-specific hardware identifier (HID) to specify the type of SMB-HC (do not use “ACPI0001” or “ACPI0005”). Using a vendor-specific HID allows the correct software to be loaded to service this segment’s SMBus address space.

```
Device(SMB0)
{
    Name(_HID, "<Vendor-Specific HID>")  // Vendor-Specific HID
      :
}
```

Regardless of the type of hardware, some OS software element (for example, the SMBus HC driver) must register with OSPM to support all SMBus operation regions defined for the segment. This software allows the generic SMBus interface defined in this section to be used on a specific hardware implementation by translating between the conceptual (for example, SMBus address space) and physical (for example, process of writing/reading registers) models. Because of this linkage, SMBus operation regions must be defined immediately within the scope of the corresponding SMBus device.

16.1.3 ASL Language and Terms

```
ASLCode                     := DefinitionBlockTerm

DefinitionBlockTerm         := DefinitionBlock(
                                   AMLFileName,          //StringData
                                   TableSignature,       //StringData
                                   ComplianceRevision,   //ByteConst
                                   OEMID,                //StringData
                                   TableID,              //StringData
                                   OEMRevision           //DWordConst
                               ) {TermList}

TermList                    := Nothing | <Term TermList>
Term                        := Object | Type1Opcode | Type2Opcode

CompilerDirective           := IncludeTerm | ExternalTerm

ObjectList                  := Nothing | <Object ObjectList>
Object                      := CompilerDirective | NamedObject | NameSpaceModifier

DataObject                  := BufferData | PackageData | IntegerData | StringData
DataRefObject               := DataObject | ObjectReference | DDBHandle
```

```
ComputationalData           := BufferData | IntegerData | StringData
BufferData                  := Type5Opcode | BufferTerm
PackageData                 := PackageTerm
IntegerData                 := Type3Opcode | Integer | ConstTerm
StringData                  := Type4Opcode | String

NamedObject                 := BankFieldTerm | CreateBitFieldTerm | CreateByteFieldTerm
                               | CreateDWordFieldTerm | CreateFieldTerm |
                               CreateQWordFieldTerm | CreateWordFieldTerm |
                               DataRegionTerm | DeviceTerm | EventTerm | FieldTerm |
                               IndexFieldTerm | MethodTerm | MutexTerm | OpRegionTerm |
                               PowerResTerm | ProcessorTerm | ThermalZoneTerm

NameSpaceModifier           := AliasTerm | NameTerm | ScopeTerm

UserTerm                    := NameString(               //NameString=>Method
                                   ArgList
                               ) => Nothing | DataRefObject
ArgList                     := Nothing | <TermArg ArgListTail>
ArgListTail                 := Nothing | <',' TermArg ArgListTail>
TermArg                     := Type2Opcode | DataObject | ArgTerm | LocalTerm |
                               NameString
Target                      := Nothing | SuperName
```

Deleted: Ref

```
Type1Opcode             := BreakTerm | BreakPointTerm | ContinueTerm | FatalTerm |
                           IfElseTerm | LoadTerm | NoOpTerm | NotifyTerm |
                           ReleaseTerm | ResetTerm | ReturnTerm | SignalTerm |
                           SleepTerm | StallTerm | SwitchTerm | UnloadTerm |
                           WhileTerm
                           // A Type1OpCode term can only be used standing alone on
                           a
                           // line of ASL code; because these types of terms do not
                           // return a value so they cannot be used as a term in an
                           // expression.

Type2Opcode             := AcquireTerm | AddTerm | AndTerm | ConcatTerm |
                           ConcatResTerm | CondRefOfTerm | CopyObjectTerm |
                           DecTerm | DerefOfTerm | DivideTerm |
                           FindSetLeftBitTerm | FindSetRightBitTerm | FromBCDTerm |
                           IncTerm | IndexTerm | LAndTerm | LEqualTerm |
                           LGreaterTerm | LGreaterEqualTerm | LLessTerm |
                           LLessEqualTerm | LNotTerm | LNotEqualTerm |
                           LoadTableTerm | LOrTerm | MatchTerm | MidTerm |
                           ModTerm | MultiplyTerm | NAndTerm | NOrTerm | NotTerm |
                           ObjectTypeTerm | OrTerm | RefOfTerm | ShiftLeftTerm |
                           ShiftRightTerm | SizeOfTerm | StoreTerm | SubtractTerm |
                           ToBCDTerm | ToBufferTerm | ToDecimalStringTerm |
                           ToHexStringTerm | ToIntegerTerm | ToStringTerm |
                           WaitTerm | XorTerm | UserTerm
                           // A Type2Opcode term returns a value that can be used
                           // inan expression.

Type3Opcode             := AddTerm | AndTerm | DecTerm | DivideTerm | EISAIDTerm |
                           FindSetLeftBitTerm | FindSetRightBitTerm | FromBCDTerm |
                           IncTerm | IndexTerm | LAndTerm | LEqualTerm |
                           LGreaterTerm | LGreaterEqualTerm | LLessTerm |
                           LLessEqualTerm | LNotTerm | LNotEqualTerm | LOrTerm |
                           MatchTerm | ModTerm | MultiplyTerm | NAndTerm |
                           NOrTerm | NotTerm | OrTerm | ShiftLeftTerm |
                           ShiftRightTerm | SubtractTerm | ToBCDTerm |
                           ToIntegerTerm | XorTerm
                           // A Type3Opcode evaluates to an Integer, can't have a
                           // destination and must have either Type3Opcode,
                           // Type4Opcode, Type5Opcode, ConstExprTerm, Integer,
                           // BufferTerm, Package or String for all arguments.

Type4Opcode             := ConcatTerm | MidTerm | ToDecimalStringTerm |
                           ToHexStringTerm | ToStringTerm
                           // A Type4Opcode evaluates to an String, can't have a
                           // destination and must have either Type3Opcode,
                           // Type4Opcode, Type5Opcode, ConstExprTerm, Integer,
                           // BufferTerm, PackageTerm or String for all arguments.

Type5Opcode             := ConcatTerm | ConcatResTerm | MidTerm |
                           ResourceTemplateTerm | ToBufferTerm | UnicodeTerm
                           // A Type5Opcode evaluates to a BufferTerm, can't
                           // have a destination and must have either Type3Opcode,
                           // Type4Opcode, Type5Opcode, ConstExprTerm, Integer,
                           // BufferTerm, PackageTerm or String for all arguments.

Type6Opcode             := RefOfTerm | DerefOfTerm | IndexTerm | UserTerm

IncludeTerm             := Include(
                               IncFilePathName        //StringData
                           )

ExternalTerm            := External(
                               ObjName,               //NameString
                               ObjType                //Nothing | ObjectTypeKeyword
                           )
```

```
BankFieldTerm                := BankField(
                                    RegionName,           //NameString=>OperationRegion
                                    BankName,             //NameString=>FieldUnit
                                    BankValue,            //TermArg=>Integer
                                    AccessType,           //AccessTypeKeyword
                                    LockRule,             //LockRuleKeyword
                                    UpdateRule            //UpdateRuleKeyword
                                ) {FieldUnitList}
```

```
FieldUnitList               := Nothing | <FieldUnit FieldUnitListTail>
FieldUnitListTail           := Nothing | <',' FieldUnit FieldUnitListTail>

FieldUnit                   := FieldUnitEntry | OffsetTerm | AccessAsTerm
FieldUnitEntry              := <Nothing | NameSeg> ',' Integer

OffsetTerm                  := Offset(
                                   ByteOffset            //IntegerData
                               )

AccessAsTerm                := AccessAs(
                                   AccessType,           //AccessTypeKeyword
                                   AccessAttribute       //Nothing | ByteConstExpr |
                                                         //AccessAttribKeyword
                               )

CreateBitFieldTerm          := CreateBitField(
                                   SourceBuffer,         //TermArg=>Buffer
                                   BitIndex,             //TermArg=>Integer
                                   BitFieldName          //NameString
                               )

CreateByteFieldTerm         := CreateByteField(
                                   SourceBuffer,         //TermArg=>Buffer
                                   ByteIndex,            //TermArg=>Integer
                                   ByteFieldName         //NameString
                               )

CreateDWordFieldTerm        := CreateDWordField(
                                   SourceBuffer,         //TermArg=>Buffer
                                   ByteIndex,            //TermArg=>Integer
                                   DWordFieldName        //NameString
                               )

CreateFieldTerm             := CreateField(
                                   SourceBuffer,         //TermArg=>Buffer
                                   BitIndex,             //TermArg=>Integer
                                   NumBits,              //TermArg=>Integer
                                   FieldName             //NameString
                               )

CreateQWordFieldTerm        := CreateQWordField(
                                   SourceBuffer,         //TermArg=>Buffer
                                   ByteIndex,            //TermArg=>Integer
                                   QWordFieldName        //NameString
                               )

CreateWordFieldTerm         := CreateWordField(
                                   SourceBuffer,         //TermArg=>Buffer
                                   ByteIndex,            //TermArg=>Integer
                                   WordFieldName         //NameString
                               )

DataRegionTerm              := DataTableRegion(
                                   RegionName,           // NameString
                                   SignatureString,      // TermArg=>String
                                   OemIDString,          // TermArg=>String
                                   OemTableIDString      // TermArg=>String
                               )

DeviceTerm                  := Device(
                                   DeviceName            //NameString
                               ) {ObjectList}

EventTerm                   := Event(
                                   EventName             //NameString
                               )
```

```
FieldTerm                   := Field(
                                   RegionName,          //NameString=>OperationRegion
                                   AccessType,          //AccessTypeKeyword
                                   LockRule,            //LockRuleKeyword
                                   UpdateRule           //UpdateRuleKeyword
                               ) {FieldUnitList}

IndexFieldTerm              := IndexField(
                                   IndexName,           //NameString=>FieldUnit
                                   DataName,            //NameString=>FieldUnit
                                   AccessType,          //AccessTypeKeyword
                                   LockRule,            //LockRuleKeyword
                                   UpdateRule           //UpdateRuleKeyword
                               ) {FieldUnitList}

MethodTerm                  := Method(
                                   MethodName,          //NameString
                                   NumArgs,             //Nothing | ByteConstExpr
                                   SerializeRule,       //Nothing |
                                                        //SerializeRuleKeyword
                                   SyncLevel            //Nothing | ByteConstExpr
                               ) {TermList}

MutexTerm                   := Mutex(
                                   MutexName,           //NameString
                                   SyncLevel            //ByteConstExpr
                               )

OpRegionTerm                := OperationRegion(
                                   RegionName,          //NameString
                                   RegionSpace,         //RegionSpaceKeyword
                                   Offset,              //TermArg=>Integer
                                   Length               //TermArg=>Integer
                               )

PowerResTerm                := PowerResource(
                                   ResourceName,        //NameString
                                   SystemLevel,         //ByteConstExpr
                                   ResourceOrder        //WordConstExpr
                               ) {ObjectList}

ProcessorTerm               := Processor(
                                   ProcessorName,       //NameString
                                   ProcessorID,         //ByteConstExpr
                                   PBlockAddress,       //DWordConstExpr|Nothing (=0)
                                   PblockLength         //ByteConstExpr|Nothing (=0)
                               ) {ObjectList}

ThermalZoneTerm             := ThermalZone(
                                   ThermalZoneName      //NameString
                               ) {ObjectList}

AliasTerm                   := Alias(
                                   SourceObject,        //NameString
                                   AliasObject          //NameString
                               )

NameTerm                    := Name(
                                   ObjectName,          //NameString
                                   Object               //DataObject
                               )

ScopeTerm                   := Scope(
                                   Location             //NameString
                               ) {ObjectList}

BreakTerm                   := Break

BreakPointTerm              := BreakPoint

ContinueTerm                := Continue
```

Deleted: Ref

```
FatalTerm               := Fatal(
                               Type,                //ByteConstExpr
                               Code,                //DWordConstExpr
                               Arg                  //TermArg=>Integer
                           )

IfElseTerm              := IfTerm ElseTerm

IfTerm                  := If(
                               Predicate            //TermArg=>Integer
                           ) {TermList}

ElseTerm                := Nothing | <Else {TermList}> | <ElseIf (
                               Predicate            //TermArg=>Integer
                           ) {TermList} ElseTerm>

LoadTerm                := Load(
                               Object,              //NameString
                               DDBHandle            //SuperName
                           )

NoOpTerm                := Noop

NotifyTerm              := Notify(
                               Object,
                               //SuperName=>ThermalZone|Processor|Device
                               NotificationValue    //TermArg=>Integer
                           )

ReleaseTerm             := Release(
                               SyncObject           //SuperName
                           )

ResetTerm               := Reset(
                               SyncObject           //SuperName
                           )

ReturnTerm              := Return(
                               Arg                  //TermArg=>DataRefObject
                           )

SignalTerm              := Signal(
                               SyncObject           //SuperName
                           )

SleepTerm               := Sleep(
                               MilliSecs            //TermArg=>Integer
                           )

StallTerm               := Stall(
                               MicroSecs            //TermArg=>Integer
                           )

SwitchTerm              := Switch(
                               Predicate            //TermArg=>ComputationalData
                           ) {CaseTermList}
CaseTermList            := Nothing | CaseTerm | DefaultTerm DefaultTermList |
                           CaseTerm CaseTermList
DefaultTermList         := Nothing | CaseTerm | CaseTerm DefaultTermList
CaseTerm                := Case(
                               Value                //DataObject
                           ) {TermList}
DefaultTerm             := Default {TermList}

UnloadTerm              := Unload(
                               DDBHandle            //SuperName
                           )

WhileTerm               := While(
                               Predicate            //TermArg=>Integer
                           ) {TermList}
```

```
AcquireTerm             := Acquire(
                               SyncObject,          //SuperName=>Mutex
                               TimeoutValue         //WordConstExpr
                           ) => Boolean             // True means timed-out

AddTerm                 := Add(
                               Addend1,             //TermArg=>Integer
                               Addend2,             //TermArg=>Integer
                               Result               //Target
                           ) => Integer

AndTerm                 := And(
                               Source1,             //TermArg=>Integer
                               Source2,             //TermArg=>Integer
                               Result               //Target
                           ) => Integer

ConcatTerm              := Concatenate(
                               Source1,             //TermArg=>ComputationalData
                               Source2,             //TermArg=>ComputationalData
                               Result               //Target
                           ) => ComputationalData

ConcatResTerm           := ConcatenateResTemplate(
                               Source1,             //TermArg=>Buffer
                               Source2,             //TermArg=>Buffer
                               Result               //Target
                           ) => Buffer

CondRefOfTerm           := CondRefOf(
                               Source,              //SuperName
                               Destination          //Target
                           ) => Boolean

CopyObjectTerm          := CopyObject(
                               Source,              //TermArg=>DataRefObject
                               Result,              //NameString | LocalTerm |
                                                    ArgTerm
                           ) => DataRefObject

DecTerm                 := Decrement(
                               Addend               //SuperName
                           ) => Integer

DerefOfTerm             := DerefOf(
                               Source               //TermArg=>ObjectReference
                                                    //ObjectReference is an
                           object
                                                    //produced by terms such as
                                                    //Index, RefOf or CondRefOf.
                           ) => DataRefObject

DivideTerm              := Divide(
                               Dividend,            //TermArg=>Integer
                               Divisor,             //TermArg=>Integer
                               Remainder,           //Target
                               Result               //Target
                           ) => Integer             //returns Result
```

```
FindSetLeftBitTerm          := FindSetLeftBit(
                                   Source,              //TermArg=>Integer
                                   Result               //Target
                               ) => Integer

FindSetRightBitTerm         := FindSetRightBit(
                                   Source,              //TermArg=>Integer
                                   Result               //Target
                               ) => Integer

FromBCDTerm                 := FromBCD(
                                   BCDValue,            //TermArg=>Integer
                                   Result               //Target
                               ) => Integer

IncTerm                     := Increment(
                                   Addend               //SuperName
                               ) => Integer

IndexTerm                   := Index(
                                   Source,              //TermArg=>
                                                        //< String | Buffer |
                                PackageTerm>
                                   Index,               //TermArg=>Integer
                                   Destination          //Target
                               ) => ObjectReference

LAndTerm                    := LAnd(
                                   Source1,             //TermArg=>Integer
                                   Source2              //TermArg=>Integer
                               ) => Boolean

LEqualTerm                  := LEqual(
                                   Source1,             //TermArg=>ComputationalData
                                   Source2              //TermArg=>ComputationalData
                               ) => Boolean

LGreaterTerm                := LGreater(
                                   Source1,             //TermArg=>ComputationalData
                                   Source2              //TermArg=>ComputationalData
                               ) => Boolean

LGreaterEqualTerm           := LGreaterEqual(
                                   Source1,             //TermArg=>ComputationalData
                                   Source2              //TermArg=>ComputationalData
                               ) => Boolean

LLessTerm                   := LLess(
                                   Source1,             //TermArg=>ComputationalData
                                   Source2              //TermArg=>ComputationalData
                               ) => Boolean

LLessEqualTerm              := LLessEqual(
                                   Source1,             //TermArg=>ComputationalData
                                   Source2              //TermArg=>ComputationalData
                               ) => Boolean

LNotTerm                    := LNot(
                                   Source,              //TermArg=>ComputationalData
                               ) => Boolean
```

```
LNotEqualTerm           := LNotEqual(
                               Source1,            //TermArg=>ComputationalData
                               Source2             //TermArg=>ComputationalData
                           ) => Boolean

LoadTableTerm           := LoadTable(
                               SignatureString,    // TermArg=>String
                               OemIDString,        // TermArg=>String
                               OemTableIDString,   // TermArg=>String
                               RootPathString,     // Nothing | TermArg=>String
                               ParameterPathString,// Nothing | TermArg=>String
                               ParameterData       // Nothing |
                           TermArg=>DataRefObject
                           ) => DDBHandle

LOrTerm                 := LOr(
                               Source1,            //TermArg=>ComputationalData
                               Source2             //TermArg=>ComputationalData
                           ) => Boolean

MatchTerm               := Match(
                               SearchPackage,      //TermArg=>Package
                               Op1,                //MatchOpKeyword
                               MatchObject1,       //TermArg=>Integer
                               Op2,                //MatchOpKeyword
                               MatchObject2,       //TermArg=>Integer
                               StartIndex          //TermArg=>Integer
                           ) => Ones | Integer

MidTerm                 := Mid(
                               Source,             //TermArg=>Buffer|String
                               Index,              //TermArg=>Integer
                               Length,             //TermArg=>Integer
                               Result              //Target
                           ) => Buffer|String

ModTerm                 := Mod(
                               Dividend,           //TermArg=>Integer
                               Divisor,            //TermArg=>Integer
                               Result              //Target
                           ) => Integer            //returns Result


MultiplyTerm            := Multiply(
                               Multiplicand,       //TermArg=>Integer
                               Multiplier,         //TermArg=>Integer
                               Result              //Target
                           ) => Integer

NAndTerm                := NAnd(
                               Source1,            //TermArg=>Integer
                               Source2             //TermArg=>Integer
                               Result              //Target
                           ) => Integer

NOrTerm                 := NOr(
                               Source1,            //TermArg=>Integer
                               Source2             //TermArg=>Integer
                               Result              //Target
                           ) => Integer

NotTerm                 := Not(
                               Source,             //TermArg=>Integer
                               Result              //Target
                           ) => Integer

ObjectTypeTerm          := ObjectType(
                               Object              //SuperName
                           ) => Integer
```

```
OrTerm                  := Or(
                               Source1,              //TermArg=>Integer
                               Source2               //TermArg=>Integer
                               Result                //Target
                           ) => Integer

RefOfTerm               := RefOf(
                               Object                //SuperName
                           ) => ObjectReference

ShiftLeftTerm           := ShiftLeft(
                               Source,               //TermArg=>Integer
                               ShiftCount            //TermArg=>Integer
                               Result                //Target
                           ) => Integer

ShiftRightTerm          := ShiftRight(
                               Source,               //TermArg=>Integer
                               ShiftCount            //TermArg=>Integer
                               Result                //Target
                           ) => Integer

SizeOfTerm              := SizeOf(
                               DataObject
                               //SuperName=>String|Buffer|Package
                           ) => Integer

StoreTerm               := Store(
                               Source,               //TermArg=>DataRefObject
                               Destination           //SuperName
                           ) => DataRefObject

SubtractTerm            := Subtract(
                               Addend1,              //TermArg=>Integer
                               Addend2,              //TermArg=>Integer
                               Result                //Target
                           ) => Integer

ToBCDTerm               := ToBCD(
                               Value,                //TermArg=>Integer
                               Result                //Target
                           ) => Integer

ToBufferTerm            := ToBuffer(                                          Formatted
                               Data,                 //TermArg=>ComputationalData
                               Result                //Target
                           ) => ComputationalData

ToDecimalStringTerm     := ToDecimalString(                                   Formatted
                               Data,                 //TermArg=>ComputationalData
                               Result                //Target
                           ) => String

ToHexStringTerm         := ToHexString(                                       Formatted
                               Data,                 //TermArg=>ComputationalData
                               Result                //Target
                           ) => String

ToIntegerTerm           := ToInteger(                                         Formatted
                               Data,                 //TermArg=>ComputationalData
                               Result                //Target
                           ) => Integer

ToStringTerm            := ToString(                                          Formatted
                               Source,               //TermArg=>Buffer
                               Length,               //Nothing | TermArg=>Integer
                               Result                //Target
                           ) => String
```

```
WaitTerm                    := Wait(
                                   SyncObject,          //SuperName=>Event
                                   TimeoutValue         //TermArg=>Integer
                               ) => Boolean             // True means timed-out

XOrTerm                     := XOr(
                                   Source1,             //TermArg=>Integer
                                   Source2              //TermArg=>Integer
                                   Result               //Target
                               ) => Integer

ObjectTypeKeyword           := UnknownObj | IntObj | StrObj | BuffObj | PkgObj |
                               FieldUnitObj | DeviceObj | EventObj | MethodObj |
                               MutexObj | OpRegionObj | PowerResObj | ThermalZoneObj |
                               BuffFieldObj | DDBHandleObj

AccessTypeKeyword           := AnyAcc | ByteAcc | WordAcc | DWordAcc | QWordAcc |
                               BufferAcc
AccessAttribKeyword         := SMBQuick | SMBSendReceive | SMBByte | SMBWord | SMBBlock
                               | SMBProcessCall
                               // Note: AccessAttribKeywords are for SMBus BufferAcc
                               only.
```

```
LockRuleKeyword             := Lock | NoLock
UpdateRuleKeyword           := Preserve | WriteAsOnes | WriteAsZeros

RegionSpaceKeyword          := UserDefRegionSpace | SystemIO | SystemMemory |
                               PCI_Config | EmbeddedControl | SMBus | SystemCMOS |
                               PciBarTarget
AddressSpaceKeyword         := RegionSpaceKeyword | FFixedHW
UserDefRegionSpace          := IntegerData=>0x80-0xff

SerializeRuleKeyword        := Serialized | NotSerialized

MatchOpKeyword              := MTR | MEQ | MLE | MLT | MGE | MGT

DMATypeKeyword              := Compatibility | TypeA | TypeB | TypeF
BusMasterKeyword            := BusMaster | NotBusMaster
XferTypeKeyword             := Transfer8 | Transfer16 | Transfer8_16

ResourceTypeKeyword         := ResourceConsumer | ResourceProducer
MinKeyword                  := MinFixed | MinNotFixed
MaxKeyword                  := MaxFixed | MaxNotFixed
DecodeKeyword               := SubDecode | PosDecode
RangeTypeKeyword            := ISAOnlyRanges | NonISAOnlyRanges | EntireRange
MemTypeKeyword              := Cacheable | WriteCombining | Prefetchable | NonCacheable
ReadWriteKeyword            := ReadWrite | ReadOnly
InterruptTypeKeyword        := Edge | Level
InterruptLevel              := ActiveHigh | ActiveLow
ShareTypeKeyword            := Shared | Exclusive
IODecodeKeyword             := Decode16 | Decode10
TypeKeyword                 := TypeTranslation | TypeStatic
TranslationKeyword          := SparseTranslation | DenseTranslation
AddressKeyword              := AddressRangeMemory | AddressRangeReserved |
                               AddressRangeNVS | AddressRangeACPI

SuperName                   := NameString | ArgTerm | LocalTerm | DebugTerm |
                               Type6Opcode | UserTerm
ArgTerm                     := Arg0 | Arg1 | Arg2 | Arg3 | Arg4 | Arg5 | Arg6
LocalTerm                   := Local0 | Local1 | Local2 | Local3 | Local4 | Local5 |
                               Local6 | Local7
DebugTerm                   := Debug

LeadDigitChar               := '1'-'9'
OctalDigitChar              := '0'-'7'
HexDigitChar                := DigitChar | 'A'-'F' | 'a'-'f'


Integer                     := DecimalConst | OctalConst | HexConst
DecimalConst                := LeadDigitChar | <DecimalConst DigitChar>
OctalConst                  := '0' | <OctalConst OctalDigitChar>
HexConst                    := <0x HexDigitChar> | <0X HexDigitChar> | <HexConst
                               HexDigitChar>
ByteConst                   := Integer=>0x00-0xff
WordConst                   := Integer=>0x0000-0xffff
DWordConst                  := Integer=>0x00000000-0xffffffff
QWordConst                  := Integer=>0x0000000000000000-0xffffffffffffffff

DDBHandle                   := Integer
ObjectReference             := Integer
String                      := '"' AsciiCharList '"'
AsciiCharList               := Nothing | <EscapeSeq AsciiCharList> | <AsciiChar
                               AsciiCharList>
AsciiChar                   := 0x01-0x21 | 0x23-0x5B | 0x5D-0x7F
EscapeSeq                   := SimpleEscapeSeq | OctalEscapeSeq | HexEscapeSeq
SimpleEscapeSeq             := \' | \" | \a | \b | \f | \n | \r | \t | \v | \\
OctalEscapeSeq              :=  \ OctalDigit |
                                \ OctalDigit OctalDigit |
                                \ OctalDigit OctalDigit OctalDigit
OctalDigitChar              := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
HexEscapeSeq                :=  \x HexDigitChar |
                                \x HexDigitChar HexDigitChar
NullChar                    := 0x00
ConstTerm                   := Zero | One | Ones | Revision
```

```
Boolean                         := True | False
True                            := Ones
False                           := Zero
```

```
ByteConstExpr               := <Type3Opcode | ConstExprTerm | Integer> => ByteConst
WordConstExpr               := <Type3Opcode | ConstExprTerm | Integer> => WordConst
DwordConstExpr              := <Type3Opcode | ConstExprTerm | Integer> => DWordConst
QwordConstExpr              := <Type3Opcode | ConstExprTerm | Integer> => QWordConst
ConstExprTerm               := Zero | One | Ones


BufferTerm                  := Buffer(
                                   BuffSize             //Nothing |
                                                        //TermArg=>Integer
                               ) {StringData | ByteList}

ByteList                    := Nothing | <ByteConstExpr ByteListTail>
ByteListTail                := Nothing | <',' ByteConstExpr ByteListTail>

DWordList                   := Nothing | <DWordConstExpr DWordListTail>
DWordListTail               := Nothing | <',' DWordConstExpr DWordListTail>

PackageTerm                 := Package(
                                   NumElements          //Nothing |
                                                        //ByteConstExpr |
                                                        //TermArg=>Integer
                               ) {PackageList}

PackageList                 := Nothing | <PackageElement PackageListTail>
PackageListTail             := Nothing | <',' PackageElement PackageListTail>
PackageElement              := DataObject | NameString

EISAIDTerm                  := EISAID(
                                   EISAIDString         //StringData
                               ) => DWordConst

ResourceTemplateTerm        := ResourceTemplate() {ResourceMacroList} => Buffer

UnicodeTerm                 := Unicode(
                                   ASCIIString          //StringData
                               ) => Buffer

ResourceMacroList           := Nothing | <ResourceMacroTerm ResourceMacroList>
ResourceMacroTerm           := DMATerm | DWordIOTerm | DWordMemoryTerm |
                               EndDependentFnTerm | FixedIOTerm | InterruptTerm |
                               IOTerm | IRQNoFlagsTerm | IRQTerm | Memory24Term |
                               Memory32FixedTerm | Memory32Term | QWordIOTerm |
                               QWordMemoryTerm | RegisterTerm | StartDependentFnTerm |
                               StartDependentFnNoPriTerm | VendorLongTerm |
                               VendorShortTerm | WordBusNumberTerm | WordIOTerm

DMATerm                     := DMA(
                                   DMAType,             //DMATypeKeyword (_TYP)
                                   BusMaster,           //BusMasterKeyword (_BM)
                                   XferType,            //XferTypeKeyword (_SIZ)
                                   ResourceTag          //Nothing | NameString
                               ) {ByteList}             //List of channels (0-7)
```

```
DWordIOTerm                 := DWordIO(
                                 ResourceType,          //Nothing (ResourceConsumer)|
                                                        //ResourceTypeKeyword
                                 MinType,               //Nothing (MinNotFixed) |
                                                        //MinKeyword (_MIF)
                                 MaxType,               //Nothing (MaxNotFixed) |
                                                        //MaxKeyword (_MAF)
                                 Decode,                //Nothing (PosDecode) |
                                                        //DecodeKeyword (_DEC)
                                 RangeType,             //Nothing (EntireRange) |
                                                        //RangeTypeKeyword (_RNG)
                                 AddressGranularity,    //DWordConstExpr (_GRA)
                                 MinAddress,            //DWordConstExpr (_MIN)
                                 MaxAddress,            //DWordConstExpr (_MAX)
                                 Translation,           //DWordConstExpr (_TRA)
                                 AddressLen,            //DWordConstExpr (_LEN)
                                 ResSourceIndex,        //Nothing | ByteConstExpr
                                 ResSource,             //Nothing | StringData
                                 ResourceTag            //Nothing | NameString
                                 Type                   //Nothing | TypeKeyword
                                 TranslationType        //Nothing |
                                                        TranslationKeyword

                                                        )

DWordMemoryTerm             := DWordMemory(
                                 ResourceType,          //Nothing (ResourceConsumer)|
                                                        //ResourceTypeKeyword
                                 Decode,                //Nothing (PosDecode) |
                                                        //DecodeKeyword (_DEC)
                                 MinType,               //Nothing (MinNotFixed) |
                                                        //MinKeyword (_MIF)
                                 MaxType,               //Nothing (MaxNotFixed) |
                                                        //MaxKeyword (_MAF)
                                 MemType,               //Nothing (NonCacheable) |
                                                        //MemTypeKeyword (_MEM)
                                 ReadWriteType,         //ReadWriteKeyword (_RW)
                                 AddressGranularity,    //DWordConstExpr (_GRA)
                                 MinAddress,            //DWordConstExpr (_MIN)
                                 MaxAddress,            //DWordConstExpr (_MAX)
                                 Translation,           //DWordConstExpr (_TRA)
                                 AddressLen,            //DWordConstExpr (_LEN)
                                 ResSourceIndex,        //Nothing | ByteConstExpr
                                 ResSource,             //Nothing | StringData
                                 ResourceTag            //Nothing | NameString
                                 AddressRange           //Nothing | AddressKeyword
                             (_MTP)
                                 Type                   //Nothing | TypeKeyword
                             (_TTP)

                                 )

EndDependentFnTerm          := EndDependentFn()

FixedIOTerm                 := FixedIO(
                                 AddressBase,           //WordConstExpr (_BAS)
                                 RangeLen,              //ByteConstExpr (_LEN)
                                 ResourceTag            //Nothing | NameString
                                 )
```

Formatted

Deleted: Translation

Deleted: Translation

Formatted

```
InterruptTerm                := Interrupt(
                                    ResourceType,          //Nothing (ResourceConsumer)|
                                                           //ResourceTypeKeyword
                                    InterruptType,         //InterruptTypeKeyword
                                                           //(_LL, _HE)
                                    InterruptLevel,        //InterruptLevelKeyword
                                                           //(_LL, _HE)
                                    ShareType,             //Nothing (Exclusive)
                                                           //ShareTypeKeyword (_SHR)
                                    ResSourceIndex,        //Nothing | ByteConstExpr
                                    ResSource,             //Nothing | StringData
                                    ResourceTag            //Nothing | NameString
                                ) {DWordList}              //list of interrupts (_INT)
```

```
IOTerm                      := IO(
                                   IODecode,             //IODecodeKeyword (_DEC)
                                   MinAddress,           //WordConstExpr (_MIN)
                                   MaxAddress,           //WordConstExpr (_MAX)
                                   Alignment,            //ByteConstExpr (_ALN)
                                   RangeLen,             //ByteConstExpr (_LEN)
                                   ResourceTag           //Nothing | NameString
                               )

IRQNoFlagsTerm              := IRQNoFlags(
                                   ResourceTag           //Nothing | NameString
                               ) {ByteList}              //list of interrupts (0-15)

IRQTerm                     := IRQ(
                                   InterruptType,        //InterruptTypeKeyword
                                                         //(_LL, _HE)
                                   InterruptLevel,       //InterruptLevelKeyword
                                                         //(_LL, _HE)
                                   ShareType,            //Nothing (Exclusive)
                                                         //ShareTypeKeyword (_SHR)
                                   ResourceTag           //Nothing | NameString
                               ) {ByteList}              //list of interrupts (0-15)

Memory24Term                := Memory24(
                                   ReadWriteType,        //ReadWriteKeyword (_RW)
                                   MinAddress[23:8],     //WordConstExpr (_MIN)
                                   MaxAddress[23:8],     //WordConstExpr (_MAX)
                                   Alignment,            //WordConstExpr (_ALN)
                                   RangeLen,             //WordConstExpr (_LEN)
                                   ResourceTag           //Nothing | NameString
                               )

Memory32FixedTerm           := Memory32Fixed(
                                   ReadWriteType,        //ReadWriteKeyword (_RW)
                                   AddressBase,          //DWordConstExpr (_BAS)
                                   RangeLen,             //DWordConstExpr (_LEN)
                                   ResourceTag           //Nothing | NameString
                               )

Memory32Term                := Memory32(
                                   ReadWriteType,        //ReadWriteKeyword (_RW)
                                   MinAddress,           //DWordConstExpr (_MIN)
                                   MaxAddress,           //DWordConstExpr (_MAX)
                                   Alignment,            //DWordConstExpr (_ALN)
                                   RangeLen,             //DWordConstExpr (_LEN)
                                   ResourceTag           //Nothing | NameString
                               )

QWordIOTerm                 := QWordIO(
                                   ResourceType,         //Nothing (ResourceConsumer)|
                                                         //ResourceTypeKeyword
                                   MinType,              //Nothing (MinNotFixed) |
                                                         //MinKeyword (_MIF)
                                   MaxType,              //Nothing (MaxNotFixed) |
                                                         //MaxKeyword (_MAF)
                                   Decode,               //Nothing (PosDecode) |
                                                         //DecodeKeyword (_DEC)
                                   RangeType,            //Nothing (EntireRange) |
                                                         //RangeTypeKeyword (_RNG)
                                   AddressGranularity,   //QWordConstExpr (_GRA)
                                   MinAddress,           //QWordConstExpr (_MIN)
                                   MaxAddress,           //QWordConstExpr (_MAX)
                                   Translation,          //QWordConstExpr (_TRA)
                                   AddressLen,           //QWordConstExpr (_LEN)
                                   ResSourceIndex,       //Nothing | ByteConstExpr
                                   ResSource,            //Nothing | StringData
                                   ResourceTag           //Nothing | NameString
                                   Type                  //Nothing | TypeKeyword
                                   TranslationType       //Nothing |
                                                         TranslationKeyword)
```

```
QWordMemoryTerm             := QWordMemory(
                                 ResourceType,          //Nothing (ResourceConsumer)|
                                                        //ResourceTypeKeyword
                                 Decode,                //Nothing (PosDecode) |
                                                        //DecodeKeyword (_DEC)
                                 MinType,               //Nothing (MinNotFixed) |
                                                        //MinKeyword (_MIF)
                                 MaxType,               //Nothing (MaxNotFixed) |
                                                        //MaxKeyword (_MAF)
                                 MemType,               //Nothing (NonCacheable) |
                                                        //MemTypeKeyword (_MEM)
                                 ReadWriteType,         //ReadWriteKeyword (_RW)
                                 AddressGranularity,    //QWordConstExpr (_GRA)
                                 MinAddress,            //QWordConstExpr (_MIN)
                                 MaxAddress,            //QWordConstExpr (_MAX)
                                 Translation,           //QWordConstExpr (_TRA)
                                 AddressLen,            //QWordConstExpr (_LEN)
                                 ResSourceIndex,        //Nothing | ByteConstExpr
                                 ResSource,             //Nothing | StringData
                                 ResourceTag            //Nothing | NameString
                                 AddressRange           //Nothing | AddressKeyword
                             (_MTP)
                                 Type                   //Nothing | TypeKeyword
                                                        (_TTP)

                             )

RegisterTerm                := Register(
                                 AddressSpaceID,        //AddressSpaceKeyword  (_ASI)
                                 RegisterBitWidth,      //ByteConstExpr (_RBW)
                                 RegisterOffset,        //ByteConstExpr (_RBO)
                                 RegisterAddress,       //QWordConstExpr (_ADR)
                                 )

StartDependentFnTerm        := StartDependentFn(
                                 CompatPriority,        //ByteConstExpr (0-2)
                                 PerfRobustPriority     //ByteConstExpr (0-2)
                             ) {ResourceMacroList}

StartDependentFnNoPriTerm   :=StartDependentFnNoPri() {ResourceMacroList}

VendorLongTerm              := VendorLong(
                                 ResourceTag            //Nothing | NameString
                             ) {ByteList}

VendorShortTerm             := VendorShort(
                                 ResourceTag            //Nothing | NameString
                             ) {ByteList}               //up to 7 bytes

WordBusNumberTerm           := WordBusNumber(
                                 ResourceType,          //Nothing (ResourceConsumer)|
                                                        //ResourceTypeKeyword
                                 MinType,               //Nothing (MinNotFixed) |
                                                        //MinKeyword (_MIF)
                                 MaxType,               //Nothing (MaxNotFixed) |
                                                        //MaxKeyword (_MAF)
                                 Decode,                //Nothing (PosDecode) |
                                                        //DecodeKeyword (_DEC)
                                 AddressGranularity,    //WordConstExpr (_GRA)
                                 MinAddress,            //WordConstExpr (_MIN)
                                 MaxAddress,            //WordConstExpr (_MAX)
                                 Translation,           //WordConstExpr (_TRA)
                                 AddressLen,            //WordConstExpr (_LEN)
                                 ResSourceIndex,        //Nothing | ByteConstExpr
                                 ResSource,             //Nothing | StringData
                                 ResourceTag            //Nothing | NameString
                             )
```

Formatted

Deleted: Translation

Deleted: Translation

Formatted

```
WordIOTerm                  := WordIO(
                                 ResourceType,          //Nothing (ResourceConsumer)|
                                                        //ResourceTypeKeyword
                                 MinType,               //Nothing (MinNotFixed) |
                                                        //MinKeyword (_MIF)
                                 MaxType,               //Nothing (MaxNotFixed) |
                                                        //MaxKeyword (_MAF)
                                 Decode,                //Nothing (PosDecode) |
                                                        //DecodeKeyword (_DEC)
                                 RangeType,             //Nothing (EntireRange) |
                                                        //RangeTypeKeyword (_RNG)
                                 AddressGranularity,    //WordConstExpr _GRA)
                                 MinAddress,            //WordConstExpr (_MIN)
                                 MaxAddress,            //WordConstExpr (_MAX)
                                 Translation,           //WordConstExpr (_TRA)
                                 AddressLen,            //WordConstExpr (_LEN)
                                 ResSourceIndex,        //Nothing | ByteConstExpr
                                 ResSource,             //Nothing | StringData
                                 ResourceTag            //Nothing | NameString
                                 Type                   //Nothing | TypeKeyword
                                 TranslationType        //Nothing |
                                                        TranslationKeyword

                                )
```

Deleted: <#>ASL Data Types¶

ASL provides a wide variety of data types and operators that work on these data types. It also provides both explicit and implicit conversion between these data types when used with ASL operators. To avoid these implicit conversions, the Copy operator may be used. ¶

In ASL, conversion can take place in two places during an operation. First, when the source operands are converted to the operand type expected by the operator and second, when the result of the operator are stored into the destination.¶

For example:¶

```
Store("XYZ",Local1)¶
Store(10,Local0) ¶
Add(Local0,"5",Local1)¶
```

In this case, the Add operator converts the first two operands (Local0 and "5") to Integers. Then the result of the operation (15) is converted into a String, since this is the type of Local1.¶

In some cases, the operator may take more than one type of operand (such as Integer and String). In this case, depending on the type of the operand, the highest priority conversion is applied. Table 16-4, column describes the source operand conversions available. For example:¶

```
Store(Buffer(1){},Local0)¶
Name(ABCD, Buffer(10)
{1,2,3,4,5,6,7,8,9,0})¶
CreateDWordField(ABCD,2,XYZ)¶
Name(MNOP,"1234")¶
Concatenate(XYZ,MNOP,Local0)¶
```

Concatenate can take an Integer, Buffer or String for its first two parameters and the type of the first parameter determines how the second parameter will be converted. In this example, the first parameter is of type Buffer Field (from the CreateDWordField operator). What should it be converted to: Integer, Buffer or String? According to Table 16-4, the highest priority conversion is to Integer. So XYZ (0x05040302) and MNOP (0x31,0x32,0x33,0x34) will be converted to Integers, joined together and the resulting type will be Buffer (0x02,0x03,0x04,0x05,0x31,0x32,0x33,0x34).¶

The following table describes the default source and destination conversions. If a particular conversion is not described, then it will generate a fatal error at run-time.¶

Table 16-4 ASL Data Types¶

Data Type … [1]

16.2.2 ASL Data Types

ASL provides a wide variety of data types and operators that manipulate data. It also provides mechanisms for both explicit and implicit conversion between the data types when used with ASL operators.

The table below describes each of the available data types.

Table 16-4 Summary of ASL Data Types

ASL Data Type	Description
[Uninitialized]	No assigned type or value. This is the type of all control method LocalX variables and unused ArgX variables at the beginning of method execution, as well as all uninitialized Package elements. Uninitialized objects must be initialized (via Store or CopyObject) before they may be used as source operands in ASL expressions.
Buffer	An array of bytes. Uninitialized elements are zero by default.
Buffer Field	Portion of a buffer created using CreateBitField, CreateByteField, CreateWordField, CreateQWordField, CreateField, or returned by the Index operator.
DDB Handle	Definition block handle returned by the Load operator
Debug Object	Debug output object. Formats an object and prints it to the system debug port. Has no effect if debugging is not active.
Device	Device or bus object
Event	Event synchronization object
Field Unit (within an Operation Region)	Portion of an address space, bit-aligned and of one-bit granularity. Created using Field, BankField, or IndexField.
Integer	An *n*-bit little-endian unsigned integer. In ACPI 1.0 this was at least 32-bits. In ACPI 2.0 this is at least 64.bits.
Integer Constant	Created by the ASL terms “Zero”, “One”, “Ones”, and “Revision”.
Method	Control Method (Executable AML function)
Mutex	Mutex synchronization object
Object Reference	Reference to an object created using the RefOf operator
Operation Region	Operation Region (A region within an Address Space)
Package	Collection of ASL objects with a fixed number of elements (up to 255).
Power Resource	Power Resource description object
Processor	Processor description object
String	Null-terminated ASCII string with up to 200 characters.
Thermal Zone	Thermal Zone description object

Compatibility Note: The ability to store and manipulate object references is new in ACPI 2.0. In ACPI 1.0 references could not be stored in variables, passed as parameters or returned from functions.

16.2.2.1 Data Type Conversion Overview

ASL provides two mechanisms to convert objects from one data type to another data type at run-time (during execution of the AML interpreter). The first mechanism, ***Explicit Data Type Conversion***, allows the use of explicit ASL operators to convert an object to a different data type. The second mechanism, ***Implicit Data Type Conversion***, is invoked by the AML interpreter when it is necessary to convert a data object to an expected data type before it is used or stored.

Both of these mechanisms are described in detail in the sections that follow.

16.2.2.2 Explicit Data Type Conversions

The following ASL operators are provided to *explicitly* convert an object from one data type to another:

- FromBCD — Convert an Integer to a BCD Integer
- ToBCD — Convert a BCD Integer to a standard binary Integer.
- ToBuffer — Convert an Integer, String, or Buffer to an object of type Buffer
- ToDecimalString — Convert an Integer, String, or Buffer to an object of type String. The string contains the ASCII representation of the decimal value of the source operand.
- ToHexString — Convert an Integer, String, or Buffer to an object of type String. The string contains the ASCII representation of the hexadecimal value of the source operand.
- ToInteger — Convert an Integer, String, or Buffer to an object of type Integer.
- ToString — Convert a Buffer to an object of type String.

The following ASL operators are provided to copy and transfer objects:

- CopyObject — Explicitly store a copy of the operand object to the target name. No implicit type conversion is performed. (This operator is used to avoid the implicit conversion inherent in the ASL Store operator.)
- Store — Store a copy of the operand object to the target name. Implicit conversion is performed if the target name is of a fixed data type (see below). However, Stores to method locals and arguments do not perform implicit conversion and are therefore the same as using CopyObject.

16.2.2.3 Implicit Data Type Conversions

Automatic or *Implicit* type conversions can take place at two different times during the execution of an ASL operator. First, it may be necessary to convert one or more of the source operands to the data type(s) expected by the ASL operator. Second, the result of the operation may require conversion before it is stored into the destination. (Many of the ASL operators can store their result optionally into an object specified by the last parameter. In these operators, if the destination is specified, the action is exactly as if a Store operator had been used to place the result in the destination.)

Such data conversions are performed by an AML interpreter during execution of AML code and are known collectively as ***Implicit Operand Conversions***. As described briefly above, there are two different types of implicit operand conversion:

1. Conversion of a source operand from a mismatched data type to the correct data type required by an ASL operator, called ***Implicit Source Conversion***. This conversion occurs when a source operand must be converted to the operand type expected by the operator. Any or all of the source operands may be converted in this manner before the execution of the ASL operator can proceed.
2. Conversion of the result of an operation to the existing type of a target operand before it is stored into the target operand, called ***Implicit Result Conversion***. This conversion occurs when the target is a fixed type such as a named object or a field. When storing to a method Local or Arg, no conversion is required because these data types are of variable type (the store simply overwrites any existing object and the existing type).

Formatted: Bullets and Numbering

16.2.2.3.1 Implicit Source Operand Conversion

During the execution of an ASL operator, each source operand is processed by the AML interpreter as follows:

- If the operand is of the type expected by the operator, no conversion is necessary.
- If the operand type is incorrect, attempt to convert it to the proper type.
- For the Concatenate operator, the data type of the first operand dictates both the required type of the second operand and the type of the result object. (The second operator is converted, if necessary, to match the type of the first operand.)
- If conversion is impossible, abort the running control method and issue a fatal error.

Formatted: Bullets and Numbering

An implicit source conversion will be attempted anytime a source operand contains a data type that is different that the type expected by the operator. For example:

```
Store (“5678”, Local1)
Add (0x1234, Local1, BUF1)
```

In the Add statement above, *Local1* contains a String object and must undergo conversion to an Integer object before the Add operation can proceed.

In some cases, the operator may take more than one type of operand (such as Integer and String). In this case, depending on the type of the operand, the highest priority conversion is applied. Table 16-4 describes the source operand conversions available. For example:

```
Store (Buffer(1){}, Local0)
Name (ABCD, Buffer(10){1,2,3,4,5,6,7,8,9,0})
CreateDWordField (ABCD, 2, XYZ)
Name (MNOP, ”1234”)
Concatenate (XYZ, MNOP, Local0)
```

The Concatenate operator can take an Integer, Buffer or String for its first two parameters and the type of the first parameter determines how the second parameter will be converted. In this example, the first parameter is of type Buffer Field (from the CreateDWordField operator). What should it be converted to: Integer, Buffer or String? According to Table 16-4, the highest priority conversion is to Integer. Therefore, both of the following objects will be converted to Integers:

```
XYZ (0x05040302)
MNOP (0x31, 0x32, 0x33, 0x34)
```

And will then joined together and the resulting type and value will be:

```
Buffer (0x02, 0x03 ,0x04, 0x05, 0x31, 0x32, 0x33, 0x34).
```

16.2.2.3.2 Implicit Result Object Conversion

Formatted: Bullets and Numbering

For all ASL operators that generate and store a result value (including the Store operator), the result object is processed and stored by the AML interpreter as follows:

- If the ASL operator is one of the *explicit* conversion operators (ToString, ToInteger, etc., and the CopyObject operator), no conversion is performed. (In other words, the result object is stored directly to the target and completely overwrites any existing object already stored at the target.)
- If the target is a method local or argument (LocalX or ArgX), no conversion is performed and the result is stored directly to the target.
- If the target is a fixed type such as a named object or field object, an attempt is made to convert the source to the existing target type before storing.
- If conversion is impossible, abort the running control method and issue a fatal error.

Formatted: Bullets and Numbering

An implicit result conversion can occur anytime the result of an operator is stored into an object that is of a fixed type. For example:

```
Name (BUF1, Buffer(10))
Add (0x1234, 0x789A, BUF1)
```

Since BUF1 is a named object of fixed type *Buffer*, the Integer result of the Add operation must be converted to a Buffer before it is stored into BUF1.

Formatted: Bullets and Numbering

16.2.2.4 Data Types and Type Conversions

The following table lists the available ASL data types and the available data type conversions (if any) for each. The entry for each data type is fully cross-referenced, showing both the types to which the object may be converted as well as all other types that may be converted to the data type.

The allowable conversions apply to both explicit and implicit conversions.

Table 16-4a Data Types and Type Conversions

ASL Data Type	Can be implicitly or explicitly converted to these Data Types: (In priority order)	Can be implicitly or explicitly converted from these Data Types:
[Uninitialized]	None. Causes a fatal error when used as a source operand in any ASL statement.	Integer, String, Buffer, Package, DDB Handle, Object Reference
Buffer	Integer, String, Debug Object	Integer, String
Buffer Field	Integer, Buffer, String, Debug Object	Integer, Buffer, String
DDB Handle	Integer, Debug Object	Integer
Debug Object	None. Causes a fatal error when used as a source operand in any ASL statement.	Integer, String, Buffer, Package, Field Unit, Buffer Field, DDB Handle
Device	None	None
Event	None	None
Field Unit (within an Operation Region)	Integer, Buffer, String, Debug Object	Integer, Buffer, String
Integer	Buffer, Buffer Field, DDB Handle, Field Unit, String, Debug Object	Buffer, String
Integer Constant	Integer, Debug Object	None. Also, storing any object to a constant is a no-op, not an error.
Method	None	None
Mutex	None	None
Object Reference	None	None
Operation Region	None	None
Package	Debug Object	None
String	Integer, Buffer, Debug Object	Integer, Buffer
Power Resource	None	None
Processor	None	None
Thermal Zone	None	None

16.2.2.5 Data Type Conversion Rules

Formatted: Bullets and Numbering

The following table presents the detailed data conversion rules for each of the allowable data type conversions. These conversion rules are implemented by the AML Interpreter and apply to all conversion types — explicit conversions, implicit source conversions, and implicit result conversions.

Table 16-4b Object Conversion Rules

To convert from an object of this Data Type	To an object of this Data Type	This action is performed by the AML Interpreter:
Buffer	Buffer Field	The contents of the buffer are copied to the Buffer Field. If the buffer is smaller than the size of the buffer field, it is zero extended. If the buffer is larger than the size of the buffer field, the upper bits are truncated. Compatibility Note: This conversion is new in ACPI 2.0. The behavior in ACPI 1.0 was undefined.
	Debug Object	Each buffer byte is displayed as hexadecimal integer, delimited by spaces and/or commas.
	Field Unit	The entire contents of the buffer are copied to the Field Unit. If the buffer is larger (in bits) than the size of the Field Unit, it is broken into pieces and completely written to the Field Unit, lower chunks first. If the integer (or the last piece of the integer, if broken up) is smaller or equal in size to the Field Unit, then it is zero extended before being written.
	Integer	The contents of the buffer are copied to the Integer, starting with the least-significant bit and continuing until the buffer has been completely copied — up to the maximum number of bits in an Integer (64 in ACPI 2.0).
	String	The entire contents of the buffer are converted to a string of two-character hexadecimal numbers, each separated by a space. A fatal error is generated if greater than two hundred ASCII characters are created.
Buffer Field	[See Rule]	If the Buffer Field is smaller than or equal to the size of an Integer (in bits), it will be treated as an Integer. Otherwise, it will be treated as a Buffer. (See the conversion rules for the Integer and Buffer data types.)
	Debug Object	Each byte is displayed as hexadecimal integer , delimited by spaces and/or commas
DDB Handle	[See Rule]	The object is treated as an Integer (See conversion rules for the Integer data type.)
Field Unit	[See Rule]	If the Field Unit is smaller than or equal to the size of an Integer (in bits), it will be treated as an Integer. Otherwise, it will be treated as a Buffer. (See the conversion rules for the Integer and Buffer data types.)
	Debug Object	Each byte is displayed as hexadecimal integer , delimited by spaces and/or commas

To convert from an object of this Data Type	To an object of this Data Type	This action is performed by the AML Interpreter:
Integer	Buffer	The Integer overwrites the entire Buffer object. If the integer requires more bits than the size of the Buffer, then the integer is truncated before being copied to the Buffer. If the integer contains fewer bits than the size of the buffer, the Integer is zero-extended to fill the entire buffer
	Buffer Field	The Integer overwrites the entire Buffer Field. If the integer is smaller than the size of the buffer field, it is zero-extended. If the integer is larger than the size of the buffer field, the upper bits are truncated. Compatibility Note: This conversion is new in ACPI 2.0. The behavior in ACPI 1.0 was undefined.
	Debug Object	Displayed as a hexadecimal integer.
	Field Unit	The Integer overwrites the entire Field Unit. If the integer is smaller than the size of the buffer field, it is zero-extended. If the integer is larger than the size of the buffer field, the upper bits are truncated.
	String	Creates an ASCII hexadecimal string.
Package	Package	All existing contents (if any) of the target package are deleted, and the contents of the source package are copied into the target package. (In other words, overwrites the same as any other object.)
	Debug Object	Each element of the package is displayed based on its type.
String	Buffer	The string is treated as a Buffer, with each ASCII character copied to one Buffer byte. If the string is longer than the buffer, it is truncated. If the string is shorter than the buffer, the buffer size is reduced
	Buffer Field	The string is treated as a buffer. If this buffer is smaller than the size of the buffer field, it is zero extended. If the buffer is larger than the size of the buffer field, the upper bits are truncated. Compatibility Note: This conversion is new in ACPI 2.0. The behavior in ACPI 1.0 was undefined.
	Debug Object	Each byte displayed as an ASCII character
	Field Unit	Each character of the string is written, starting with the first, to the Field Unit. If the Field Unit is less than eight bits, then the upper bits of each character are lost. If the Field Unit is greater than eight bits, then the additional bits are zeroed.
	Integer	The ASCII string is interpreted as a hexadecimal constant. Starts with the first hexadecimal ASCII character ('0'-'9', 'A'-'F', 'a', 'f') and ends with the first non-hexadecimal character.

16.2.2.6 Rules for Storing and Copying Objects

The table below lists the actions performed when storing objects to different types of named targets. ASL provides the following types of "store" operations:

- The Store operator is used to explicitly store an object to a location, with implicit conversion support of the source object.

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

- Many of the ASL operators can store their result optionally into an object specified by the last parameter. In these operators, if the destination is specified, the action is exactly as if a Store operator had been used to place the result in the destination.
- The CopyObject operator is used to explicitly store a copy of an object to a location, with no implicit conversion support.

Table 16-4c Object Storing and Copying Rules

When Storing an object of any data type to this type of Target location	This action is performed by the Store operator or any ASL operator with a Target operand:	This action is performed by the CopyObject operator:
Method **ArgX** variable	The object is copied to the destination with no conversion applied, with one exception. If the ArgX contains an Object Reference, an automatic de-reference occurs and the object is copied to the target of the Object Reference instead of overwriting the contents of ArgX	
Method **LocalX** variable	The object is copied to the destination with no conversion applied. Even if LocalX contains an Object Reference, it is overwritten.	
Field Unit or Buffer Field	The object is copied to the destination after implicit result conversion is applied	Fields permanently retain their type and cannot be changed. Therefore, CopyObject can only be used to copy an object of type Integer or Buffer to fields.
Named data object	The object is copied to the destination after implicit result conversion is applied to match the existing type of the named location	The object and type are copied to the named location.

16.2.3.3.1.14 Mutex (Declare Synchronization/Mutex Object)

```
MutexTerm                   := Mutex(
                                   MutexName,          //NameString
                                   SyncLevel           //ByteConstExpr
                               )
```

Creates a data mutex synchronization object named *MutexName*, with level from 0 to 15 specified by *SyncLevel*.

Deleted: A *SyncLevel* of *n* allows *n*+1 mutex owners

A synchronization object provides a control method with a mechanism for waiting for certain events. To prevent deadlocks, wherever more than one synchronization object must be owned, the synchronization objects must always be released in the order opposite the order in which they were acquired. The *SyncLevel* parameter declares the logical nesting level of the synchronization object. All **Acquire** terms must refer to a synchronization object with an equal or greater *SyncLevel* to current level, and all **Release** terms must refer to a synchronization object with equal or lower *SyncLevel* to the current level.

Mutex synchronization provides the means for mutually exclusive ownership. Ownership is acquired using an **Acquire** term and is released using a **Release** term. Ownership of a Mutex must be relinquished before completion of any invocation. For example, the top-level control method cannot exit while still holding ownership of a Mutex. Acquiring ownership of a Mutex can be nested.

Deleted: The *SyncLevel* check is not performed on a Mutex when the ownership count is nesting.

The SyncLevel of a thread before acquiring any mutexes is zero. The SyncLevel of the Global Lock (_GL) is zero.

Deleted: A method marked serialized has an inherent mutex of SyncLevel 0 unless SyncLevel is explicitly specified.

16.2.3.4.1.7 Load (Load Definition Block)

```
LoadTerm                    := Load(
                                   Object,              //NameString
                                   DDBHandle            //SuperName
                               )
```

Performs a run-time load of a Definition Block. The *Object* parameter can either refer to an operation region field or an operation region directly. If the object is an operation region, the operation region must be in SystemMemory space. The Definition Block should contain a DESCRIPTION_HEADER of type SSDT. The Definition Block must be totally contained within the supplied operation region or operation region field. OSPM reads this table into memory, the checksum is verified, and then it is loaded into the ACPI namespace. The *DDBHandle* parameter is the handle to the Differentiating Definition Block that can be used to unload the Definition Block at a future time.

The OS can also check the OEM Table ID and Revision ID against a database for a newer revision Definition Block of the same OEM Table ID and load it instead.

The default namespace location to load the Definition Block is relative to the current namespace. The new Definition Block can override this by specifying absolute names or by adjusting the namespace location using the **Scope** operator.

Loading a Definition Block is a synchronous operation. Upon completion of the operation, the Definition Block has been loaded. The control methods defined in the Definition Block are not executed during load time.

16.2.3.4.1.16 Switch – Select Code To Execute Based On Expression

```
SwitchTerm                  := Switch(
                                   Predicate            //ComputationalData
                               ) {CaseTermList}
DefaultTermList             := Nothing | CaseTerm | CaseTerm DefaultTermList
CaseTermList                := Nothing | CaseTerm CaseTermList | DefaultTerm
                               DefaultTermList
CaseTerm                    := Case(DataObject) {TermList}
DefaultTerm                 := Default {TermList}
```

The **Switch**, **Case** and **Default** statements help simplify the creation of conditional and branching code. The **Switch** statement transfers control to a statement within its body.

If the **Case** *value* is an Integer, Buffer or String, then control passes to the statement that matches the value of **Switch(***Predicate***)**. If the **Case** value is a Package, then control passes if any member of the package matches the **Switch**(*Predicate*). The **Switch** *CaseTermList* can include any number of **Case** instances, but no two **Case** *value*s (or members of a *value*, if *value* is a Package) within the same **Switch** statement can contain the same value.

Execution of the statement body begins at the selected statement's TermList and proceeds until the end of the body or until an **ExitSwitch** (or other valid **Exit**x) statement transfers control out of the body.

Use of the **Switch** statement usually looks something like this:

Deleted: Differentiated

Deleted: or PSDT

Deleted: T

Deleted: is read

```
Switch ( expression )
{
    Case ( value ) {
        Statements executed if Lequal(expression, value)

    }
    Case ( Package() {value,value,value}) {
        Statements executed if Lequal(expression, any value in package)
    Default {
        statements executed if expression does not equal
        any case constant-expression
    }
}
```

The **Default** statement is executed if no **Case** *value* matches the value of **switch (** *expression* **)**. If the **Default** statement is omitted, and no **Case** match is found, none of the statements in the **Switch** body are executed. There can be at most one **Default** statement. The **Default** statement need not come at the end; it can appear anywhere in the body of the **Switch** statement.

A **Case** or **Default** term can only appear inside a **Switch** statement. Switch statements can be nested.

Compatibility Note: The **Switch, Case,** and **Default** terms are new to ACPI 2.0. However, their implementation is backward compatible with ACPI 1.0 AML interpreters.

Compiler Note: The following example demonstrates how the Switch statement should be translated into ACPI 1.0-compatible AML:

```
Switch (Add(ABCD(),1)
{
    Case(1) {
    …statements1…
    }
    Case(Package() {4,5,6}) {
    …statements2…
    }
    Default {
    …statements3…
    }
}
```

is translated as:

```
While(One)
{
    Name(_T_I,0)                    // Create Integer temporary variable for result
    Store(Add(ABCD(),1),_T_I)
    If (LEqual(_T_I,1)) {
        …statements1…
    }
    Else {
    If (LNotEqual(Match(Package() {4,5,6},MEQ,_T_I,MTR,0,0),Ones)) {
        …statements2…
    }
    Else {
        …statements3…
    }
    }
    Break
}
```

Deleted: Zero

Note: If the compiler is unable to determine the type of the expression, then it should generate a warning and assume integer type. The warning should indicate that the ASL should use one of the type conversion operators (Int, Buff, DecStr or HexStr). For example:

```
Switch(ABCD())        // Can't determine the type because methods can return anything.
{
    …case statements…
}
```

will generate a warning and the following code:

```
Name(_T_I,0)
Store(ABCD(),_T_I)
```

To remove the warning, the code should be:

```
Switch(Int(ABCD()))
{
    …case statements…
}
```

Deleted: Differentiated

16.2.3.4.1.17 Unload (Unload Definition Block)

```
UnloadTerm                  := Unload(
                                   DDBHandle            //TermArg=>DDBHandle
                               )
```

Performs a run-time unload of a Definition Block that was loaded using a **Load** term. Loading or unloading a Definition Block is a synchronous operation, and no control method execution occurs during the function. On completion of the **Unload** operation, the Definition Block has been unloaded (all the namespace objects created as a result of the corresponding **Load** operation will be removed from the namespace).

16.2.3.4.2Type 2 Opcodes

```
Type2Opcode                 := AcquireTerm | AddTerm | AndTerm | ConcatTerm |
                               ConcateResTerm | CondRefOfTerm | DecTerm | DerefOfTerm |
                               DivideTerm | FindSetLeftBitTerm | FindSetRightBitTerm |
                               FromBCDTerm | IncTerm | IndexTerm | LAndTerm |
                               LEqualTerm | LGreaterTerm | LGreaterEqualTerm |
                               LLessTerm | LLessEqualTerm | LNotTerm | LNotEqualTerm |
                               LoadTableTerm | LOrTerm | MatchTerm | MidTerm |
                               ModTerm | MultiplyTerm | NAndTerm | NOrTerm | NotTerm |
                               ObjectTypeTerm | OrTerm | RefOfTerm | ShiftLeftTerm |
                               ShiftRightTerm | SizeOfTerm | StoreTerm | SubtractTerm |
                               ToBCDTerm | ToBufferTerm | ToDecimalStringTerm |
                               ToHexStringTerm | ToIntegerTerm | ToStringTerm |
                               WaitTerm | XorTerm | UserTerm
```

The ASL terms for Type 2 Opcodes are listed in the following table.

Table 16-9 Type 2 Opcodes

ASL Statement	Description
Acquire	Acquire a mutex
Add	Add two values
And	Bitwise And
Concatenate	Concatenate two strings, integers or buffers
ConcatenateResTemplate	Concatenate two resource templates
CondRefOf	Conditional reference to an object
Decrement	Decrement a value
DerefOf	Dereference an object reference
Divide	Divide
FindSetLeftBit	Index of first least significant bit set
FindSetRightBit	Index of first most significant bit set
FromBCD	Convert from BCD to numeric
Increment	Increment a value
Index	Reference the nth element/byte/character of a package, buffer or string
LAnd	Logical And

Table 16-9 Type 2 Opcodes *(continued)*

ASL Statement	Description
LEqual	Logical Equal
LGreater	Logical Greater
LGreaterEqual	Logical Not less
LLess	Logical Less
LLessEqual	Logical Not greater
LNot	Logical Not
LNotEqual	Logical Not equal
LoadTable	Load Table from RSDT/XSDT
LOr	Logical Or
Match	Search for match in package array
Mid	Returns a portion of buffer or string
Mod	Modulo
Multiply	Multiply
NAnd	Bitwise Nand
NOr	Bitwise Nor
Not	Bitwise Not
ObjectType	Type of object
Or	Bitwise Or
RefOf	Reference to an object
ShiftLeft	Shift value left
ShiftRight	Shift value right
SizeOf	Get the size of a buffer, string, or package
Store	Store value
Subtract	Subtract values
ToBCD	Convert numeric to BCD
ToBuffer	Convert data type to buffer
ToDecimalString	Convert data type to decimal string
ToHexString	Convert data type to hexadecimal string
ToInteger	Convert data type to integer
ToString	Copy ASCII string from buffer
Wait	Wait
Xor	Bitwise Xor

16.2.3.4.2.4 ToBuffer (Convert Data Type to Buffer)

```
ToBufferTerm            := ToBuffer(
                               Data,                //TermArg=>ComputationalData
                               Result               //Target
                           ) => Buffer
```

Formatted

Data must be evaluated to integer, string, or buffer. *Data* is then converted to buffer type and the result is optionally stored into *Result*. If *Data* was an integer, it is converted into 4 bytes of buffer, taking the least significant type of integer as the first byte of buffer. If *Data* is a buffer, no conversion is performed.

16.2.3.4.2.8 CopyObject (Copy Object)

Deleted: –

```
CopyObjectTerm          := CopyObject(
                               Source,              //SuperName=>DataRefObject
                               Destination          //NameString | LocalTerm |
                                                    // ArgTerm
                           ) => DataRefObject
```

Converts the contents of the *Source* to a DataRefObject using the conversion rules in 16.2.2 and then copies the results without conversion to the object referred to by *Destination*. If *Destination* is already an initialized object of type DataRefObject, the original contents of *Destination* are discarded and replaced with *Source*. Otherwise, a fatal error is generated.

Compatibility Note: The CopyObject operator is new in ACPI 2.0.

16.2.3.4.2.10 ToDecimalString (Convert Data Type to Decimal String)

```
ToDecimalStringTerm     := ToDecimalString(
                               Data,                //TermArg=>ComputationalData
                               Result               //Target
                           ) => String
```

Formatted

Data must be evaluated to integer, string, or buffer. *Data* is then converted to a decimal string, and the result is optionally stored into *Result*. If *Data* is already a string, no action is performed. If *Data* is a buffer, it is converted to a string of decimal values separated by commas.

16.2.3.4.2.16 ToHexString (Convert Data Type to Hexadecimal String)

```
ToHexStringTerm         := ToHexString(
                               Data,                //TermArg=>ComputationalData
                               Result               //Target
                           ) => String
```

Formatted

Data must be evaluated to integer, string, or buffer. *Data* is then converted to a hexadecimal string, and the result is optionally stored into *Result*. If *Data* is already a string, no action is performed. If *Data* is a buffer, it is converted to a string of hexadecimal values separated by commas.

16.2.3.4.2.19 ToInteger (Convert Data Type to Integer)

```
ToIntegerTerm           := ToInteger(
                               Data,                //TermArg=>ComputationalData
                               Result               //Target
                           ) => Integer
```

Formatted

Data must be evaluated to integer, string, or buffer. *Data* is then converted to integer type and the result is optionally stored into *Result*. If *Data* was a string, it must be either a decimal or hexadecimal numeric string (in other words, prefixed by “0x”) and the value must not exceed the maximum of an integer value. If the value is exceeding the maximum, the result of the conversion is unpredictable. If *Data* was a Buffer, the first 8 bytes of the buffer are converted to an integer, taking the first byte as the least significant byte of the integer.

16.2.3.4.2.26 LNot (Logical Not)

```
LNotTerm                    := LNot(
                                   Source,              //TermArg=>Integer
                               ) => Boolean
```

Source1 is evaluated as an integer. If the value is zero True is returned; otherwise, False is returned.

Deleted: non-

16.2.3.4.2.37 ObjectType (Object Type)

```
ObjectTypeTerm              := ObjectType(
                                   Object               //SuperName
                               ) => Integer
```

The execution result of this operation is an integer that has the numeric value of the object type for *Object*. The object type codes are listed in Table 16-12. Notice that if this operation is performed on an object reference such as one produced by the **Alias, Index,** or **RefOf** statements, the object type of the base object is returned. For typeless objects such as pre-defined scope names (in other words, _SB, _GPE, and so on), the type value 0 (**Uninitialized**) is returned.

Deleted: 4

16.2.3.4.2.44 ToString (Create ASCII String From Buffer)

```
ToStringTerm                := ToString(
                                   Source,              //TermArg=>Buffer
                                   Length,              //Nothing | TermArg=>Integer
                                   Result               //Target
                               ) => String
```

Formatted

Source is evaluated as a buffer. Starting with the first byte, the contents of the buffer are copied into the string until the number of characters specified by *Length* is reached. If *Length* is not specified or is **Ones**, then the contents of the buffer are copied until a null (0) character is found. In any case, a fatal error will be generated if the number of characters copied exceeds 200 (not including the terminating null). The result is copied into the *Result*.

16.2.4.5 ASL Macro for I/O Port Descriptor

The following macro generates a short I/O descriptor:

```
IO(
    Decode16 | Decode10,          // _DEC
    WordConstExpr,                // _MIN, Address minimum
    WordConstExpr,                // _MAX, Address max
    ByteConstExpr,                // _ALN, Base alignment
    ByteConstExpr                 // _LEN, Range length
    NameString | Nothing          // A name to refer back to this resource

)
```

16.2.4.6 ASL Macro for Fixed I/O Port Descriptor

The following macro generates a short Fixed I/O descriptor:

```
FixedIO(
    WordConstExpr,              // _BAS, Address base
    ByteConstExpr               // _LEN, Range length
    NameString | Nothing        // A name to refer back to this resource
    )
```

Deleted: The following macro generates a short Fixed I/O descriptor:¶
¶
FixedIO(¶
WordConstExpr, // _BAS, Address base¶
ByteConstExpr // _LEN, Range length¶
NameString | Nothing // A name to refer back to this resource¶
)

Formatted

16.2.4.7 ASL Macro for Short Vendor-Defined Descriptor

The following macro generates a short Vendor-Defined descriptor:

```
VendorShort(
    NameString | Nothing          // A name to refer back to this resource
    )
    {
    ByteConstExpr [, ByteConstExpr ...]   // List of bytes, up to 7 bytes
        }
```

17.2.1 Name Objects Encoding

```
LeadNameChar                := 'A'-'Z' | '_'
DigitChar                   := '0'-'9'
NameChar                    := DigitChar | LeadNameChar
RootChar                    := '\'
ParentPrefixChar            := '^'

'A'-'Z'                     := 0x41-0x5a
'_'                         := 0x5f
'0'-'9'                     := 0x30-0x39
'\'                         := 0x5c
'^'                         := 0x5e

NameSeg                     := <LeadNameChar NameChar NameChar NameChar>
                               // Notice that NameSegs shorter than 4 characters are
                               // filled with trailing '_'s.
NameString                  := <RootChar NamePath> | <PrefixPath NamePath>
PrefixPath                  := Nothing | <'^' PrefixPath>
NamePath                    := NameSeg | DualNamePath | MultiNamePath | NullName

DualNamePath                := DualNamePrefix NameSeg NameSeg
DualNamePrefix              := 0x2e
MultiNamePath               := MultiNamePrefix SegCount NameSeg(SegCount)
MultiNamePrefix             := 0x2f
SegCount                    := ByteData
                               // SegCount can be from 1 to 255.
                               // MultiNamePrefix(35) => 0x2f 0x23
                               // and following by 35 NameSegs.
                               // So, the total encoding length
                               // will be 1 + 1 + 35*4 = 142.
                               // Notice that:
                               //   DualNamePrefix NameSeg NameSeg
                               // has a smaller encoding than the
                               // equivalent encoding of:
                               //    MultiNamePrefix(2) NameSeg NameSeg
SimpleName                  := NameString | ArgObj | LocalObj
SuperName                   := SimpleName | DebugObj | Type6Opcode
NullName                    := 0x00
Target                      := SuperName | NullName
```

Compaq/Intel/Microsoft/Phoenix/Toshiba

ASL Data Types

ASL provides a wide variety of data types and operators that work on these data types. It also provides both explicit and implicit conversion between these data types when used with ASL operators. To avoid these implicit conversions, the Copy operator may be used.

In ASL, conversion can take place in two places during an operation. First, when the source operands are converted to the operand type expected by the operator and second, when the result of the operator are stored into the destination.

For example:

```
Store("XYZ",Local1)
Store(10,Local0)
Add(Local0,"5",Local1)
```

In this case, the Add operator converts the first two operands (Local0 and "5") to Integers. Then the result of the operation (15) is converted into a String, since this is the type of Local1.

In some cases, the operator may take more than one type of operand (such as Integer and String). In this case, depending on the type of the operand, the highest priority conversion is applied. Table 16-4, column describes the source operand conversions available. For example:

```
Store(Buffer(1){},Local0)
Name(ABCD, Buffer(10) {1,2,3,4,5,6,7,8,9,0})
CreateDWordField(ABCD,2,XYZ)
Name(MNOP,"1234")
Concatenate(XYZ,MNOP,Local0)
```

Concatenate can take an Integer, Buffer or String for its first two parameters and the type of the first parameter determines how the second parameter will be converted. In this example, the first parameter is of type Buffer Field (from the CreateDWordField operator). What should it be converted to: Integer, Buffer or String? According to Table 16-4, the highest priority conversion is to Integer. So XYZ (0x05040302) and MNOP (0x31,0x32,0x33,0x34) will be converted to Integers, joined together and the resulting type will be Buffer (0x02,0x03,0x04,0x05,0x31,0x32,0x33,0x34).

The following table describes the default source and destination conversions. If a particular conversion is not described, then it will generate a fatal error at run-time.

Table 16-4 ASL Data Types

Data Type	Description	Default Source Conversion (Operand)	Destination Conversion From…	What Happens
Uninitialized	No assigned type. The type of all Localx variables at the beginning of a Method's execution and uninitialized Package elements.	Nothing. Generates a fatal error when used as an operand.	Integer	Integer
			String	String
			Buffer	Buffer
			Package	Package
			DDB Handle	DDB Handle
			Object Reference	Object Reference

Table 16-4 ASL Data Types *(continued)*

Data Type	Description	Default Source Conversion (Operand)	Destination Conversion From…	What Happens
Integer	An *n*-bit little-endian, unsigned integer. In ACPI 1.0 this was at least 32-bits. In ACPI 2.0 this is at least 64.bits.	Integer Buffer String DDB Handle	String	The ASCII string is interpreted as a hexadecimal content. Starts with the first hexadecimal ASCII character (‘0’-‘9’, ‘A’-‘F’, ‘a’, ‘f’) and ends with the first non-hexadecimal character.
			Buffer	The contents of the buffer, starting with the least - significant bit and continuing through the minimum of the most significant bit number (in other words, # of bytes * 8) in the buffer or the number of bits in an Integer (at least 64 in ACPI 2.0) are copied as an Integer.
String	Null-terminated ASCII string with up to 200 characters.	String Integer Buffer	Integer	Creates an ASCII hexadecimal string.
			Buffer	Converted to a string of two-character hexadecimal numbers, separated by a space. Fatal error if greater than two hundred ASCII characters generated.
			Package	Generates an error.
Buffer		Buffer Integer String	Integer String	If the integer requires more bits than the size of the Buffer, then the integer is truncated before being copied to the Buffer. If the integer contains fewer significant bits than the size of the buffer, then the Integer is zero-extended to fill the entire buffer.
				The string is treated as a Buffer, with each ASCII character making one Buffer byte.

Table 16-4 ASL Data Types *(continued)*

Data Type	Description	Default Source Conversion (Operand)	Destination Conversion From…	What Happens
Package	Collection of ASL objects with a fixed number of members (up to 255).	Package	Package	All contents of the package are removed. Contents of the source are copied into the package.
Operation Region Field Unit	Bit-aligned variable in an address space.	Integer Buffer String If the Field Unit is larger than the size of an Integer, it will be treated as a Buffer.	Integer	If the integer requires more bits than the size of the Field Unit, it is broken into pieces and written to the Field Unit, least significant bits first. If the integer (or the last piece of the integer, if broken up) is smaller or equal in size to the Field Unit, then it is zero extended before being written.
			String	Each character of the string is written, starting with the first, to the Field Unit. If the Field Unit is less than eight bits, then the upper bits of each character is lost. If the Field Unit is greater than eight bits, then the additional bits are zeroed.
			Buffer	If the buffer requires more bits than the size of the Field Unit, it is broken into pieces and written to the Field Unit, lower chunks first. If the integer (or the last piece of the integer, if broken up) is smaller or equal in size to the Field Unit, then it is zero extended before being written.

Table 16-4 ASL Data Types *(continued)*

Data Type	Description	Default Source Conversion (Operand)	Destination Conversion From…	What Happens
Buffer Field	Piece of a buffer created using CreateBitField, CreateByteField, CreateWordField, CreateQWordField, CreateField or returned by the Index command.	Integer Buffer String	Integer	If the integer is smaller than the size of the buffer field, it is zero-extended. If the integer is larger than the size of the buffer field, the upper bits are truncated. **Compatibility Note:** New in ACPI 2.0. The behavior in ACPI 1.0 was undefined.
			String	The string is treated as a buffer. If this buffer is smaller than the size of the buffer field, it is zero extended. If the buffer is larger than the size of the buffer field, the upper bits are truncated. **Compatibility Note:** New in ACPI 2.0. The behavior in ACPI 1.0 was undefined.
			Buffer	If this buffer is smaller than the size of the buffer field, it is zero extended. If the buffer is larger than the size of the buffer field, the upper bits are truncated. **Compatibility Note:** New in ACPI 2.0. The behavior in ACPI 1.0 was undefined.
DDB Handle	Definition block handle	DDB Handle Integer	Integer	DDB Handle
Device	Device or bus	Nothing	None	Generates an error.
Event	Event	Nothing	None	Generates an error.
Method	Method (function)	Nothing	None	Generates an error.
Mutex	Mutex	Nothing	None	Generates an error.
Operation Region	Operation Region	Nothing	None	Generates an error.
Power Resource	Power Resource	Nothing	None	Generates an error.
Processor	Processor	Nothing	None	Generates an error.
Thermal Zone	Thermal Zone	Nothing	None	Generates an error.

Table 16-4 ASL Data Types *(continued)*

Data Type	Description	Default Source Conversion (Operand)	Destination Conversion From…	What Happens
Debug Object	Debug-output object. Has no effect if debugging is not active.	Nothing. Will generate an error when used as a source.	Integer	Displayed as hexadecimal integer.
			String	Display as ASCII characters.
			Buffer	Each byte displayed as hexadecimal integer , delimited.
			Package	Each element of the package displayed based on its type.
			Operation Region Field Unit	Displayed as hexadecimal integer (if less than or equal to the size of an integer). Otherwise displayed as a buffer.
			Buffer Field	Displayed as a hexadecimal integer.
			DDB Handle	Displayed including information about the DDB.
Zero, One, Ones	Integer constants	Integer	None	Cannot be a destination.
Object Reference	Reference to an object.	Object Reference	Object Reference	Object Reference

Many of the ASL operators can store their result optionally into an object specified by the last parameter. In these operators, if the destination is specified, the action is exactly as if a **Store** operator had been used to place the result in the destination.

Compatibility Note: The ability to store and manipulate object references is new in ACPI 2.0. In ACPI 1.0 references could not be stored in variables, passed as parameters or returned from functions.