ACPI Source Language (ASL) Operator Reference

This document contains the ASL Operator Reference of the ACPI Specification 4.0a.

This document is intended only as quick ASL Operator Reference. The full ACPI Specification can be downloaded from:

http://www.acpi.info

18.3 ASL Operator Summary

	Operator Name	Page	Description
1.	**Acquire**	579	Acquire a mutex
2.	**Add**	579	Integer Add
3.	**Alias**	580	Define a name alias
4.	**And**	580	Integer Bitwise And
5.	**ArgX**	580	Method argument data objects
6.	**BankField**	580	Declare fields in a banked configuration object
7.	**Break**	581	Continue following the innermost enclosing While
8.	**BreakPoint**	582	Used for debugging, stops execution in the debugger
9.	**Buffer**	582	Declare Buffer object
10.	**Case**	582	Expression for conditional execution
11.	**Concatenate**	583	Concatenate two strings, integers or buffers
12.	**ConcatenateResTemplate**	583	Concatenate two resource templates
13.	**CondRefOf**	583	Conditional reference to an object
14.	**Continue**	584	Continue innermost enclosing While loop
15.	**CopyObject**	584	Copy and existing object
16.	**CreateBitField**	584	Declare a bit field object of a buffer object
17.	**CreateByteField**	585	Declare a byte field object of a buffer object
18.	**CreateDWordField**	585	Declare a DWord field object of a buffer object
19.	**CreateField**	585	Declare an arbitrary length bit field of a buffer object
20.	**CreateQWordField**	585	Declare a QWord field object of a buffer object
21.	**CreateWordField**	586	Declare a Word field object of a buffer object
22.	**DataTableRegion**	586	Declare a Data Table Region
23.	**Debug**	587	Debugger output
24.	**Decrement**	587	Decrement an Integer
25.	**Default**	587	Default execution path in Switch()
26.	**DefinitionBlock**	588	Declare a Definition Block
27.	**DerefOf**	588	Dereference an object reference
28.	**Device**	588	Declare a bus/device object
29.	**Divide**	590	Integer Divide
30.	**DMA**	590	DMA Resource Descriptor macro
31.	**DWordIO**	591	DWord IO Resource Descriptor macro
32.	**DWordMemory**	592	DWord Memory Resource Descriptor macro
33.	**DWordSpace**	594	DWord Space Resource Descriptor macro
34.	**EisaId**	595	EISA ID String to Integer conversion macro
35.	**Else**	595	Alternate conditional execution
36.	**ElseIf**	596	Conditional execution
37.	**EndDependentFn**	597	End Dependent Function Resource Descriptor macro
38.	**Event**	597	Declare an event synchronization object
39.	**ExtendedIO**	597	Extended IO Resource Descriptor macro
40.	**ExtendedMemory**	599	Extended Memory Resource Descriptor macro
41.	**ExtendedSpace**	600	Extended Space Resource Descriptor macro
42.	**External**	601	Declare external objects
43.	**Fatal**	602	Fatal error check
44.	**Field**	602	Declare fields of an operation region object
45.	**FindSetLeftBit**	605	Index of first least significant bit set
46.	**FindSetRightBit**	605	Index of first most significant bit set
47.	**FixedIO**	605	Fixed I/O Resource Descriptor macro
48.	**FromBCD**	606	Convert from BCD to numeric
49.	**Function**	606	Declare control method
50.	**If**	607	Conditional execution
51.	**Include**	607	Include another ASL file
52.	**Increment**	608	Increment a Integer
53.	**Index**	608	Indexed Reference to member object
54.	**IndexField**	610	Declare Index/Data Fields
55.	**Interrupt**	611	Interrupt Resource Descriptor macro
56.	**IO**	612	IO Resource Descriptor macro
57.	**IRQ**	613	Interrupt Resource Descriptor macro
58.	**IRQNoFlags**	613	Short Interrupt Resource Descriptor macro
59.	**LAnd**	614	Logical And
60.	**LEqual**	614	Logical Equal
61.	**LGreater**	614	Logical Greater
62.	**LGreaterEqual**	615	Logical Not less
63.	**LLess**	615	Logical Less
64.	**LLessEqual**	615	Logical Not greater
65.	**LNot**	616	Logical Not

18.4 ASL Operator Summary By Type

Operator Name	Page	Description
// ASL compiler controls		
External	601	Declare external objects
Include	607	Include another ASL file
// ACPI table management		
DefinitionBlock	588	Declare a Definition Block
Load	616	Load definition block
LoadTable	617	Load Table from RSDT/XSDT
Unload	648	Unload definition block
// Miscellaneous named object creation		
Alias	580	Define a name alias
Buffer	582	Declare Buffer object
Device	588	Declare a bus/device object
Function	606	Declare a control method
Method	621	Declare a control method
Name	624	Declare a Named object
Package	629	Declare a package object
PowerResource	630	Declare a power resource object
Processor	630	Declare a processor package
Scope	637	Open named scope
ThermalZone	644	Declare a thermal zone package.
// Operation Regions		
BankField	580	Declare fields in a banked configuration object
DataTableRegion	586	Declare a Data Table Region
Field	602	Declare fields of an operation region object
IndexField	610	Declare Index/Data Fields
OperationRegion	627	Declare an operational region
// Buffer Fields		
CreateBitField	584	Declare a bit field object of a buffer object
CreateByteField	585	Declare a byte field object of a buffer object
CreateDWordField	585	Declare a DWord field object of a buffer object
CreateField	585	Declare an arbitrary length bit field of a buffer object
CreateQWordField	585	Declare a QWord field object of a buffer object
CreateWordField	586	Declare a Word field object of a buffer object
// Synchronization		
Acquire	579	Acquire a mutex
Event	597	Declare an event synchronization object
Mutex	624	Declare a mutex synchronization object
Notify	626	Notify Object of event
Release	636	Release a synchronization object
Reset	636	Reset a synchronization object
Signal	639	Signal a synchronization object
Wait	649	Wait on an Event
// Object references		
CondRefOf	583	Conditional reference to an object
DerefOf	588	Dereference an object reference
RefOf	635	Create Reference to an object

```
// Integer arithmetic

Add                         579    Integer Add
And                         580    Integer Bitwise And
Decrement                   587    Decrement an Integer
Divide                      590    Integer Divide
FindSetLeftBit              605    Index of first least significant bit set
FindSetRightBit             605    Index of first most significant bit set
Increment                   608    Increment a Integer
Mod                         623    Integer Modulo
Multiply                    623    Integer Multiply
NAnd                        625    Integer Bitwise Nand
NOr                         625    Integer Bitwise Nor
Not                         625    Integer Bitwise Not
Or                          629    Integer Bitwise Or
ShiftLeft                   638    Integer shift value left
ShiftRight                  639    Integer shift value right
Subtract                    641    Integer Subtract
Xor                         654    Integer Bitwise Xor

// Logical operators

LAnd                        614    Logical And
LEqual                      614    Logical Equal
LGreater                    614    Logical Greater
LGreaterEqual               615    Logical Not less
LLess                       615    Logical Less
LLessEqual                  615    Logical Not greater
LNot                        616    Logical Not
LNotEqual                   616    Logical Not equal
LOr                         618    Logical Or

// Method execution control

Break                       581    Continue following the innermost enclosing While
BreakPoint                  582    Used for debugging, stops execution in the debugger
Case                        582    Expression for conditional execution
Continue                    584    Continue innermost enclosing While loop
Default                     587    Default execution path in Switch()
Else                        595    Alternate conditional execution
ElseIf                      596    Conditional execution
Fatal                       602    Fatal error check
If                          607    Conditional execution
NoOp                        625    No operation
Return                      637    Return from method execution
Sleep                       639    Sleep n milliseconds (yields the processor)
Stall                       640    Delay n microseconds (does not yield the processor)
Switch                      642    Select code to execute based on expression value
While                       649    Conditional loop

// Data type conversion and manipulation

Concatenate                 583    Concatenate two strings, integers or buffers
CopyObject                  584    Copy and existing object
Debug                       587    Debugger output
EisaId                      595    EISA ID String to Integer conversion macro
FromBCD                     606    Convert from BCD to numeric
Index                       608    Indexed Reference to member object
Match                       618    Search for match in package array
Mid                         623    Return a portion of buffer or string
ObjectType                  626    Type of object
SizeOf                      639    Get the size of a buffer, string, or package
Store                       641    Store object
Timer                       644    Get 64-bit timer value
ToBCD                       645    Convert Integer to BCD
ToBuffer                    645    Convert data type to buffer
ToDecimalString             645    Convert data type to decimal string
ToHexString                 646    Convert data type to hexadecimal string
ToInteger                   646    Convert data type to integer
ToString                    646    Copy ASCII string from buffer
ToUUID                      647    Convert Ascii string to UUID
```

```
Unicode                         648     String to Unicode conversion macro

// Resource Descriptor macros

ConcatenateResTemplate          583     Concatenate two resource templates
DMA                             590     DMA Resource Descriptor macro
DWordIO                         591     DWord IO Resource Descriptor macro
DWordMemory                     592     DWord Memory Resource Descriptor macro
DWordSpace                      594     DWord Space Resource Descriptor macro
EndDependentFn                  597     End Dependent Function Resource Descriptor macro
ExtendedIO                      597     Extended I/O Resource Descriptor macro
ExtendedMemory                  599     Extended Memory Resource Descriptor macro
ExtendedSpace                   600     Extended Space Resource Descriptor macro
FixedIO                         605     Fixed I/O Resource Descriptor macro
Interrupt                       611     Interrupt Resource Descriptor macro
IO                              612     IO Resource Descriptor macro
IRQ                             613     Interrupt Resource Descriptor macro
IRQNoFlags                      613     Short Interrupt Resource Descriptor macro
Memory24                        619     Memory Resource Descriptor macro
Memory32                        620     Memory Resource Descriptor macro
Memory32Fixed                   621     Memory Resource Descriptor macro
QWordIO                         631     QWord IO Resource Descriptor macro
QWordMemory                     632     QWord Memory Resource Descriptor macro
QWordSpace                      634     Qword Space Resource Descriptor macro
Register                        635     Generic register Resource Descriptor macro
ResourceTemplate                637     Resource to buffer conversion macro
StartDependentFn                640     Start Dependent Function Resource Descriptor macro
StartDependentFnNoPri           641     Start Dependent Function Resource Descriptor macro
VendorLong                      648     Vendor Resource Descriptor
VendorShort                     649     Vendor Resource Descriptor
WordBusNumber                   650     Word Bus number Resource Descriptor macro
WordIO                          651     Word IO Resource Descriptor macro
WordSpace                       652     Word Space Resource Descriptor macro

// Constants

One                             627     Constant One Object (1)
Ones                            627     Constant Ones Object (-1)
Revision                        637     Constant revision object
Zero                            654     Constant Zero object (0)

// Control method objects

ArgX                            580     Method argument data objects
LocalX                          618     Method local data objects
```

18.5 ASL Operator Reference

This section describes each of the ASL operators. The syntax for each operator is given, with a description of each argument and an overall description of the operator behavior. Example ASL code is provided for the more complex operators.

ASL operators can be categorized as follows:

- Named Object creation
- Method execution control (If, Else, While, etc.)
- Integer math
- Logical operators
- Resource Descriptor macros
- Object conversion
- Utility/Miscellaneous

18.5.1 Acquire (Acquire a Mutex)

Syntax

```
Acquire (SyncObject, TimeoutValue) => Boolean
```

Arguments

SynchObject must be a mutex synchronization object. *TimeoutValue* is evaluated as an Integer.

Description

Ownership of the Mutex is obtained. If the Mutex is already owned by a different invocation, the current execution thread is suspended until the owner of the Mutex releases it or until at least *TimeoutValue* milliseconds have elapsed. A Mutex can be acquired more than once by the same invocation.

This operation returns **True** if a timeout occurred and the mutex ownership was not acquired. A *TimeoutValue* of 0xFFFF (or greater) indicates that there is no timeout and the operation will wait indefinitely.

18.5.2 Add (Integer Add)

Syntax

```
Add (Addend1, Addend2, Result) => Integer
```

Arguments

Addend1 and *Addend2* are evaluated as Integers.

Description

The operands are added and the result is optionally stored into *Result.* Overflow conditions are ignored and the result of overflows simply loses the most significant bits.

18.5.3 Alias (Declare Name Alias)

Syntax

Alias (*SourceObject, AliasObject***)**

Arguments

SourceObject is any named object. *AliasObject* is a NameString.

Description

Creates a new object named *AliasObject* that refers to and acts exactly the same as *SourceObject*.

AliasObject is created as an alias of *SourceObject* in the namespace. The *SourceObject* name must already exist in the namespace. If the alias is to a name within the same definition block, the *SourceObject* name must be logically ahead of this definition in the block.

Example

The following example shows the use of an **Alias** term:

```
Alias (\SUS.SET.EVEN, SSE)
```

18.5.4 And (Integer Bitwise And)

Syntax

And (*Source1, Source2, Result***)** => Integer

Arguments

Source1 and *Source2* are evaluated as Integers.

Description

A bitwise AND is performed and the result is optionally stored into *Result*.

18.5.5 Arg*x* (Method Argument Data Objects)

Syntax

Arg0 | Arg1 | Arg2 | Arg3 | Arg4 | Arg5 | Arg6

Description

Up to 7 argument-object references can be passed to a control method. On entry to a control method, only the argument objects that are passed are usable.

18.5.6 BankField (Declare Bank/Data Field)

Syntax

BankField (*RegionName, BankName, BankValue, AccessType, LockRule, UpdateRule***) {**FieldUnitList**}**

Arguments

RegionName is the name of the host Operation Region. *BankName* is the name of the bank selection register.

Accessing the contents of a banked field data object will occur automatically through the proper bank setting, with synchronization occurring on the operation region that contains the *BankName* data variable, and on the Global Lock if specified by the *LockRule*.

The *AccessType*, *LockRule*, *UpdateRule*, and *FieldUnitList* are the same format as the **Field** operator.

Description

This operator creates data field objects. The contents of the created objects are obtained by a reference to a bank selection register.

This encoding is used to define named data field objects whose data values are fields within a larger object selected by a bank-selected register.

Example

The following is a block of ASL sample code using *BankField:*

- Creates a 4-bit bank-selected register in system I/O space.
- Creates overlapping fields in the same system I/O space that are selected via the bank register.

```
//
// Define a 256-byte operational region in SystemIO space
// and name it GIO0

OperationRegion (GIO0, SystemIO, 0x125, 0x100)

// Create some fields in GIO including a 4-bit bank select register

Field (GIO0, ByteAcc, NoLock, Preserve) {
    GLB1, 1,
    GLB2, 1,
    Offset (1),        // Move to offset for byte 1
    BNK1, 4
}

// Create FET0 & FET1 in bank 0 at byte offset 0x30

BankField (GIO0, BNK1, 0, ByteAcc, NoLock, Preserve) {
    Offset (0x30),
    FET0, 1,
    FET1, 1
}

// Create BLVL & BAC in bank 1 at the same offset

BankField (GIO0, BNK1, 1, ByteAcc, NoLock, Preserve) {
    Offset (0x30),
    BLVL, 7,
    BAC,  1
}
```

18.5.7 Break (Break from While)

Syntax

Break

Description

Break causes execution to continue immediately following the innermost enclosing **While** or **Switch** scope, in the current Method. If there is no enclosing **While** or **Switch** within the current Method, a fatal error is generated.

Compatibility Note: In ACPI 1.0, the Break operator continued immediately following the innermost "code package." Starting in ACPI 2.0, the Break operator was changed to exit the innermost "While" or "Switch" package. This should have no impact on existing code, since the ACPI 1.0 definition was, in practice, useless.

18.5.8 BreakPoint (Execution Break Point)

Syntax

BreakPoint

Description

Used for debugging, the **Breakpoint** opcode stops the execution and enters the AML debugger. In the non-debug version of the AML interpreter, **BreakPoint** is equivalent to **Noop**.

18.5.9 Buffer (Declare Buffer Object)

Syntax

Buffer (*BufferSize***) {***String* or *ByteList***}** => Buffer

Arguments

Declares a Buffer of size *BufferSize* and optional initial value of *String* or *ByteList.*

Description

The optional *BufferSize* parameter specifies the size of the buffer and the initial value is specified in *Initializer* ByteList. If *BufferSize* is not specified, it defaults to the size of initializer. If the count is too small to hold the value specified by initializer, the initializer size is used. For example, all four of the following examples generate the same data in namespace, although they have different ASL encodings:

```
Buffer (10)   {"P00.00A"}
Buffer (Arg0) {0x50, 0x30, 0x30, 0x2e, 0x30, 0x30, 0x41}
Buffer (10)   {0x50, 0x30, 0x30, 0x2e, 0x30, 0x30, 0x41, 0x00, 0x00, 0x00}
Buffer ()     {0x50, 0x30, 0x30, 0x2e, 0x30, 0x30, 0x41, 0x00, 0x00, 0x00}
```

18.5.10 Case (Expression for Conditional Execution)

Syntax

Case (*Value***) {**TermList**}**

Arguments

Value specifies an Integer, Buffer, String or Package object. *TermList* is a sequence of executable ASL expressions.

Description

Execute code based upon the value of a **Switch** statement.

If the **Case** *Value* is an Integer, Buffer or String, then control passes to the statement that matches the value of the enclosing **Switch (***Value***)**. If the **Case** value is a Package, then control passes if any member of the package matches the **Switch** (*Value*). 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 start of the *TermList* and proceeds until the end of the TermList body or until a **Break** or **Continue** operator transfers control out of the body.

18.5.11 Concatenate (Concatenate Data)

Syntax

Concatenate (*Source1, Source2, Result***)** => ComputationalData

Arguments

Source1 and *Source2* must each evaluate to an integer, a string, or a buffer. The data type of *Source1* dictates the required type of *Source2* and the type of the result object. *Source2* is implicitly converted if necessary to match the type of *Source1*.

Description

Source2 is concatenated to *Source1* and the result data is optionally stored into *Result.*

Table 18-16 Concatenate Data Types

Source1 Data Type	Source2 Data Type (➔ Converted Type)	Result Data Type
Integer	Integer/String/Buffer ➔ Integer	Buffer
String	Integer/String/Buffer ➔ String	String
Buffer	Integer/String/Buffer ➔ Buffer	Buffer

18.5.12 ConcatenateResTemplate (Concatenate Resource Templates)

Syntax

ConcatenateResTemplate (*Source1, Source2, Result***)** => Buffer

Arguments

Source1 and *Source2* are evaluated as Resource Template buffers.

Description

The resource descriptors from *Source2* are appended to the resource descriptors from *Source1*. Then a new end tag and checksum are appended and the result is stored in *Result*, if specified. If either *Source1* or *Source2* is exactly 1 byte in length, a run-time error occurs. An empty buffer is treated as a resource template with only an end tag.

18.5.13 CondRefOf (Create Object Reference Conditionally)

Syntax

CondRefOf (*Source, Result***)** => Boolean

Arguments

Attempts to create a reference to the *Source* object. The *Source* of this operation can be any object type (for example, data package, device object, and so on), and the result data is optionally stored into *Result.*

Description

On success, the *Destination* object is set to refer to *Source* and the execution result of this operation is the value **True**. On failure, *Destination* is unchanged and the execution result of this operation is the value **False**. This can be used to reference items in the namespace that may appear dynamically (for example, from a dynamically loaded definition block).

CondRefOf is equivalent to **RefOf** except that if the *Source* object does not exist, it is fatal for **RefOf** but not for **CondRefOf**.

18.5.14 Continue (Continue Innermost Enclosing While)

Syntax

```
Continue
```

Description

Continue causes execution to continue at the start of the innermost enclosing **While** scope, in the currently executing Control Method, at the point where the condition is evaluated. If there is no enclosing **While** within the current Method, a fatal error is generated.

18.5.15 CopyObject (Copy and Store Object)

Syntax

```
CopyObject (Source, Destination) => DataRefObject
```

Arguments

Converts the contents of the *Source* to a DataRefObject using the conversion rules in 18.2.5 and then copies the results without conversion to the object referred to by *Destination*.

Description

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 was first introduced new in ACPI 2.0.

18.5.16 CreateBitField (Create 1-Bit Buffer Field)

Syntax

```
CreateBitField (SourceBuffer, BitIndex, BitFieldName)
```

Arguments

SourceBuffer is evaluated as a buffer. *BitIndex* is evaluated as an integer. *BitFieldName* is a NameString.

Description

A new buffer field object named *BitFieldName* is created for the bit of *SourceBuffer* at the bit index of *BitIndex*. The bit-defined field within *SourceBuffer* must exist.*BitFieldName* is created for the bit of *SourceBuffer* at the bit index of *BitIndex*. The bit-defined field within *SourceBuffer* must exist.

18.5.17 CreateByteField (Create 8-Bit Buffer Field)

Syntax

CreateByteField (*SourceBuffer, ByteIndex, ByteFieldName***)**

Arguments

SourceBuffer is evaluated as a buffer. *ByteIndex* is evaluated as an integer. *ByteFieldName* is a NameString.

Description

A new buffer field object named *ByteFieldName* is created for the byte of *SourceBuffer* at the byte index of *ByteIndex*. The byte-defined field within *SourceBuffer* must exist.

18.5.18 CreateDWordField (Create 32-Bit Buffer Field)

Syntax

CreateDWordField (*SourceBuffer, ByteIndex, DWordFieldName***)**

Arguments

SourceBuffer is evaluated as a buffer. *ByteIndex* is evaluated as an integer. *DWordFieldName* is a NameString.

Description

A new buffer field object named *DWordFieldName* is created for the DWord of *SourceBuffer* at the byte index of *ByteIndex*. The DWord-defined field within *SourceBuffer* must exist.

18.5.19 CreateField (Create Arbitrary Length Buffer Field)

Syntax

CreateField (*SourceBuffer, BitIndex, NumBits, FieldName***)**

Arguments

SourceBuffer is evaluated as a buffer. *BitIndex* and *NumBits* are evaluated as integers. *FieldName* is a NameString.

Description

A new buffer field object named *FieldName* is created for the bits of *SourceBuffer* at *BitIndex* for *NumBits*. The entire bit range of the defined field within *SourceBuffer* must exist. If *NumBits* evaluates to zero, a fatal exception is generated.

18.5.20 CreateQWordField (Create 64-Bit Buffer Field)

Syntax

CreateQWordField (*SourceBuffer, ByteIndex, QWordFieldName***)**

Arguments

SourceBuffer is evaluated as a buffer. *ByteIndex* is evaluated as an integer. *QWordFieldName* is a NameString.

Description

A new buffer field object named *QWordFieldName* is created for the QWord of *SourceBuffer* at the byte index of *ByteIndex*. The QWord-defined field within *SourceBuffer* must exist.

18.5.21 CreateWordField (Create 16-Bit Buffer Field)

Syntax

```
CreateWordField (SourceBuffer, ByteIndex, WordFieldName)
```

Arguments

SourceBuffer is evaluated as a buffer. *ByteIndex* is evaluated as an integer. *WordFieldName* is a NameString.

Description

A new bufferfield object named *WordFieldName* is created for the word of *SourceBuffer* at the byte index of *ByteIndex.* The word-defined field within *SourceBuffer* must exist.

18.5.22 DataTableRegion (Create Data Table Operation Region)

Syntax

```
DataTableRegion (RegionName, SignatureString, OemIDString, OemTableIDString)
```

Arguments

Creates a new region named *RegionName. SignatureString*, *OemIDString* and *OemTableIDString* are evaluated as strings.

Description

A Data Table Region is a special Operation Region whose RegionSpace is SystemMemory . Any table referenced by a Data Table Region must be in memory marked by AddressRangeReserved or AddressRangeNVS.

The memory referred to by the Data Table Region is the memory that is occupied by the table referenced in XSDT that is identified by *SignatureString*, *OemIDString* and *OemTableIDString*. Any Field object can reference RegionName

The base address of a Data Table region is the address of the first byte of the header of the table identified by *SignatureString*, *OemIDString* and *OemTableIDString*. The length of the region is the length of the table.

18.5.23 Debug (Debugger Output)

Syntax

Debug

Description

The debug data object is a virtual data object. Writes to this object provide debugging information. On at least debug versions of the interpreter, any writes into this object are appropriately displayed on the system's native kernel debugger. All writes to the debug object are otherwise benign. If the system is in use without a kernel debugger, then writes to the debug object are ignored. The following table relates the ASL term types that can be written to the Debug object to the format of the information on the kernel debugger display.

Table 18-17 Debug Object Display Formats

ASL Term Type	Display Format
Numeric data object	All digits displayed in hexadecimal format.
String data object	String is displayed.
Object reference	Information about the object is displayed (for example, object type and object name), but the object is not evaluated.

The Debug object is a write-only object; attempting to read from the debug object is not supported.

18.5.24 Decrement (Integer Decrement)

Syntax

Decrement (*Minuend***)** => Integer

Arguments

Minuend is evaluated as an Integer.

Description

This operation decrements the *Minuend* by one and the result is stored back to *Minuend*. Equivalent to **Subtract (***Minuend*, 1, *Minuend***)**. Underflow conditions are ignored and the result is Ones.

18.5.25 Default (Default Execution Path in Switch)

Syntax

Default {TermList**}**

Arguments

TermList is a sequence of executable ASL expressions.

Description

Within the body of a **Switch** (page 548) statement, the statements specified by *TermList* will be executed if no **Case** (page 489) statement value matches the Switch statement value. If **Default** 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 in the immediate scope of the parent Switch statement. The **Default** statement can appear anywhere in the body of the **Switch** statement.

18.5.26 DefinitionBlock (Declare Definition Block)

Syntax

```
DefinitionBlock (AMLFileName, TableSignature, ComplianceRevision, OEMID,
     TableID, OEMRevision) {TermList}
```

Arguments

AMLFileName is a string that specifies the desired name of the translated output AML file. *TableSignature* is a string that contains the 4-character ACPI signature. *ComplianceRevision* is an 8-bit value. *OEMID* is a 6-character string, *TableId* is an 8-character string, and *OEMRevision* is a 32-bit value. *TermList* is a sequence of executable ASL expressions.

Description

The **DefinitionBlock** term specifies the unit of data and/or AML code that the OS will load as part of the Differentiated Definition Block or as part of an additional Definition Block.

This unit of data and/or AML code describes either the base system or some large extension (such as a docking station). The entire DefinitionBlock will be loaded and compiled by the OS as a single unit, and can be unloaded by the OS as a single unit.

Note: For compatibility with ACPI versions before ACPI 2.0, the bit width of Integer objects is dependent on the *ComplianceRevision* of the DSDT. If the *ComplianceRevision* is less than 2, all integers are restricted to 32 bits. Otherwise, full 64-bit integers are used. The version of the DSDT sets the global integer width for all integers, including integers in SSDTs.

18.5.27 DerefOf (Dereference an Object Reference)

Syntax

```
DerefOf (Source) => Object
```

Arguments

Returns the object referred by the *Source* object reference.

Description

If the *Source* evaluates to an object reference, the actual contents of the object referred to are returned. If the *Source* evaluates to a string, the string is evaluated as an ASL name (relative to the current scope) and the contents of that object are returned. If the object specified by *Source* does not exist then a fatal error is generated.

Compatibility Note: The use of a String with **DerefOf** was first introduced in ACPI 2.0.

18.5.28 Device (Declare Bus/Device Package)

Syntax

```
Device (DeviceName) {ObjectList}
```

Arguments

Creates a Device object of name *DeviceName*, which represents either a bus or a device or any other similar hardware. **Device** opens a name scope.

Description

A Bus/Device Package is one of the basic ways the Differentiated Definition Block describes the hardware devices in the system to the operating software. Each Bus/Device Package is defined somewhere in the hierarchical namespace corresponding to that device's location in the system. Within the namespace of the device are other names that provide information and control of the device, along with any sub-devices that in turn describe sub-devices, and so on.

For any device, the BIOS provides only information that is added to the device in a non-hardware standard manner. This type of value-added function is expressible in the ACPI Definition Block such that operating software can use the function.

The BIOS supplies Device Objects only for devices that are obtaining some system-added function outside the device's normal capabilities and for any Device Object required to fill in the tree for such a device. For example, if the system includes a PCI device (integrated or otherwise) with no additional functions such as power management, the BIOS would not report such a device; however, if the system included an integrated ISA device below the integrated PCI device (device is an ISA bridge), then the system would include a Device Package for the ISA device with the minimum feature being added being the ISA device's ID and configuration information and the parent PCI device, because it is required to get the ISA Device Package placement in the namespace correct.

Example

The following block of ASL sample code shows a nested use of Device objects to describe an IDE controller connected to the root PCI bus.

```
Device (IDE0) {        // primary controller
    Name (_ADR, 0)     // put PCI Address (device/function) here

    // define region for IDE mode register

    OperationRegion (PCIC, PCI_Config, 0x50, 0x10)
    Field (PCIC, AnyAcc, NoLock, Preserve) {
        …
    }
    Device (PRIM) {               // Primary adapter
        Name (_ADR, 0)         // Primary adapter = 0
        …
        Method (_STM, 2) {
            …
        }
        Method (_GTM) {
            …
        }
        Device (MSTR) {            // master channel
            Name (_ADR, 0)
            Name (_PR0, Package () {0, PIDE})

            Name (_GTF) {
                …
            }
        }
        Device (SLAV) {
            Name (_ADR, 1)
            Name (_PR0, Package () {0, PIDE})
            Name (_GTF) {
                …
            }
        }
    }
}
```

18.5.29 Divide (Integer Divide)

Syntax

```
Divide (Dividend, Divisor, Remainder, Result) => Integer
```

Arguments

Dividend and *Divisor* are evaluated as Integers.

Description

Dividend is divided by *Divisor*, then the resulting remainder is optionally stored into *Remainder* and the resulting quotient is optionally stored into *Result*. Divide-by-zero exceptions are fatal.

The function return value is the *Result (quotient).*

18.5.30 DMA (DMA Resource Descriptor Macro)

Syntax

```
DMA (DmaType, IsBusMaster, DmaTransferSize, DescriptorName) {DmaChannelList}
     => Buffer
```

Arguments

DmaType specifies the type of DMA cycle: ISA compatible (**Compatibility**), EISA Type A (**TypeA**), EISA Type B (**TypeB**) or EISA Type F (**TypeF**). The 2-bit field *DescriptorName*._TYP is automatically created to refer to this portion of the resource descriptor, where ‘0’ is Compatibility, ‘1’ is TypeA, ‘2’ is TypeB and ‘3’ is TypeF.

IsBusMaster specifies whether this device can generate DMA bus master cycles (**BusMaster**) or not (**NotBusMaster**). If nothing is specified, then BusMaster is assumed. The 1-bit field *DescriptorName*._BM is automatically created to refer to this portion of the resource descriptor, where ‘0’ is NotBusMaster and ‘1’ is BusMaster.

DmaTransferSize specifies the size of DMA cycles the device is capable of generating: 8-bit (**Transfer8**), 16-bit (**Transfer16**) or both 8 and 16-bit (**Transfer8_16**). The 2-bit field *DescriptorName*._SIZ is automatically created to refer to this portion of the resource descriptor, where ‘0’ is Transfer8, ‘1’ is Transfer8_16 and ‘2’ is Transfer16.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

DmaChannelList is a comma-delimited list of integers in the range 0 through 7 that specify the DMA channels used by the device. There may be no duplicates in the list.

Description

The **DMA** macro evaluates to a buffer which contains a DMA resource descriptor. The format of the DMA resource descriptor can be found in “DMA Descriptor” (page 225). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.31 DWordIO (DWord IO Resource Descriptor Macro)

Syntax

```
DWordIO (ResourceUsage, IsMinFixed, IsMaxFixed, Decode, ISARanges,
     AddressGranularity, AddressMinimum, AddressMaximum, AddressTranslation,
     RangeLength, ResourceSourceIndex, ResourceSource, DescriptorName,
     TranslationType, TranslationDensity)
```

Arguments

ResourceUsage specifies whether the I/O range is consumed by this device (**ResourceConsumer**) or passed on to child devices (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

IsMinFixed specifies whether the minimum address of this I/O range is fixed (**MinFixed**) or can be changed (**MinNotFixed**). If nothing is specified, then MinNotFixed is assumed. The 1-bit field *DescriptorName*._MIF is automatically created to refer to this portion of the resource descriptor, where '1' is MinFixed and '0' is MinNotFixed.

IsMaxFixed specifies whether the maximum address of this I/O range is fixed (**MaxFixed**) or can be changed (**MaxNotFixed**). If nothing is specified, then MaxNotFixed is assumed. The 1-bit field *DescriptorName*._MAF is automatically created to refer to this portion of the resource descriptor, where '1' is MaxFixed and '0' is MaxNotFixed.

Decode specifies whether or not the device decodes the I/O range using positive (**PosDecode**) or subtractive (**SubDecode**) decode. If nothing is specified, then PosDecode is assumed. The 1-bit field *DescriptorName*._DEC is automatically created to refer to this portion of the resource descriptor, where '1' is SubDecode and '0' is PosDecode.

ISARanges specifies whether the I/O ranges specifies are limited to valid ISA I/O ranges (**ISAOnly**), valid non-ISA I/O ranges (**NonISAOnly**) or encompass the whole range without limitation (**EntireRange**). The 2-bit field *DescriptorName*._RNG is automatically created to refer to this portion of the resource descriptor, where '1' is NonISAOnly, '2' is ISAOnly and '0' is EntireRange.

AddressGranularity evaluates to a 32-bit integer that specifies the power-of-two boundary (- 1) on which the I/O range must be aligned. The 32-bit field *DescriptorName*._GRA is automatically created to refer to this portion of the resource descriptor.

AddressMinimum evaluates to a 32-bit integer that specifies the lowest possible base address of the I/O range. The value must have '0' in all bits where the corresponding bit in *AddressGranularity* is '1'. For bridge devices which translate addresses, this is the address on the secondary bus. The 32-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 32-bit integer that specifies the highest possible base address of the I/O range. The value must have '0' in all bits where the corresponding bit in *AddressGranularity* is '1'. For bridge devices which translate addresses, this is the address on the secondary bus. The 32-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressTranslation evaluates to a 32-bit integer that specifies the offset to be added to a secondary bus I/O address which results in the corresponding primary bus I/O address. For all non-bridge devices or bridges which do not perform translation, this must be '0'. The 32-bit field *DescriptorName*._TRA is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 32-bit integer that specifies the total number of bytes decoded in the I/O range. The 32-bit field *DescriptorName*._LEN is automatically created to refer to this portion of the resource descriptor.

ResourceSourceIndex is an optional argument which evaluates to an 8-bit integer that specifies the resource descriptor within the object specified by *ResourceSource*. If this argument is specified, the *ResourceSource* argument must also be specified.

ResourceSource is an optional argument which evaluates to a string containing the path of a device which produces the pool of resources from which this I/O range is allocated. If this argument is specified, but the *ResourceSourceIndex* argument is not specified, a value of zero is assumed.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

TranslationType is an optional argument that specifies whether the resource type on the secondary side of the bus is different (**TypeTranslation**) from that on the primary side of the bus or the same (**TypeStatic**). If TypeTranslation is specified, then the secondary side of the bus is Memory. If TypeStatic is specified, then the secondary side of the bus is I/O. If nothing is specified, then TypeStatic is assumed. The 1-bit field *DescriptorName*._TTP is automatically created to refer to this portion of the resource descriptor, where '1' is TypeTranslation and '0' is TypeStatic. See _TTP (page 248) for more information

TranslationDensity is an optional argument that specifies whether or not the translation from the primary to secondary bus is sparse (**SparseTranslation**) or dense (**DenseTranslation**). It is only used when *TranslationType* is **TypeTranslation**. If nothing is specified, then DenseTranslation is assumed. The 1-bit field *DescriptorName*._TRS is automatically created to refer to this portion of the resource descriptor, where '1' is SparseTranslation and '0' is DenseTranslation. See _TRS (page 248) for more information.

Description

The **DWordIO** macro evaluates to a buffer which contains a 32-bit I/O range resource descriptor. The format of the 32-bit I/O range resource descriptor can be found in "DWord Address Space Descriptor " (page 238). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.32 DWordMemory (DWord Memory Resource Descriptor Macro)

Syntax

```
DWordMemory (ResourceUsage, Decode, IsMinFixed, IsMaxFixed, Cacheable,
     ReadAndWrite, AddressGranularity, AddressMinimum, AddressMaximum,
     AddressTranslation, RangeLength, ResourceSourceIndex, ResourceSource,
     DescriptorName, MemoryType, TranslationType)
```

Arguments

ResourceUsage specifies whether the Memory range is consumed by this device (**ResourceConsumer**) or passed on to child devices (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

Decode specifies whether or not the device decodes the Memory range using positive (**PosDecode**) or subtractive (**SubDecode**) decode. If nothing is specified, then PosDecode is assumed. The 1-bit field *DescriptorName*._DEC is automatically created to refer to this portion of the resource descriptor, where '1' is SubDecode and '0' is PosDecode.

IsMinFixed specifies whether the minimum address of this Memory range is fixed (**MinFixed**) or can be changed (**MinNotFixed**). If nothing is specified, then MinNotFixed is assumed. The 1-bit field *DescriptorName*._MIF is automatically created to refer to this portion of the resource descriptor, where '1' is MinFixed and '0' is MinNotFixed.

IsMaxFixed specifies whether the maximum address of this Memory range is fixed (**MaxFixed**) or can be changed (**MaxNotFixed**). If nothing is specified, then MaxNotFixed is assumed. The 1-bit field *DescriptorName.*_MAF is automatically created to refer to this portion of the resource descriptor, where '1' is MaxFixed and '0' is MaxNotFixed.

Cacheable specifies whether or not the memory region is cacheable (**Cacheable**), cacheable and write-combining (**WriteCombining**), cacheable and prefetchable (**Prefetchable**) or uncacheable (**NonCacheable**). If nothing is specified, then NonCacheable is assumed. The 2-bit field *DescriptorName*._MEM is automatically created to refer to this portion of the resource descriptor, where '1' is Cacheable, '2' is WriteCombining, '3' is Prefetchable and '0' is NonCacheable.

ReadAndWrite specifies whether or not the memory region is read-only (**ReadOnly**) or read/write (**ReadWrite**). If nothing is specified, then ReadWrite is assumed. The 1-bit field *DescriptorName*._RW is automatically created to refer to this portion of the resource descriptor, where '1' is ReadWrite and '0' is ReadOnly.

AddressGranularity evaluates to a 32-bit integer that specifies the power-of-two boundary (- 1) on which the Memory range must be aligned. The 32-bit field *DescriptorName*._GRA is automatically created to refer to this portion of the resource descriptor.

AddressMinimum evaluates to a 32-bit integer that specifies the lowest possible base address of the Memory range. The value must have '0' in all bits where the corresponding bit in *AddressGranularity* is '1'. For bridge devices which translate addresses, this is the address on the secondary bus. The 32-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 32-bit integer that specifies the highest possible base address of the Memory range. The value must have '0' in all bits where the corresponding bit in *AddressGranularity* is '1'. For bridge devices which translate addresses, this is the address on the secondary bus. The 32-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressTranslation evaluates to a 32-bit integer that specifies the offset to be added to a secondary bus I/O address which results in the corresponding primary bus I/O address. For all non-bridge devices or bridges which do not perform translation, this must be '0'. The 32-bit field *DescriptorName*._TRA is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 32-bit integer that specifies the total number of bytes decoded in the Memory range. The 32-bit field *DescriptorName*._LEN is automatically created to refer to this portion of the resource descriptor.

ResourceSourceIndex is an optional argument which evaluates to an 8-bit integer that specifies the resource descriptor within the object specified by *ResourceSource*. If this argument is specified, the *ResourceSource* argument must also be specified.

ResourceSource is an optional argument which evaluates to a string containing the path of a device which produces the pool of resources from which this Memory range is allocated. If this argument is specified, but the *ResourceSourceIndex* argument is not specified, a zero value is assumed.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

MemoryType is an optional argument that specifies the memory usage. The memory can be marked as normal (**AddressRangeMemory**), used as ACPI NVS space (**AddressRangeNVS**), used as ACPI reclaimable space (**AddressRangeACPI**) or as system reserved (**AddressRangeReserved**). If nothing is specified, then AddressRangeMemory is assumed. The 2-bit field *DescriptorName*._MTP is automatically created in order to refer to this portion of the resource descriptor, where '0' is AddressRangeMemory, '1' is AddressRangeReserved, '2' is AddressRangeACPI and '3' is AddressRangeNVS.

TranslationType is an optional argument that specifies whether the resource type on the secondary side of the bus is different (**TypeTranslation**) from that on the primary side of the bus or the same (**TypeStatic**). If TypeTranslation is specified, then the secondary side of the bus is I/O. If TypeStatic is specified, then the secondary side of the bus is I/O. If nothing is specified, then TypeStatic is assumed. The 1-bit field *DescriptorName*._TTP is automatically created to refer to this portion of the resource descriptor, where '1' is TypeTranslation and '0' is TypeStatic. See _TTP (page 248) for more information.

Description

The **DWordMemory** macro evaluates to a buffer which contains a 32-bit memory resource descriptor. The format of the 32-bit memory resource descriptor can be found in “DWord Address Space Descriptor ” (page 238). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.33 DWordSpace (DWord Space Resource Descriptor Macro)

Syntax

```
DWordSpace (ResourceType, ResourceUsage, Decode, IsMinFixed, IsMaxFixed,
     TypeSpecificFlags, AddressGranularity, AddressMinimum, AddressMaximum,
     AddressTranslation, RangeLength, ResourceSourceIndex, ResourceSource,
     DescriptorName)
```

Arguments

ResourceType evaluates to an 8-bit integer that specifies the type of this resource. Acceptable values are 0xC0 through 0xFF.

ResourceUsage specifies whether the Memory range is consumed by this device (**ResourceConsumer**) or passed on to child devices (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

Decode specifies whether or not the device decodes the Memory range using positive (**PosDecode**) or subtractive (**SubDecode**) decode. If nothing is specified, then PosDecode is assumed. The 1-bit field *DescriptorName*._DEC is automatically created to refer to this portion of the resource descriptor, where ‘1’ is SubDecode and ‘0’ is PosDecode.

IsMinFixed specifies whether the minimum address of this Memory range is fixed (**MinFixed**) or can be changed (**MinNotFixed**). If nothing is specified, then MinNotFixed is assumed. The 1-bit field *DescriptorName*._MIF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MinFixed and ‘0’ is MinNotFixed.

IsMaxFixed specifies whether the maximum address of this Memory range is fixed (**MaxFixed**) or can be changed (**MaxNotFixed**). If nothing is specified, then MaxNotFixed is assumed. The 1-bit field *DescriptorName.*_MAF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MaxFixed and ‘0’ is MaxNotFixed.

TypeSpecificFlags evaluates to an 8-bit integer. The flags are specific to the *ResourceType*.

AddressGranularity evaluates to a 32-bit integer that specifies the power-of-two boundary (- 1) on which the Memory range must be aligned. The 32-bit field *DescriptorName*._GRA is automatically created to refer to this portion of the resource descriptor.

AddressMinimum evaluates to a 32-bit integer that specifies the lowest possible base address of the Memory range. The value must have ‘0’ in all bits where the corresponding bit in *AddressGranularity* is ‘1’. For bridge devices which translate addresses, this is the address on the secondary bus. The 32-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 32-bit integer that specifies the highest possible base address of the Memory range. The value must have ‘0’ in all bits where the corresponding bit in *AddressGranularity* is ‘1’. For bridge devices which translate addresses, this is the address on the secondary bus. The 32-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressTranslation evaluates to a 32-bit integer that specifies the offset to be added to a secondary bus I/O address which results in the corresponding primary bus I/O address. For all non-bridge devices or bridges which do not perform translation, this must be ‘0’. The 32-bit field *DescriptorName*._TRA is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 32-bit integer that specifies the total number of bytes decoded in the Memory range. The 32-bit field *DescriptorName*._LEN is automatically created to refer to this portion of the resource descriptor.

ResourceSourceIndex is an optional argument which evaluates to an 8-bit integer that specifies the resource descriptor within the object specified by *ResourceSource*. If this argument is specified, the *ResourceSource* argument must also be specified.

ResourceSource is an optional argument which evaluates to a string containing the path of a device which produces the pool of resources from which this Memory range is allocated. If this argument is specified, but the *ResourceSourceIndex* argument is not specified, a zero value is assumed.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

Description

The **DWordSpace** macro evaluates to a buffer which contains a 32-bit Address Space resource descriptor. The format of the 32-bit Address Space resource descriptor can be found in “DWord Address Space Descriptor ” (page 238). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.34 EISAID (EISA ID String To Integer Conversion Macro)

Syntax

```
EISAID (EisaIdString) => DWordConst
```

Arguments

The *EisaIdString* must be a String object of the form “UUUNNNN”, where “U” is an uppercase letter and “N” is a hexadecimal digit. No asterisks or other characters are allowed in the string.

Description

Converts *EisaIdString*, a 7-character text string argument, into its corresponding 4-byte numeric EISA ID encoding. It can be used when declaring IDs for devices that have EISA IDs.

Example

```
EISAID (“PNP0C09”)    // This is a valid invocation of the macro.
```

18.5.35 Else (Alternate Execution)

Syntax

```
Else {TermList}
```

Arguments

TermList is a sequence of executable ASL statements.

Description

If *Predicate* evaluates to 0 in an **If** statement, then control is transferred to the Else portion, which can consist of zero or more **ElseIf** statements followed by zero or one **Else** statements. If the *Predicate* of any **ElseIf** statement evaluates to non-zero, the statements in its term list are executed and then control is transferred past the end of the final Else term. If no *Predicate* evaluates to non-zero, then the statements in the **Else** term list are executed.

Example

The following example checks Local0 to be zero or non-zero. On non-zero, CNT is incremented; otherwise, CNT is decremented.

```
If (LGreater (Local0, 5)
{
    Increment (CNT)
} Else If (Local0) {
    Add (CNT, 5, CNT)
}
Else
{
    Decrement (CNT)
}
```

18.5.36 ElseIf (Alternate/Conditional Execution)

Syntax

ElseIf (*Predicate***)**

Arguments

Predicate is evaluated as an Integer.

Description

If the *Predicate* of any **ElseIf** statement evaluates to non-zero, the statements in its term list are executed and then control is transferred past the end of the final **Else**. If no *Predicate* evaluates to non-zero, then the statements in the **Else** term list are executed.

Compatibility Note: The **ElseIf** operator was first introduced in ACPI 2.0, but is backward compatible with the ACPI 1.0 specification. An ACPI 2.0 and later ASL compiler must synthesize **ElseIf** from the **If**. and **Else** opcodes available in 1.0. For example:

```
If (predicate1)
{
    …statements1…
}
ElseIf (predicate2)
{
    …statements2…
}
Else
{
    …statements3…
}
```

is translated to the following:

```
If (predicate1)
{
    …statements1…
}
Else
{
    If (predicate2)
    {
        …statements2…
    }
    Else
    {
        …statements3…
    }
}
```

18.5.37 EndDependentFn (End Dependent Function Resource Descriptor Macro)

Syntax

```
EndDependentFn () => Buffer
```

Description

The **EndDependentFn** macro generates an end-of-dependent-function resource descriptor buffer inside of an ResourceTemplate (page 544). It must be matched with a StartDependentFn (page 547) or StartDependentFnNoPri (page 547) macro.

18.5.38 Event (Declare Event Synchronization Object)

Syntax

```
Event (EventName)
```

Arguments

Creates an event synchronization object named *EventName*.

Description

For more information about the uses of an event synchronization object, see the ASL definitions for the Wait, Signal, and Reset function operators.

18.5.39 ExtendedIO (Extended IO Resource Descriptor Macro)

Syntax

```
ExtendedIO (ResourceUsage, IsMinFixed, IsMaxFixed, Decode, ISARanges,
     AddressGranularity, AddressMinimum, AddressMaximum, AddressTranslation,
     RangeLength, TypeSpecificAttributes, DescriptorName, TranslationType,
     TranslationDensity)
```

Arguments

ResourceUsage specifies whether the Memory range is consumed by this device (**ResourceConsumer**) or passed on to child devices (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

IsMinFixed specifies whether the minimum address of this I/O range is fixed (**MinFixed**) or can be changed (**MinNotFixed**). If nothing is specified, then MinNotFixed is assumed. The 1-bit field *DescriptorName*._MIF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MinFixed and ‘0’ is MinNotFixed.

IsMaxFixed specifies whether the maximum address of this I/O range is fixed (**MaxFixed**) or can be changed (**MaxNotFixed**). If nothing is specified, then MaxNotFixed is assumed. The 1-bit field *DescriptorName*._MAF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MaxFixed and ‘0’ is MaxNotFixed.

Decode specifies whether or not the device decodes the I/O range using positive (**PosDecode**) or subtractive (**SubDecode**) decode. If nothing is specified, then PosDecode is assumed. The 1-bit field *DescriptorName*._DEC is automatically created to refer to this portion of the resource descriptor, where ‘1’ is SubDecode and ‘0’ is PosDecode.

ISARanges specifies whether the I/O ranges specifies are limited to valid ISA I/O ranges (**ISAOnly**), valid non-ISA I/O ranges (**NonISAOnly**) or encompass the whole range without limitation (**EntireRange**). The 2-bit field *DescriptorName*._RNG is automatically created to refer to this portion of the resource descriptor, where '1' is NonISAOnly, '2' is ISAOnly and '0' is EntireRange.

AddressGranularity evaluates to a 64-bit integer that specifies the power-of-two boundary (- 1) on which the I/O range must be aligned. The 64-bit field *DescriptorName*._GRA is automatically created to refer to this portion of the resource descriptor.

AddressMinimum evaluates to a 64-bit integer that specifies the lowest possible base address of the I/O range. The value must have '0' in all bits where the corresponding bit in *AddressGranularity* is '1'. For bridge devices which translate addresses, this is the address on the secondary bus. The 64-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 64-bit integer that specifies the highest possible base address of the I/O range. The value must have '0' in all bits where the corresponding bit in *AddressGranularity* is '1'. For bridge devices which translate addresses, this is the address on the secondary bus. The 64-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressTranslation evaluates to a 64-bit integer that specifies the offset to be added to a secondary bus I/O address which results in the corresponding primary bus I/O address. For all non-bridge devices or bridges which do not perform translation, this must be '0'. The 64-bit field *DescriptorName*._TRA is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 64-bit integer that specifies the total number of bytes decoded in the I/O range. The 64-bit field *DescriptorName*._LEN is automatically created to refer to this portion of the resource descriptor.

TranslationType is an optional argument that specifies whether the resource type on the secondary side of the bus is different (**TypeTranslation**) from that on the primary side of the bus or the same (**TypeStatic**). If TypeTranslation is specified, then the secondary side of the bus is Memory. If TypeStatic is specified, then the secondary side of the bus is I/O. If nothing is specified, then TypeStatic is assumed. The 1-bit field *DescriptorName*. _TTP is automatically created to refer to this portion of the resource descriptor, where '1' is TypeTranslation and '0' is TypeStatic. See _TTP (page 248) for more information

TranslationDensity is an optional argument that specifies whether or not the translation from the primary to secondary bus is sparse (**SparseTranslation**) or dense (**DenseTranslation**). It is only used when *TranslationType* is **TypeTranslation**. If nothing is specified, then DenseTranslation is assumed. The 1-bit field *DescriptorName*._TRS is automatically created to refer to this portion of the resource descriptor, where '1' is SparseTranslation and '0' is DenseTranslation. See _TRS (page 248) for more information.

TypeSpecificAttributes is an optional argument that specifies attributes specific to this resource type. See section 6.4.3.5.4.1,"Type Specific Attributes".

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operatorsDescription

The **ExtendedIO** macro evaluates to a buffer which contains a 64-bit I/O resource descriptor, which describes a range of I/O addresses. The format of the 64-bit I/O resource descriptor can be found in "Extended Address Space Descriptor" (page 242). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.40 ExtendedMemory (Extended Memory Resource Descriptor Macro)

Syntax

```
ExtendedMemory (ResourceUsage, Decode, IsMinFixed, IsMaxFixed, Cacheable,
     ReadAndWrite, AddressGranularity, AddressMinimum, AddressMaximum,
     AddressTranslation, RangeLength, TypeSpecificAttributes,
     DescriptorName, MemoryType, TranslationType)
```

Arguments

ResourceUsage specifies whether the Memory range is consumed by this device (**ResourceConsumer**) or passed on to child devices (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

Decode specifies whether or not the device decodes the Memory range using positive (**PosDecode**) or subtractive (**SubDecode**) decode. If nothing is specified, then PosDecode is assumed. The 1-bit field *DescriptorName*._DEC is automatically created to refer to this portion of the resource descriptor, where ‘1’ is SubDecode and ‘0’ is PosDecode.

IsMinFixed specifies whether the minimum address of this Memory range is fixed (**MinFixed**) or can be changed (**MinNotFixed**). If nothing is specified, then MinNotFixed is assumed. The 1-bit field *DescriptorName*. _MIF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MinFixed and ‘0’ is MinNotFixed.

IsMaxFixed specifies whether the maximum address of this Memory range is fixed (**MaxFixed**) or can be changed (**MaxNotFixed**). If nothing is specified, then MaxNotFixed is assumed. The 1-bit field *DescriptorName.* _MAF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MaxFixed and ‘0’ is MaxNotFixed.

Cacheable specifies whether or not the memory region is cacheable (**Cacheable**), cacheable and write-combining (**WriteCombining**), cacheable and prefetchable (**Prefetchable**) or uncacheable (**NonCacheable**). If nothing is specified, then NonCacheable is assumed. The 2-bit field *DescriptorName*._MEM is automatically created to refer to this portion of the resource descriptor, where ‘1’ is Cacheable, ‘2’ is WriteCombining, ‘3’ is Prefetchable and ‘0’ is NonCacheable.

ReadAndWrite specifies whether or not the memory region is read-only (**ReadOnly**) or read/write (**ReadWrite**). If nothing is specified, then ReadWrite is assumed. The 1-bit field *DescriptorName*._RW is automatically created to refer to this portion of the resource descriptor, where ‘1’ is ReadWrite and ‘0’ is ReadOnly.

AddressGranularity evaluates to a 64-bit integer that specifies the power-of-two boundary (- 1) on which the Memory range must be aligned. The 64-bit field *DescriptorName*._GRA is automatically created to refer to this portion of the resource descriptor.

AddressMinimum evaluates to a 64-bit integer that specifies the lowest possible base address of the Memory range. The value must have ‘0’ in all bits where the corresponding bit in *AddressGranularity* is ‘1’. For bridge devices which translate addresses, this is the address on the secondary bus. The 64-bit field *DescriptorName* ._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 64-bit integer that specifies the highest possible base address of the Memory range. The value must have ‘0’ in all bits where the corresponding bit in *AddressGranularity* is ‘1’. For bridge devices which translate addresses, this is the address on the secondary bus. The 64-bit field *DescriptorName* ._MAX is automatically created to refer to this portion of the resource descriptor.

AddressTranslation evaluates to a 64-bit integer that specifies the offset to be added to a secondary bus I/O address which results in the corresponding primary bus I/O address. For all non-bridge devices or bridges which do not perform translation, this must be ‘0’. The 64-bit field *DescriptorName*. _TRA is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 64-bit integer that specifies the total number of bytes decoded in the Memory range. The 64-bit field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

MemoryType is an optional argument that specifies the memory usage. The memory can be marked as normal (**AddressRangeMemory**), used as ACPI NVS space (**AddressRangeNVS**), used as ACPI reclaimable space (**AddressRangeACPI**) or as system reserved (**AddressRangeReserved**). If nothing is specified, then AddressRangeMemory is assumed. The 2-bit field *DescriptorName*. _MTP is automatically created in order to refer to this portion of the resource descriptor, where '0' is AddressRangeMemory, '1' is AddressRangeReserved, '2' is AddressRangeACPI and '3' is AddressRangeNVS.

TranslationType is an optional argument that specifies whether the resource type on the secondary side of the bus is different (**TypeTranslation**) from that on the primary side of the bus or the same (**TypeStatic**). If TypeTranslation is specified, then the secondary side of the bus is I/O. If TypeStatic is specified, then the secondary side of the bus is I/O. If nothing is specified, then TypeStatic is assumed. The 1-bit field *DescriptorName*. _TTP is automatically created to refer to this portion of the resource descriptor, where '1' is TypeTranslation and '0' is TypeStatic. See _TTP (page 248) for more information.

TypeSpecificAttributes is an optional argument that specifies attributes specific to this resource type. See section 6.4.3.5.4.1,"Type Specific Attributes".

Description

The **ExtendedMemory** macro evaluates to a buffer which contains a 64-bit memory resource descriptor, which describes a range of memory addresses. The format of the 64-bit memory resource descriptor can be found in "Extended Address Space Descriptor" (page 242). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.41 ExtendedSpace (Extended Address Space Resource Descriptor Macro)

Syntax

```
ExtendedSpace (ResourceType, ResourceUsage, Decode, IsMinFixed, IsMaxFixed,
     TypeSpecificFlags, AddressGranularity, AddressMinimum, AddressMaximum,
     AddressTranslation, RangeLength, TypeSpecificAttributes,
     DescriptorName)
```

Arguments

ResourceType evaluates to an 8-bit integer that specifies the type of this resource. Acceptable values are 0xC0 through 0xFF.

ResourceUsage specifies whether the Memory range is consumed by this device (**ResourceConsumer**) or passed on to child devices (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

Decode specifies whether or not the device decodes the Memory range using positive (**PosDecode**) or subtractive (**SubDecode**) decode. If nothing is specified, then PosDecode is assumed. The 1-bit field *DescriptorName*. _DEC is automatically created to refer to this portion of the resource descriptor, where '1' is SubDecode and '0' is PosDecode.

IsMinFixed specifies whether the minimum address of this Memory range is fixed (**MinFixed**) or can be changed (**MinNotFixed**). If nothing is specified, then MinNotFixed is assumed. The 1-bit field *DescriptorName*. _MIF is automatically created to refer to this portion of the resource descriptor, where '1' is MinFixed and '0' is MinNotFixed.

IsMaxFixed specifies whether the maximum address of this Memory range is fixed (**MaxFixed**) or can be changed (**MaxNotFixed**). If nothing is specified, then MaxNotFixed is assumed. The 1-bit field *DescriptorName.* _MAF is automatically created to refer to this portion of the resource descriptor, where '1' is MaxFixed and '0' is MaxNotFixed.

TypeSpecificFlags evaluates to an 8-bit integer. The flags are specific to the *ResourceType*.

AddressGranularity evaluates to a 64-bit integer that specifies the power-of-two boundary (- 1) on which the Memory range must be aligned. The 64-bit field *DescriptorName*. _GRA is automatically created to refer to this portion of the resource descriptor.

AddressMinimum evaluates to a 64-bit integer that specifies the lowest possible base address of the Memory range. The value must have '0' in all bits where the corresponding bit in *AddressGranularity* is '1'. For bridge devices which translate addresses, this is the address on the secondary bus. The 64-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 64-bit integer that specifies the highest possible base address of the Memory range. The value must have '0' in all bits where the corresponding bit in *AddressGranularity* is '1'. For bridge devices which translate addresses, this is the address on the secondary bus. The 64-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressTranslation evaluates to a 64-bit integer that specifies the offset to be added to a secondary bus I/O address which results in the corresponding primary bus I/O address. For all non-bridge devices or bridges which do not perform translation, this must be '0'. The 64-bit field *DescriptorName*._TRA is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 64-bit integer that specifies the total number of bytes decoded in the Memory range. The 64-bit field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor.

TypeSpecificAttributes is an optional argument that specifies attributes specific to this resource type. See section 6.4.3.5.4.1,"Type Specific Attributes".

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

Description

The **ExtendedSpace** macro evaluates to a buffer which contains a 64-bit Address Space resource descriptor, which describes a range of addresses. The format of the 64-bit AddressSpace descriptor can be found in "Extended Address Space Descriptor" (page 242). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.42 External (Declare External Objects)

Syntax

```
External (ObjectName, ObjectType, ReturnType, ParameterTypes)
```

Arguments

ObjectName is a NameString.

ObjectType is an optional *ObjectTypeKeyword* (e.g. **IntObj**, **PkgObj**, etc.). If not specified, "**UnknownObj**" type is assumed.

ReturnType is optional. If the specified *ObjectType* is **MethodObj**, then this specifies the type or types of object returned by the method. If the method does not return an object, then nothing is specified or **UnknownObj** is specified. To specify a single return type, simply use the *ObjectTypeKeyword*. To specify multiple possible return types, enclose the comma-separated *ObjectTypeKeyword*s with braces. For example: **`{IntObj, BuffObj}`**.

ParameterTypes is optional. If the specified *ObjectType* is **MethodObj**, this specifies both the number and type of the method parameters. It is a comma-separated, variable-length list of the expected object type or types for each of the method parameters, enclosed in braces. For each parameter, the parameter type consists of either an *ObjectTypeKeyword* or a comma-separated sub-list of *ObjectTypeKeywords* enclosed in braces. There can be no more than seven parameters in total.Description

The External directive informs the ASL compiler that the object is declared external to this table so that no errors will be generated for an undeclared object. The ASL compiler will create the external object at the specified place in the namespace (if a full path of the object is specified), or the object will be created at the current scope of the External term.

External is especially useful for use in secondary SSDTs, when the required scopes and objects are declared in the main DSDT.

Example

This example shows the use of **External** in conjunction with **Scope** within an SSDT:

```
DefinitionBlock ("ssdt.aml", "SSDT", 2, "X", "Y", 0x00000001)
{
    External (\_SB.PCI0, DeviceObj)

    Scope (\_SB.PCI0)
    {
    }
}
```

18.5.43 Fatal (Fatal Error Check)

Syntax

`Fatal (`*`Type, Code, Arg`***`)`**

Arguments

This operation is used to inform the OS that there has been an OEM-defined fatal error.

Description

In response, the OS must log the fatal event and perform a controlled OS shutdown in a timely fashion.

18.5.44 Field (Declare Field Objects)

Syntax

`Field (`*`RegionName, AccessType, LockRule, UpdateRule`***`) {`***`FieldUnitList`***`}`**

Arguments

RegionName is a namestring that refers to the host operation region.

AccessType defines the default access width of the field definition and is any one of the following: **AnyAcc**, **ByteAcc**, **WordAcc**, **DWordAcc**, or **QWordAcc**. In general, accesses within the parent object are performed naturally aligned. If desired, *AccessType* set to a value other than **AnyAcc** can be used to force minimum access width. Notice that the parent object must be able to accommodate the *AccessType* width. For example, an access type of **WordAcc** cannot read the last byte of an odd-length operation region. The exceptions to natural alignment are the access types used for a non-linear SMBus device. These will be discussed in detail below. Not all access types are meaningful for every type of operational region.

LockRule is a flag that indicates whether the Global Lock is to be used when accessing this field and is one of the following: **Lock** or **NoLock**. If *LockRule* is set to **Lock**, accesses to modify the component data objects will acquire and release the Global Lock. If both types of locking occur, the Global Lock is acquired after the parent object Mutex.

UpdateRule is used to specify how the unmodified bits of a field are treated and is any one of the following: **Preserve**, **WriteAsOnes**, or **WriteAsZeros**. For example, if a field defines a component data object of 4 bits in the middle of a **WordAcc** region, when those 4 bits are modified the *UpdateRule* specifies how the other 12 bits are treated.

FieldUnitList is a variable-length list of individual field unit definitions, separated by commas. Each entry in the field unit list is one of the following:

FieldUnitName, *BitLength*
Offset (*ByteOffset*)
AccessAs (*AccessType*, *AccessAttribute*)

FieldUnitName is the ACPI name for the field unit (1 to 4 characters), and *BitLength* is the length of the field unit in bits. **Offset** is used to specify the byte offset of the next defined field unit. This can be used instead of defining the bit lengths that need to be skipped. **AccessAs** is used to define the access type and attributes for the remaining field units within the list.

Description

Declares a series of named data objects whose data values are fields within a larger object. The fields are parts of the object named by *RegionName*, but their names appear in the same scope as the **Field** term.

For example, the field operator allows a larger operation region that represents a hardware register to be broken down into individual bit fields that can then be accessed by the bit field names. Extracting and combining the component field from its parent is done automatically when the field is accessed.

When reading from a ***FieldUnit***, returned values are normalized (shifted and masked to the proper length.) The data type of an individual FieldUnit can be either a **Buffer** or an **Integer**, depending on the bit length of the FieldUnit. If the FieldUnit is smaller than or equal to the size of an Integer (in bits), it will be treated as an Integer. If the FieldUnit is larger than the size of an Integer, it will be treated as a Buffer. The size of an Integer is indicated by the DSDT header's *Revision* field. A revision less than 2 indicates that the size of an Integer is 32 bits. A value greater than or equal to 2 signifies that the size of an Integer is 64 bits. For more information about data types and FieldUnit type conversion rules, see section 18.2.5.7, "Data Type Conversion Rules".

Accessing the contents of a field data object provides access to the corresponding field within the parent object. If the parent object supports Mutex synchronization, accesses to modify the component data objects will acquire and release ownership of the parent object around the modification.

The following table relates region types declared with an OperationRegion term to the different access types supported for each region.

Table 18-18 OperationRegion Region Types and Access Types

Region Type	Permitted Access Type(s)	Description
SystemMemory	**ByteAcc, WordAcc, DWordAcc, QWordAcc, or AnyAcc**	All access allowed
SystemIO	**ByteAcc, WordAcc, DWordAcc, QWordAcc, or AnyAcc**	All access allowed
PCI_Config	**ByteAcc, WordAcc, DWordAcc, QWordAcc, or AnyAcc**	All access allowed
EmbeddedControl	**ByteAcc**	Byte access only
SMBus	**BufferAcc**	Reads and writes to this operation region involve the use of a region specific data buffer. (See below.)
CMOS	**ByteAcc**	Byte access only
PciBarTarget	**ByteAcc, WordAcc, DWordAcc, QWordAcc, or AnyAcc**	All access allowed
IPMI	**BufferAcc**	Reads and writes to this operation region involve the use of a region specific data buffer. (See below.)

The named FieldUnit data objects are provided in the FieldList as a series of names and bit widths. Bits assigned no name (or NULL) are skipped. The ASL compiler supports the **Offset (**ByteOffset**)** macro within a FieldList to skip to the bit position of the supplied byte offset, and the **AccessAs** macro to change access within the field list.

SMBus and IPMI regions are inherently non-linear, where each offset within the respective address space represents a variable sized (0 to 32 bytes) field. Given this uniqueness, these operation regions include restrictions on their field definitions and require the use of a region-specific data buffer when initiating transactions. For more information on the SMBus data buffer format, see section 14, “ACPI System Management Bus Interface Specification,”. For more information on the IPMI data buffer format, see section 5.5.2.4.3, “Declaring IPMI Operation Regions”.

Example

```
OperationRegion (MIOC, PCI_Config, Zero, 0xFF)
Field (MIOC, AnyAcc, NoLock, Preserve)
{
    Offset  (0x58),
    HXGB,   32,
    HXGT,   32,
    GAPE,   8,
    MR0A,   4,
    MR0B,   4
}
```

18.5.45 FindSetLeftBit (Find First Set Left Bit)

Syntax

FindSetLeftBit (*Source, Result***)** => Integer

Arguments

Source is evaluated as an Integer.

Description

The one-based bit location of the first MSb (most significant set bit) is optionally stored into *Result*. The result of 0 means no bit was set, 1 means the left-most bit set is the first bit, 2 means the left-most bit set is the second bit, and so on.

18.5.46 FindSetRightBit (Find First Set Right Bit)

Syntax

FindSetRightBit (*Source, Result***)** => Integer

Arguments

Source is evaluated as an Integer.

Description

The one-based bit location of the most LSb (least significant set bit) is optionally stored in *Result.* The result of 0 means no bit was set, 32 means the first bit set is the thirty-second bit, 31 means the first bit set is the thirty-first bit, and so on.

18.5.47 FixedIO (Fixed IO Resource Descriptor Macro)

Syntax

FixedIO (*AddressBase, RangeLength, DescriptorName***)** => Buffer

Arguments

AddressBase evaluates to a 16-bit integer. It describes the starting address of the fixed I/O range. The field *DescriptorName*. _BAS is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to an 8-bit integer. It describes the length of the fixed I/O range. The field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor.

DescriptorName evaluates to a name string which refers to the entire resource descriptor.

Description

The **FixedIO** macro evaluates to a buffer which contains a fixed I/O resource descriptor. The format of the fixed I/O resource descriptor can be found in “Fixed Location I/O Port Descriptor ” (page 228). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.48 FromBCD (Convert BCD To Integer)

Syntax

FromBCD (*BCDValue, Result***)** => Integer

Arguments

BCDValue is evaluated as an Integer.

Description

The **FromBCD** operation is used to convert *BCDValue* to a numeric format and store the numeric value into *Result*.

18.5.49 Function (Declare Control Method)

Syntax

Function (*FunctionName, ReturnType, ParameterTypes***) {**TermList**}**

Arguments

ReturnType is optional and specifies the type(s) of the object(s) returned by the method. If the method does not return an object, then nothing is specified or **UnknownObj** is specified. To specify a single return type, simply use the *ObjectTypeKeyword* (e.g. **IntObj**, **PkgObj**, etc.). To specify multiple possible return types, enclose the comma-separated *ObjectTypeKeyword*s with braces. For example: **{IntObj, BuffObj}**.

ParameterTypes specifies both the number and type of the method parameters. It is a comma-separated, variable-length list of the expected object type or types for each of the method parameters, enclosed in braces. For each parameter, the parameter type consists of either an *ObjectTypeKeyword* or a comma-separated sub-list of *ObjectTypeKeywords* enclosed in braces. There can be no more than seven parameters in total.

Description

Function declares a named package containing a series of terms that collectively represent a control method. A control method is a procedure that can be invoked to perform computation. **Function** opens a name scope.

System software executes a control method by executing the terms in the package in order. For more information on method execution, see section 5.5.2, “Control Method Execution.”

The current namespace location used during name creation is adjusted to be the current location on the namespace tree. Any names created within this scope are “below” the name of this package. The current namespace location is assigned to the method package, and all namespace references that occur during control method execution for this package are relative to that location.

Functions are equivalent to a **Method** that specifies **NotSerialized**. As such, a function should not create any named objects, since a second thread that might re-enter the function will cause a fatal error if an attempt is made to create the same named object twice.

Compatibility Note: New for ACPI 3.0

Example

The following block of ASL sample code shows the use of **Function** for defining a control method:

```
Function (EXAM, IntObj, {StrObj, {IntObj, StrObj}})
{
    Name (Temp,"")
    Store (Arg0, Temp)            // could have used Arg1
    Return (SizeOf (Concatenate (Arg1, Temp)))
}
```

This declaration is equivalent to:

```
Method (EXAM, 2, NotSerialized, 0, IntObj, {StrObj, {IntObj, StrObj}})
{
…
}
```

18.5.50 If (Conditional Execution)

Syntax

If (*Predicate***) {**TermList**}**

Arguments

Predicate is evaluated as an Integer.

Description

If the *Predicate* is non-zero, the term list of the **If** term is executed.

Example

The following examples all check for bit 3 in **Local0** being set, and clear it if set.

```
// example 1

If (And (Local0, 4))
{
    XOr (Local0, 4, Local0)
}

// example 2

Store (4, Local2)
If (And (Local0, Local2))
{
    XOr (Local0, Local2, Local0)
}
```

18.5.51 Include (Include Additional ASL File)

Syntax

Include (*FilePathName***)**

Arguments

FilePathname is a StringData data type that contains the full OS file system path.

Description

Include another file that contains ASL terms to be inserted in the current file of ASL terms. The file must contain elements that are grammatically correct in the current scope.

Example

```
Include ("dataobj.asl")
```

18.5.52 Increment (Integer Increment)

Syntax

Increment (*Addend***)** => Integer

Arguments

Addend is evaluated as an Integer.

Description

Add one to the Addend and place the result back in Addend. Equivalent to **Add (***Addend*, 1, *Addend***)**. Overflow conditions are ignored and the result of an overflow is zero.

18.5.53 Index (Indexed Reference To Member Object)

Syntax

Index (*Source, Index, Destination***)** => ObjectReference

Arguments

Source is evaluated to a buffer, string, or package data type. *Index* is evaluated to an integer. The reference to the nth object (where n = *Index*) within *Source* is optionally stored as a reference into *Destination.*

Description

When *Source* evaluates to a Buffer, Index returns a reference to a Buffer Field containing the nth byte in the buffer. When *Source* evaluates to a String, Index returns a reference to a Buffer Field containing the nth character in the string. When *Source* evaluates to a Package, Index returns a reference to the nth object in the package.

18.5.53.1 Index with Packages

The following example ASL code shows a way to use the **Index** term to store into a local variable the sixth element of the first package of a set of nested packages:

```
Name (IO0D, Package () {
     Package () {
        0x01, 0x03F8, 0x03F8, 0x01, 0x08, 0x01, 0x25, 0xFF, 0xFE, 0x00, 0x00
     },
     Package () {
        0x01, 0x02F8, 0x02F8, 0x01, 0x08, 0x01, 0x25, 0xFF, 0xBE, 0x00, 0x00
     },
     Package () {
        0x01, 0x03E8, 0x03E8, 0x01, 0x08, 0x01, 0x25, 0xFF, 0xFA, 0x00, 0x00
     },
     Package () {
        x01, 0x02E8, 0x02E8, 0x01, 0x08, 0x01, 0x25, 0xFF, 0xBA, 0x00, 0x00
     },
     Package() {
        0x01, 0x0100, 0x03F8, 0x08, 0x08, 0x02, 0x25, 0x20, 0x7F, 0x00, 0x00
     }
})

// Get the 6th element of the first package

Store (DeRefOf (Index (DeRefOf (Index (IO0D, 0)), 5)), Local0)
```

Note: DeRefOf is necessary in the first operand of the **Store** operator in order to get the actual object, rather than just a reference to the object. If **DeRefOf** were not used, then Local0 would contain an object reference to the sixth element in the first package rather than the number 1.

18.5.53.2 Index with Buffers

The following example ASL code shows a way to store into the third byte of a buffer:

```
Name (BUFF, Buffer () {0x01, 0x02, 0x03, 0x04, 0x05})

// Store 0x55 into the third byte of the buffer

Store (0x55, Index (BUFF, 2))
```

The **Index** operator returns a reference to an 8-bit Buffer Field (similar to that created using **CreateByteField**).

If *Source* is evaluated to a buffer data type, the *ObjectReference* refers to the byte at *Index* within *Source*. If *Source* is evaluated to a buffer data type, a **Store** operation will only change the byte at *Index* within *Source*.

The following example ASL code shows the results of a series of **Store** operations:

```
Name (SRCB, Buffer () {0x10, 0x20, 0x30, 0x40})
Name (BUFF, Buffer () {0x1, 0x2, 0x3, 0x4})
```

The following will store 0x78 into the 3rd byte of the destination buffer:

```
Store (0x12345678, Index (BUFF, 2))
```

The following will store 0x10 into the 2nd byte of the destination buffer:

```
Store (SRCB, Index (BUFF, 1))
```

The following will store 0x41 (an ‘A’) into the 4th byte of the destination buffer:

```
Store (“ABCDEFGH”, Index (BUFF, 3))
```

Compatibility Note: First introduced in ACPI 2.0. In ACPI 1.0, the behavior of storing data larger than 8-bits into a buffer using Index was undefined.

18.5.53.3 Index with Strings

The following example ASL code shows a way to store into the 3rd character in a string:

```
Name (STR, “ABCDEFGHIJKL”)

// Store ‘H’ (0x48) into the third character to the string

Store (“H”, Index (STR, 2))
```

The **Index** operator returns a reference to an 8-bit Buffer Field (similar to that created using **CreateByteField**).

Compatibility Note: First introduced in ACPI 2.0.

18.5.54 IndexField (Declare Index/Data Fields)

Syntax

```
IndexField (IndexName, DataName, AccessType, LockRule, UpdateRule)
     {FieldUnitList}
```

Arguments

IndexName and *DataName* refer to field unit objects. *AccessType*, *LockRule*, *UpdateRule*, and *FieldList* are the same format as the **Field** term.

Description

Creates a series of named data objects whose data values are fields within a larger object accessed by an index/data-style reference to *IndexName* and *DataName*.

This encoding is used to define named data objects whose data values are fields within an index/data register pair. This provides a simple way to declare register variables that occur behind a typical index and data register pair.

Accessing the contents of an indexed field data object will automatically occur through the *DataName* object by using an *IndexName* object aligned on an *AccessType* boundary, with synchronization occurring on the operation region that contains the index data variable, and on the Global Lock if specified by *LockRule*.

The value written to the *IndexName* register is defined to be a byte offset that is aligned on an *AccessType* boundary. For example, if *AccessType* is **DWordAcc**, valid index values are 0, 4, 8, etc. This value is always a byte offset and is independent of the width or access type of the *DataName* register.

Example

The following is a block of ASL sample code using **IndexField***:*

Creates an index/data register in system I/O space made up of 8-bit registers.

- Creates a FET0 field within the indexed range.

```
Method (EX1) {
    // Define a 256-byte operational region in SystemIO space
    // and name it GIO0

    OperationRegion (GIO0, 1, 0x125, 0x100)

    // Create a field named Preserve structured as a sequence
    // of index and data bytes

    Field (GIO0, ByteAcc, NoLock, WriteAsZeros) {
        IDX0, 8,
        DAT0, 8,
            .
            .
            .
    }
    // Create an IndexField within IDX0 & DAT0 which has
    // FETs in the first two bits of indexed offset 0,
    // and another 2 FETs in the high bit on indexed
    // 2F and the low bit of indexed offset 30

    IndexField (IDX0, DAT0, ByteAcc, NoLock, Preserve) {
        FET0, 1,
        FET1, 1,
        Offset (0x2f),     // skip to byte offset 2f
        , 7,               // skip another 7 bits
        FET3, 1,
        FET4, 1
    }

    // Clear FET3 (index 2F, bit 7)

    Store (Zero, FET3)

} // End EX1
```

18.5.55 Interrupt (Interrupt Resource Descriptor Macro)

Syntax

```
Interrupt (ResourceUsage, EdgeLevel, ActiveLevel, Shared,
     ResourceSourceIndex, ResourceSource, DescriptorName) {InterruptList} =>
     Buffer
```

Arguments

ResourceUsage describes whether the device consumes the specified interrupt (**ResourceConsumer**) or produces it for use by a child device (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

EdgeLevel describes whether the interrupt is edge triggered (**Edge**) or level triggered (**Level**). The field *DescriptorName*. _HE is automatically created to refer to this portion of the resource descriptor, where ‘1’ is Edge and ‘0’ is Level.

ActiveLevel describes whether the interrupt is active-high (**ActiveHigh**) or active-low (**ActiveLow**). The field *DescriptorName*. _LL is automatically created to refer to this portion of the resource descriptor, where ‘1’ is ActiveHigh and ‘0’ is ActiveLow.

Shared describes whether the interrupt can be shared with other devices (**Shared**) or not (**Exclusive**). The field *DescriptorName*. _SHR is automatically created to refer to this portion of the resource descriptor, where ‘1’ is Shared and ‘0’ is Exclusive. If nothing is specified, then Exclusive is assumed.

ResourceSourceIndex evaluates to an integer between 0x00 and 0xFF and describes the resource source index. If it is not specified, then it is not generated. If this argument is specified, the *ResourceSource* argument must also be specified.

ResourceSource evaluates to a string which uniquely identifies the resource source. If it is not specified, it is not generated. If this argument is specified, but the *ResourceSourceIndex* argument is not specified, a zero value is assumed.

DescriptorName evaluates to a name string which refers to the entire resource descriptor.

InterruptList is a comma-delimited list on integers, at least one value is required. Each integer represents a 32-bit interrupt number. At least one interrupt must be defined, and there may be no duplicates in the list. The field "*DescriptorName*. _INT" is automatically created to refer to this portion of the resource descriptor.

Description

The **Interrupt** macro evaluates to a buffer that contains an interrupt resource descriptor. The format of the interrupt resource descriptor can be found in "Extended Interrupt Descriptor " (page 249). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.56 IO (IO Resource Descriptor Macro)

Syntax

```
IO (Decode, AddressMin, AddressMax, AddressAlignment, RangeLength,
     DescriptorName) => Buffer
```

Argument

Decode describes whether the I/O range uses 10-bit decode (**Decode10**) or 16-bit decode (**Decode16**). The field *DescriptorName*. _DEC is automatically created to refer to this portion of the resource descriptor, where '1' is **Decode16** and '0' is **Decode10**.

AddressMin evaluates to a 16-bit integer that specifies the minimum acceptable starting address for the I/O range. It must be an even multiple of *AddressAlignment*. The field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMax evaluates to a 16-bit integer that specifies the maximum acceptable starting address for the I/O range. It must be an even multiple of *AddressAlignment*. The field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressAlignment evaluates to an 8-bit integer that specifies the alignment granularity for the I/O address assigned. The field *DescriptorName*. _ALN is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to an 8-bit integer that specifies the number of bytes in the I/O range. The field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

Description

The **IO** macro evaluates to a buffer which contains an IO resource descriptor. The format of the IO descriptor can be found in "I/O Port Descriptor" (page 227). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.57 IRQ (Interrupt Resource Descriptor Macro)

Syntax

```
IRQ (EdgeLevel, ActiveLevel, Shared, DescriptorName) {InterruptList} =>
     Buffer
```

Arguments

EdgeLevel describes whether the interrupt is edge triggered (**Edge**) or level triggered (**Level**). The field *DescriptorName*. _HE is automatically created to refer to this portion of the resource descriptor, where '1' is **Edge** and **ActiveHigh** and '0' is **Level** and **ActiveLow**.

ActiveLevel describes whether the interrupt is active-high (**ActiveHigh**) or active-low (**ActiveLow**). The field *DescriptorName*. _LL is automatically created to refer to this portion of the resource descriptor, where '1' is **Edge** and **ActiveHigh** and '0' is **Level** and **ActiveLow**.

Shared describes whether the interrupt can be shared with other devices (**Shared**) or not (**Exclusive**). The field *DescriptorName*. _SHR is automatically created to refer to this portion of the resource descriptor, where '1' is **Shared** and '0' is **Exclusive**. If nothing is specified, then **Exclusive** is assumed.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

InterruptList is a comma-delimited list of integers in the range 0 through 15, at least one value is required. There may be no duplicates in the list.

Description

The **IRQ** macro evaluates to a buffer that contains an IRQ resource descriptor. The format of the IRQ descriptor can be found in "IRQ Descriptor" (page 225). The macro produces the three-byte form of the descriptor. The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.58 IRQNoFlags (Interrupt Resource Descriptor Macro)

Syntax

```
IRQNoFlags (DescriptorName) {InterruptList} => Buffer
```

Arguments

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer.

InterruptList is a comma-delimited list of integers in the range 0 through 15, at least one value is required. There may be no duplicates in the list Description

The **IRQNoFlags** macro evaluates to a buffer which contains an active-high, edge-triggered IRQ resource descriptor. The format of the IRQ descriptor can be found in IRQ Descriptor (page 225). The macro produces the two-byte form of the descriptor. The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.59 LAnd (Logical And)

Syntax

```
LAnd (Source1, Source2) => Boolean
```

Arguments

Source1 and *source2* are evaluated as integers.

Description

If both values are non-zero, True is returned: otherwise, False is returned.

18.5.60 LEqual (Logical Equal)

Syntax

```
LEqual (Source1, Source2) => Boolean
```

Arguments

Source1 and *Source2* must each evaluate to an integer, a string, or a buffer. The data type of *Source1* dictates the required type of *Source2*. *Source2* is implicitly converted if necessary to match the type of *Source1*.

Description

If the values are equal, True is returned; otherwise, False is returned. For integers, a numeric compare is performed. For strings and buffers, True is returned only if both lengths are the same and the result of a byte-wise compare indicates exact equality.

18.5.61 LGreater (Logical Greater)

Syntax

```
LGreater (Source1, Source2) => Boolean
```

Arguments

Source1 and *Source2* must each evaluate to an integer, a string, or a buffer. The data type of *Source1* dictates the required type of *Source2*. *Source2* is implicitly converted if necessary to match the type of *Source1*.

Description

If *Source1* is greater than *Source2*, True is returned; otherwise, False is returned. For integers, a numeric comparison is performed. For strings and buffers, a lexicographic comparison is performed. **True** is returned if a byte-wise (unsigned) compare discovers at least one byte in *Source1* that is numerically greater than the corresponding byte in *Source2*. **False** is returned if at least one byte in *Source1* is numerically less than the corresponding byte in *Source2*. In the case of byte-wise equality, **True** is returned if the length of *Source1* is greater than *Source2*, **False** is returned if the length of *Source1* is less than or equal to *Source2*.

18.5.62 LGreaterEqual (Logical Greater Than Or Equal)

Syntax

```
LGreaterEqual (Source1, Source2) => Boolean
```

Arguments

Source1 and *Source2* must each evaluate to an integer, a string, or a buffer. The data type of *Source1* dictates the required type of *Source2*. *Source2* is implicitly converted if necessary to match the type of *Source1*.

Description

If *Source1* is greater than or equal to *Source2*, True is returned; otherwise, False is returned. Equivalent to `LNot(LLess())`. See the description of the LLess operator.

18.5.63 LLess (Logical Less)

Syntax

```
LLess (Source1, Source2) => Boolean
```

Arguments

Source1 and *Source2* must each evaluate to an integer, a string, or a buffer. The data type of *Source1* dictates the required type of *Source2*. *Source2* is implicitly converted if necessary to match the type of *Source1*.

Description

If *Source1* is less than *Source2*, True is returned; otherwise, False is returned. For integers, a numeric comparison is performed. For strings and buffers, a lexicographic comparison is performed. **True** is returned if a byte-wise (unsigned) compare discovers at least one byte in *Source1* that is numerically less than the corresponding byte in *Source2*. **False** is returned if at least one byte in *Source1* is numerically greater than the corresponding byte in *Source2*. In the case of byte-wise equality, **True** is returned if the length of *Source1* is less than *Source2*, **False** is returned if the length of *Source1* is greater than or equal to *Source2*.

18.5.64 LLessEqual (Logical Less Than Or Equal)

Syntax

```
LLessEqual (Source1, Source2) => Boolean
```

Arguments

Source1 and *Source2* must each evaluate to an integer, a string, or a buffer. The data type of *Source1* dictates the required type of *Source2*. *Source2* is implicitly converted if necessary to match the type of *Source1*.

Description

If *Source1* is less than or equal to *Source2*, True is returned; otherwise False is returned. Equivalent to `LNot(LGreater()).` See the description of the LGreater operator.

18.5.65 LNot (Logical Not)

Syntax

```
LNot (Source) => Boolean
```

Arguments

Source is evaluated as an integer.

Description

If the value is zero True is returned; otherwise, False is returned.

18.5.66 LNotEqual (Logical Not Equal))

Syntax

```
LNotEqual (Source1, Source2) => Boolean
```

Arguments

Source1 and *Source2* must each evaluate to an integer, a string, or a buffer. The data type of *Source1* dictates the required type of *Source2*. *Source2* is implicitly converted if necessary to match the type of *Source1*.

Description

If *Source1* is not equal to *Source2*, True is returned; otherwise False is returned. Equivalent to `LNot(LEqual())`.See the description of the LEqual operator.

18.5.67 Load (Load Definition Block)

Syntax

```
Load (Object, DDBHandle)
```

Arguments

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 an ACPI 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 Definition Block that can be used to unload the Definition Block at a future time via the Unload operator.

Description

Performs a run-time load of a Definition Block. Any table referenced by **Load** must be in memory marked as AddressRangeReserved or AddressRangeNVS.

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 root of the 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.

18.5.68 LoadTable (Load Definition Block From XSDT)

Syntax

```
LoadTable (SignatureString, OEMIDString, OEMTableIDString, RootPathString,
     ParameterPathString, ParameterData) => DDBHandle
```

Arguments

The XSDT is searched for a table where the Signature field matches *SignatureString*, the OEM ID field matches *OEMIDString*, and the OEM Table ID matches *OEMTableIDString*. All comparisons are case sensitive. If the *SignatureString* is greater than four characters, the *OEMIDString* is greater than six characters, or the *OEMTableID* is greater than eight characters, a run-time error is generated. 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 *RootPathString* specifies the root of the Definition Block. It is evaluated using normal scoping rules, assuming that the scope of the **LoadTable** instruction is the current scope. The new Definition Block can override this by specifying absolute names or by adjusting the namespace location using the **Scope** operator. If *RootPathString* is not specified, "\" is assumed

If *ParameterPathString* and *ParameterData* are specified, the data object specified by *ParameterData* is stored into the object specified by *ParameterPathString* after the table has been added into the namespace. If the first character of *ParameterPathString* is a backslash ('\') or caret ('^') character, then the path of the object is *ParameterPathString*. Otherwise, it is *RootPathString.ParameterPathString*. If the specified object does not exist, a run-time error is generated.

The handle of the loaded table is returned. If no table matches the specified signature, then 0 is returned.

Description

Performs a run-time load of a Definition Block from the XSDT. Any table referenced by **LoadTable** must be in memory marked by AddressRangeReserved or AddressRangeNVS. Note: OSPM loads the DSDT and all SSDTs during initialization. As such, Definition Blocks to be conditionally loaded via **LoadTable** must contain signatures other than "SSDT".

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.

Example

```
Store (LoadTable ("OEM1", "MYOEM", "TABLE1", "\\_SB.PCI0","MYD",
                  Package () {0,"\\_SB.PCI0"}), Local0)
```

This operation would search through the RSDT or XSDT for a table with the signature "OEM1," the OEM ID of "MYOEM," and the table ID of "TABLE1." If not found, it would store **Zero** in Local0. Otherwise, it will store a package containing 0 and "_SB.PCI0" into the variable at _SB.PCI0.MYD.

18.5.69 Local*x* (Method Local Data Objects)

Syntax

Local0 | **Local1** | **Local2** | **Local3** | **Local4** | **Local5** | **Local6** | **Local7**

Description

Up to 8 local objects can be referenced in a control method. On entry to a control method, these objects are uninitialized and cannot be used until some value or reference is stored into the object. Once initialized, these objects are preserved in the scope of execution for that control method.

18.5.70 LOr (Logical Or)

Syntax

LOr (*Source1, Source2***)** => Boolean

Arguments

Source1 and *Source2* are evaluated as integers.

Description

If either value is non-zero, True is returned; otherwise, False is returned.

18.5.71 Match (Find Object Match)

Syntax

Match (*SearchPackage, Op1, MatchObject1, Op2, MatchObject2, StartIndex***)** =>
Ones | Integer

Arguments

SearchPackage is evaluated to a package object and is treated as a one-dimension array. Each package element must evaluate to either an integer, a string, or a buffer. Uninitialized package elements and elements that do not evaluate to integers, strings, or buffers are ignored. *Op1* and *Op2* are match operators. *MatchObject1* and *MatchObject2* are the objects to be matched and must each evaluate to either an integer, a string, or a buffer. *StartIndex* is the starting index within the *SearchPackage*.

Description

A comparison is performed for each element of the package, starting with the index value indicated by *StartIndex* (0 is the first element). If the element of *SearchPackage* being compared against is called *P[i]***,** then the comparison is:

If (*P[i] Op1 MatchObject1***) and (***P[i] Op2 MatchObject2***) then Match =>** i is returned.

If the comparison succeeds, the index of the element that succeeded is returned; otherwise, the constant object **Ones** is returned. The data type of the MatchObject dictates the required type of the package element. If necessary, the package element is implicitly converted to match the type of the MatchObject. If the implicit conversion fails for any reason, the package element is ignored (no match.)

Op1 and *Op2* have the values and meanings listed in the Table 18-19.

Table 18-19 Match Term Operator Meanings

Operator	Encoding	Macro
TRUE – A don't care, always returns TRUE	0	**MTR**
EQ – Returns TRUE if P[i] == MatchObject	1	**MEQ**
LE – Returns TRUE if P[i] <= MatchObject	2	**MLE**
LT – Returns TRUE if P[i] < MatchObject	3	**MLT**
GE – Returns TRUE if P[i] >= MatchObject	4	**MGE**
GT – Returns TRUE if P[i] > MatchObject	5	**MGT**

Example

Following are some example uses of **Match**:

```
Name (P1,
Package () {1981, 1983, 1985, 1987, 1989, 1990, 1991, 1993, 1995, 1997, 1999, 2001}
)

// match 1993 == P1[i]
Match (P1, MEQ, 1993, MTR, 0, 0)     // -> 7, since P1[7] == 1993

// match 1984 == P1[i]
Match (P1, MEQ, 1984, MTR, 0, 0)     // -> ONES (not found)

// match P1[i] > 1984 and P1[i] <= 2000
Match (P1, MGT, 1984, MLE, 2000, 0)  // -> 2, since P1[2]>1984 and P1[2]<=2000

// match P1[i] > 1984 and P1[i] <= 2000, starting with 3rd element
Match (P1, MGT, 1984, MLE, 2000, 3)  // -> 3, first match at or past Start
```

18.5.72 Memory24 (Memory Resource Descriptor Macro)

Syntax

Memory24 (*ReadAndWrite*, *AddressMinimum*, *AddressMaximum*, *AddressAlignment*, *RangeLength*, *DescriptorName***)**

Arguments

ReadAndWrite specifies whether or not the memory region is read-only (**ReadOnly**) or read/write (**ReadWrite**). If nothing is specified, then ReadWrite is assumed. The 1-bit field *DescriptorName*._RW is automatically created to refer to this portion of the resource descriptor, where '1' is ReadWrite and '0' is ReadOnly.

AddressMinimum evaluates to a 16-bit integer that specifies bits [8:23] of the lowest possible base address of the memory range. All other bits are assumed to be zero. The value must be an even multiple of *AddressAlignment*. The 16-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 16-bit integer that specifies bits [8:23] of the highest possible base address of the memory range. All other bits are assumed to be zero. The value must be an even multiple of *AddressAlignment*. The 16-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressAlignment evaluates to a 16-bit integer that specifies bits [0:15] of the required alignment for the memory range. All other bits are assumed to be zero. The address selected must be an even multiple of this value. The 16-bit field *DescriptorName*. _ALN is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 16-bit integer that specifies the total number of bytes decoded in the memory range. The 16-bit field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor. The range length provides the length of the memory range in 256 byte blocks.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

Description

The **Memory24** macro evaluates to a buffer which contains an 24-bit memory descriptor. The format of the 24-bit memory descriptor can be found in "24-Bit Memory Range Descriptor " (page 231). The macro is designed to be used inside of a ResourceTemplate (page 544).

NOTE: The use of **Memory24** is deprecated and should not be used in new designs.

18.5.73 Memory32 (Memory Resource Descriptor Macro)

Syntax

```
Memory32 (ReadAndWrite, AddressMinimum, AddressMaximum, AddressAlignment,
     RangeLength, DescriptorName)
```

Arguments

ReadAndWrite specifies whether or not the memory region is read-only (**ReadOnly**) or read/write (**ReadWrite**). If nothing is specified, then ReadWrite is assumed. The 1-bit field *DescriptorName*._RW is automatically created to refer to this portion of the resource descriptor, where '1' is ReadWrite and '0' is ReadOnly.

AddressMinimum evaluates to a 32-bit integer that specifies the lowest possible base address of the memory range. The value must be an even multiple of *AddressAlignment*. The 32-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 32-bit integer that specifies the highest possible base address of the memory range. The value must be an even multiple of *AddressAlignment*. The 32-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressAlignment evaluates to a 32-bit integer that specifies the required alignment for the memory range. The address selected must be an even multiple of this value. The 32-bit field *DescriptorName*. _ALN is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 32-bit integer that specifies the total number of bytes decoded in the memory range. The 32-bit field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor. The range length provides the length of the memory range in 1 byte blocks.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

Description

The **Memory32** macro evaluates to a buffer which contains a 32-bit memory descriptor, which describes a memory range with a minimum, a maximum and an alignment. The format of the 32-bit memory descriptor can be found in "32-Bit Memory Range Descriptor " (page 232). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.74 Memory32Fixed (Memory Resource Descriptor Macro)

Syntax

```
Memory32Fixed (ReadAndWrite, AddressBase, RangeLength, DescriptorName)
```

Arguments

ReadAndWrite specifies whether or not the memory region is read-only (**ReadOnly**) or read/write (**ReadWrite**). If nothing is specified, then ReadWrite is assumed. The 1-bit field *DescriptorName*._RW is automatically created to refer to this portion of the resource descriptor, where '1' is ReadWrite and '0' is ReadOnly.

AddressBase evaluates to a 32-bit integer that specifies the base address of the memory range. The 32-bit field *DescriptorName*. _BAS is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 32-bit integer that specifies the total number of bytes decoded in the memory range. The 32-bit field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

Description

The **Memory32Fixed** macro evaluates to a buffer which contains a 32-bit memory descriptor, which describes a fixed range of memory addresses. The format of the fixed 32-bit memory descriptor can be found in 32-Bit Fixed Memory Range Descriptor (page 233). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.75 Method (Declare Control Method)

Syntax

```
Method (MethodName, NumArgs, SerializeRule, SyncLevel, ReturnType,
     ParameterTypes) {TermList}
```

Arguments

MethodName is evaluated as a Namestring data type.

NumArgs is optional and is the required number of arguments to be passed to the method, evaluated as an Integer data type. If not specified, the default value is zero arguments. Up to 7 arguments may be passed to a method. These arguments may be referenced from within the method as **Arg0** through **Arg6**.

SerializeRule is optional and is a flag that defines whether the method is serialized or not and is one of the following: **Serialized** or **NotSerialized**. A method that is serialized cannot be reentered by additional threads. If not specified, the default is **NotSerialized**.

SyncLevel is optional and specifies the synchronization level for the method (0 – 15). If not specified, the default sync level is zero.

ReturnType is optional and specifies the type(s) of the object(s) returned by the method. If the method does not return an object, then nothing is specified or **UnknownObj** is specified. To specify a single return type, simply use the *ObjectTypeKeyword* (e.g. **IntObj**, **PkgObj**, etc.). To specify multiple possible return types, enclose the comma-separated *ObjectTypeKeyword*s with braces. For example: **{IntObj, BuffObj}**.

ParameterTypes is optional and specifies the type of the method parameters. It is a comma-separated, variable-length list of the expected object type or types for each of the method parameters, enclosed in braces. For each parameter, the parameter type consists of either an *ObjectTypeKeyword* or a comma-separated sub-list of *ObjectTypeKeywords* enclosed in braces. If *ParameterTypes* is specified, the number of parameters must match *NumArgs*.

TermList is a variable-length list of executable ASL statements representing the body of the control method.

Description

Creates a new control method of name *MethodName.* This is a named package containing a series of object references that collectively represent a control method, which is a procedure that can be invoked to perform computation. **Method** opens a name scope.

System software executes a control method by referencing the objects in the package in order. For more information on method execution, see section 5.5.2, “Control Method Execution.”

The current namespace location used during name creation is adjusted to be the current location on the namespace tree. Any names created within this scope are “below” the name of this package. The current namespace location is assigned to the method package, and all namespace references that occur during control method execution for this package are relative to that location.

If a method is declared as **Serialized**, an implicit mutex associated with the method object is acquired at the specified *SyncLevel*. If no *SyncLevel* is specified, *SyncLevel 0* is assumed. The serialize rule can be used to prevent reentering of a method. This is especially useful if the method creates namespace objects. Without the serialize rule, the reentering of a method will fail when it attempts to create the same namespace object.

There are eight local variables automatically available for each method, referenced as **Local0** through **Local7**. These locals may be used to store any type of ASL object.

Also notice that all namespace objects created by a method have temporary lifetime. When method execution exits, the created objects will be destroyed.

Examples

The following block of ASL sample code shows a use of **Method** for defining a control method that turns on a power resource.

```
Method (_ON) {
    Store (One, GIO.IDEP)         // assert power
    Sleep (10)                    // wait 10ms
    Store (One, GIO.IDER)         // de-assert reset#
    Stall (10)                    // wait 10us
    Store (Zero, GIO.IDEI)        // de-assert isolation
}
```

This method is an implementation of _SRS (Set Resources). It shows the use of a method argument and two method locals.

```
Method (_SRS, 1, NotSerialized)
{
    CreateWordField (Arg0, One, IRQW)
    Store (\_SB.PCI0.PID1.IENA, Local1)
    Or (IRQW, Local1, Local1)
    Store (Local1, \_SB.PCI0.PID1.IENA)
    FindSetRightBit (IRQW, Local0)
    If (Local0)
    {
        Decrement (Local0)
        Store (Local0, \_SB.PCI0.PID1.IN01)
    }
}
```

18.5.76 Mid (Extract Portion of Buffer or String)

Syntax

```
Mid (Source, Index, Length, Result) => Buffer or String
```

Arguments

Source is evaluated as either a Buffer or String. *Index* and *Length* are evaluated as Integers.

Description

If *Source* is a buffer, then *Length* bytes, starting with the *Index*th byte (zero-based) are optionally copied into *Result*. If *Index* is greater than or equal to the length of the buffer, then the result is an empty buffer. Otherwise, if *Index* + *Length* is greater than or equal to the length of the buffer, then only bytes up to and including the last byte are included in the result.

If *Source* is a string, then *Length* characters, starting with the *Index*th character (zero-based) are optionally copied into *Result*. If *Index* is greater than or equal to the length of the buffer, then the result is an empty string. Otherwise, if *Index* + *Length* is greater than or equal to the length of the string, then only bytes up to an including the last character are included in the result.

18.5.77 Mod (Integer Modulo)

Syntax

```
Mod (Dividend, Divisor, Result) => Integer
```

Arguments

Dividend and *Divisor* are evaluated as Integers.

Description

The *Dividend* is divided by *Divisor*, and then the resulting remainder is optionally stored into *Result*. If *Divisor* evaluates to zero, a fatal exception is generated.

18.5.78 Multiply (Integer Multiply)

Syntax

```
Multiply (Multiplicand, Multiplier, Result) => Integer
```

Arguments

Multiplicand and *Multiplier* are evaluated as Integers.

Description

The *Multiplicand* is multiplied by *Multiplier* and the result is optionally stored into *Result.* Overflow conditions are ignored and results are undefined.

18.5.79 Mutex (Declare Synchronization/Mutex Object)

Syntax

Mutex (*MutexName, SyncLevel***)**

Arguments

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

Description

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. The *current sync level* is maintained internally for a thread, and represents the greatest *SyncLevel* among mutex objects that are currently acquired by the thread. The *SyncLevel* of a thread before acquiring any mutexes is zero. The *SyncLevel* of the Global Lock (_GL) is zero.

All **Acquire** terms must refer to a synchronization object with a *SyncLevel* that is equal or greater than the current level, and all **Release** terms must refer to a synchronization object with a *SyncLevel* that is equal 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 (can be acquired multiple times by the same thread).

18.5.80 Name (Declare Named Object)

Syntax

Name (*ObjectName, Object***)**

Arguments

Creates a new object named *ObjectName.* Attaches *Object* to *ObjectName* in the Global ACPI namespace.

Description

Creates *ObjectName* in the namespace, which references the *Object.*

Example

The following example creates the name PTTX in the root of the namespace that references a package.

```
Name (\PTTX,                  // Port to Port Translate Table
    Package () {Package () {0x43, 0x59}, Package) {0x90, 0xFF}}
)
```

The following example creates the name CNT in the root of the namespace that references an integer data object with the value 5.

```
Name (\CNT, 5)
```

18.5.81 NAnd (Integer Bitwise Nand)

Syntax

```
NAnd (Source1, Source2, Result) => Integer
```

Arguments

Source1 and *Source2* are evaluated as Integers.

Description

A bitwise **NAND** is performed and the result is optionally stored in *Result.*

18.5.82 NoOp Code (No Operation)

Syntax

```
NoOp
```

Description

This operation has no effect.

18.5.83 NOr (Integer Bitwise Nor)

Syntax

```
NOr (Source1, Source2, Result) => Integer
```

Arguments

Source1 and *Source2* are evaluated as Integers.

Description

A bitwise **NOR** is performed and the result is optionally stored in *Result.*

18.5.84 Not (Integer Bitwise Not)

Syntax

```
Not (Source, Result) => Integer
```

Arguments

Source is evaluated as an integer data type.

Description

A bitwise **NOT** is performed and the result is optionally stored in *Result*.

18.5.85 Notify (Notify Object of Event)

Syntax

```
Notify (Object, NotificationValue)
```

Arguments

Notifies the OS that the *NotificationValue* for the *Object* has occurred. *Object* must be a reference to a device, processor, or thermal zone object.

Description

Object type determines the notification values. For example, the notification values for a thermal zone object are different from the notification values used for a device object. Undefined notification values are treated as reserved and are ignored by the OS.

For lists of defined Notification values, see section 5.6.5, “Device Object Notifications.”

18.5.86 ObjectType (Get Object Type)

Syntax

```
ObjectType (Object) => Integer
```

Arguments

Object is any valid object.

Description

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 18-20. 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 predefined scope names (in other words, **_SB**, **_GPE**, etc.), the type value 0 (**Uninitialized**) is returned.

Table 18-20 Values Returned By the ObjectType Operator

Value	Object
0	Uninitialized
1	Integer
2	String
3	Buffer
4	Package
5	Field Unit
6	Device
7	Event
8	Method
9	Mutex
10	Operation Region
11	Power Resource

Value	Object
12	Processor
13	Thermal Zone
14	Buffer Field
15	DDB Handle
16	Debug Object
>16	*Reserved*

18.5.87 One (Constant One Object)

Syntax

One

Description

The constant **One** object is an object of type Integer that will always read the LSB as set and all other bits as clear (that is, the value of 1). Writes to this object are not allowed.

18.5.88 Ones (Constant Ones Object)

Syntax

Ones

Description

The constant **Ones** object is an object of type Integer that will always read as all bits set. Writes to this object are not allowed.

18.5.89 OperationRegion (Declare Operation Region)

Syntax

OperationRegion (*RegionName, RegionSpace, Offset, Length***)**

Arguments

Declares an operation region named *RegionName*. *Offset* is the offset within the selected *RegionSpace* at which the region starts (byte-granular), and *Length* is the length of the region in bytes.

Description

An Operation Region is a type of data object where read or write operations to the data object are performed in some hardware space. For example, the Definition Block can define an Operation Region within a bus, or system I/O space. Any reads or writes to the named object will result in accesses to the I/O space.

Operation regions are regions in some space that contain hardware registers for *exclusive* use by ACPI control methods. In general, no hardware register (at least byte-granular) within the operation region accessed by an ACPI control method can be shared with any accesses from any other source, with the exception of using the Global Lock to share a region with the firmware. The entire Operation Region can be allocated for exclusive use to the ACPI subsystem in the host OS.

Operation Regions that are defined within the scope of a method are the exception to this rule. These Operation Regions are known as “Dynamic” since the OS has no idea that they exist or what registers they use until the control method is executed. Using a Dynamic SystemIO or SystemMemory Operation Region is not recommended since the OS cannot *guarantee* exclusive access. All other types of Operation Regions may be Dynamic.

Operation Regions define the overall base address and length of a hardware region, but they cannot be accessed directly by AML code. A **Field** object containing one or more **FieldUnits** is used to overlay the Operation Region in order to access individual areas of the Region. An individual FieldUnit within an Operation Region may be as small as one bit, or as large as the length of the entire Region. FieldUnit values are normalized (shifted and masked to the proper length.) The data type of a FieldUnit can be either a **Buffer** or an **Integer**, depending on the bit length of the FieldUnit. If the FieldUnit is smaller than or equal to the size of an Integer (in bits), it will be treated as an Integer. If the FieldUnit is larger than the size of an Integer, it will be treated as a Buffer. The size of an Integer is indicated by the DSDT header’s *Revision* field. A revision less than 2 indicates that the size of an Integer is 32 bits. A value greater than or equal to 2 signifies that the size of an Integer is 64 bits. For more information about data types and FieldUnit type conversion rules, see section 18.2.5.7, “Data Type Conversion Rules”.

An Operation Region object implicitly supports Mutex synchronization. Updates to the object, or a **Field** data object for the region**,** will automatically synchronize on the Operation Region object; however, a control method may also explicitly synchronize to a region to prevent other accesses to the region (from other control methods). Notice that according to the control method execution model, control method execution is non-preemptive. Because of this, explicit synchronization to an Operation Region needs to be done only in cases where a control method blocks or yields execution and where the type of register usage requires such synchronization.

There are eight predefined Operation Region types specified in ACPI:

Name (*RegionSpace* Keyword)	**Value**
SystemMemory	0
SystemIO	1
PCI_Config	2
EmbeddedControl	3
SMBus	4
CMOS	5
PCIBARTarget	6
IPMI	7
Reserved	0x08-0x7F

In addition, OEMs may define Operation Regions types **0x80** to **0xFF**.

Example

The following example ASL code shows the use of **OperationRegion** combined with **Field** to describe IDE 0 and 1 controlled through general I/O space, using one FET.

```
OperationRegion (GIO, SystemIO, 0x125, 0x1)
Field (GIO, ByteAcc, NoLock, Preserve) {
    IDEI,   1,      // IDEISO_EN    - isolation buffer
    IDEP,   1,      // IDE_PWR_EN   - power
    IDER,   1       // IDERST#_EN   - reset#
}
```

18.5.90 Or (Integer Bitwise Or)

Syntax

```
Or (Source1, Source2, Result) => Integer
```

Arguments

Source1 and *Source2* are evaluated as Integers.

Description

A bitwise **OR** is performed and the result is optionally stored in *Result.*

18.5.91 Package (Declare Package Object)

Syntax

```
Package (NumElements) {PackageList} => Package
```

Arguments

NumElements is evaluated as an integer data type. *PackageList* is an initializer list of objects.

Description

Declares an unnamed aggregation of data items, constants, and/or references to control methods. The size of the package is *NumElements*. *PackageList* contains the list data items, constants, and/or control method references used to initialize the package.

If *NumElements* is absent, it is set to match the number of elements in the PackageList. If *NumElements* is present and greater than the number of elements in the PackageList, the default entry of type Uninitialized (see **ObjectType**) is used to initialize the package elements beyond those initialized from the PackageList.

Evaluating an undefined element will yield an error, but elements can be assigned values to make them defined. It is an error for *NumElements* to be less than the number of elements in the PackageList. It is an error for *NumElements* to exceed 255.

There are two types of package elements in the PackageList: data objects and references to control methods.

Examples

Example 1:

```
Package () {
    3,
    9,
    “ACPI 1.0 COMPLIANT”,
    Package () {
        “CheckSum=>”,
        Package () {7, 9}
    },
    0
}
```

Example 2: This example defines and initializes a two-dimensional array.

```
Package () {
    Package () {11, 12, 13},
    Package () {21, 22, 23}
}
```

Example 3: This encoding allocates space for ten things to be defined later (see the **Name** and **Index** term definitions).

```
Package (10) {}
```

Note: The ability to create variable-sized packages was first introduced in ACPI 2.0. ACPI 1.0 only allowed fixed-size packages with up to 255 elements.

18.5.92 PowerResource (Declare Power Resource)

Syntax

PowerResource (*ResourceName, SystemLevel, ResourceOrder***) {**ObjectList**}**

Arguments

Declares a power resource named *ResourceName*. **PowerResource** opens a name scope.

Description

For a definition of the **PowerResource** term, see section 7.1, “Declaring a Power Resource Object.”

18.5.93 Processor (Declare Processor)

Syntax

Processor (*ProcessorName, ProcessorID, PBlockAddress, PblockLength***)**
{ObjectList**}**

Arguments

Declares a named processor object named *ProcessorName*. **Processor** opens a name scope. Each processor is required to have a unique *ProcessorID* value that is unique from any other *ProcessorID* value.

For each processor in the system, the ACPI BIOS declares one processor object in the namespace anywhere within the _SB scope. For compatibility with operating systems implementing ACPI 1.0, the processor object may also be declared under the _PR scope. An ACPI-compatible namespace may define Processor objects in either the _SB or _PR scope but not both.

PBlockAddress provides the system I/O address for the processors register block. Each processor can supply a different such address. *PBlockLength* is the length of the processor register block, in bytes and is either 0 (for no P_BLK) or 6. With one exception, all processors are required to have the same *PBlockLength*. The exception is that the boot processor can have a non-zero *PBlockLength* when all other processors have a zero *PBlockLength*. It is valid for every processor to have a PBlockLength of 0.

Description

The following block of ASL sample code shows a use of the **Processor** term.

```
Processor (
    \_PR.CPU0,      // Namespace name
    1,
    0x120,          // PBlk system IO address
    6               // PBlkLen
) {ObjectList}
```

The ObjectList is an optional list that may contain an arbitrary number of ASL Objects. Processor-specific objects that may be included in the ObjectList include _PTC, _CST, _PCT, _PSS, _PPC, _PSD, _TSD, _CSD, _PDC, _TPC, _TSS, and _OSC. These processor-specific objects can only be specified when the processor object is declared within the _SB scope. For a full definition of these objects, see section 8, “Processor Configuration and Control.”

18.5.94 QWordIO (QWord IO Resource Descriptor Macro)

Syntax

```
QWordIO (ResourceUsage, IsMinFixed, IsMaxFixed, Decode, ISARanges,
     AddressGranularity, AddressMinimum, AddressMaximum, AddressTranslation,
     RangeLength, ResourceSourceIndex, ResourceSource, DescriptorName,
     TranslationType, TranslationDensity)
```

Arguments

ResourceUsage specifies whether the I/O range is consumed by this device (**ResourceConsumer**) or passed on to child devices (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

IsMinFixed specifies whether the minimum address of this I/O range is fixed (**MinFixed**) or can be changed (**MinNotFixed**). If nothing is specified, then MinNotFixed is assumed. The 1-bit field *DescriptorName*. _MIF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MinFixed and ‘0’ is MinNotFixed.

IsMaxFixed specifies whether the maximum address of this I/O range is fixed (**MaxFixed**) or can be changed (**MaxNotFixed**). If nothing is specified, then MaxNotFixed is assumed. The 1-bit field *DescriptorName.* _MAF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MaxFixed and ‘0’ is MaxNotFixed.

Decode specifies whether or not the device decodes the I/O range using positive (**PosDecode**) or subtractive (**SubDecode**) decode. If nothing is specified, then PosDecode is assumed. The 1-bit field *DescriptorName*. _DEC is automatically created to refer to this portion of the resource descriptor, where ‘1’ is SubDecode and ‘0’ is PosDecode.

ISARanges specifies whether the I/O ranges specifies are limited to valid ISA I/O ranges (**ISAOnly**), valid non-ISA I/O ranges (**NonISAOnly**) or encompass the whole range without limitation (**EntireRange**). The 2-bit field *DescriptorName*._RNG is automatically created to refer to this portion of the resource descriptor, where ‘1’ is NonISAOnly, ‘2’ is ISAOnly and ‘0’ is EntireRange.

AddressGranularity evaluates to a 64-bit integer that specifies the power-of-two boundary (- 1) on which the I/O range must be aligned. The 64-bit field *DescriptorName*. _GRA is automatically created to refer to this portion of the resource descriptor.

AddressMinimum evaluates to a 64-bit integer that specifies the lowest possible base address of the I/O range. The value must have ‘0’ in all bits where the corresponding bit in *AddressGranularity* is ‘1’. For bridge devices which translate addresses, this is the address on the secondary bus. The 64-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 64-bit integer that specifies the highest possible base address of the I/O range. The value must have ‘0’ in all bits where the corresponding bit in *AddressGranularity* is ‘1’. For bridge devices which translate addresses, this is the address on the secondary bus. The 64-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressTranslation evaluates to a 64-bit integer that specifies the offset to be added to a secondary bus I/O address which results in the corresponding primary bus I/O address. For all non-bridge devices or bridges which do not perform translation, this must be ‘0’. The 64-bit field *DescriptorName*._TRA is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 64-bit integer that specifies the total number of bytes decoded in the I/O range. The 64-bit field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor.

ResourceSourceIndex is an optional argument which evaluates to an 8-bit integer that specifies the resource descriptor within the object specified by *ResourceSource*. If this argument is specified, the *ResourceSource* argument must also be specified.

ResourceSource is an optional argument which evaluates to a string containing the path of a device which produces the pool of resources from which this I/O range is allocated. If this argument is specified, but the *ResourceSourceIndex* argument is not specified, a zero value is assumed.

TranslationType is an optional argument that specifies whether the resource type on the secondary side of the bus is different (**TypeTranslation**) from that on the primary side of the bus or the same (**TypeStatic**). If TypeTranslation is specified, then the secondary side of the bus is Memory. If TypeStatic is specified, then the secondary side of the bus is I/O. If nothing is specified, then TypeStatic is assumed. The 1-bit field *DescriptorName*. _TTP is automatically created to refer to this portion of the resource descriptor, where '1' is TypeTranslation and '0' is TypeStatic. See _TTP (page 248) for more information

TranslationDensity is an optional argument that specifies whether or not the translation from the primary to secondary bus is sparse (**SparseTranslation**) or dense (**DenseTranslation**). It is only used when *TranslationType* is **TypeTranslation**. If nothing is specified, then DenseTranslation is assumed. The 1-bit field *DescriptorName*. _TRS is automatically created to refer to this portion of the resource descriptor, where '1' is SparseTranslation and '0' is DenseTranslation. See _TRS (page 248) for more information.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

Description

The **QWordIO** macro evaluates to a buffer which contains a 64-bit I/O resource descriptor, which describes a range of I/O addresses. The format of the 64-bit I/O resource descriptor can be found in QWord Address Space Descriptor (page 235). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.95 QWordMemory (QWord Memory Resource Descriptor Macro)

Syntax

```
QWordMemory (ResourceUsage, Decode, IsMinFixed, IsMaxFixed, Cacheable,
     ReadAndWrite, AddressGranularity, AddressMinimum, AddressMaximum,
     AddressTranslation, RangeLength, ResourceSourceIndex, ResourceSource,
     DescriptorName, MemoryType, TranslationType)
```

Arguments

ResourceUsage specifies whether the Memory range is consumed by this device (**ResourceConsumer**) or passed on to child devices (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

Decode specifies whether or not the device decodes the Memory range using positive (**PosDecode**) or subtractive (**SubDecode**) decode. If nothing is specified, then PosDecode is assumed. The 1-bit field *DescriptorName*. _DEC is automatically created to refer to this portion of the resource descriptor, where '1' is SubDecode and '0' is PosDecode.

IsMinFixed specifies whether the minimum address of this Memory range is fixed (**MinFixed**) or can be changed (**MinNotFixed**). If nothing is specified, then MinNotFixed is assumed. The 1-bit field *DescriptorName*. _MIF is automatically created to refer to this portion of the resource descriptor, where '1' is MinFixed and '0' is MinNotFixed.

IsMaxFixed specifies whether the maximum address of this Memory range is fixed (**MaxFixed**) or can be changed (**MaxNotFixed**). If nothing is specified, then MaxNotFixed is assumed. The 1-bit field *DescriptorName*. _MAF is automatically created to refer to this portion of the resource descriptor, where '1' is MaxFixed and '0' is MaxNotFixed.

Cacheable specifies whether or not the memory region is cacheable (**Cacheable**), cacheable and write-combining (**WriteCombining**), cacheable and prefetchable (**Prefetchable**) or uncacheable (**NonCacheable**). If nothing is specified, then NonCacheable is assumed. The 2-bit field *DescriptorName*._MEM is automatically created to refer to this portion of the resource descriptor, where '1' is Cacheable, '2' is WriteCombining, '3' is Prefetchable and '0' is NonCacheable.

ReadAndWrite specifies whether or not the memory region is read-only (**ReadOnly**) or read/write (**ReadWrite**). If nothing is specified, then ReadWrite is assumed. The 1-bit field *DescriptorName*._RW is automatically created to refer to this portion of the resource descriptor, where '1' is ReadWrite and '0' is ReadOnly.

AddressGranularity evaluates to a 64-bit integer that specifies the power-of-two boundary (- 1) on which the Memory range must be aligned. The 64-bit field *DescriptorName*. _GRA is automatically created to refer to this portion of the resource descriptor.

AddressMinimum evaluates to a 64-bit integer that specifies the lowest possible base address of the Memory range. The value must have '0' in all bits where the corresponding bit in *AddressGranularity* is '1'. For bridge devices which translate addresses, this is the address on the secondary bus. The 64-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 64-bit integer that specifies the highest possible base address of the Memory range. The value must have '0' in all bits where the corresponding bit in *AddressGranularity* is '1'. For bridge devices which translate addresses, this is the address on the secondary bus. The 64-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressTranslation evaluates to a 64-bit integer that specifies the offset to be added to a secondary bus I/O address which results in the corresponding primary bus I/O address. For all non-bridge devices or bridges which do not perform translation, this must be '0'. The 64-bit field *DescriptorName*._TRA is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 64-bit integer that specifies the total number of bytes decoded in the Memory range. The 64-bit field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor.

ResourceSourceIndex is an optional argument which evaluates to an 8-bit integer that specifies the resource descriptor within the object specified by *ResourceSource*. If this argument is specified, the *ResourceSource* argument must also be specified.

ResourceSource is an optional argument which evaluates to a string containing the path of a device which produces the pool of resources from which this Memory range is allocated. If this argument is specified, but the *ResourceSourceIndex* argument is not specified, a zero value is assumed.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

MemoryType is an optional argument that specifies the memory usage. The memory can be marked as normal (**AddressRangeMemory**), used as ACPI NVS space (**AddressRangeNVS**), used as ACPI reclaimable space (**AddressRangeACPI**) or as system reserved (**AddressRangeReserved**). If nothing is specified, then AddressRangeMemory is assumed. The 2-bit field *DescriptorName*. _MTP is automatically created in order to refer to this portion of the resource descriptor, where '0' is AddressRangeMemory, '1' is AddressRangeReserved, '2' is AddressRangeACPI and '3' is AddressRangeNVS.

TranslationType is an optional argument that specifies whether the resource type on the secondary side of the bus is different (**TypeTranslation**) from that on the primary side of the bus or the same (**TypeStatic**). If TypeTranslation is specified, then the secondary side of the bus is I/O. If TypeStatic is specified, then the secondary side of the bus is I/O. If nothing is specified, then TypeStatic is assumed. The 1-bit field *DescriptorName*. _TTP is automatically created to refer to this portion of the resource descriptor, where '1' is TypeTranslation and '0' is TypeStatic. See _TTP (page 248) for more information.

Description

The **QWordMemory** macro evaluates to a buffer which contains a 64-bit memory resource descriptor, which describes a range of memory addresses. The format of the 64-bit memory resource descriptor can be found in “QWord Address Space Descriptor ” (page 235). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.96 QWordSpace (QWord Space Resource Descriptor Macro)

Syntax

```
QWordSpace (ResourceType, ResourceUsage, Decode, IsMinFixed, IsMaxFixed,
     TypeSpecificFlags, AddressGranularity, AddressMinimum, AddressMaximum,
     AddressTranslation, RangeLength, ResourceSourceIndex, ResourceSource,
     DescriptorName)
```

Arguments

ResourceType evaluates to an 8-bit integer that specifies the type of this resource. Acceptable values are 0xC0 through 0xFF.

ResourceUsage specifies whether the Memory range is consumed by this device (**ResourceConsumer**) or passed on to child devices (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

Decode specifies whether or not the device decodes the Memory range using positive (**PosDecode**) or subtractive (**SubDecode**) decode. If nothing is specified, then PosDecode is assumed. The 1-bit field *DescriptorName*. _DEC is automatically created to refer to this portion of the resource descriptor, where ‘1’ is SubDecode and ‘0’ is PosDecode.

IsMinFixed specifies whether the minimum address of this Memory range is fixed (**MinFixed**) or can be changed (**MinNotFixed**). If nothing is specified, then MinNotFixed is assumed. The 1-bit field *DescriptorName*. _MIF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MinFixed and ‘0’ is MinNotFixed.

IsMaxFixed specifies whether the maximum address of this Memory range is fixed (**MaxFixed**) or can be changed (**MaxNotFixed**). If nothing is specified, then MaxNotFixed is assumed. The 1-bit field *DescriptorName.* _MAF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MaxFixed and ‘0’ is MaxNotFixed.

TypeSpecificFlags evaluates to an 8-bit integer. The flags are specific to the *ResourceType*.

AddressGranularity evaluates to a 64-bit integer that specifies the power-of-two boundary (- 1) on which the Memory range must be aligned. The 64-bit field *DescriptorName*. _GRA is automatically created to refer to this portion of the resource descriptor.

AddressMinimum evaluates to a 64-bit integer that specifies the lowest possible base address of the Memory range. The value must have ‘0’ in all bits where the corresponding bit in *AddressGranularity* is ‘1’. For bridge devices which translate addresses, this is the address on the secondary bus. The 64-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 64-bit integer that specifies the highest possible base address of the Memory range. The value must have ‘0’ in all bits where the corresponding bit in *AddressGranularity* is ‘1’. For bridge devices which translate addresses, this is the address on the secondary bus. The 64-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressTranslation evaluates to a 64-bit integer that specifies the offset to be added to a secondary bus I/O address which results in the corresponding primary bus I/O address. For all non-bridge devices or bridges which do not perform translation, this must be ‘0’. The 64-bit field *DescriptorName*._TRA is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 64-bit integer that specifies the total number of bytes decoded in the Memory range. The 64-bit field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor.

ResourceSourceIndex is an optional argument which evaluates to an 8-bit integer that specifies the resource descriptor within the object specified by *ResourceSource*. If this argument is specified, the *ResourceSource* argument must also be specified.

ResourceSource is an optional argument which evaluates to a string containing the path of a device which produces the pool of resources from which this Memory range is allocated. If this argument is specified, but the *ResourceSourceIndex* argument is not specified, a zero value is assumed.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

Description

The **QWordSpace** macro evaluates to a buffer which contains a 64-bit Address Space resource descriptor, which describes a range of addresses. The format of the 64-bit AddressSpace descriptor can be found in “QWord Address Space Descriptor ” (page 235). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.97 RefOf (Create Object Reference)

Syntax

```
RefOf (Object) => ObjectReference
```

Arguments

Object can be any object type (for example, a package, a device object, and so on).

Description

Returns an object reference to *Object*. If the *Object* does not exist, the result of a **RefOf** operation is fatal. Use the **CondRefOf** term in cases where the *Object* might not exist.

The primary purpose of **RefOf()** is to allow an object to be passed to a method as an argument to the method without the object being evaluated at the time the method was loaded.

18.5.98 Register (Generic Register Resource Descriptor Macro)

Syntax

```
Register (AddressSpaceKeyword, RegisterBitWidth, RegisterBitOffset,
     RegisterAddress, AccessSize, DescriptorName)
```

Arguments

AddressSpaceKeyword specifies the address space where the register exists. The register can exist in I/O space (**SystemIO**), memory (**SystemMemory**), PCI configuration space (**PCI_Config**), embedded controller space (**EmbeddedControl**), SMBus (**SMBus**) or fixed-feature hardware (**FFixedHW**). The 8-bit field *DescriptorName*. _ASI is automatically created in order to refer to this portion of the resource descriptor. See _ASI (page 251) for more information, including a list of valid values and their meanings.

RegisterBitWidth evaluates to an 8-bit integer that specifies the number of bits in the register. The 8-bit field *DescriptorName*. _RBW is automatically created in order to refer to this portion of the resource descriptor. See _RBW (page 251) for more information.

RegisterBitOffset evaluates to an 8-bit integer that specifies the offset in bits from the start of the register indicated by *RegisterAddress*. The 8-bit field *DescriptorName*. _RBO is automatically created in order to refer to this portion of the resource descriptor. See _RBO (page 251) for more information.

RegisterAddress evaluates to a 64-bit integer that specifies the register address. The 64-bit field *DescriptorName*. _ADR is automatically created in order to refer to this portion of the resource descriptor. See _ADR (page 251) for more information.

AccessSize evaluates to an 8-bit integer that specifies the size of data values used when accessing the address space as follows:

0 - Undefined (legacy)
1 - Byte access
2 - Word access
3 - DWord access
4 - QWord access

The 8-bit field *DescriptorName*. _ASZ is automatically created in order to refer to this portion of the resource descriptor. See _ASZ(page 251) for more information. For backwards compatibility, the *AccesSize* parameter is optional when invoking the Register macro. If the *AccessSize* parameter is not supplied then the *AccessSize* field will be set to zero. In this case, OSPM will assume the access size.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

Description

The **Register** macro evaluates to a buffer which contains a generic register resource descriptor. The format of the generic register resource descriptor can be found in “Generic Register Descriptor ” (page 251). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.99 Release (Release a Mutex Synchronization Object)

Syntax

```
Release (SyncObject)
```

Arguments

SynchObject must be a mutex synchronization object.

Description

If the mutex object is owned by the current invocation, ownership for the Mutex is released once. It is fatal to release ownership on a Mutex unless it is currently owned. A Mutex must be totally released before an invocation completes.

18.5.100 Reset (Reset an Event Synchronization Object)

Syntax

```
Reset (SyncObject)
```

Arguments

SynchObject must be an Event synchronization object.

Description

This operator is used to reset an event synchronization object to a non-signaled state. See also the Wait and Signal function operator definitions.

18.5.101 ResourceTemplate (Resource To Buffer Conversion Macro)

Syntax

```
ResourceTemplate () {ResourceMacroList} => Buffer
```

Description

For a full definition of the ResourceTemplateTerm macro, see section 18.2.3 “ASL Resource Templates” (page 560)

18.5.102 Return (Return from Method Execution)

Syntax

```
Return
Return ()
Return (Arg)
```

Arguments

Arg is optional and can be any valid object or reference.

Description

Returns control to the invoking control method, optionally returning a copy of the object named in *Arg*. If no *Arg* object is specified, a Return(**Zero**) is generated by the ASL compiler.

Note: in the absence of an explicit **Return ()** statement, the return value to the caller is undefined.

18.5.103 Revision (Constant Revision Object)

Syntax

```
Revision
```

Description

The constant **Revision** object is an object of type Integer that will always read as the revision of the AML interpreter.

18.5.104 Scope (Open Named Scope)

Syntax

```
Scope (Location) {ObjectList}
```

Arguments

Opens and assigns a base namespace scope to a collection of objects. All object names defined within the scope are created relative to *Location*. Note that *Location* does not have to be below the surrounding scope, but can refer to any location within the namespace. The **Scope** term itself does not create objects, but only locates objects within the namespace; the actual objects are created by other ASL terms.

Description

The object referred to by *Location* must already exist in the namespace and be one of the following object types that has a namespace scope associated with it:

- A predefined scope such as: \ (root), **_SB**, **\GPE**, **_PR**, **_TZ**, etc.
- Device
- Processor
- Thermal Zone
- Power Resource

The **Scope** term alters the current namespace location to the existing *Location.* This causes the defined objects within *ObjectList* to be created relative to this new location in the namespace.

Note: When creating secondary SSDTs, it is often required to use the **Scope** operator to change the namespace location in order create objects within some part of the namespace that has been defined by the main DSDT. Use the **External** operator to declare the scope location so that the ASL compiler will not issue an error for an undefined *Location*.

Examples

The following example ASL code uses the **Scope** operator and creates several objects:

```
Scope (\PCI0)
{
    Name (X, 3)
    Scope (\)
    {
        Method (RQ) {Return (0)}
    }
    Name (^Y, 4)
}
```

The created objects are placed in the ACPI namespace as shown:

```
\PCI0.X
\RQ
\Y
```

This example shows the use of **External** in conjunction with **Scope** within an SSDT:

```
DefinitionBlock ("ssdt.aml", "SSDT", 2, "X", "Y", 0x00000001)
{
    External (\_SB.PCI0, DeviceObj)

    Scope (\_SB.PCI0)
    {
    }
}
```

18.5.105 ShiftLeft (Integer Shift Left)

Syntax

ShiftLeft (*Source, ShiftCount*, *Result***)** => Integer

Arguments

Source and *ShiftCount* are evaluated as Integers.

Description

Source is shifted left with the least significant bit zeroed *ShiftCount* times. The result is optionally stored into *Result.*

18.5.106 ShiftRight (Integer Shift Right)

Syntax

```
ShiftRight (Source, ShiftCount, Result) => Integer
```

Arguments

Source and *ShiftCount* are evaluated as Integers.

Description

Source is shifted right with the most significant bit zeroed *ShiftCount* times. The result is optionally stored into *Result.*

18.5.107 Signal (Signal a Synchronization Event)

Syntax

```
Signal (SyncObject)
```

Arguments

SynchObject must be an Event synchronization object.

Description

The Event object is signaled once, allowing one invocation to acquire the event.

18.5.108 SizeOf (Get Data Object Size)

Syntax

```
SizeOf (ObjectName) => Integer
```

Arguments

ObjectName must be a buffer, string or package object.

Description

Returns the size of a buffer, string, or package data object.

For a buffer, it returns the size in bytes of the data. For a string, it returns the size in bytes of the string, not counting the trailing NULL. For a package, it returns the number of elements. For an object reference, the size of the referenced object is returned. Other data types cause a fatal run-time error.

18.5.109 Sleep (Milliseconds Sleep)

Syntax

```
Sleep (MilliSeconds)
```

Arguments

The **Sleep** term is used to implement long-term timing requirements. Execution is delayed for at least the required number of milliseconds.

Description

The implementation of **Sleep** is to round the request up to the closest sleep time supported by the OS and relinquish the processor.

18.5.110 Stall (Stall for a Short Time)

Syntax

Stall (*MicroSeconds***)**

Arguments

The **Stall** term is used to implement short-term timing requirements. Execution is delayed for at least the required number of microseconds.

Description

The implementation of **Stall** is OS-specific, but must not relinquish control of the processor. Because of this, delays longer than 100 microseconds must use **Sleep** instead of **Stall**.

18.5.111 StartDependentFn (Start Dependent Function Resource Descriptor Macro)

Syntax

StartDependentFn (*CompatibilityPriority, PerformancePriority***) {***ResourceList***}**

Arguments

CompatibilityPriority indicates the relative compatibility of the configuration specified by *ResourceList* relative to the PC/AT. 0 = Good, 1 = Acceptable, 2 = Sub-optimal.

PerformancePriority indicates the relative performance of the configuration specified by *ResourceList* relative to the other configurations. 0 = Good, 1 = Acceptable, 2 = Sub-optimal.

ResourceList is a list of resources descriptors which must be selected together for this configuration.

Description

The **StartDependentFn** macro evaluates to a buffer which contains a start dependent function resource descriptor, which describes a group of resources which must be selected together. Each subsequent StartDependentFn or StartDependentFnNoPri resource descriptor introduces a new choice of resources for configuring the device, with the last choice terminated with an EndDependentFn resource descriptor. The format of the start dependent function resource descriptor can be found in “Start Dependent Functions Descriptor” (page 226). This macro generates the two-byte form of the resource descriptor. The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.112 StartDependentFnNoPri (Start Dependent Function Resource Descriptor Macro)

Syntax

StartDependentFnNoPri () {*ResourceList***}**

Description

The **StartDependentFnNoPri** macro evaluates to a buffer which contains a start dependent function resource descriptor, which describes a group of resources which must be selected together. Each subsequent StartDependentFn or StartDependentFnNoPri resource descriptor introduces a new choice of resources for configuring the device, with the last choice terminated with an EndDependentFn resource descriptor. The format of the start dependent function resource descriptor can be found in "Start Dependent Functions Descriptor" (page 226). This macro generates the one-byte form of the resource descriptor. The macro is designed to be used inside of a ResourceTemplate (page 544).

This is similar to StartDependentFn (page 547) with both *CompatibilityPriority* and *PerformancePriority* set to 1, but is one byte shorter.

18.5.113 Store (Store an Object)

Syntax

Store (*Source, Destination***)** => DataRefObject

Arguments

This operation evaluates *Source***,** converts it to the data type of *Destination***,** and writes the result into *Destination.* For information on automatic data-type conversion, see section 16.2.2, "ASL Data Types."

Description

Stores to OperationRegion Field data types may relinquish the processor depending on the region type.

All stores (of any type) to the constant **Zero**, constant **One**, or constant **Ones** object are not allowed. Stores to read-only objects are fatal. The execution result of the operation depends on the type of *Destination*. For any type other than an operation region field, the execution result is the same as the data written to *Destination.* For operation region fields with an *AccessType* of **ByteAcc**, **WordAcc**, **DWordAcc**, **QWordAcc** or **AnyAcc**, the execution result is the same as the data written to *Destination* as in the normal case, but when the *AccessType* is **BufferAcc**, the operation region handler may modify the data when it is written to the *Destination* so that the execution result contains modified data.

Example

The following example creates the name CNT that references an integer data object with the value 5 and then stores CNT to Local0. After the Store operation, Local0 is an integer object with the value 5.

```
Name (CNT, 5)
Store (CNT, Local0)
```

18.5.114 Subtract (Integer Subtract)

Syntax

Subtract (*Minuend, Subtrahend, Result***)** => Integer

Arguments

Minuend and *Subtrahend* are evaluated as Integers.

Description

Subtrahend is subtracted from *Minuend*, and the result is optionally stored into *Result.* Underflow conditions are ignored and the result simply loses the most significant bits.

18.5.115 Switch (Select Code To Execute Based On Expression)

Syntax

Switch (*Expression***) {***CaseTermList***}**

Arguments

Expression is an ASL expression that evaluates to an Integer, String or Buffer.

Description

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

If the **Case** *Value* is an Integer, Buffer or String, then control passes to the statement that matches the value of **Switch (***Expression***)**. If the **Case** value is a Package, then control passes if any member of the package matches the **Switch** (*Value*) 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 have the same value.

Execution of the statement body begins at the selected TermList and proceeds until the TermList end of body or until a **Break** or **Continue** statement transfers control out of the body.

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 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 were first introduced in ACPI 2.0. However, their implementation is backward compatible with ACPI 1.0 AML interpreters.

Example

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

```
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
    }
}
```

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:

```
Name (_T_I, 0)                // Create Integer temporary variable for result
While (One)
{
    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
}
```

The **While (One)** is emitted to enable the use of **Break** and **Continue** within the Switch statement. Temporary names emitted by the ASL compiler should appear at the top level of the method, since the Switch statement could appear within a loop and thus attempt to create the name more than once.

Note: If the ASL compiler is unable to determine the type of the expression, then it will generate a warning and assume a type of Integer. The warning will indicate that the code should use one of the type conversion operators (Such as ToInteger, ToBuffer, ToDecimalString or ToHexString). Caution: Some of these operators are defined starting with ACPI 2.0 and as such may not be supported by ACPI 1.0b compatible interpreters.

For example:

```
Switch (ABCD ())   // Cannot 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 (ToInteger (ABCD ()))
{
    …case statements…
}
```

18.5.116 ThermalZone (Declare Thermal Zone)

Syntax

```
ThermalZone (ThermalZoneName) {ObjectList}
```

Arguments

Declares a Thermal Zone object named *ThermalZoneName*. **ThermalZone** opens a name scope.

Each use of a **ThermalZone** term declares one thermal zone in the system. Each thermal zone in a system is required to have a unique *ThermalZoneName*.

Description

A thermal zone may be declared in the namespace anywhere within the _SB scope. For compatibility with operating systems implementing ACPI 1.0, a thermal zone may also be declared under the _TZ scope. An ACPI-compatible namespace may define Thermal Zone objects in either the _SB or _TZ scope but not both.

For example ASL code that uses a ThermalZone statement, see section 12, “Thermal Management.”

18.5.117 Timer (Get 64-Bit Timer Value)

Syntax

```
Timer => Integer
```

Description

The timer opcode returns a monotonically increasing value that can be used by ACPI methods to measure time passing, this enables speed optimization by allowing AML code to mark the passage of time independent of OS ACPI interpreter implementation.

The **Sleep** opcode can only indicate waiting for longer than the time specified.

The value resulting from this opcode is 64-bits. It is monotonically increasing, but it is not guaranteed that every result will be unique, i.e. two subsequent instructions may return the same value. The only guarantee is that each subsequent evaluation will be greater-than or equal to the previous ones.

The period of this timer is 100 nanoseconds. While the underlying hardware may not support this granularity, the interpreter will do the conversion from the actual timer hardware frequency into 100 nanosecond units.

Users of this opcode should realize that a value returned only represents the time at which the opcode itself executed. There is no guarantee that the next opcode in the instruction stream will execute in any particular time bound.

The OSPM can implement this using the ACPI Timer and keep track of overrun. Other implementations are possible. This provides abstraction away from chipset differences

Compatibility Note: New for ACPI 3.0

18.5.118 ToBCD (Convert Integer to BCD)

Syntax

```
ToBCD (Value, Result) => Integer
```

Arguments

Value is evaluated as an integer

Description

The **ToBCD** operator is used to convert *Value* from a numeric (Integer) format to a BCD format and optionally store the numeric value into *Result*.

18.5.119 ToBuffer (Convert Data to Buffer)

Syntax

```
ToBuffer (Data, Result) => Buffer
```

Arguments

Data must be an Integer, String, or Buffer data type.

Description

Data is converted to buffer type and the result is optionally stored into *Result*. If *Data* is an integer, it is converted into n bytes of buffer (where n is 4 if the definition block has defined integers as 32-bits or 8 if the definition block has defined integers as 64-bits as indicated by the Definition Block table header's Revision field), taking the least significant byte of integer as the first byte of buffer. If *Data* is a buffer, no conversion is performed. If *Data* is a string, each ASCII string character is copied to one buffer byte, including the string null terminator. A null (zero-length) string will be converted to a zero-length buffer.

18.5.120 ToDecimalString (Convert Data to Decimal String)

Syntax

```
ToDecimalString (Data, Result) => String
```

Arguments

Data must be an Integer, String, or Buffer data type.

Description

Data is 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. (Each byte of the buffer is converted to a single decimal value.) A zero-length buffer will be converted to a null (zero-length) string.

18.5.121 ToHexString (Convert Data to Hexadecimal String)

Syntax

```
ToHexString (Data, Result) => String
```

Arguments

Data must be an Integer, String, or Buffer data type.

Description

Data is 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. A zero-length buffer will be converted to a null (zero-length) string.

18.5.122 ToInteger (Convert Data to Integer)

Syntax

```
ToInteger (Data, Result) => Integer
```

Arguments

Data must be an Integer, String, or Buffer data type.

Description

Data is converted to integer type and the result is optionally stored into *Result*. If *Data* is 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. A null (zero-length) string is illegal. If *Data* is 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. A zero-length buffer is illegal. If *Data* is an integer, no action is performed.

18.5.123 ToString (Convert Buffer To String)

Syntax

```
ToString (Source, Length, Result) => String
```

Arguments

Source is evaluated as a buffer. *Length* is evaluated as an integer data type.

Description

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 or a null (0) character is found. If *Length* is not specified or is **Ones**, then the contents of the buffer are copied until a null (0) character is found. If the source buffer has a length of zero, a zero length (null terminator only) string will be created. The result is copied into the *Result*.

18.5.124 ToUUID (Convert String to UUID Macro)

Syntax

ToUUID (*AsciiString***)** => Buffer

Arguments

AsciiString is evaluated as a String data type.

Description

This macro will convert an ASCII string to a 128-bit buffer. The string must have the following format:

aabbccdd-eeff-gghh-iijj-kkllmmnnoopp

where *aa – pp* are one byte hexadecimal numbers, made up of hexadecimal digits. The resulting buffer has the following format:

Table 18-21 UUID Buffer Format

String	Offset In Buffer
aa	**3**
bb	**2**
cc	**1**
dd	**0**
ee	**5**
ff	**4**
gg	**7**
hh	**6**
ii	**8**
jj	**9**
kk	**10**
ll	**11**
mm	**12**
nn	**13**
oo	**14**
pp	**15**

Compatibility Note: New for ACPI 3.0

18.5.125 Unicode (String To Unicode Conversion Macro)

Syntax

```
Unicode (String) => Buffer
```

Arguments

This macro will convert a string to a Unicode (UTF-16) string contained in a buffer. The format of the Unicode string is 16 bits per character, with a 16-bit null terminator.

18.5.126 Unload (Unload Definition Block)

Syntax

```
Unload (Handle)
```

Arguments

Handle is evaluated as a DDBHandle data type.

Description

Performs a run-time unload of a Definition Block that was loaded using a **Load** term or **LoadTable** 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).

18.5.127 VendorLong (Long Vendor Resource Descriptor)

Syntax

```
VendorLong (DescriptorName) {VendorByteList}
```

Arguments

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer.

VendorByteList evaluates to a comma-separated list of 8-bit integer constants, where each byte is added verbatim to the body of the VendorLong resource descriptor. A maximum of n bytes can be specified. UUID and UUID specific descriptor subtype are part of the *VendorByteList*.

Description

The **VendorLong** macro evaluates to a buffer which contains a vendor-defined resource descriptor. The format of the long form of the vendor-defined resource descriptor can be found in Vendor-Defined Descriptor (page 232). The macro is designed to be used inside of a ResourceTemplate (page 544).

This is similar to VendorShort (page 555), except that the number of allowed bytes in *VendorByteList* is 65,533 (instead of 7).

18.5.128 VendorShort (Short Vendor Resource Descriptor)

Syntax

VendorShort (*DescriptorName***) {***VendorByteList***}**

Arguments

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer.

Description

The **VendorShort** macro evaluates to a buffer which contains a vendor-defined resource descriptor. The format of the short form of the vendor-defined resource descriptor can be found in “Vendor-Defined Descriptor” (page 229). The macro is designed to be used inside of a ResourceTemplate (page 544).

This is similar to VendorLong (page 555), except that the number of allowed bytes in *VendorByteList* is 7 (instead of 65,533).

18.5.129 Wait (Wait for a Synchronization Event)

Syntax

Wait (*SyncObject, TimeoutValue***) => Boolean**

Arguments

SynchObject must be an event synchronization object. *TimeoutValue* is evaluated as an Integer. The calling method blocks while waiting for the event to be signaled.

Description

The pending signal count is decremented. If there is no pending signal count, the processor is relinquished until a signal count is posted to the Event or until at least *TimeoutValue* milliseconds have elapsed.

This operation returns a non-zero value if a timeout occurred and a signal was not acquired. A *TimeoutValue* of 0xFFFF (or greater) indicates that there is no time out and the operation will wait indefinitely.

18.5.130 While (Conditional Loop)

Syntax

While (*Predicate***) {**TermList**}**

Arguments

Predicate is evaluated as an integer.

Description

If the *Predicate* is non-zero, the list of terms in TermList is executed. The operation repeats until the *Predicate* evaluates to zero.

Note: Creation of a named object more than once in a given scope is not allowed. As such, unconditionally creating named objects within a **While** loop must be avoided. A fatal error will be generated on the second iteration of the loop, during the attempt to create the same named object a second time.

18.5.131 WordBusNumber (Word Bus Number Resource Descriptor Macro)

Syntax

```
WordBusNumber (ResourceUsage, IsMinFixed, IsMaxFixed, Decode,
     AddressGranularity, AddressMinimum, AddressMaximum, AddressTranslation,
     RangeLength, ResourceSourceIndex, ResourceSource, DescriptorName)
```

Arguments

ResourceUsage specifies whether the bus range is consumed by this device (**ResourceConsumer**) or passed on to child devices (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

IsMinFixed specifies whether the minimum address of this bus number range is fixed (**MinFixed**) or can be changed (**MinNotFixed**). If nothing is specified, then MinNotFixed is assumed. The 1-bit field *DescriptorName*. _MIF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MinFixed and ‘0’ is MinNotFixed.

IsMaxFixed specifies whether the maximum address of this bus number range is fixed (**MaxFixed**) or can be changed (**MaxNotFixed**). If nothing is specified, then MaxNotFixed is assumed. The 1-bit field *DescriptorName.* _MAF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MaxFixed and ‘0’ is MaxNotFixed.

Decode specifies whether or not the device decodes the bus number range using positive (**PosDecode**) or subtractive (**SubDecode**) decode. If nothing is specified, then PosDecode is assumed. The 1-bit field *DescriptorName*. _DEC is automatically created to refer to this portion of the resource descriptor, where ‘1’ is SubDecode and ‘0’ is PosDecode.

AddressGranularity evaluates to a 16-bit integer that specifies the power-of-two boundary (- 1) on which the bus number range must be aligned. The 16-bit field *DescriptorName*. _GRA is automatically created to refer to this portion of the resource descriptor.

AddressMinimum evaluates to a 16-bit integer that specifies the lowest possible bus number for the bus number range. The value must have ‘0’ in all bits where the corresponding bit in *AddressGranularity* is ‘1’. For bridge devices which translate addresses, this is the address on the secondary bus. The 16-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 16-bit integer that specifies the highest possible bus number for the bus number range. The value must have ‘0’ in all bits where the corresponding bit in *AddressGranularity* is ‘1’. For bridge devices which translate addresses, this is the address on the secondary bus. The 16-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressTranslation evaluates to a 16-bit integer that specifies the offset to be added to a secondary bus bus number which results in the corresponding primary bus bus number. For all non-bridge devices or bridges which do not perform translation, this must be ‘0’. The 16-bit field *DescriptorName*._TRA is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 16-bit integer that specifies the total number of bus numbers decoded in the bus number range. The 16-bit field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor.

ResourceSourceIndex is an optional argument which evaluates to an 8-bit integer that specifies the resource descriptor within the object specified by *ResourceSource*. If this argument is specified, the *ResourceSource* argument must also be specified.

ResourceSource is an optional argument which evaluates to a string containing the path of a device which produces the pool of resources from which this I/O range is allocated. If this argument is specified, but the *ResourceSourceIndex* argument is not specified, a zero value is assumed.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

Description

The **WordBusNumber** macro evaluates to a buffer which contains a 16-bit bus-number resource descriptor. The format of the 16-bit bus number resource descriptor can be found in “Word Address Space Descriptor ” (page 240). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.132 WordIO (Word IO Resource Descriptor Macro)

Syntax

```
WordIO (ResourceUsage, IsMinFixed, IsMaxFixed, Decode, ISARanges,
     AddressGranularity, AddressMinimum, AddressMaximum, AddressTranslation,
     RangeLength, ResourceSourceIndex, ResourceSource, DescriptorName,
     TranslationType, TranslationDensity)
```

Arguments

ResourceUsage specifies whether the I/O range is consumed by this device (**ResourceConsumer**) or passed on to child devices (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

IsMinFixed specifies whether the minimum address of this I/O range is fixed (**MinFixed**) or can be changed (**MinNotFixed**). If nothing is specified, then MinNotFixed is assumed. The 1-bit field *DescriptorName*. _MIF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MinFixed and ‘0’ is MinNotFixed.

IsMaxFixed specifies whether the maximum address of this I/O range is fixed (**MaxFixed**) or can be changed (**MaxNotFixed**). If nothing is specified, then MaxNotFixed is assumed. The 1-bit field *DescriptorName.* _MAF is automatically created to refer to this portion of the resource descriptor, where ‘1’ is MaxFixed and ‘0’ is MaxNotFixed.

Decode specifies whether or not the device decodes the I/O range using positive (**PosDecode**) or subtractive (**SubDecode**) decode. If nothing is specified, then PosDecode is assumed. The 1-bit field *DescriptorName*. _DEC is automatically created to refer to this portion of the resource descriptor, where ‘1’ is SubDecode and ‘0’ is PosDecode.

ISARanges specifies whether the I/O ranges specifies are limited to valid ISA I/O ranges (**ISAOnly**), valid non-ISA I/O ranges (**NonISAOnly**) or encompass the whole range without limitation (**EntireRange**). The 2-bit field *DescriptorName*._RNG is automatically created to refer to this portion of the resource descriptor, where ‘1’ is NonISAOnly, ‘2’ is ISAOnly and ‘0’ is EntireRange.

AddressGranularity evaluates to a 16-bit integer that specifies the power-of-two boundary (- 1) on which the I/O range must be aligned. The 16-bit field *DescriptorName*. _GRA is automatically created to refer to this portion of the resource descriptor.

AddressMinimum evaluates to a 16-bit integer that specifies the lowest possible base address of the I/O range. The value must have ‘0’ in all bits where the corresponding bit in *AddressGranularity* is ‘1’. For bridge devices which translate addresses, this is the address on the secondary bus. The 16-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 16-bit integer that specifies the highest possible base address of the I/O range. The value must have ‘0’ in all bits where the corresponding bit in *AddressGranularity* is ‘1’. For bridge devices which translate addresses, this is the address on the secondary bus. The 16-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressTranslation evaluates to a 16-bit integer that specifies the offset to be added to a secondary bus I/O address which results in the corresponding primary bus I/O address. For all non-bridge devices or bridges which do not perform translation, this must be '0'. The 16-bit field *DescriptorName*._TRA is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 16-bit integer that specifies the total number of bytes decoded in the I/O range. The 16-bit field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor.

ResourceSourceIndex is an optional argument which evaluates to an 8-bit integer that specifies the resource descriptor within the object specified by *ResourceSource*. If this argument is specified, the *ResourceSource* argument must also be specified.

ResourceSource is an optional argument which evaluates to a string containing the path of a device which produces the pool of resources from which this I/O range is allocated. If this argument is specified, but the *ResourceSourceIndex* argument is not specified, a zero value is assumed.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

TranslationType is an optional argument that specifies whether the resource type on the secondary side of the bus is different (**TypeTranslation**) from that on the primary side of the bus or the same (**TypeStatic**). If TypeTranslation is specified, then the secondary side of the bus is Memory. If TypeStatic is specified, then the secondary side of the bus is I/O. If nothing is specified, then TypeStatic is assumed. The 1-bit field *DescriptorName*. _TTP is automatically created to refer to this portion of the resource descriptor, where '1' is TypeTranslation and '0' is TypeStatic. See _TTP (page 248) for more information

TranslationDensity is an optional argument that specifies whether or not the translation from the primary to secondary bus is sparse (**SparseTranslation**) or dense (**DenseTranslation**). It is only used when *TranslationType* is **TypeTranslation**. If nothing is specified, then DenseTranslation is assumed. The 1-bit field *DescriptorName*. _TRS is automatically created to refer to this portion of the resource descriptor, where '1' is SparseTranslation and '0' is DenseTranslation. See _TRS (page 248) for more information.

Description

The **WordIO** macro evaluates to a buffer which contains a 16-bit I/O range resource descriptor. The format of the 16-bit I/O range resource descriptor can be found in "Word Address Space Descriptor " (page 240). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.133 WordSpace (Word Space Resource Descriptor Macro))

Syntax

```
WordSpace (ResourceType, ResourceUsage, Decode, IsMinFixed, IsMaxFixed,
     TypeSpecificFlags, AddressGranularity, AddressMinimum, AddressMaximum,
     AddressTranslation, RangeLength, ResourceSourceIndex, ResourceSource,
     DescriptorName)
```

Arguments

ResourceType evaluates to an 8-bit integer that specifies the type of this resource. Acceptable values are 0xC0 through 0xFF.

ResourceUsage specifies whether the bus range is consumed by this device (**ResourceConsumer**) or passed on to child devices (**ResourceProducer**). If nothing is specified, then ResourceConsumer is assumed.

Decode specifies whether or not the device decodes the bus number range using positive (**PosDecode**) or subtractive (**SubDecode**) decode. If nothing is specified, then PosDecode is assumed. The 1-bit field *DescriptorName*. _DEC is automatically created to refer to this portion of the resource descriptor, where '1' is SubDecode and '0' is PosDecode.

IsMinFixed specifies whether the minimum address of this bus number range is fixed (**MinFixed**) or can be changed (**MinNotFixed**). If nothing is specified, then MinNotFixed is assumed. The 1-bit field *DescriptorName*. _MIF is automatically created to refer to this portion of the resource descriptor, where '1' is MinFixed and '0' is MinNotFixed.

IsMaxFixed specifies whether the maximum address of this bus number range is fixed (**MaxFixed**) or can be changed (**MaxNotFixed**). If nothing is specified, then MaxNotFixed is assumed. The 1-bit field *DescriptorName.* _MAF is automatically created to refer to this portion of the resource descriptor, where '1' is MaxFixed and '0' is MaxNotFixed.

TypeSpecificFlags evaluates to an 8-bit integer. The flags are specific to the *ResourceType*.

AddressGranularity evaluates to a 16-bit integer that specifies the power-of-two boundary (- 1) on which the bus number range must be aligned. The 16-bit field *DescriptorName*. _GRA is automatically created to refer to this portion of the resource descriptor.

AddressMinimum evaluates to a 16-bit integer that specifies the lowest possible bus number for the bus number range. The value must have '0' in all bits where the corresponding bit in *AddressGranularity* is '1'. For bridge devices which translate addresses, this is the address on the secondary bus. The 16-bit field *DescriptorName*._MIN is automatically created to refer to this portion of the resource descriptor.

AddressMaximum evaluates to a 16-bit integer that specifies the highest possible bus number for the bus number range. The value must have '0' in all bits where the corresponding bit in *AddressGranularity* is '1'. For bridge devices which translate addresses, this is the address on the secondary bus. The 16-bit field *DescriptorName*._MAX is automatically created to refer to this portion of the resource descriptor.

AddressTranslation evaluates to a 16-bit integer that specifies the offset to be added to a secondary bus bus number which results in the corresponding primary bus bus number. For all non-bridge devices or bridges which do not perform translation, this must be '0'. The 16-bit field *DescriptorName*._TRA is automatically created to refer to this portion of the resource descriptor.

RangeLength evaluates to a 16-bit integer that specifies the total number of bus numbers decoded in the bus number range. The 16-bit field *DescriptorName*. _LEN is automatically created to refer to this portion of the resource descriptor.

ResourceSourceIndex is an optional argument which evaluates to an 8-bit integer that specifies the resource descriptor within the object specified by *ResourceSource*. If this argument is specified, the *ResourceSource* argument must also be specified.

ResourceSource is an optional argument which evaluates to a string containing the path of a device which produces the pool of resources from which this I/O range is allocated. If this argument is specified, but the *ResourceSourceIndex* argument is not specified, a zero value is assumed.

DescriptorName is an optional argument that specifies a name for an integer constant that will be created in the current scope that contains the offset of this resource descriptor within the current resource template buffer. The predefined descriptor field names may be appended to this name to access individual fields within the descriptor via the Buffer Field operators.

Description

The **WordSpace** macro evaluates to a buffer which contains a 16-bit Address Space resource descriptor. The format of the 16-bit Address Space resource descriptor can be found in "Word Address Space Descriptor " (page 240). The macro is designed to be used inside of a ResourceTemplate (page 544).

18.5.134 XOr (Integer Bitwise Xor)

Syntax

```
XOr (Source1, Source2, Result) => Integer
```

Arguments

Source1 and *Source2* are evaluated as Integers.

Description

A bitwise **XOR** is performed and the result is optionally stored into *Result.*

18.5.135 Zero (Constant Zero Object)

Syntax

```
Zero
```

Description

The constant **Zero** object is an object of type Integer that will always read as all bits clear. Writes to this object are not allowed.