

ACPI MACHINE LANGUAGE (AML) SPECIFICATION

This chapter formally defines the ACPI Machine Language (AML), which is the virtual machine language for ACPI control methods on an ACPI-compatible OS. ACPI control methods can be written directly in AML, but people usually write them in ASL and then compile to AML.

AML is the language processed by the ACPI AML interpreter. It is primarily a declarative language. It's best not to think of it as a stream of code, but rather as a set of declarations that the ACPI AML interpreter will compile into the ACPI Namespace at definition block load time. For example, notice that DefByte allocates an anonymous integer variable with a byte-size initial value in ACPI namespace, and passes in an initial value. The byte in the AML stream that defines the initial value is not the address of the variable's storage location.

An OEM or platform firmware vendor needs to write ASL and be able to single-step AML for debugging. (Debuggers and other ACPI control method language tools are expected to be AML-level tools, not source-level tools.) An ASL translator implementer must understand how to read ASL and generate AML. An AML interpreter author must understand how to execute AML.

AML and ASL are different languages, though they are closely related.

All ACPI-compatible operating systems must support AML. A given user can define some arbitrary source language (to replace ASL) and write a tool to translate it to AML. However, the ACPI group will support a single translator for a single language, ASL.

20.1 Notation Conventions

The notation conventions in the table below help the reader to interpret the AML formal grammar.

Table 20.1: AML Grammar Notation Conventions

Notation Convention	Description	Example
0xdd	Refers to a byte value expressed as 2 hexadecimal digits.	0x21
Number in bold.	Denotes the encoding of the AML term.	
Term => Evaluated Type	Shows the resulting type of the evaluation of Term.	
Single quotes (' ')	Indicate constant characters.	'A' => 0x41
Term := Term Term ...	The term to the left of := can be expanded into the sequence of terms on the right.	aterm := bterm cterm means that aterm can be expanded into the two-term sequence of bterm followed by cterm.

continues on next page

Table 20.1 – continued from previous page

Notation Convention	Description	Example
Term Term Term ...	Terms separated from each other by spaces form an ordered list.	
Angle brackets (< >)	used to group items.	<a b> <c d> means either a b or c d.
Bar symbol ()	Separates alternatives.	<p>aterm := bterm [cterm dterm] means the following constructs are possible:</p> <p style="padding-left: 40px;">bterm cterm dterm</p> <p>aterm := [bterm cterm] dterm means the following constructs are possible:</p> <p style="padding-left: 40px;">bterm dterm cterm dterm</p>
Dash character (-)	Indicates a range.	1-9 means a single digit in the range 1 to 9 inclusive.
Parenthesized term following another term.	The parenthesized term is the repeat count of the previous term.	aterm(3) means aterm aterm aterm. bterm(n) means n number of bterms.

20.2 AML Grammar Definition

This section defines the byte values that make up an AML byte stream.

The AML encoding can be categorized into the following groups:

- Table and Table Header encoding
- Name objects encoding
- Data objects encoding
- Package length encoding
- Term objects encoding
- Miscellaneous objects encoding

20.2.1 Table and Table Header Encoding

AMLCode := *DefBlockHeader TermList*

DefBlockHeader := *TableSignature TableLength SpecCompliance CheckSum OemID OemTableID OemRevision CreatorID CreatorRevision*

TableSignature := DWordData // As defined in section 5.2.3.

TableLength := DWordData // Length of the table in bytes including the block header.

SpecCompliance := *ByteData* // The revision of the structure.

CheckSum := *ByteData* // Byte checksum of the entire table.

OemID :=

ByteData(6) // OEM ID of up to 6 characters. If the OEM ID is shorter than 6 characters, it can be terminated with a NULL character.

OemTableID :=

ByteData(8) // OEM Table ID of up to 8 characters. If the OEM Table ID is shorter than 8 characters,
it can be terminated with a NULL character.

OemRevision := DWordData // OEM Table Revision.

CreatorID := DWordData // Vendor ID of the ASL compiler.

CreatorRevision := DWordData // Revision of the ASL compiler.

20.2.2 Name Objects Encoding

LeadNameChar := 'A'-'Z' | '_'

DigitChar := '0' - '9'

NameChar := *DigitChar* | *LeadNameChar*

RootChar := '^'

ParentPrefixChar := '^'

'A'-'Z' := 0x41 - 0x5A

'_' := 0x5F

'0'-'9' := 0x30 - 0x39

'^' := 0x5C

'^' := 0x5E

NameSeg :=

<leadnamechar namechar namechar namechar>

// Notice that NameSegs shorter than 4 characters are filled with trailing underscores ('_'s).

NameString := *<rootchar namepath>* | *<prefixpath namepath>*

PrefixPath := Nothing | *<'^' prefixpath>*

NamePath := NameSeg | *DualNamePath* | *MultiNamePath* | *NullName*

DualNamePath := *DualNamePrefix NameSeg NameSeg*

DualNamePrefix := 0x2E

MultiNamePath := *MultiNamePrefix SegCount NameSeg(SegCount)*

MultiNamePrefix := 0x2F

SegCount := ByteData

// SegCount can be from 1 to 255. For example: MultiNamePrefix(35) is

// encoded as 0x2f 0x23 and followed by 35 NameSegs. So, the total

// encoding length will be 1 + 1 + 35*4 = 142. Notice that:

// DualNamePrefix NameSeg NameSeg has a smaller encoding than the

// encoding of: MultiNamePrefix(2) NameSeg NameSeg

SimpleName := *NameString* | *ArgObj* | *LocalObj*

SuperName := *SimpleName* | *DebugObj* | *ReferenceTypeOpcode*

NullName := 0x00

Target := *SuperName* | *NullName*

20.2.3 Data Objects Encoding

ComputationalData := *ByteConst* | *WordConst* | *DWordConst* | *QWordConst* | *String* | *ConstObj* | *RevisionOp* | *DefBuffer*

DataObject := *ComputationalData* | *DefPackage* | *DefVarPackage*

DataRefObject := *DataObject* | *ObjectReference*

ByteConst := *BytePrefix* *ByteData*

BytePrefix := 0x0A

WordConst := *WordPrefix* *WordData*

WordPrefix := 0x0B

DWordConst := *DWordPrefix* *DWordData*

DWordPrefix := 0x0C

QWordConst := *QWordPrefix* *QWordData*

QWordPrefix := 0x0E

String := *StringPrefix* *AsciiCharList* *NullChar*

StringPrefix := 0x0D

ConstObj := *ZeroOp* | *OneOp* | *OnesOp*

ByteList := *Nothing* | *<bytedata bytelist>*

ByteData := 0x00 - 0xFF

***WordData* := *ByteData*[0:7] *ByteData*[8:15] // 0x0000-0xFFFF**

***DWordData* := *WordData*[0:15] *WordData*[16:31] // 0x00000000-0xFFFFFFFF**

***QWordData* := *DWordData*[0:31] *DWordData*[32:63] // 0x0000000000000000-0xFFFFFFFFFFFFFFFF**

AsciiCharList := *Nothing* | *<asciichar asciicharlist>*

AsciiChar := 0x01 - 0x7F

NullChar := 0x00

ZeroOp := 0x00

OneOp := 0x01

OnesOp := 0xFF

RevisionOp := *ExtOpPrefix* 0x30

ExtOpPrefix := 0x5B

20.2.4 Package Length Encoding

PkgLength :=

PkgLeadByte |
 <*pkgleadbyte bytedata*> |
 <*pkgleadbyte bytedata bytedata*> |
 <*pkgleadbyte bytedata bytedata bytedata*>

PkgLeadByte :=

<bit 7-6: bytedata count that follows (0-3)>
 <bit 5-4: only used if pkglength < 63>
 <bit 3-0: least significant package length nybble>

Note: The high 2 bits of the first byte reveal how many follow bytes are in the PkgLength. If the PkgLength has only one byte, bit 0 through 5 are used to encode the package length (in other words, values 0-63). If the package length value is more than 63, more than one byte must be used for the encoding in which case bit 4 and 5 of the PkgLeadByte are reserved and must be zero. If the multiple bytes encoding is used, bits 0-3 of the PkgLeadByte become the least significant 4 bits of the resulting package length value. The next ByteData will become the next least significant 8 bits of the resulting value and so on, up to 3 ByteData bytes. Thus, the maximum package length is 2*28.

20.2.5 Term Objects Encoding

Object := *NamespaceModifierObj* | *NamedObj*

TermObj := *Object* | *StatementOpcode* | *ExpressionOpcode*

TermList := Nothing | <*termobj termlist*>

TermArg := *ExpressionOpcode* | *DataObject* | *ArgObj* | *LocalObj*

MethodInvocation := *NameString TermArgList*

TermArgList := Nothing | <*termarg termarglist*>

20.2.5.1 Namespace Modifier Objects Encoding

NamespaceModifierObj := *DefAlias* | *DefName* | *DefScope*

DefAlias := *AliasOp NameString NameString*

AliasOp := 0x06

DefName := *NameOp NameString DataRefObject*

NameOp := 0x08

DefScope := *ScopeOp PkgLength NameString TermList*

ScopeOp := 0x10

20.2.5.2 Named Objects Encoding

NamedObj := *DefBankField* | *DefCreateBitField* | *DefCreateByteField* | *DefCreateDWordField* | *DefCreateField* | *DefCreateQWordField* | *DefCreateWordField* | *DefDataRegion* | *DefExternal* | *DefOpRegion* | *DefPowerRes* | *DefThermalZone*

DefBankField := *BankFieldOp* *PkgLength* *NameString* *NameString* *BankValue* *FieldFlags* *FieldList*

BankFieldOp := *ExtOpPrefix* 0x87

BankValue := *TermArg* => Integer

FieldFlags :=

ByteData // bit 0-3: AccessType

// 0 AnyAcc

// 1 ByteAcc

// 2 WordAcc

// 3 DWordAcc

// 4 QWordAcc

// 5 BufferAcc

// 6 Reserved

// 7-15 Reserved

// bit 4: LockRule

// 0 NoLock

// 1 Lock

// bit 5-6: UpdateRule

// 0 Preserve

// 1 WriteAsOnes

// 2 WriteAsZeros

// bit 7: Reserved (must be 0)

FieldList := Nothing | <*fieldelement fieldlist*>

NamedField := *NameSeg* *PkgLength*

ReservedField := 0x00 *PkgLength*

AccessField := 0x01 *AccessType* *AccessAttrib*

AccessType :=

ByteData // Bits 0:3 - Same as AccessType bits of FieldFlags.

// Bits 4:5 - Reserved

// Bits 7:6 - 0 = AccessAttrib = Normal Access Attributes

// 1 = AccessAttrib = AttribBytes (x)

// 2 = AccessAttrib = AttribRawBytes (x)

// 3 = AccessAttrib = AttribRawProcessBytes (x)

//

// x' is encoded as bits 0:7 of the AccessAttrib byte.

AccessAttrib :=

ByteData // If AccessType is BufferAcc for the SMB or

// GPIO OpRegions, AccessAttrib can be one of

// the following values:

```

// 0x02 AttribQuick
// 0x04 AttribSendReceive
// 0x06 AttribByte
// 0x08 AttribWord
// 0x0A AttribBlock
// 0x0C Attrib ProcessCall
// 0x0D AttribBlockProcessCall

ConnectField := <0x02 NameString> | <0x02 BufferData

DefCreateBitField := CreateBitFieldOp SourceBuff BitIndex NameString
CreateBitFieldOp := 0x8D

SourceBuff := TermArg => Buffer

BitIndex := TermArg => Integer

DefCreateByteField := CreateByteFieldOp SourceBuff ByteIndex NameString
CreateByteFieldOp := 0x8C

ByteIndex := TermArg => Integer

DefCreateDWordField := CreateDWordFieldOp SourceBuff ByteIndex NameString
CreateDWordFieldOp := 0x8A

DefCreateField := CreateFieldOp SourceBuff BitIndex NumBits NameString
CreateFieldOp := ExtOpPrefix 0x13

NumBits := TermArg => Integer

DefCreateQWordField := CreateQWordFieldOp SourceBuff ByteIndex NameString
CreateQWordFieldOp := 0x8F

DefCreateWordField := CreateWordFieldOp SourceBuff ByteIndex NameString
CreateWordFieldOp := 0x8B

DefDataRegion := DataRegionOp NameString TermArg TermArg TermArg
DataRegionOp := ExOpPrefix 0x88

DefDevice := DeviceOp PkgLength NameString TermList
DeviceOp := ExtOpPrefix 0x82

DefEvent := EventOp NameString
EventOp := ExtOpPrefix 0x02

DefExternal := ExternalOp NameString ObjectType ArgumentCount
ExternalOp := 0x15

ObjectType := ByteData
ArgumentCount := ByteData (0 - 7)

DefField := FieldOp PkgLength NameString FieldFlags FieldList
FieldOp := ExtOpPrefix 0x81

DefIndexField := IndexFieldOp PkgLength NameString NameString FieldFlags FieldList

```

IndexFieldOp := *ExtOpPrefix* 0x86

DefMethod := *MethodOp PkgLength NameString MethodFlags TermList*

MethodOp := 0x14

MethodFlags :=

- ByteData // bit 0-2: ArgCount (0-7)
- // bit 3: SerializeFlag
- // 0 NotSerialized
- // 1 Serialized
- // bit 4-7: SyncLevel (0x00-0x0f)

DefMutex := *MutexOp NameString SyncFlags*

MutexOp := *ExtOpPrefix* 0x01

SyncFlags := ByteData // bits 0-3: SyncLevel (0x00-0x0f), bits 4-7: Reserved (must be 0)

DefOpRegion := *OpRegionOp NameString RegionSpace RegionOffset RegionLen*

OpRegionOp := *ExtOpPrefix* 0x80

RegionSpace :=

- ByteData // 0x00 SystemMemory
- // 0x01 SystemIO
- // 0x02 PCI_Config
- // 0x03 EmbeddedControl
- // 0x04 SMBus
- // 0x05 System CMOS
- // 0x06 PciBarTarget
- // 0x07 IPMI
- // 0x08 GeneralPurposeIO
- // 0x09 GenericSerialBus
- // 0x0A PCC
- // 0x80-0xFF: OEM Defined

RegionOffset := *TermArg* => Integer

RegionLen := *TermArg* => Integer

DefPowerRes := *PowerResOp PkgLength NameString SystemLevel ResourceOrder TermList*

PowerResOp := *ExtOpPrefix* 0x84

SystemLevel := *ByteData*

ResourceOrder := *WordData*

ProcID := *ByteData*

PblkAddr := *DWordData*

PblkLen := *ByteData*

DefThermalZone := *ThermalZoneOp PkgLength NameString TermList*

ThermalZoneOp := *ExtOpPrefix* 0x85

ExtendedAccessField := 0x03 *AccessType ExtendedAccessAttrib AccessLength*

ExtendedAccessAttrib := ByteData // 0x0B AttribBytes, 0x0E AttribRawBytes, 0x0F AttribRawProcess

FieldElement := *NamedField* | *ReservedField* | *AccessField* | *ExtendedAccessField* | *ConnectField*

20.2.5.3 Statement Opcodes Encoding

StatementOpcode := *DefBreak* | *DefBreakPoint* | *DefContinue* | *DefFatal* | *DefIfElse* | *DefNoop* | *DefNotify* | *DefRelease* | *DefReset* | *DefReturn* | *DefSignal* | *DefSleep* | *DefStall* | *DefWhile*

DefBreak := *BreakOp*

BreakOp := 0xA5

DefBreakPoint := *BreakPointOp*

BreakPointOp := 0xCC

DefContinue := *ContinueOp*

ContinueOp := 0x9F

DefElse := Nothing | *<elseop pkglength termlist>*

ElseOp := 0xA1

DefFatal := *FatalOp FatalType FatalCode FatalArg*

FatalOp := *ExtOpPrefix* 0x32

FatalType := *ByteData*

FatalCode := *DWordData*

FatalArg := *TermArg* => Integer

DefIfElse := *IfOp PkgLength Predicate TermList DefElse*

IfOp := 0xA0

Predicate := *TermArg* => Integer

DefNoop := *NoopOp*

NoopOp := 0xA3

DefNotify := *NotifyOp NotifyObject NotifyValue*

NotifyOp := 0x86

NotifyObject := *SuperName* => ThermalZone | Processor | Device

NotifyValue := *TermArg* => Integer

DefRelease := *ReleaseOp MutexObject*

ReleaseOp := *ExtOpPrefix* 0x27

MutexObject := *SuperName*

DefReset := *ResetOp EventObject*

ResetOp := *ExtOpPrefix* 0x26

EventObject := *SuperName*

DefReturn := *ReturnOp ArgObject*

ReturnOp := 0xA4

ArgObject := *TermArg => DataRefObject*
 DefSignal := *SignalOp EventObject*
 SignalOp := *ExtOpPrefix 0x24*
 DefSleep := *SleepOp MsecTime*
 SleepOp := *ExtOpPrefix 0x22*
 MsecTime := *TermArg => Integer*
 DefStall := *StallOp UsecTime*
 StallOp := *ExtOpPrefix 0x21*
 UsecTime := *TermArg => ByteData*
 DefWhile := *WhileOp PkgLength Predicate TermList*
 WhileOp := 0xA2

20.2.5.4 Expression Opcodes Encoding

ExpressionOpcode := *DefAcquire | DefAdd | DefAnd | DefBuffer | DefConcat | DefConcatRes | DefCondRefOf | DefCopyObject | DefDecrement | DefDerefOf | DefDivide | DefFindSetLeftBit | DefFindSetRightBit | DefFromBCD | DefIncrement | DefIndex | DefLAnd | DefLEqual | DefLGreater | DefLGreaterEqual | DefLLess | DefLLessEqual | DefMid | DefLNot | DefLNotEqual | DefLoadTable | DefLOR | DefMatch | DefMod | DefMultiply | DefNAnd | DefNOR | DefNot | DefObjectType | DefOr | DefPackage | DefVarPackage | DefRefOf | DefShiftLeft | DefShiftRight | DefSizeOf | DefStore | DefSubtract | DefTimer | DefToBCD | DefToBuffer | DefToDecimalString | DefToHexString | DefToInteger | DefToString | DefWait | DefXOR | MethodInvocation*

ReferenceTypeOpcode := *DefRefOf | DefDerefOf | DefIndex | UserTermObj*

DefAcquire := *AcquireOp MutexObject Timeout*

AcquireOp := *ExtOpPrefix 0x23*

Timeout := *WordData*

DefAdd := *AddOp Operand Operand Target*

AddOp := 0x72

Operand := *TermArg => Integer*

DefAnd := *AndOp Operand Operand Target*

AndOp := 0x7B

DefBuffer := *BufferOp PkgLength BufferSize ByteList*

BufferOp := 0x11

BufferSize := *TermArg => Integer*

DefConcat := *ConcatOp Data Data Target*

ConcatOp := 0x73

Data := *TermArg => ComputationalData*

DefConcatRes := *ConcatResOp BufData BufData Target*

ConcatResOp := 0x84

BufData := *TermArg* => *Buffer*
 DefCondRefOf := *CondRefOfOp SuperName Target*
 CondRefOfOp := *ExtOpPrefix* 0x12
 DefCopyObject := *CopyObjectOp TermArg SimpleName*
 CopyObjectOp := 0x9D
 DefDecrement := *DecrementOp SuperName*
 DecrementOp := 0x76
 DefDerefOf := *DerefOfOp ObjReference*
 DerefOfOp := 0x83
 ObjReference := *TermArg* => *ObjectReference* | String
 DefDivide := *DivideOp Dividend Divisor Remainder Quotient*
 DivideOp := 0x78
 Dividend := *TermArg* => Integer
 Divisor := *TermArg* => Integer
 Remainder := *Target*
 Quotient := *Target*
 DefFindSetLeftBit := *FindSetLeftBitOp Operand Target*
 FindSetLeftBitOp := 0x81
 DefFindSetRightBit := *FindSetRightBitOp Operand Target*
 FindSetRightBitOp := 0x82
 DefFromBCD := *FromBCDOp BCDValue Target*
 FromBCDOp := *ExtOpPrefix* 0x28
 BCDValue := *TermArg* => Integer
 DefIncrement := *IncrementOp SuperName*
 IncrementOp := 0x75
 DefIndex := *IndexOp BuffPkgStrObj IndexValue Target*
 IndexOp := 0x88
 BuffPkgStrObj := *TermArg* => *Buffer*, *Package*, or *String*
 IndexValue := *TermArg* => Integer
 DefLAnd := *LandOp Operand Operand*
 LandOp := 0x90
 DefLEqual := *LequalOp Operand Operand*
 LequalOp := 0x93
 DefLGreater := *LgreaterOp Operand Operand*
 LgreaterOp := 0x94
 DefLGreaterEqual := *LgreaterEqualOp Operand Operand*

LgreaterEqualOp := *LnotOp LlessOp*
 DefLLess := *LlessOp Operand Operand*
 LlessOp := 0x95
 DefLLessEqual := *LlessEqualOp Operand Operand*
 LlessEqualOp := *LnotOp LgreaterOp*
 DefLNot := *LnotOp Operand*
 LnotOp := 0x92
 DefLNotEqual := *LnotEqualOp Operand Operand*
 LnotEqualOp := *LnotOp LequalOp*
 DefLoad := *LoadOp NameString Target*
 LoadOp := *ExtOpPrefix* 0x20
 DefLoadTable := *LoadTableOp TermArg TermArg TermArg TermArg TermArg TermArg*
 LoadTableOp := *ExtOpPrefix* 0x1F
 DefLor := *LorOp Operand Operand*
 LorOp := 0x91
 DefMatch := *MatchOp SearchPkg MatchOpcode Operand MatchOpcode Operand StartIndex*
 MatchOp := 0x89
 SearchPkg := *TermArg => Package*
MatchOpcode :=
 ByteData // 0 MTR
 // 1 MEQ
 // 2 MLE
 // 3 MLT
 // 4 MGE
 // 5 MGT
 StartIndex := *TermArg => Integer*
 DefMid := *MidOp MidObj TermArg TermArg Target*
 MidOp := 0x9E
 MidObj := *TermArg => Buffer | String*
 DefMod := *ModOp Dividend Divisor Target*
 ModOp := 0x85
 DefMultiply := *MultiplyOp Operand Operand Target*
 MultiplyOp := 0x77
 DefNAnd := *NandOp Operand Operand Target*
 NandOp := 0x7C
 DefNOr := *NorOp Operand Operand Target*
 NorOp := 0x7E

DefNot := *NotOp Operand Target*
 NotOp := 0x80
 DefObjectType := *ObjectTypeOp <SimpleName | DebugObj | DefRefOf | DefDerefOf | DefIndex>*
 ObjectTypeOp := 0x8E
 DefOr := *OrOp Operand Operand Target*
 OrOp := 0x7D
 DefPackage := *PackageOp PkgLength NumElements PackageElementList*
 PackageOp := 0x12
 DefVarPackage := *VarPackageOp PkgLength VarNumElements PackageElementList*
 VarPackageOp := 0x13
 NumElements := *ByteData*
 VarNumElements := *TermArg => Integer*
 PackageElementList := *Nothing | <packageelement packageelementlist>*
 PackageElement := *DataRefObject | NameString*
 DefRefOf := *RefOfOp SuperName*
 RefOfOp := 0x71
 DefShiftLeft := *ShiftLeftOp Operand ShiftCount Target*
 ShiftLeftOp := 0x79
 ShiftCount := *TermArg => Integer*
 DefShiftRight := *ShiftRightOp Operand ShiftCount Target*
 ShiftRightOp := 0x7A
 DefSizeOf := *SizeOfOp SuperName*
 SizeOfOp := 0x87
 DefStore := *StoreOp TermArg SuperName*
 StoreOp := 0x70
 DefSubtract := *SubtractOp Operand Operand Target*
 SubtractOp := 0x74
 DefTimer := *TimerOp*
 TimerOp := 0x5B 0x33
 DefToBCD := *ToBCDOp Operand Target*
 ToBCDOp := *ExtOpPrefix* 0x29
 DefToBuffer := *ToBufferOp Operand Target*
 ToBufferOp := 0x96
 DefToDecimalString := *ToDecimalStringOp Operand Target*
 ToDecimalStringOp := 0x97
 DefToHexString := *ToHexStringOp Operand Target*

ToHexStringOp := 0x98
DefToInteger := *ToIntegerOp Operand Target*
ToIntegerOp := 0x99
DefToString := *ToStringOp TermArg LengthArg Target*
LengthArg := *TermArg* => Integer
ToStringOp := 0x9C
DefWait := *WaitOp EventObject Operand*
WaitOp := *ExtOpPrefix* 0x25
DefXOr := *XorOp Operand Operand Target*
XorOp := 0x7F

20.2.6 Miscellaneous Objects Encoding

Miscellaneous objects include:

- Arg objects
- Local objects
- Debug objects

20.2.6.1 Arg Objects Encoding

ArgObj := Arg0Op | Arg1Op | Arg2Op | Arg3Op | Arg4Op | Arg5Op | Arg6Op
Arg0Op := 0x68
Arg1Op := 0x69
Arg2Op := 0x6A
Arg3Op := 0x6B
Arg4Op := 0x6C
Arg5Op := 0x6D
Arg6Op := 0x6E

20.2.6.2 Local Objects Encoding

LocalObj := Local0Op | Local1Op | Local2Op | Local3Op | Local4Op | Local5Op | Local6Op | Local7Op
Local0Op := 0x60
Local1Op := 0x61
Local2Op := 0x62
Local3Op := 0x63
Local4Op := 0x64
Local5Op := 0x65
Local6Op := 0x66

Local7Op := 0x67

20.2.6.3 Debug Objects Encoding

DebugObj := *DebugOp*

DebugOp := *ExtOpPrefix* 0x31

20.3 AML Byte Stream Byte Values

The following table lists all byte values that can be found in an AML byte stream, and the meaning of each byte value. This table is useful for debugging AML code.

Table 20.2: AML Byte Stream Byte Values

Encoding Value	Encoding Name	Encoding Group	Fixed List Arguments	Variable List Arguments
0x00	ZeroOp	Data Object	—	—
0x01	OneOp	Data Object	—	—
0x02-0x05	—	—	—	—
0x06	AliasOp	Term Object	NameString NameString	—
0x07	—	—	—	—
0x08	NameOp	Term Object	NameString DataRefObject	—
0x09	—	—	—	—
0x0A	BytePrefix	Data Object	ByteData	—
0x0B	WordPrefix	Data Object	WordData	—
0x0C	DWordPrefix	Data Object	DWordData	—
0x0D	StringPrefix	Data Object	AsciiCharList NullChar	—
0x0E	QWordPrefix	Data Object	QWordData	—
0x0F	—	—	—	—
0x10	ScopeOp	Term Object	NameString	TermList
0x11	BufferOp	Term Object	TermArg	ByteList
0x12	PackageOp	Term Object	ByteData	Package TermList
0x13	VarPackageOp	Term Object	TermArg	Package TermList
0x14	MethodOp	Term Object	NameString ByteData	TermList
0x15	ExternalOp	Name Object	NameString ByteData ByteData	—
0x16-0x2D	—	—	—	—
0x2E (':')	DualNamePrefix	Name Object	NameSeg NameSeg	—
0x2F ('/')	MultiNamePrefix	Name Object	ByteData NameSeg(N)	—
0x30-0x39 ('0'-'9')	DigitChar	Name Object	—	—
0x3A-0x40	—	—	—	—
0x41-0x5A ('A'-'Z')	NameChar	Name Object	—	—
0x5B ('[')	ExtOpPrefix	—	ByteData	—
0x5B 0x00	—	—	—	—
0x5B 0x01	MutexOp	Term Object	NameString ByteData	—
0x5B 0x02	EventOp	Term Object	NameString	—

continues on next page

Table 20.2 – continued from previous page

Encoding Value	Encoding Name	Encoding Group	Fixed List Arguments	Variable List Arguments
0x5B 0x12	CondRefOfOp	Term Object	SuperName SuperName	—
0x5B 0x13	CreateFieldOp	Term Object	TermArg TermArg TermArg NameString	—
0x5B 0x1F	LoadTableOp	Term Object	TermArg TermArg TermArg TermArg TermArg TermArg	—
0x5B 0x20	LoadOp	Term Object	NameString SuperName	—
0x5B 0x21	StallOp	Term Object	TermArg	—
0x5B 0x22	SleepOp	Term Object	TermArg	—
0x5B 0x23	AcquireOp	Term Object	SuperName WordData	—
0x5B 0x24	SignalOp	Term Object	SuperName	—
0x5B 0x25	WaitOp	Term Object	SuperName TermArg	—
0x5B 0x26	ResetOp	Term Object	SuperName	—
0x5B 0x27	ReleaseOp	Term Object	SuperName	—
0x5B 0x28	FromBCDOP	Term Object	TermArg Target	—
0x5B 0x29	ToBCD	Term Object	TermArg Target	—
0x5B 0x2A	Reserved	—	—	—
0x5B 0x30	RevisionOp	Data Object	—	—
0x5B 0x31	DebugOp	Debug Object	—	—
0x5B 0x32	FatalOp	Term Object	ByteData DWordData TermArg	—
0x5B 0x33	TimerOp	Term Object	—	—
0x5B 0x80	OpRegionOp	Term Object	NameString ByteData TermArg TermArg	—
0x5B 0x81	FieldOp	Term Object	NameString ByteData	FieldList
0x5B 0x82	DeviceOp	Term Object	NameString	TermList
0x5B 0x83	<i>Permanently Reserved</i>	—	Use of this opcode for ProcessorOp was deprecated in ACPI 6.4, and is not to be reused.	—
0x5B 0x84	PowerResOp	Term Object	NameString ByteData WordData	TermList
0x5B 0x85	ThermalZoneOp	Term Object	NameString	TermList
0x5B 0x86	IndexFieldOp	Term Object	NameString NameString ByteData	FieldList
0x5B 0x87	BankFieldOp	Term Object	NameString NameString TermArg ByteData	FieldList
0x5B 0x88	DataRegionOp	Term Object	NameString TermArg TermArg TermArg	—
0x5B 0x80 - 0x5B 0xFF	—	—	—	—
0x5C ('\')	RootChar	Name Object	—	—
0x5D	—	—	—	—
0x5E ('^')	ParentPrefixChar	Name Object	—	—
0x5F('_')	NameChar—	Name Object	—	—
0x60 ('')	Local0Op	Local Object	—	—
0x61 ('a')	Local1Op	Local Object	—	—
0x62 ('b')	Local2Op	Local Object	—	—
0x63 ('c')	Local3Op	Local Object	—	—
0x64 ('d')	Local4Op	Local Object	—	—
0x65 ('e')	Local5Op	Local Object	—	—

continues on next page

Table 20.2 – continued from previous page

Encoding Value	Encoding Name	Encoding Group	Fixed List Arguments	Variable List Arguments
0x66 ('f')	Local6Op	Local Object	—	—
0x67 ('g')	Local7Op	Local Object	—	—
0x68 ('h')	Arg0Op	Arg Object	—	—
0x69 ('i')	Arg1Op	Arg Object	—	—
0x6A ('j')	Arg2Op	Arg Object	—	—
0x6B ('k')	Arg3Op	Arg Object	—	—
0x6C ('l')	Arg4Op	Arg Object	—	—
0x6D ('m')	Arg5Op	Arg Object	—	—
0x6E ('n')	Arg6Op	Arg Object	—	—
0x6F	—	—	—	—
0x70	StoreOp	Term Object	TermArg SuperName	—
0x71	RefOfOp	Term Object	SuperName	—
0x72	AddOp	Term Object	TermArg TermArg Target	—
0x73	ConcatOp	Term Object	TermArg TermArg Target	—
0x74	SubtractOp	Term Object	TermArg TermArg Target	—
0x75	IncrementOp	Term Object	SuperName	—
0x76	DecrementOp	Term Object	SuperName	—
0x77	MultiplyOp	Term Object	TermArg TermArg Target	—
0x78	DivideOp	Term Object	TermArg TermArg Target Target	—
0x79	ShiftLeftOp	Term Object	TermArg TermArg Target	—
0x7A	ShiftRightOp	Term Object	TermArg TermArg Target	—
0x7B	AndOp	Term Object	TermArg TermArg Target	—
0x7C	NandOp	Term Object	TermArg TermArg Target	—
0x7D	OrOp	Term Object	TermArg TermArg Target	—
0x7E	NorOp	Term Object	TermArg TermArg Target	—
0x7F	XorOp	Term Object	TermArg TermArg Target	—
0x80	NotOp	Term Object	TermArg Target	—
0x81	FindSetLeftBitOp	Term Object	TermArg Target	—
0x82	FindSetRightBitOp	Term Object	TermArg Target	—
0x83	DerefOfOp	Term Object	TermArg	—
0x84	ConcatResOp	Term Object	TermArg TermArg Target	—
0x85	ModOp	Term Object	TermArg TermArg Target	—
0x86	NotifyOp	Term Object	SuperName TermArg	—
0x87	SizeOfOp	Term Object	SuperName	—
0x88	IndexOp	Term Object	TermArg TermArg Target	—
0x89	MatchOp	Term Object	TermArg ByteData TermArg ByteData TermArg TermArg	—
0x8A	CreateDWordFieldOp	Term Object	TermArg TermArg NameString	—
0x8B	CreateWordFieldOp	Term Object	TermArg TermArg NameString	—
0x8C	CreateByteFieldOp	Term Object	TermArg TermArg NameString	—
0x8D	CreateBitFieldOp	Term Object	TermArg TermArg NameString	—
0x8E	ObjectTypeOp	Term Object	SuperName	—
0x8F	CreateQWordFieldOp	Term Object	TermArg TermArg NameString	—
0x90	LandOp	Term Object	TermArg TermArg	—
0x91	LorOp	Term Object	TermArg TermArg	—
0x92	LnotOp	Term Object	TermArg	—

continues on next page

Table 20.2 – continued from previous page

Encoding Value	Encoding Name	Encoding Group	Fixed List Arguments	Variable List Arguments
0x92 0x93	LNotEqualOp	Term Object	TermArg TermArg	—
0x92 0x94	LLessEqualOp	Term Object	TermArg TermArg	—
0x92 0x95	LGreaterEqualOp	Term Object	TermArg TermArg	—
0x93	LEqualOp	Term Object	TermArg TermArg	—
0x94	LGreaterOp	Term Object	TermArg TermArg	—
0x95	LLessOp	Term Object	TermArg TermArg	—
0x96	ToBufferOp	Term Object	TermArg Target	—
0x97	ToDecimalStringOp	Term Object	TermArg Target	—
0x98	ToHexStringOp	Term Object	TermArg Target	—
0x99	ToIntegerOp	Term Object	TermArg Target	—
0x9A-0x9B	—	—	—	—
0x9C	ToStringOp	Term Object	TermArg TermArg Target	—
0x9D	CopyObjectOp	Term Object	TermArg SimpleName	—
0x9E	MidOp	Term Object	TermArg TermArg TermArg Target	—
0x9F	ContinueOp	Term Object	—	—
0xA0	IfOp	Term Object	TermArg	TermList
0xA1	ElseOp	Term Object	—	TermList
0xA2	WhileOp	Term Object	TermArg	TermList
0xA3	NoopOp	Term Object	—	—
0xA4	ReturnOp	Term Object	TermArg	—
0xA5	BreakOp	Term Object	—	—
0xA6-0xCB	—	—	—	—
0xCC	BreakPointOp	Term Object	—	—
0xCD-0xFE	—	—	—	—
0xFF	OnesOp	Data Object	—	—

20.4 AML Encoding of Names in the Namespace

Assume the following namespace exists:

```

\\
S0
  MEM
    SET
    GET
S1
  MEM
    SET
    GET
  CPU
    SET
    GET

```

Assume further that a definition block is loaded that creates a node \S0.CPU.SET, and loads a block using it as a root. Assume the loaded block contains the following names:

```

STP1
^GET
^^PCI0
^^PCI0.SBS
\\S2
\\S2.ISA.COM1
^^^S3
^^^S2.MEM
^^^S2.MEM.SET
Scope(\S0.CPU.SET.STP1) {
    XYZ
    ^ABC
    ^ABC.DEF
}

```

This will be encoded in AML as:

```

'STP1'
ParentPrefixChar 'GET_'
ParentPrefixChar ParentPrefixChar 'PCI0'
ParentPrefixChar ParentPrefixChar DualNamePrefix 'PCI0' 'SBS_'
RootChar 'S2__'
RootChar MultiNamePrefix 3 'S2__' 'ISA_' 'COM1'
ParentPrefixChar ParentPrefixChar ParentPrefixChar 'S3__'
ParentPrefixChar ParentPrefixChar ParentPrefixChar DualNamePrefix 'S2__' 'MEM_'
ParentPrefixChar ParentPrefixChar ParentPrefixChar MultiNamePrefix 3 'S2__' 'MEM_'
↪ 'SET_'

```

After the block is loaded, the namespace will look like this (names added to the namespace by the loading operation are shown in bold):

```

\\
  S0
    MEM
      SET
      GET
    CPU
      SET
        STP1
          XYZ
          ABC
          DEF
        GET
      PCI0
      SBS
  S1
    MEM
      SET
      GET
    CPU
      SET
      GET
  S2
    ISA
      COM1
    MEM
      SET
  S3

```

ACPI DATA TABLES AND TABLE DEFINITION LANGUAGE

There are two fundamental types of ACPI tables:

- Tables that contain AML code produced from the ACPI Source Language (ASL). These include the DSDT, any SSDTs, and sometimes OEM-specific tables (OEMx).
- Tables that contain simple data and no AML byte code. These types of tables are known as ACPI Data Tables. They include tables such as the FADT, MADT, EC DT, SRAT, etc. - essentially any table other than a DSDT or SSDT.
- The first type of table is generated using an ASL compiler and this language is specified in section 18.

The second type of table, the ACPI Data Table, is addressed by this section.

This section describes a simple language (the Table Definition Language or TDL) that can be used to generate any ACPI data table. It simplifies the table generation for platform firmware vendors and can automatically generate fields such as table lengths, subtable lengths, checksums, flag fields, etc.

21.1 Types of ACPI Data Tables

In the context of a compiler for the Table Definition Language (TDL), there are two types of ACPI Data Tables:

- ACPI tables that are “known” to the compiler. These would typically include all of the basic ACPI tables defined in the ACPI specification such as the FADT, MADT, EC DT, etc. Since these tables are fully specified (usually via the ACPI specification, but from other sources as well), the TDL compiler knows all details of these tables – including all required data types, optional or required sub-tables, etc.
- ACPI tables that are unknown to the compiler. These may include tables that are not defined in the ACPI specification such as MCFG, DBG P, etc., or simply new ACPI tables that have not yet been implemented in the compiler.

One of the goals of the ACPI Table Definition Language is to support both cases above. Most ACPI tables will be known to the compiler (and will be the easiest to specify in TDL), but the language is general enough to allow the definition of new ACPI tables that are unknown or unimplemented in the compiler.

An additional goal of TDL is to support the output of a disassembler that formats an existing table into TDL. This enables disassembler/change/compile operations.

21.2 ACPI Table Definition Language Specification

The following section defines the ACPI Table Definition Language (TDL). The grammar notation follows the same rules as the ASL source language (See [Section 19.2.1](#)). Full definition of the various data types follows the ASL grammar specification.

21.2.1 Overview of the Table Definition Language (TDL)

Most ACPI tables share the following structure (all except FACS):

- A common, 36 byte header containing the table signature, length, checksum, revision, and other data.
- A table body which contains the specific table data.

The Table Definition Language allows the definition of an ACPI table via a collection of fields. Each line of TDL source code is a field, and corresponds to a single data item in the definition of the table.

For example, the C definition of the common ACPI table header is as follows:

```
typedef struct acpi_table_header
{
    char Signature[4];
    UINT32 Length;
    UINT8 Revision;
    UINT8 Checksum;
    char OemId[6];
    char OemTableId[8];
    UINT32 OemRevision;
    char AslCompilerId[4];
    UINT32 AslCompilerRevision;
} ACPI_TABLE_HEADER;
```

In the Table Definition Language, an ACPI table header can be described as follows:

```
: "ECDT"
: 00000000
: 01
: 00
: "OEM "
: "MACHINE1"
: 00000001
: ""
: 00000000
```

Additionally and optionally, it can also be described with accompanying field names:

```
Signature : "ECDT" [Embedded Controller Boot Resources Table]
Table Length : 00000000
Revision : 01
Checksum : 00
Oem ID : "OEM "
Oem Table ID : "MACHINE1"
Oem Revision : 00000001
Asl Compiler ID : ""
Asl Compiler Revision : 00000000
```