AML

From OSDev Wiki

This page or section is a stub. You can help the wiki by *accurately* contributing (https://wiki.osdev.org/index.php?title=AML&action=edit) to it.

ACPI Machine Language (AML) is the platform independent code that ACPI utilizes. A knowledge of it is required to even shutdown the computer. It is found in the DSDT and SSDT tables, which are in turn found by parsing the RSDT or XSDT.

AML code is byte code which is parsed from the beginning of each table when that table is read. It contains definitions of devices and objects within the ACPI namespace. By parsing the code, paying attention to all appropriate control-flow statements, an AML interpreter can build up a database of all devices within a system and the properties and functions they support (in reference to configuration and power management).

The specification is available from the UEFI website (https://uefi.org/specifications) . In addition, Intel provides a reference implementation in its ACPICA software.

ASL and AML

ASL is ACPI source language. It is a more human-readable form of the byte code that is AML. This difference is similar to that between assembly code and the actual binary machine code. The Intel ASL assembler (iasl) is freely available on many Linux distributions and can convert in either direction between these formats.

Sample ASL code

The following is a very simple example of ASL code for a DSDT.

test.asl:

```
DefinitionBlock ("test.aml", "DSDT", 1, "OEMID ", "TABLEID  ", 0x00000000)
{
    Scope (_SB)
    {
        Device (PCI0)
        {
            Name (_HID, EisaId ("PNP0A03"))
        }
    }
}
```

This can then be compiled to AML by running 'iasl test.asl' to generate test.aml.

Here is an actual device off an HP pavilion g6 (RTC). This is embedded in a much larger DefinitionBlock:

```
Device (RTC)
{
    Name (_HID, EisaId ("PNP0B00"))  // _HID: Hardware ID
    Name (_CRS, ResourceTemplate ()  // _CRS: Current Resource Se
    {
        IO (Decode16,
            0x0070,             // Range Minimum
            0x0070,             // Range Maximum
            0x01,               // Alignment
            0x08,               // Length
            )
        IRQNoFlags ()
            {8}
    })
    OperationRegion (CMS0, SystemCMOS, Zero, 0x40)
    Field (CMS0, ByteAcc, NoLock, Preserve)
    {
        RTSE,   8,
        Offset (0x02),
        RTMN,   8,
        Offset (0x04),
        RTHR,   8,
        Offset (0x06),
        RTDY,   8,
        RTDE,   8
    }
}
```

Notice that the region is in the CMOS.

Following on from the RTC, here is a PS/2 Keyboard off the same PC:

```
Device (PS2K)
{
    Name (_HID, EisaId ("PNP0303"))  // _HID: Hardware ID
    Method (_STA, 0, NotSerialized)  // _STA: Status
    {
        Return (0x0F)
    }

    Name (_CRS, ResourceTemplate ()  // _CRS: Current Resource Se
    {
        IO (Decode16,
            0x0060,             // Range Minimum
            0x0060,             // Range Maximum
            0x01,               // Alignment
```

```
                    0x01,               // Length
                    )
                IO (Decode16,
                    0x0064,             // Range Minimum
                    0x0064,             // Range Maximum
                    0x01,               // Alignment
                    0x01,               // Length
                    )
                IRQ (Edge, ActiveHigh, Exclusive, )
                    {1}
            })
            Name (_PRS, ResourceTemplate ()  // _PRS: Possible Resource S
            {
                StartDependentFn (0x00, 0x00)
                {
                    FixedIO (
                        0x0060,             // Address
                        0x01,               // Length
                        )
                    FixedIO (
                        0x0064,             // Address
                        0x01,               // Length
                        )
                    IRQNoFlags ()
                        {1}
                }
                EndDependentFn ()
            })
            Name (_PRW, Package (0x02)  // _PRW: Power Resources for Wake
            {
                0x18,
                0x03
            })
            Method (_PSW, 1, NotSerialized)  // _PSW: Power State Wake
            {
                Store (Arg0, KBWK)
            }
        }
```

the above 2 samples were gathered via a utility I wrote to snatch DSDT data from the Registry, disassembled by iASL.--Bellezzasolo 14:58, 20 January 2013 (CST)

This defines one table (the DSDT), and requests that the output AML be placed in a file called test.aml. The "OEMID " is a 6 character string defining the name of the OEM who made the firmware of the system, the "TABLEID " is an 8 character string defining the name of the table - it is normally OEM specific. The last entry in the DefinitionBlock line is the OEM revision id. It defines one namespace within the root namespace which is called _SB. Note that all device/scope/object names are 4 characters long, therefore this is sometimes represented as "_SB ". _SB is a special name in the ACPI namespace called "System Bus", which is the main scope under which all devices and bus objects are found. See the ACPI specification table 5-67 for a list of predefined names.

Within the _SB scope we define a device called PCI0, which has one object within it called _HID. _HID is again a predefined name that refers to a device's Plug and Play hardware ID, in this case the built-in macro EisaId is used to generate the value 0x030ad041 from "PNP0A03", which is the PNP ID of a PCI root bus. A reasonably complete list of PNP ids is available from http://tuxmobil.org/pnp_ids.html.

AML Opcodes

This table comes from the ACPI specification, where it is provided mainly for use during debugging an AML parser. For example, if your parser fails and the next byte (that it couldn't parse) is "0x72" then you could refer to this table to see that it is an "Add" operation. Don't let this table confuse you into thinking that AML consists a linear flow of simple instructions to be decoded one at a time. Most AML consists of nested, recursively defined structures and lists.

Value (Hex)	Name
0x00	ZeroOp
0x01	OneOp
0x06	AliasOp
0x08	NameOp
0x0A	BytePrefix
0x0B	WordPrefix
0x0C	DWordPrefix
0x0D	StringPrefix
0x0E	QWordPrefix
0x10	ScopeOp
0x11	BufferOp
0x12	PackageOp
0x13	VarPackageOp
0x14	MethodOp
0x2E (‘.’)	DualNamePrefix
0x2F (‘/’)	MultiNamePrefix
0x30-0x39 ('0'-'9')	DigitChar
0x41-0x5A (‘A’-‘Z’)	NameChar
0x5B (‘[’)	ExtOpPrefix
0x5B 0x01	MutexOp
0x5B 0x02	EventOp
0x5B 0x12	CondRefOfOp
0x5B 0x13	CreateFieldOp
0x5B 0x1F	LoadTableOp
0x5B 0x20	LoadOp
0x5B 0x21	StallOp

0x5B 0x22	SleepOp
0x5B 0x23	AcquireOp
0x5B 0x24	SignalOp
0x5B 0x25	WaitOp
0x5B 0x26	ResetOp
0x5B 0x27	ReleaseOp
0x5B 0x28	FromBCDOp
0x5B 0x29	ToBCD
0x5B 0x2A	UnloadOp
0x5B 0x30	RevisionOp
0x5B 0x31	DebugOp
0x5B 0x32	FatalOp
0x5B 0x33	TimerOp
0x5B 0x80	OpRegionOp
0x5B 0x81	FieldOp
0x5B 0x82	DeviceOpList
0x5B 0x83	ProcessorOp
0x5B 0x84	PowerResOp
0x5B 0x85	ThermalZoneOpList
0x5B 0x86	IndexFieldOp
0x5B 0x87	BankFieldOp
0x5B 0x88	DataRegionOp
0x5C (‘\’)	RootChar
0x5E (‘^’)	ParentPrefixChar
0x5F(‘_’)	NameChar
0x60 (‘`’)	Local0Op
0x61 (‘a’)	Local1Op
0x62 (‘b’)	Local2Op
0x63 (‘c’)	Local3Op
0x64 (‘d’)	Local4Op
0x65 (‘e’)	Local5Op
0x66 (‘f’)	Local6Op
0x67 (‘g’)	Local7Op
0x68 (‘h’)	Arg0Op
0x69 (‘i’)	Arg1Op
0x6A (‘j’)	Arg2Op

0x6B (‘k’)	Arg3Op
0x6C (‘l’)	Arg4Op
0x6D (‘m’)	Arg5Op
0x6E (‘n’)	Arg6Op
0x70	StoreOp
0x71	RefOfOp
0x72	AddOp
0x73	ConcatOp
0x74	SubtractOp
0x75	IncrementOp
0x76	DecrementOp
0x77	MultiplyOp
0x78	DivideOp
0x79	ShiftLeftOp
0x7A	ShiftRightOp
0x7B	AndOp
0x7C	NandOp
0x7D	OrOp
0x7E	NorOp
0x7F	XorOp
0x80	NotOp
0x81	FindSetLeftBitOp
0x82	FindSetRightBitOp
0x83	DerefOfOp
0x84	ConcatResOp
0x85	ModOp
0x86	NotifyOp
0x87	SizeOfOp
0x88	IndexOp
0x89	MatchOp
0x8A	CreateDWordFieldOp
0x8B	CreateWordFieldOp
0x8C	CreateByteFieldOp
0x8D	CreateBitFieldOp
0x8E	TypeOp
0x8F	CreateQWordFieldOp

0x90	LandOp
0x91	LorOp
0x92	LnotOp
0x92 0x93	LNotEqualOp
0x92 0x94	LLessEqualOp
0x92 0x95	LGreaterEqualOp
0x93	LEqualOp
0x94	LGreaterOp
0x95	LLessOp
0x96	ToBufferOp
0x97	ToDecimalStringOp
0x98	ToHexStringOp
0x99	ToIntegerOp
0x9C	ToStringOp
0x9D	CopyObjectOp
0x9E	MidOp
0x9F	ContinueOp
0xA0	IfOp
0xA1	ElseOp
0xA2	WhileOp
0xA3	NoopOp
0xA4	ReturnOp
0xA5	BreakOp
0xCC	BreakPointOp
0xFF	OnesOp