ASL 2.0 Introduction and Overview

21 June 2016, version 1.1

Version 6.0 of the Advanced Configuration and Power interface (ACPI) specification introduced support for standard symbolic operators and expressions in the ACPI Source Language (ASL). These changes are known as ASL 2.0 and are detailed in chapter 19 of the ACPI specification. This document intends to increase visibility and further explain the benefits and implementation of these changes.

Legacy ASL

As more developers begin working with ACPI, the limitations of the ASL language itself have become more apparent. The main mathematical and logical operators of ASL are expressed in Polish notation, which places operators to the left of operands, as shown in the examples below:

```
Add (X, Y, Z)
LEqual (X, Y)
```

Building complex expressions with this notation can make ASL code practically unreadable by humans. Even experienced BIOS engineers can find ASL expressions difficult to parse, while developers new to the language might find ASL to be inaccessible compared to languages that use a more common notation.

ASL 2.0

Fortunately version 6.0 of the ACPI specification has introduced standard symbolic operators and infix expressions for ASL that are similar to those used in C and other common algebraic languages. The following examples show how the legacy ASL syntax above is expressed in ASL 2.0:

```
Z = X + Y
If (X == Y)
```

The ASL 2.0 operator precedence and associativity rules are identical to the C language. These new symbolic operators are backwards compatible with existing legacy ASL operators, and are implemented with only minor changes to the ASL compiler and AML disassembler.

These changes to the ASL language are significant enough to warrant changing the name of the language to ASL 2.0, as compared to legacy ASL.

Benefits of ASL 2.0

Existing ASL compilers can be upgraded to ASL 2.0 with only minor changes that allow them to compile the ASL 2.0 operators and expressions. The resulting AML byte code is identical to code compiled from legacy ASL code.

Similar rule changes in the AML disassembler enable developers to disassemble AML byte code into ASL 2.0 code. *This allows developers to automatically convert existing legacy ASL code into functionally equivalent ASL 2.0 code without any additional steps or compatibility concerns.* The table in the *Symbolic Operators and Expressions* section below lists legacy ASL operators, expressions, and associations along with their ASL 2.0 equivalents.

Example use cases

Example 1

To demonstrate the benefits of ASL 2.0, consider the following example of legacy ASL code:

```
If (LOr (LOr (LEqual (And (R510, 0x03FB), 0x02E0), LEqual (
   And (R520, 0x03FB), 0x02E0)), LOr (LEqual (And (R530,
   0x03FB), 0x02E0), LEqual (And (R540, 0x03FB), 0x02E0))))
{
   And (MEMB, 0xFFFFFFF0, SRMB)
   Store (MEMB, Local2)
   Store (PDBM, Local1)
   And (PDBM, 0xFFFFFFFFFFFFFFF9, PDBM)
   Store (SRMB, MEMB)
   Or (PDBM, 0x02, PDBM)
}
```

Example 2

When we compile this Legacy ASL code, and then disassemble the resulting AML byte code using the ASL 2.0 rules, we effectively convert the Legacy ASL code into the following ASL 2.0 code:

```
If (((R510 & 0x03FB) == 0x02E0) ||
    ((R520 & 0x03FB) == 0x02E0) ||
    ((R530 & 0x03FB) == 0x02E0) ||
    ((R540 & 0x03FB) == 0x02E0))
{
    SRMB = (MEMB & 0xFFFFFFF0)
    Local2 = MEMB
    Local1 = PDBM
    PDBM &= 0xFFFFFFFFFFFFFFF9
    MEMB = SRMB
    PDBM |= 0x02
}
```

This second example code block is exactly equivalent to the legacy ASL code in Example 1 above. Note how the notational changes implemented by ASL 2.0 make the converted code much easier to read and write. Developers who are unfamiliar with legacy ASL code can intuitively understand ASL 2.0 because the operators and expressions are so similar to those used in common algebraic languages such as C.

Future developments

The current implementation of ASL 2.0 does not automatically retain comments in legacy ASL code. This is important to note before converting any legacy ASL code into ASL 2.0 code. While the rule changes in the AML compiler and disassembler do greatly improve the readability and accessibility of ASL, any comments in the code are lost during the conversion process.

A future build utility that could retain all code comments throughout the conversion from legacy ASL to ASL 2.0 is in development.

Symbolic Operators and Expressions

The ASL 2.0 operators and legacy ASL equivalents are shown below. You can find additional information in section *19.1 ASL 2.0 Symbolic Operators and Expressions* in version 6.1 of the ACPI specification.

Math operators

ASL 2.0 syntax	Legacy ASL equivalent
`Z = X + Y`	`Add (X, Y, Z)`
`Z = X / Y`	`Divide (X, Y, , Z)`
`Z = X % Y`	`Mod (X, Y, Z)`

Z = X * Y	Multiply (X, Y, Z)
Z = X - Y	Subtract (X, Y, Z)
Z = X << Y	ShiftLeft (X, Y, Z)
Z = X >> Y	ShiftRight (X, Y, Z)
Z = X & Y	And (X, Y, Z)
Z = X | Y	Or (X, Y, Z)
Z = X ^ Y	Xor (X, Y, Z)
Z = ~X	Not (X, Z)
X++	Increment (X)
X--	Decrement (X)

Logical operators

ASL 2.0 syntax	Legacy ASL equivalent
(X == Y)	LEqual (X, Y)
(X != Y)	LNotEqual (X, Y)
(X < Y)	LLess (X, Y)
(X > Y)	LGreater (X, Y)
(X <= Y)	LLessEqual (X, Y)
(X >= Y)	LGreaterEqual (X, Y)
(X && Y)	LAnd (X, Y)
(X || Y)	LOr (X, Y)
!X	LNot (X)

Assignment and compound assignment operations

ASL 2.0 syntax	Legacy ASL equivalent
`X = Y`	`Store (Y, X)`
`X += Y`	`Add (X, Y, X)`
`X /= Y`	`Divide (X, Y, , X)`
`X %= Y`	`Mod (X, Y, X)`
`X *= Y`	`Multiply (X, Y, X)`
`X -= Y`	`Subtract (X, Y, X)`
`X <<= Y`	`ShiftLeft (X, Y, X)`
`X >>= Y`	`ShiftRight (X, Y, X)`
`X &= Y`	`And (X, Y, X)`
`X |= Y`	`Or (X, Y, X)`
`X ^= Y`	`Xor (X, Y, X)`
`Z = X[Y]`	`Index (X, Y, Z)`

Resources

The complete syntax of of ASL opcode, terminal, root, and data terms can be found in the ACPI Specification version 6.1, section *19.5 ASL Operator Summary by Type,* available on http://uefi.org/specifications.