# EFI ASL 简介



freevanx@gmail.com

2010/04/XX

## 一、ASL 基本准则

1. 变量命名不超过 4 个字符，且不能以数字开头。（好像很怀念的感觉）
2. 变量或者函数命名，不分大小写（VB? = Very bad）
3. Scope 形成作用域，概念类似于 C++中的 namespace，Java 中的 package。（不知道的话赶快去复习一下吧）
4. Device 定义也会形成自己的作用域，类似于 C++中 class 的概念
5. Method 或者 Function 定义函数，函数可以定义在 Device 下或者 Scope 下，但是不能脱离 Scope 定义单独的函数，也就是说，函数必须依附于对象（Scope or device）
6. 以"_"字符开头的函数，都是系统保留的，不得给自己的函数取这样的名字
7. ASL 中没有运算符号（逻辑或者算术都是如此），但有与此等价的相应系统函数代替
8. 符号“\”引用根作用域，“^”引用父级或称上级作用域
9. 作用域，或者称路径，有相对和绝对之分。相对作用域从当前作用域开始，向上延伸。也就是说在当前作用域中使用函数和变量时，解析器会首先从当前作用域中寻找它的定义，如果找不到，则会从父级或称上级作用域中继续寻找，一直找到当前作用域的 root 为止。绝对路径，则从定义此变量或者函数的 root 作用域开始，一级级的写下去，一直写到此变量的作用域，作用域的引用使用符号“.”，例如 \SB.PCI0.ABCD 。
10. 函数最多可传递 8 个参数，在函数里用 Arg0~Arg7 引用，不可以自己定义名字
11. 在函数中最多可以使用 8 个局部变量，用 Local0~Local7 表示，不用定义，但是在把局部变量的值赋给其他变量之前，局部变量必须是有效的值，也就是说，至少有一次把值赋给局部变量的操作
12. 声明变量时不需要显式声明其类型，只学系统和应用型语言的童鞋可能会感到强大的不适应，而会 Perl 或者 Python 这类 Script 语言的童鞋则见怪不怪。

## 二、数据类型，赋值，基本运算

1. 数据类型 ASL 中支持的数据类型有，
   a) 整数- Integer，
   b) 字符串 - String，
   c) 事件 - Event，
   d) 数组 - Buffer，
   e) 对象集合 - Package
2. 定义变量

```
Name(MYTS, 0)                   // 定义一个整数
Name(TSTR, "Hello ASL")         // 定义一个字符串
```

3. 赋值

   最常用的赋值函数只有一个，即 Store()， 如

```
Store(0x1234, Local0)               //  Local0 = 0x1234;
Store("Hello ASL", Local0)     // Local0 = "Hello ASL"
```

4. 算术运算

   算数运算的操作符如下图所示，请牢记第一节列出的准则，千万不要使用 + - * / 等符号计算

```
// Integer arithmetic

Add                             575     Integer Add
And                             576     Integer Bitwise And
Decrement                       583     Decrement an Integer
Divide                          586     Integer Divide
FindSetLeftBit                  601     Index of first least significant bit set
FindSetRightBit                 601     Index of first most significant bit set
Increment                       604     Increment a Integer
Mod                             619     Integer Modulo
Multiply                        619     Integer Multiply
NAnd                            621     Integer Bitwise Nand
NOr                             621     Integer Bitwise Nor
Not                             621     Integer Bitwise Not
Or                              625     Integer Bitwise Or
ShiftLeft                       634     Integer shift value left
ShiftRight                      635     Integer shift value right
Subtract                        637     Integer Subtract
Xor                             650     Integer Bitwise Xor
```

举例如下：

```
Add(3, 5, Local0)                   // Local0 = 3 +5
And (0xF4, 0x39, Local0)        // Local0 = 0xF4 & 0x39
Divide(100, 9, Local1, Local0)      // Local0 = 100/9, Local1 = 100 % 9
Mod(100, 9, Local0)             // Local0 = 100 % 9
Multiply(34, 25, Local0)            // Local0 = 34 * 25
Nor(0x34, 0xF8, Local0)         // Local0 = (~0x34) & (~0xF8), 无对应的操作符
Not(0x00, Local0)                   // Local0 = ~0x00
Or(0x3F, 0xF4, Local0)              // Local0 = 0x3F | 0xF4
ShiftLeft(3, 6, Local0)             // Local0 = 3 << 6;
ShiftRight(0xF0, 6, Local0)         // Local0 = 0xF0 >> 6;
Subtract(100, 24, Local0)       // Local0 = 100 - 24;
Xor(0x3F, 0x90, Local0)             // Local0 = 0x3F ^ 0x90
```

除了上面列出的使用方法之外，这些函数还会将计算结果通过返回值传递回来，这样就方便一次连接很长的算式。

例如： `Store(Add(5, 4), Local0)        // Local0 = 5 + 4`

下面我们使用 ASL 计算一个简单的数学算式

```
Local0 = (5+3) * 12 + 100/9 - 100%9
```

在将算式转换为 ASL 算式的时候，我们先从整个算式优先级最低的开始，然后向高一级优先级递进。接下来我们将展示这一过程，注意，在最后一步转换完成前，所展示的代码都是伪代码，不能实际用于 ASL 中。

第一步，从最低优先级最后的“-”开始，转换

```
Store(Subtract((5+3) * 12 + 100/9, 100%9), Local0)
```

第二步，从和“-”相同优先级，但是位置靠前的“+”开始转换

```
Store(Subtract(Add((5+3) * 12 , 100/9), 100%9), Local0)
```

第三步，“*”“/”“%”3 个运算处于同一优先级，由于这个运算比较简单，我们将同时转换这 3 个运算符号

```
Store(Subtract(Add(Multiply((5+3)  , 12) , Divide(100,9)), Mod(100,9)), Local0)
```

然后，最后一步，我们将括号内的加法转换

```
Store(Subtract(Add(Multiply(Add(5,3), 12) , Divide(100,9)), Mod(100,9)), Local0)
```

最终，我们成功得到一个完全用 ASL 写成的算式。

以上，我们展示了从最低优先级到最高优先级逐步转换，将一个 C 语言或者数学算式转换成一个 ASL 算式的过程。当然，从最高优先级到最低优先级转换也同样可行，我们将会在后续的一些示例中展示这种方法。

5. 逻辑运算

ASL 中逻辑运算的函数如下图所示，运算的方法和数学中逻辑运算相同，不过要使用逻辑运算函数代替符号

```
// Logical operators

LAnd                            610     Logical And
LEqual                          610     Logical Equal
LGreater                        610     Logical Greater
LGreaterEqual                   611     Logical Not less
LLess                           611     Logical Less
LLessEqual                      611     Logical Not greater
LNot                            612     Logical Not
LNotEqual                       612     Logical Not equal
LOr                             614     Logical Or
```

举例如下：

```
Store (LAnd(1, 0), Local0)          // Local0 = 1 & 0;
Store (LEqual(45, 32), Local0)      // Local0 = (45 == 32)
Store (LGreater(40, 32), Local0)    // Local0 = (40 > 32)
Store(LGreaterEqual(43, 40), Local0)  // Local0 = (43 >= 40)
Store(LLess(30, 40), Local0)          // Local0 = (30 < 40)
Store(LLessEqual(30, 40), Local0)     // Local0 = (30 <= 40)
Store(LNot(1), Local0)                // Local0 = ! 1
Store(LNotEqual(30, 40), Local0)      // Local0 = (30 != 40)
Store(LOr(1, 0), Local0)              // Local0 = (1 | 0)
```

接下来，将展示上一节所说的，按照从最高优先级开始转换的方法，将一个包含逻辑和算术运算的算式转换成 ASL 语言写成的算式。

算式： Local0 = 30*( 5== 3) + (50/4 >= 30)

第一步，将最高优先级的“==”和“/”转换成 ASL 符号

```
Local0 = 30 * LEqual(5, 3) + (Divide(50, 4) >= 30)
```

第二步，将“*”和“>=”转换成 ASL

```
Local0 = Multiply(30 , LEqual(5, 3)) + (LGreaterEqual(Divide(50, 4) ,30))
```

第三步，将“+”和“=”转换成 ASL 符号

```
Store(Add (Multiply(30, LEqual(5, 3)), LGreaterEqual(Divide(50, 4) ,30)), Local0)
```

三、函数，流程控制

1. 定义函数

```
Method(TMED)
{}
```

2. 定义有两个输入参数的函数

```
Method(TMED, 2)
```

```
{
    // Arg0 - 第一个输入参数
    // Arg1 - 第二个输入参数
}
```

3. 在函数中使用局部变量

```
Method(TMED, 2)
{
    Store(Arg0, Local0)
    Store(Arg1, Local1)
    Add(Local0, Local1, Local0)
}
```

4. 在函数中使用返回值

```
Method(TMED, 2)
{
    Store(Arg0, Local0)
    Store(Arg1, Local1)
    Add(Local0, Local1, Local0)
    Return(Local0)
}
```

5. 调用函数

```
TMED(3, 5)
```

6. 保存函数的返回值

```
Store(TMED(4, 5),  TMPD)
```

与常见的高级语言一样，ASL 中也有与之相应的流程控制语句，现将列表如下：

```
// Method execution control

Break                   577     Continue following the innermost enclosing While
BreakPoint              578     Used for debugging, stops execution in the debugger
Case                    578     Expression for conditional execution
Continue                580     Continue innermost enclosing While loop
Default                 583     Default execution path in Switch()
Else                    591     Alternate conditional execution
ElseIf                  592     Conditional execution
Fatal                   598     Fatal error check
If                      603     Conditional execution
NoOp                    621     No operation
Return                  633     Return from method execution
Sleep                   635     Sleep n milliseconds (yields the processor)
Stall                   636     Delay n microseconds (does not yield the processor)
Switch                  638     Select code to execute based on expression value
While                   645     Conditional loop
```

分支控制 If, ElseIf, Else

借用 ACPI 中的 sample，展示如下

```
// 示例1， 展示If的用法
If (And (Local0, 4))
{
    XOr (Local0, 4, Local0)
}
//示例2， 展示If的用法
```

```
  Store (4, Local2)

  If (And (Local0, Local2))
  {
      XOr (Local0, Local2, Local0)
  }

// 示例三，展示 ElseIf 的用法
If (LEqual (local0, "Microsoft Windows NT"))
{
        Store (3, TOOS)
}
ElseIf (LEqual (Local0, "Microsoft Windows"))
{
        Store (1, TOOS)
}
ElseIf (LEqual (Local0, "Microsoft WindowsME:Millennium Edition"))
{
        Store (2, TOOS)
}

//示例四，展示 Else 的用法
  If (LGreater (Local0, 5)
  {
      Increment (CNT)
  } Else If (Local0) {

      Add (CNT, 5, CNT)
  } Else
  {
      Decrement (CNT)
  }

分支控制 Switch, Case
// 示例，展示 switch case default 的用法
    switch(Arg2)
    {
        case(0)
        {
            switch(Arg1)
            {
                case(0) {return (Buffer() {0x1F})}
                case(1) {return (Buffer() {0x3F})}
```

```
            }
            return (Buffer() {0x7F})
        }
        case(1)
        {
        … function 1 code …
        Return(Zero)
        }
        case(2)
        {
        … function 2 code …
        Return(Buffer(){0x00})
        }
        case(3) { … function 3 code …}
        case(4) { … function 4 code …}
        default {BreakPoint }
    }
```

循环控制 While, Break, Continue

```
//示例，展示While的用法
    While(LEqual(Local0, 45))
    {
        Noop
    }

//示例，展示Break的用法
    While(LEqual(Local0, 45))
    {
        If(Mod (Local0, 4))
        {
            Break
        }
        Noop
    }

//示例，展示Continue的用法
    While(LEqual(Local0, 45))
    {
        If(Mod (Local0, 4))
        {
            Continue
        }
        Noop
    }
```

四、OperationRegion 的使用，IO，Memory，PCI，EC 读写

OperationRegion 是 ACPI 定义的一种操作 Register 方式，可以操作的 Register 包括 IO Memory PCI 等等，ACPI 4.0 支持的 OperationRegion 共有以下几种。此外，用户还可以自己写一个 ACPI 驱动，注册自己的 OperationRegion。

| 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**.

就当前来说，并不是上面所有的 OperationRegion 都受到支持且可以使用，一些 OperationRegion，受限于编译器，OS 下 AML 的 Interpreter 支持等等因素，是不能确定能够在当前 ASL 中使用的，类似的 OperationRegion 有 CMOS，PCIBARTarget，IPMI，SMBus。另外，对于其他的一些 OperationRegion，ACPI Spec 有一些特殊的规定

1. OS 必须保证 SystemIO OperationRegion 在任何情况下都可以使用
2. OS 必须保证 PCI Root Bus 下的 PCI_Config OperationRegion 一定可用
3. OS 必须保证，SystemMemory OperationRegion 在访问通过 Memory Map Report 的 Memory 时，一定可用。事实上，这一条就是说明，只要是在有效地址空间中的 Memory 访问，OS 必须保证 Memory OperationRegion 可用

此外其他 OperationRegion，必须通过_Reg Method 去判断，如果 OperationRegion 已经 connect，则此时此 OperationRegion 可用，如果没有 connect，或者已经 Disconnect，则不可通过此 OperationRegion 去访问设备的地址空间。

IO OperationRegion

接下来，将展示一个 IO OperationRegion 的使用，我们使用定义的 OperationRegion，将 debug code 输出到 80 Port

```
//示例开始
OperationRegion (DBGP, SystemIO, 0x80, 4)
Field (DBGP, ByteAcc, Lock, Preserve)
{
        P80L,      8
}
```

```
Store(0xA3, P80L)         // 输出 A3 到 80 port
```

Memory OperationRegion

接下来，我们通过 Memory OperationRegion 展示一个相当好用函数。我们知道，对于 PCIe 设备来说，有两种访问方法，一种是通过传统的 PCI 兼容方式，另外一种是通过 MMIO，不但可以访问 PCI 的 256 byte 的 PCI Space，而且可以访问全部的 4K PCI space，那么接下来，我们将展示这样一组函数，他能够让你在 ASL 里面自由的访问 PCI Express 的 Configuration Space。

```
//示例开始
#define          PCIE_BASE        0xE0000000
Method (RDPB, 1)
 {
        Add (Arg0, PCIE_BASE, Local0)         // Add PCI Express MMIO base address
        OperationRegion (PECF, SystemMemory, Local0, 0x1)

        Field (PECF, ByteAcc, Nolock, Preserve)
        {
            MCFG, 8 ,
        }

        Return (MCFG)
}
```

上面展示的函数是直接通过 MMIO 访问 PCI Express 设备的 configuration space，但是上面函数只适合比较熟练的开发者使用，因为需要自己计算 PCIe 设备的地址，那么，我们把上面的函数稍微转换一下，变成下面的函数，这样就够直观了。

```
// Arg0 - bus no
// Arg1 - dev no
// Arg2 - func no
// Arg3 - register offset
Method (RDPB, 4)
 {
        ShiftLeft(Arg0, 20, Local0)               // 计算 bus number PCIe address
        Or(ShiftLeft(Arg1, 15), Local0, Local0)   // 计算 device number PCIe address
        Or(ShiftLeft(Arg2, 12), Local0, Local0)   // 计算 function number PCIe address
        Or(Arg3, Local0, Local0)                  // 计算 register，形成最终 PCIe address
        Add (Arg0, PCIE_BASE, Local0)         // Add PCI Express MMIO base address
        OperationRegion (PECF, SystemMemory, Local0, 0x1)
        Field (PECF, ByteAcc, Nolock, Preserve)
        {
            MCFG, 8 ,
        }

        Return (MCFG)
}
```

有了上面的函数，我们可以直接在 ASL 读某个 PCIe 设备的一个 byte，例如，我们读 PCI 设备，bus 0，dev 31， func 3， register 0x40，可以使用如下语句：

```
Store(RDPB(0, 31, 3, 0x40), Local0)
```

上面是读 PCIe Register 的函数，接下来，我们将上面的函数稍作修改，写一个写 byte 到 PCIe register 的函数。

```
// Arg0 - bus no
// Arg1 - dev no
// Arg2 - func no
// Arg3 - register offset
// Arg4 - value
Method (WRPB, 4)
 {
        ShiftLeft(Arg0, 20, Local0)               // 计算 bus number PCIe address
        Or(ShiftLeft(Arg1, 15), Local0, Local0)   // 计算 device number PCIe address
        Or(ShiftLeft(Arg2, 12), Local0, Local0)   // 计算 function number PCIe address
        Or(Arg3, Local0, Local0)                  // 计算 register，形成最终 PCIe address
        Add (Arg0, PCIE_BASE, Local0)         // Add PCI Express MMIO base address
        OperationRegion (PECF, SystemMemory, Local0, 0x1)
        Field (PECF, ByteAcc, Nolock, Preserve)
        {
            MCFG, 8 ,
        }

        Store(Arg4, MCFG)
}
```

可以使用以下语句，将一个 byte 写入到 bus 0， dev 31， func 3， register 0x40

```
WRPB(0, 31, 3, 0x40, 5)
```

PCI_CONFIG OperationRegion

可以通过定义 PCI OperationRegion 来访问 PCI 和 PCI Express 设备的配置空间。不过，可以从 PCI OperationRegion 的定义看到，OperationRegion 本身只定义 Register 在 Configuration Space 的位置和长度，但不能确定 PCI 设备 bus number，device number，和 function number，所以也就不能确定设备的地址。 ACPI 引入了其他一些 Method 来确定 ACPI 中 PCI 设备的地址。 _SEG 函数定义 PCI 设备的 Segment，在 x86 架构中来说，一般不使用_SEG，所有的 PCI 设备默认都在 Segment 0. _BBN 函数定义 PCI Root Bridge 的 bus number，一般来说在 PCI Root Bridge 下定义_BBN(0)，意指从 bus 0 开始。有了 segment 和 bus，最后我们在 PCI 设备下定义一个_ADR 属性，确定此 PCI 设备的 device number 和 function number，_ADR 的返回值中，高 16 位表示 device number，低 16 位表示 function number，接下来将展示一段 sample code,读写 bus 0 ,device 29，function 1 USB controller 的 PCI Configuration Space

```
//示例开始
Name(_ADR, 0x001d0001)                  // Device (HI WORD)=29, Func (LO WORD)=1
```

```
OperationRegion(USBR, PCI_Config, 0xC4, 1)
Field(USBR, ANYACC, NOLOCK, PRESERVE)
{
        URES, 8
}

Method(TEMD)
{
        Store(3, URES)
}
```

EC OperationRegion

EC OperationRegion 是定义 EC Space 操作的，可以在 ASL 里面定义 EC OperationRegion，直接读写 EC OperationRegion，OS 的 ACPI 或者 EC Driver 或将这些操作转换为对 EC Space 的读写。根据 ACPI spec，对于一个读操作，driver 会向 EC 发送 0x80 command 读其中的 value，对于写操作，driver 会向 EC 发 0x81 command 将一个 value 写道 EC Space。接下来定义一个 EC OperationRegion，假设 EC Space offset 0x00 是 CPU 的温度。

//示例开始

```
  OperationRegion (ECF2, EmbeddedControl, 0x00, 0xFF)
  Field (ECF2, ByteAcc, Lock, Preserve)
  {
        CTMP,  8,               //CPU Temp
  }
  Store(CTMP, Local0)         // Read CPU temperature from EC Space
```

与上面其他 OperationRegion 不同的是，EC OperationRegion 并不是任何时刻都可以使用，所以我们要 follow ACPI spec，在同一 scope 中定义一个_Reg 属性，来判断 EC OperationRegion 是否可用

//示例开始

```
  Name(ECON, 0)       // Variable to remember EC OperationRegion Status
   Method(_REG, 0x2)
    {
        if(LEqual(Arg0, 0x03))        // Is it EC OperationRegion? Yes EC =3
        {
            If(Arg1)                  // Is OperationRegion Connect?
            {
                Store(0x01, ECON) // Available
            } Else                    // OperationRegion Disconnect
             {
                Store(0x00, ECON) // unavailable
            }
        }
    }
```