ACPI Source Language (ASL) Tutorial

version 20190625

License

0 Prerequisites

This tutorial assumes that the reader is familiar with ACPI concepts illustrated in the Introduction to ACPI paper. If not, the reader is highly encouraged to read the paper available here: https://acpica.org/sites/acpica/files/ACPI-Introduction.pdf.

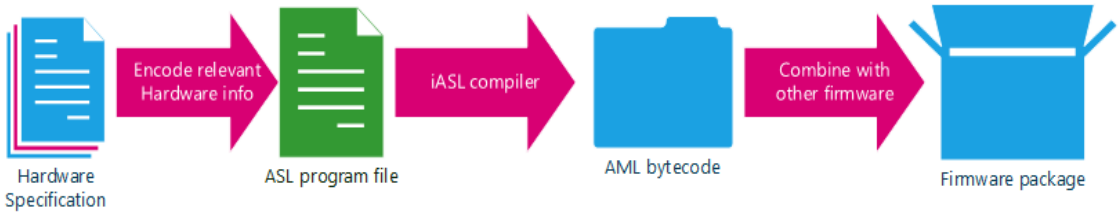
1 Overview

1.1 Operating Systems and ACPI

One role of an operating system (OS) is to configure and manage the system's hardware resources. These resources could include timers, removable devices, and so on. In order to do so, the OS must be able to correctly find and configure devices and system components.

Some components have a hardware infrastructure so that operating systems can easily enumerate and configure certain devices. Other devices cannot be enumerated natively, and their configuration may be dependent on the platform or motherboard. Devices that cannot be enumerated natively can encode their platform-specific information in the Advanced Configuration and Power Interface (ACPI) firmware so they can be enumerated by the OS. ACPI firmware helps the OS by providing information about devices that cannot be enumerated natively.

1.2 ACPI Overview



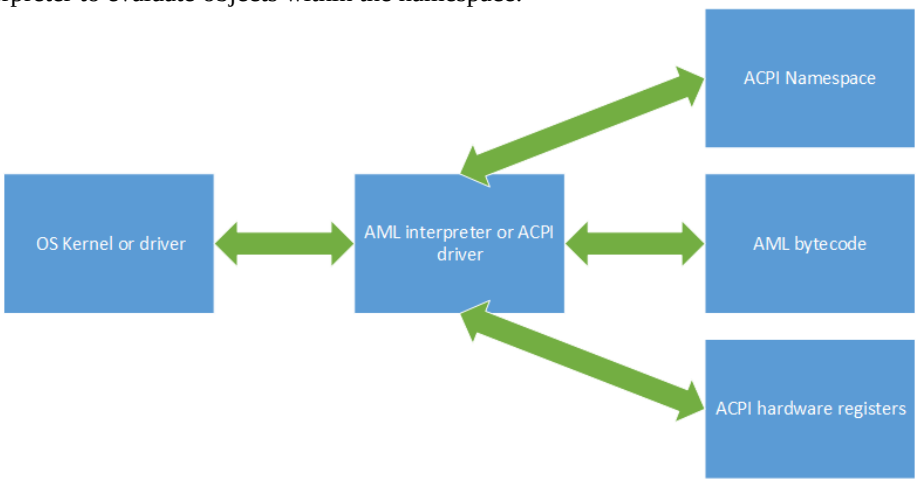
Generally, ACPI development starts with datasheets that describe hardware components. Firmware developers translate relevant portions of the hardware specification to a file containing code written in ACPI source language (ASL). This ASL file is compiled to ACPI machine language (AML) bytecode. AML is packaged along with other firmware code and stored in the platform's non-volatile read-only memory.

This tutorial introduces ASL, a programming language with syntax similar to C and also touches on the basics of other components defined by the ACPI specification.

Once the operating system boots, the AML interpreter starts building the ACPI namespace from AML tables (DSDT and SSDT) contained inside of the firmware package. The ACPI namespace is a tree-like data structure that maps variable names to internal objects. When the OS queries the AML interpreter, the interpreter searches the namespace for the requested variable, evaluates the object associated with the variable, and returns the result of the computation. This is similar to the act of loading a program file in an interpreter, like Python, and interactively invoking functions from the

program file. Another similar example is loading SQL files in a database management system and submitting queries to the database from an interactive SQL prompt.

The ACPI namespace is owned by the AML interpreter that resides in kernel space. The interpreter acts as a mediator between the ACPI namespace and other OS kernel components. Operating systems are not allowed to directly alter the ACPI namespace. The primary operations that the OS performs with the AML interpreter are to load firmware tables and to query the interpreter to evaluate objects within the namespace.



The internal objects associated with the ACPI variable names represent data, device hierarchies, and subroutines that are used for configuration and power management. These objects are evaluated according to the ACPI specification, which may result in change to hardware registers owned by the AML interpreter or the ACPI namespace.

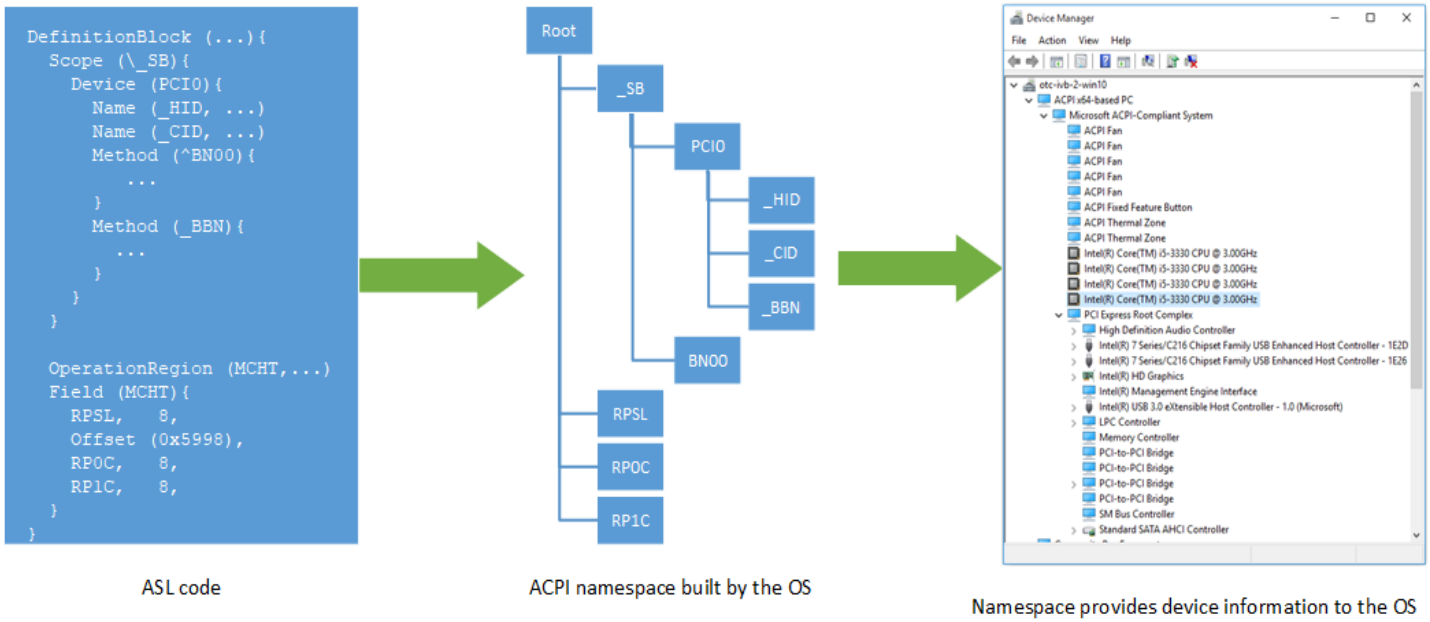
1.2.1 Example

ASL files typically have content that looks like this:

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x0)
{
    Scope (\_SB)
    {
        Device (PCI0)
        {
            Name (INT1, 0x1234)
            Name (_HID, EisaId ("PNP0A08") /* PCI Express Bus */)
            Name (_CID, EisaId ("PNP0A03") /* PCI Bus */)
            Method (^BN00, 0, NotSerialized)
            {
                Return (0x12 + INT1)
            }

            Method (_BBN, 0, NotSerialized)
            {
                Return (BN00 ())
            }

            Name (_UID, 0x00) // _UID: Unique ID

            OperationRegion (MCHT, SystemMemory, 0xFED10000, 0x6000)
            Field (MCHT, ByteAcc, NoLock, Preserve)
            {
                Offset (0x5994),
                RPSL,   8,
                Offset (0x5998),
```

```
                    RP0C,    8,
                    RP1C,    8,
                    RPNC,    8
                }
            }
        }
    }
```

Once a file containing ASL similar to this example is compiled, the file is packaged as firmware. During the initialization of the OS, a namespace hierarchy is created and helps the OS find devices and initialize drivers for them. The following diagram illustrates this concept.



2 ASL declarations

There are two kinds of operators in ASL: operators that create variables and data to populate the ACPI namespace and operators that perform actions on data. This section provides an introduction to the operators that create variables and data to populate the ACPI namespace.

2.1 ASL foundation: the DefinitionBlock

ASL's syntax is similar to C, but there are notable semantic differences, like data types and scoping rules. The fundamental language construct of ASL is the DefinitionBlock. All ASL code must reside inside of DefinitionBlock declarations. ASL code found outside of any DefinitionBlock will be regarded as invalid. Each DefinitionBlock is also called a "table".

The syntax to declare a `DefinitionBlock` is as follows:

```
DefinitionBlock (AMLFileName, TableSignature, ComplianceRevision,
                 OEMID, TableID, OEMRevision)
{
    TermList // A list of ASL terms
}
```

The DefinitionBlock syntax definitions are:

- `AMLFileName` —Name of the AML file (string). Can be a null string.
- `TableSignature` —Signature of the AML file (could be DSDT or SSDT) (4-character string)
- `ComplianceRevision` —A value of 2 or greater enables 64-bit arithmetic; a value of 1 or less enables 32-bit arithmetic (8 bit unsigned integer)
- `OEMID` —ID of the original equipment manufacturer (OEM) developing the ACPI table (6-character string)
- `TableID` —A specific identifier for the table (8-character string)
- `OEMRevision` —Revision number set by the OEM (32-bit number)

A `DefinitionBlock` contains a list of ASL terms. In general, this list is comprised of ASL code that adds variable names to the ACPI namespace. The AMLFileName, OEMID, TableID, and OEMRevision parameters will not be explained in this tutorial. Consult the ACPI specification for more information on these parameters. For simplicity, this tutorial this will use the following format for DefinitionBlocks:

```
DefinitionBlock ("", DSDT, 2, "", "", 0x0)
{
    // A list of ASL terms
}
```

2.2 Populating the ACPI Namespace with named objects

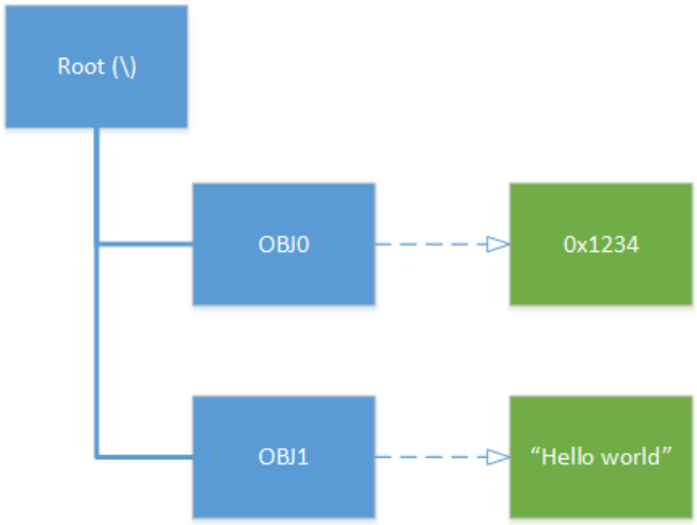
Previously, the ACPI namespace has been described as a mapping from variable names to internal objects. However, in the ACPI specification those "variable names" are also referred to as "object names," and the variables called by those object names can be referred to as "named objects". For consistency with the ACPI specification, we will use that terminology in this tutorial from this point onward.

The simplest way to add a named object to the namespace is by using the ASL `Name` keyword, for example:

```
DefinitionBlock ("", DSDT, 2, "", "", 0x0)
{
```

```
    Name(OBJ0, 0x1234)
    Name(OBJ1, "Hello world")
}
```

This `DefinitionBlock` adds named objects called `OBJ0` and `OBJ1` to the ACPI namespace. In the namespace, `OBJ0` is bound to an object with a value of `0x1234` and `OBJ1` is bound to a string object with a value of `"Hello world"`.



The Name keyword is defined in the ACPI specification as the following:

```
Name (ObjectName, Object)
```

The Name keyword creates a new object named ObjectName and attaches Object to ObjectName in the global ACPI namespace.
`ObjectName` is a four-letter variable name (also called NameSeg) that starts with an alphabetical letter or _ (underscore) and contains up to three or more additional letters, numbers, or underscores. Lowercase letters are converted to uppercase during compilation. Although NameSegs shorter than four characters are padded with additional underscores, this tutorial will use NameSegs that are four characters. Only four characters are allowed in a NameSeg because four bytes fit nicely into a DWORD. The following examples are valid named object declarations:

```
Name (lowr, 0x0)  Name (UPPR, 0x0)  Name (MiXd, 0x0)
Name (___A, 0x0)  Name (X, 0x0)     Name (ABC,0x0)
```

These next named object declarations are invalid:

```
Name (!@#~, 0x0)  Name (TOOLONG, 0x0)     Name (,0x0)
```

Named objects are bound to the types and values of objects in the ACPI namespace through the use of specific keywords. Adding named objects to the ACPI namespace allows the OS to query the AML interpreter to fetch the value of the given named object.

2.2.1 Introduction to iASL

The Intel ASL compiler (iASL) is used to translate ASL to AML bytecode. This short section introduces how to use iASL tools. You can find information on how to build and install the ACPICA tools in the appendix.

First, create a file called `dsdt.asl` in a text editor, and then enter the following:

```
DefinitionBlock ("", "DSDT", 2,"","",0x0)
{
       Name (OBJ1, 0x1234)
       Name (OBJ2, "HELLO WORLD")
       Method (TEST, 2)
       {
           printf ("Arg0 %o", Arg0)
           printf ("Arg2 %o", Arg2)
       }
}
```

To compile the `dsdt.asl` file , open a command line terminal and enter the following command:

```
iasl dsdt.asl
```

This should result in output similar to the following:

```
Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20181031
Copyright (c) 2000 - 2018 Intel Corporation

ASL Input:  dsdt.asl - 15 lines, 340 bytes, 2 keywords
AML Output: dsdt.aml - 62 bytes, 2 named objects, 0 executable
opcodes
```

This output indicates that compilation has completed successfully and the AML has been generated in an output file called `dsdt.aml`. The AML inside this file defines an ACPI Namespace. Typically, the ACPI Namespace is created inside the operating system kernel space. However, in this tutorial we will demonstrate how to simulate creation of the ACPI Namespace in user space. This is useful because it can help prototype ASL without having to constantly reflash new firmware images, and it allows developers to interact with the ACPI Namespace with some limitations.

2.2.2 Introduction to Acpiexec

This user space simulation can be done through *acpiexec*, a userspace version of the AML interpreter that simulates hardware accesses that happen while executing instructions. Execute the following command in the terminal to load `dsdt.aml`:

```
acpiexec dsdt.aml
```

This results in the following output:

```
Input file dsdt.aml, Length 0x3E (62) bytes

ACPI: RSDP 0x000000000068D480 000024 (v02 Intel )
```

```
ACPI: XSDT 0x0000000001FFFC50 000034 (v00 Intel  AcpiExec 00001001
INTL 20181031)
ACPI: FACP 0x000000000068D2A0 000114 (v05 Intel  AcpiExec 00001001
INTL 20181031)
ACPI: DSDT 0x0000000002005470 00003E (v02 Intel  _DSDT_01 00000001
INTL 20181031)
ACPI: FACS 0x000000000068D440 000040

Initializing Namespace objects:
Table [DSDT: _DSDT_01] (id 01) - 2 Objects with   0 Devices,   0 Regions,
      0 Methods (0/0/0 Serial/Non/Cvt)
ACPI: 1 ACPI AML tables successfully acquired and loaded

Completing Region/Field/Buffer/Package initialization:
      Initialized 0/0 Regions 0/0 Fields 0/0 Buffers 0/0 Packages (11
nodes)
Initializing General Purpose Events (GPEs):
      Initialized GPE 00 to 7F [_GPE] 16 regs on interrupt 0x0 (SCI)
      Initialized GPE 80 to FF [_GPE] 16 regs on interrupt 0x0 (SCI)
Completing Region/Field/Buffer/Package initialization:
      Initialized 0/0 Regions 0/0 Fields 0/0 Buffers 0/0 Packages (13
nodes)
Initializing Device/Processor/Thermal objects and executing _INI/_STA
methods:
      Executed 0 _INI methods requiring 0 _STA executions (examined 2
objects)
```

The line below indicates that the DSDT was loaded successfully.

```
Table [DSDT: _DSDT_01] (id 01) -    2 Objects with   0 Devices,   0
Regions,    0 Methods (0/0/0 Serial/Non/Cvt)
ACPI: 1 ACPI AML tables successfully acquired and loaded
-
```

To view the namespace, type the `n` command (short for namespace).

```
- n
ACPI Namespace (from Namespace Root):
 0  _GPE Scope          0x21436c0 00
 0  _PR_ Scope          0x2143720 00
 0  _SB_ Device         0x2143780 00 Notify Object: 0x2148590
 0  _SI_ Scope          0x21437e0 00
 0  _TZ_ Device         0x2143840 00
 0  _REV Integer        0x21438a0 00 = 0000000000000002
 0  _OS_ String         0x2143980 00 Len 14 "Microsoft Windows NT"
 0  _GL_ Mutex          0x2143a60 00 Object 0x2143ac0
 0  _OSI Method         0x2143ba0 00 Args 1 Len 0000 Aml (nil)
 0  OBJ1 Integer        0x2149430 01 = 0000000000001234
```

```
 0  OBJ2 String          0x2149770 01 Len 0B "HELLO WORLD"
 0  TEST Method          0x2149850 01 Args 2 Len 001A Aml 0x21494b5
 0  _TI_ Untyped         0x2148e20 00
 1    _T97 Method        0x2148e80 00 Args 1 Len 0023 Aml 0x2148ee0

Namespace node count: 14

-
```

In the output, `OBJ1` and `OBJ2` exist in the namespace along with several other items.

To simulate the evaluation of `OBJ1`, type `evaluate OBJ1`.

```
- evaluate OBJ1
Evaluating \OBJ1
Evaluation of \OBJ1 returned object 0xffac90, external buffer
length 18
  [Integer] = 0000000000001234
```

As expected, the evaluation of OBJ1 returned an integer with a value of `0x1234`.

You can evaluate control methods in the same way with parameters separated by spaces. To evaluate a sample control method, type `evaluate TEST "Hello world" 0xABCD`

```
- evaluate TEST "Hello world" 0xABCD
Evaluating \TEST
ACPI Debug:  "arg0 Hello world"
ACPI Debug:  "arg1 000000000000ABCD"
No object was returned from evaluation of \TEST
-
```

To exit acpiexec, enter `q`.

2.3 Simple ASL Types

In the previous example, `TST0` was bound to `0x1234`, and `TST1` was bound to `"Hello world"`. In ASL, the name segments also have a type. In this case, `TST0` is an `Integer` and `TST1` is a `String` type.

Integers and strings are prevalent in other languages, but ASL was invented specifically to describe devices, bitfields, and other low-level constructs. For this reason, ASL also has object types that are not found in other languages. These include Device, OperationRegion, ThermalZones, and a few others. Before describing these domain-specific types, we will discuss some simple ASL types. These types are similar to ones that appear in other programming languages.

The following are several ASL types that are similar to other languages:

- `Integer` —An unsigned 64-bit or 32-bit integer. The size depends on the `ComplianceRevision` field in the `DefinitionBlock`.
- `String` —A null-terminated ASCII string.
- `Buffer` —An array of bytes.
- `Package` —An array of ASL objects.
- `Object Reference` —A reference to an object created by `RefOf`, `Index`, or `CondRefOf` operators. This is similar to pointers found in other programming languages.
- `Method` –An executable AML function. Also called control method.

2.3.1 ASL buffers and package declarations

Recall that the syntax for defining `Integer` and `String` objects are done by using the `Name` keyword. The syntax for defining `Buffer` and `Package` objects are similar to `Integer` and `String` but require additional keywords. Here is an example:

```
Name (BUF1, Buffer(3){0x00, 0x01, 0x02})
Name (BUF2, Buffer(){0x00, 0x01, 0x02, 0x03})
```

The above code snippet describes two buffer objects: `BUF1` and `BUF2`. The use of `buffer` as a parameter indicates that the contents inside `{}` are encoded as a buffer. Each element of the comma-separated list is a value between 0x00 and 0xff. There is an optional parameter to this operator that denotes the length of the buffer. If the length parameter is not present, a length will be automatically inserted during compilation.

A package is an array containing ASL objects. The elements of packages can include `Integer`, `String`, `Buffer`, `Package`, or other named objects. The following are examples of package declarations:

```
Name (INT1, 0xABCD)
Name (PKG1, Package(3){0x1234, "Hello world", INT1})
Name (PKG2, Package(){INT1, "Good bye"})
Name (PKG3,
    Package(){
        Package() {0x00, 0x01, 0x02},
        Package() {0x03, 0x04, 0x05}
    })
Name (PKG4, Package(){
        "ASL is fun",
        Package() {0xff, 0xfe, 0xfd, 0xfc, fb}})
Name (PKG5, Package(){
    0x4321,
    Buffer() {0x1}
})
```

Notice that `PKG3` contains two elements that are packages. ASL supports multiple nesting of packages. This is similar to n-dimensional arrays in languages like C. However, ASL packages can contain different types within packages. `PKG4` is a package containing a string and a package. This example is also a valid package declaration.

2.4 Operation Regions and Fields

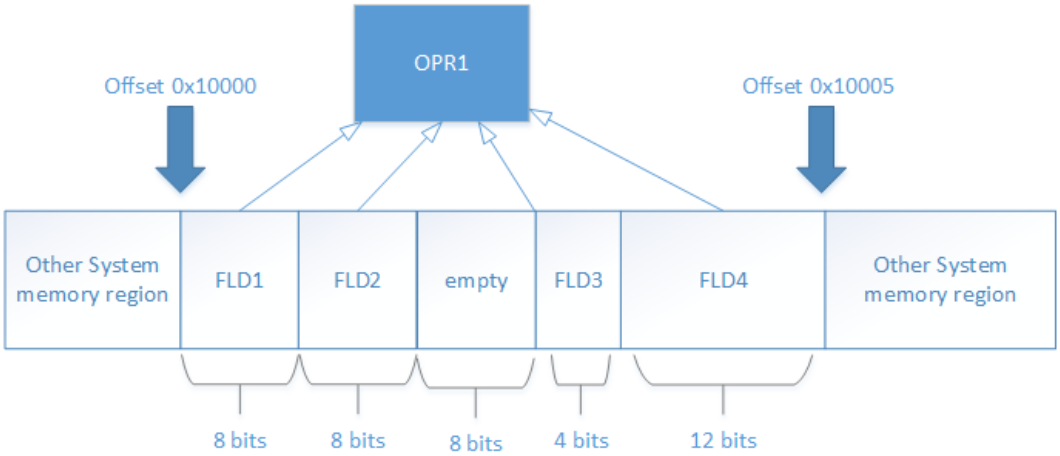
There may be a need for ASL code to access system memory or hardware registers in ASL. These areas may contain important information that was initialized at boot time. They may also represent hardware registers that result in I/O or memory requests. These registers and memory regions can be defined in ASL with the `OperationRegion` keyword and the `Field` keyword.

`OperationRegion` defines a named object as a certain type (such as SystemMemory, SystemIO, PCIConfig, etc.), and gives the starting address and length. The `Field` keyword defines individual bit fields inside of an `OperationRegion`. The individual field units that are used in control methods to access data in this operation region reside at a particular offset. The following example declares an OperationRegion and field units.

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x1)
{
    OperationRegion(OPR1, SystemMemory, 0x10000, 0x5)
    Field (OPR1)
    {
        FLD1, 8
        FLD2, 8
```

```
        Offset (3), //Start the next field unit at byte offset 3
        FLD3, 4
        FLD4, 12
    }
}
```

This operation region is called OPR1. It represents system memory, and it starts at address 0x10000 with a length of 5 bytes. FLD1 through FLD4 are declared inside of OPR1. FLD1 and FLD2 span 8 bits each, while FLD3 starts at byte offset 3 and spans 4 bits, and FLD4 spans 12 bits. The primary motivation for operation region and field declaration is to read and write values to FLD1 through FLD4 inside of control methods (discussed later).



There are many operation region subtypes other than SystemMemory. To learn more, consult the ACPI specification.

3 Scopes

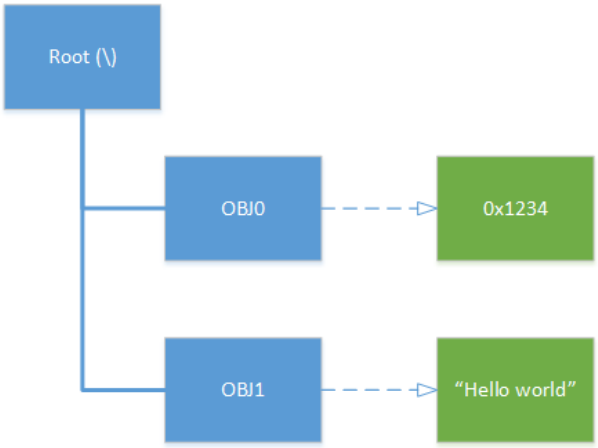
3.1 Defining scopes using Devices

The ACPI namespace is a tree-like data structure that describes a hierarchy of named objects. Each layer in this hierarchy is called a scope. Once a scope is defined, additional named objects can be inserted in the defined scope.

The contents inside of DefinitionBlock declarations represent the top-most scope of the ACPI namespace called the root scope. Therefore, declaring a named object inside a DefinitionBlock will add the named object to the root scope. Consider the following table:

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x1)
{
    Name (OBJ0, 0x1234)
    Name (OBJ1, "Hello world!")
}
```

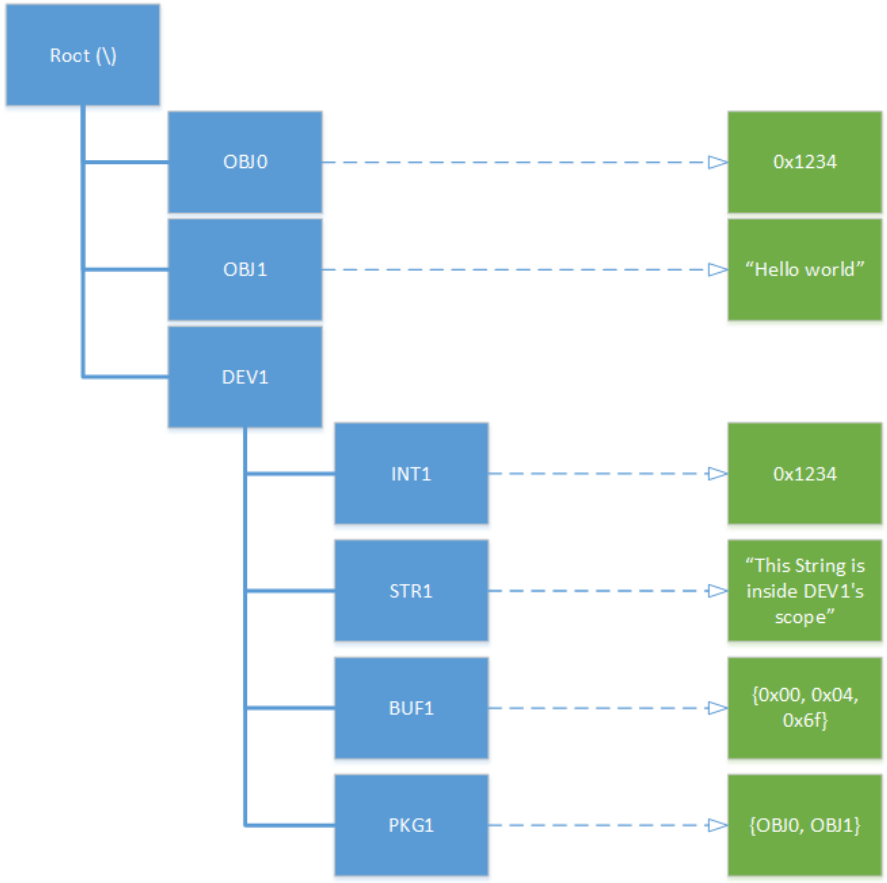
Loading this `Definitionblock` will result in a namespace like this:



One role of ASL is to describe devices that are not natively enumerable. The ASL `Device` object represents a device and defines a scope. The contents of this scope provide information about the device. Here is an example of a device declaration:

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x1)
{
    Name (OBJ0, 0x1234)
    Name (OBJ1, "Hello world!")
    Device (DEV1)
    {
        Name (INT1, 0x1234)
        Name (STR1, "This string is inside of DEV1's scope")
        Name (BUF1, Buffer() {0x00, 0x04, 0x6f})
        Name (PKG1, Package() {OBJ0, OBJ1})
    }
}
```

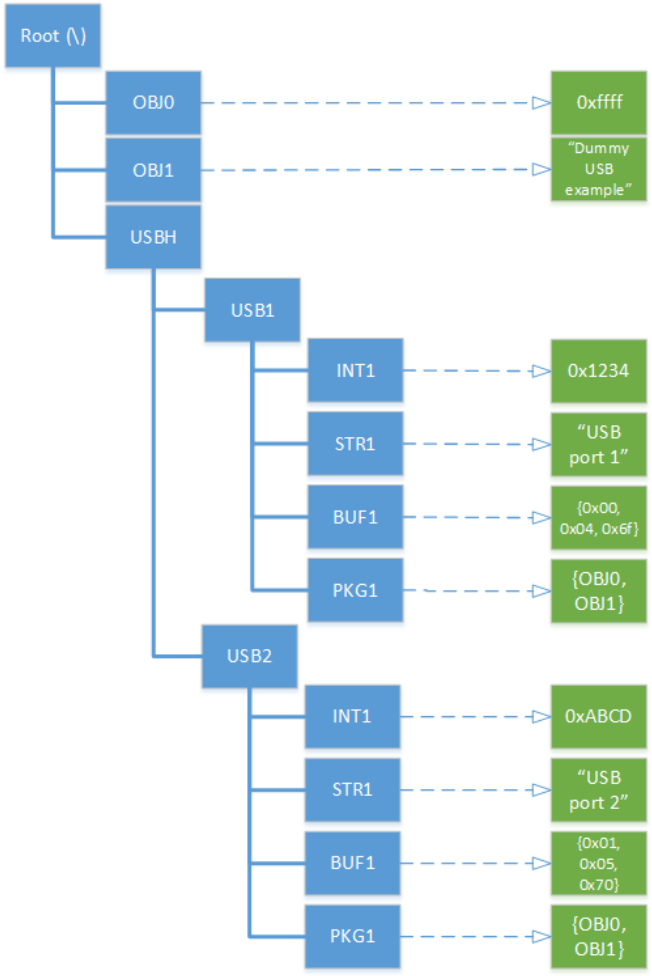
The braces after `Device (DEV1)` represent the scope of the device. INT1, STR1, BUF1, and PKG1 contain information about DEV1. This results in a namespace that looks like this:



Device declarations could be nested as well:

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x1)
{
    Name (OBJ0, 0xffff)
    Name (OBJ1, "Dummy USB example")

    // USB host controller
    // This device can contain many ports
    Device (USBH)
    {
        Device (USB1) // USB port #1
        {
            Name (INT1, 0x1234)
            Name (STR1, "USB port 1")
            Name (BUF1, Buffer() {0x00, 0x04, 0x6f})
            Name (PKG1, Package() {OBJ0, OBJ1})
        }
        Device (USB2) // USB port #2
        {
            Name (INT1, 0xABCD)
            Name (STR1, "USB port 2")
            Name (BUF1, Buffer() {0x01, 0x05, 0x70})
            Name (PKG1, Package() {OBJ0, OBJ1})
        }
    }
}
```

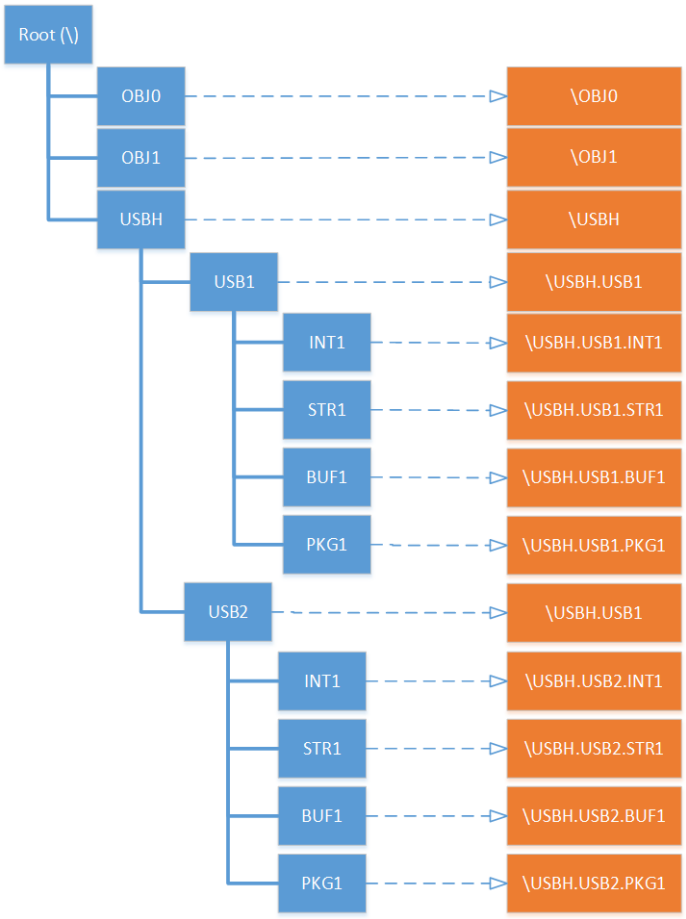
In the example above, `USBH` represents a device that serves as a parent device of USB1 and USB2. This hierarchy of named objects may be useful to describe large devices that have different components. For example, a USB controller may have many USB ports. Each port can be described as a separate device that is a part of the USB controller. This results in a namespace that looks like this:



Notice that USB1 and USB2 contain the same named objects. This is allowed because these objects are in a different scope. If the same named objects were defined under the same scope, this would result in a compiler error because it is illegal to declare two named objects that have the same NameSeg in the same scope. Similar to many other programming languages, multiple definitions of the same name within the same scope are not allowed in ASL.

3.2 NamePaths

The above namespace contains nested scopes, and there may be situations where other ASL code may need to refer to one of the objects underneath USB1. In order to do so, individual objects underneath USB1 can be referenced by their full pathname. Below is the same namespace annotated with full pathnames for each named object:



The `\` represents the root scope. Therefore, `\OBJ0` indicates that `OBJ0` is defined at the root scope. `USB1` is declared inside of `USBH` and can be referenced as `USBH.USB1`. The . (dot) operator is called the path separator. This indicates that `USB1` is in `USBH`'s scope. For any named object that defines a scope, the dot symbol indicates that the name segment before the symbol is inside the scope of the NameSeg after the dot symbol. For `INT1` declared in `USB1`, the full pathname would be `\USBH.USB1.INT1`.

4 Predefined Names

Once the ACPI namespace is populated, the OS will start searching for various named objects to initialize device drivers. In order to do so, the OS looks for named objects that have a predefined meaning and use the data as intended. So far, the names INT1, STR1, BUF1, PKG1, and other objects have been used in examples. These object names do not represent any useful information about devices; they were only used to illustrate ASL concepts. If an OS were to boot using the contents of the ACPI namespace in the USB example, the object names INT1, STR1, BUF1, and PKG1 are likely to be completely useless to a driver because the driver does not need the data associated with these NameSegs.

The semantics of objects with NameSegs starting with _ are predefined by the ACPI specification. For example, the `_HID` represents a hardware ID of a device. When this named object is defined under a device, it represents the ID associated with that device. According to the specification, `_HID` needs to be defined with data that is either a string or an integer. `_CRS` is a predefined name that returns a buffer that represents the current resource setting. Below we show an example of a device that has a plug and play (PNP) ID as well as a buffer that represents an empty resource.

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x1)
{
    Device (DEV1)
    {
        Name (_HID, 0x1234)
        Name (_CRS, Buffer() {})
    }
}
```

Note that a more realistic table would contain many other predefined named objects. A list of these predefined names and their meanings can be found in section 5.6.8 of the ACPI specification. In a typical operating system, _HID will be associated with a device driver for DEV1.

5 Executable ASL

The previous section described named objects and their data types as hard-coded values. Some named objects may need to be computed by performing operations on data or by performing certain actions. In this case, a named object should be defined as an ASL control method.

5.1 Processing Data with ASL Control Methods

Control methods (commonly referred to as methods) contain executable code that perform operations on ASL data. Here is an example of a DSDT containing a method that sets `INT1` to `0`.

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x1)
{
    Name (INT1, 0x1234) // define INT1 to be 0x1234
    Method (MTH1)
    {
        INT1 = 0x00 // store 0x00 to INT1
    }
}
```

When `MTH1` runs, it stores the value `0x00` to `INT1`. This value is persistent until the OS shuts down or another control method changes the value. In other words, if the OS evaluates `INT1` after evaluating `MTH1`, the value of `INT1` will be `0x00`. Notice that `INT1` is declared outside of the declaration of `MTH1`. Methods can refer to any named objects that exist outside of its scope.

Control methods have the following syntax:

```
Method (MethodName, NumArgs, SerializeRule)
{
    TermList // A list of ASL terms and expressions
}
```

- `MethodName`—Name of the control method object.
- `NumArgs`—Number of arguments. Arguments can be referenced as `arg0` through `arg6`.
- `SerializeRule`—States whether if this control method can be entered by multiple threads. This will be discussed in detail later.

The ACPI specification contains more details on optional parameters, but this definition will be used throughout this tutorial for simplicity. The definition of `MTH1` leaves out `NumArgs` and `SerializeRule`. If these parameters are left out they are set to `0` and `NotSerialized` respectively. `TermList` is a list of ASL terms that get executed when the method name is evaluated by the OS.

Putting everything together, a method is defined by a method name, the number of parameters, a serialization rule, and a list of ASL terms. The TermList is a list of ASL operators. In a control method, the TermList operators typically operate on named objects that are in the ACPI namespace, method arguments, or local variables. Within the TermList, a `Return` keyword can be used to denote a value to return to the caller of the method.

Methods are allowed to have up to seven arguments and eight local variables. Arguments are referenced as `Arg0` through `Arg6`, and locals are referenced as `local0` through `local7`. ASL methods can be called by other ASL methods or by the OS through the AML interpreter.

5.2 Basic ASL Operators

5.2.1 Hello World with `printf` and String Operators

As with many introductions to programming languages, the following is a way to print "`Hello world`" to a console during debugging:

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x0)
{
    Method (GRT1)
    {
        printf ("Hello world")
    }
}
```

The `printf` is similar to the one found in C. The format specifier `%o` used is for named objects. The following example will append the contents of `STR1` to `"Hello world"`:

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x0)
{
    Name (STR1, "it is a wonderful day")
    Method (GRT1)
    {
        printf ("Hello world, %o", STR1)
    }
}
```

5.2.2 Integer, logical, and bitwise operators

ASL provides a variety of integer, logical, and bitwise operators.

mathematical:

- + addition
- += addition assignment
- − subtraction
- −= subtraction assignment
- * multiplication
- *= multiplication assignment
- / division
- /= division assignment
- % mod
- %= mod assignment
- ++ post increment
- −− post decrement

bit-wise:

- « left shift
- «= left shift assignment
- » right shift
- »= right shift assignment
- | bitwise or
- |= bitwise or assignment
- & bitwise and

- `&=` bitwise and assignment
- `^` bitwise xor
- `^=` bitwise xor assignment
- `~` bitwise not
- `~=` bitwise not assignment

Logical:

- `&&` logical and
- `||` logical or
- `!` logical not
- `>` greater than
- `>=` greater than or equal to
- `<` less than
- `<=` less than or equal to

Boolean types do not exist in ASL. Instead, logical operators return integers greater than or equal to `1` as "true" and `0` as "false".

5.2.3 Control flow operators

ASL also supports control flow operators such as if, else, and while. The following example contains methods that combine the above operators:

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x0)
{
    Name (INT1, 0x02)

    Method (CTDW, 1)
    {
        local0 = arg0
        while (local0)
        {
            printf ("%o",local0)
            local0--;
        }
    }
    Method (CTUP, 1)
    {
        local0 = arg0
        while (local0 < 10)
        {
            if (!(local0 % 2))
            {
                printf ("%o is even", local0)
            }
            else
            {
                printf ("%o is odd", local0)
            }
            local0++;
        }
    }
}
```

5.2.4 Exercises

1. Write a method to determine if the input number is a prime. If the number is a prime, return the number. Otherwise return `0`.

   Note: `0` and `1` are not prime numbers.

2. Write a method to return the *n*th prime number.

   Hint: Use the method from the previous exercise.

5.3 String Operators

Strings in ASL are null-terminated arrays of ASCII characters. They are frequently used for the following cases: debugging messages, macros that convert strings to buffers, and predefined names.

As previous examples have demonstrated, the `printf` macro is used to print messages to the console. This macro is a convenient way to add debugging during development. `printf` can concatenate multiple strings into one message. The `Concatenate` operator can be used if strings need to be built prior to the `printf` invocation. In the following example, DBG1 and DBG2 result in the same output:

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x0)
{
    Name (STR1, "it is a wonderful day")
    Method (DBG1)
    {
        printf ("Hello world, %o", STR1)
    }
    Method (DBG2)
    {
        local0 = Concatenate ("Hello world, ", STR1)
        printf ("%o", local0)
    }
}
```

In addition to `Concatenate`, there are several useful macros that generate buffers from strings. For example, the `ToUUID` macro takes a string of the form `aabbccdd-eeff-gghh-iijj-kkllmmnnoopp` where `aa` through `pp` represent one byte values encoded with hexadecimal characters. This string gets converted to a 16-byte buffer that looks like the following:

```
Buffer()
{
    dd, cc, bb, aa,
    ff, ee,
    hh, gg,
    ii, jj, kk, ll, mm, nn, oo, pp
}
```

This mixture of little endian and big-endian encoding UUID is called a mixed-endian format. The use of strings and the `ToUUID` macro is a convenient way to avoid having to manually encode the mixed-endian format. There are many other macros that provide similar conveniences, such as `EISAID`.

5.4 Using Buffer and Package Objects

Recall that buffers are like C arrays where each element is a single byte. One way to access elements in a buffer is to use the index operator `[]`. Like arrays in C, buffers are indexed starting at 0. However, there is one catch: An index operator in ASL returns a reference to the buffer element, so

it needs to be dereferenced using the `DeRefOf` operator before it can be used. The example below shows a method that prints each element of the buffer passed as `arg0` to the console in userspace.

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x0)
{
    Method (IDXA, 1)
    {
        local0 = sizeof(arg0)
        local1 = 0 // use this as the index value
        printf ("The size of the buffer is %o", local0)
        while (local0 > local1)
        {
            printf ("%o", Derefof(arg0[local1]))
            local1++
        }
    }
}
```

Packages are similar to buffers except their elements can contain strings, integers, buffers, and other named objects. The above method works for both buffers and packages.

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x1)
{
    Method (OVFL, 1) // Causes a buffer overrun
    {
        local0 = sizeof(arg0)
        local1 = 0 // use this as the index value
        printf ("The size of the buffer is %o", local0)
        while (local0 >= local1)
        {
            printf ("%o", derefof(arg0[local1])) //runtime error
            local1++
        }
        printf ("Complete")
    }
}
```

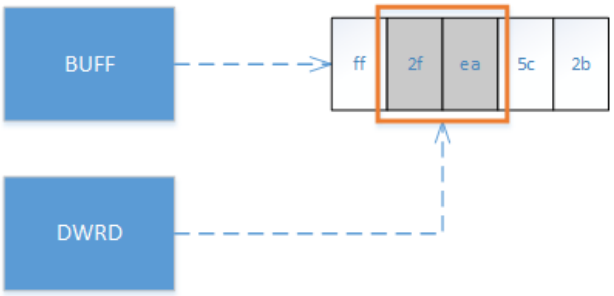
Unlike C, out-of-bounds accesses in buffers and packages in ASL result in runtime errors from the interpreter. In the example above, `OVFL` accesses the input buffer or package one past the max index when `local0 == local1`. When the runtime error occurs, the method execution is terminated. This means that `local1++` and `printf ("Complete")` are not executed after the runtime error.

5.4.1 Exercises

1. What happens when the `DeRefOf` operator is removed in `IDXA`?
2. Which component invokes the runtime error?
3. Write a method to return a buffer where the length of the buffer is passed as `arg0`.
4. Write an ASL method to compute the sum of all elements in a buffer. A buffer will be passed to this method as `arg0`.

5.5 Bit fields

Due to the low-level nature of ASL, it is often convenient to manipulate bits inside of buffers by assigning a name to a region of bits and performing operations on the region as if it were a named object. This saves the programmer from constantly indexing buffer elements.



The `CreateWordField` overlays a named object over `BUFF` that spans 2 bytes. This allows a simple approach to work with two contiguous elements of `BUFF`. It's also important to note that the CreateWordField operator adds the name `DWRD` to the ACPI namespace.

Here is the syntax:

```
CreateWordField (SourceBuffer, ByteIndex, FieldName)
```

- `SourceBuffer` –Buffer that the field will overlay
- `ByteIndex`—Starting bit offset within the source buffer
- `FieldName`—A NameSeg of this Field

Here is an example of two methods that perform similar computations:

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x0)
{
    Name (BUFF, buffer() {0xff,0x2f,0xea,0x5c})
    CreateWordField (BUFF, 0x01, WRD)

    //XCH1 and XCH2 will assign 0x00 to BUFF[1] and BUFF[2]

    Method (XCH1)
    {
        BUFF[1] = 0x00
        BUFF[2] = 0x00
    }

    Method (XCH2) // Index operator is not needed
    {
        WRD = 0x0000
    }
}
```

XCH1 and XCH2 result in the same outcomes. However, XCH2 incurs fewer instructions.

To provide more flexibility, ASL supports opcodes similar to `CreateWordField`. Below is a list of opcodes that create named objects that span the length of buffers with various lengths:

- `CreateBitField`—Length: 1 bit
- `CreateByteField`—Length: 1 byte
- `CreateWordField`—Length: 2 bytes
- `CreateDWordField`—Length: 4 bytes
- `CreateQWordField`—Length: 8 bytes
- `CreateField`—Length: Arbitrary

5.5.1 Exercises

1. What is the value of DWRD after XCH1 is evaluated?
2. Write a method to clear bits 5 through 12 of `BUFF` without using the opcodes discussed above. Write another method to do the same thing using `CreateField`.

5.6 ResourceTemplates

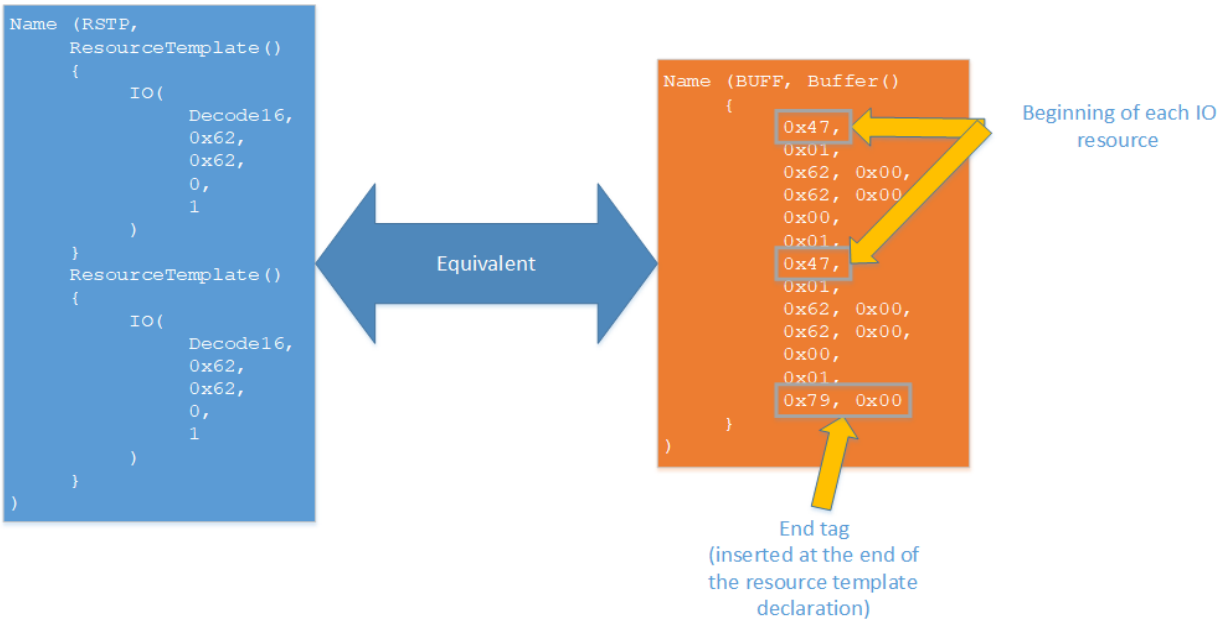
Buffers are useful for encoding information that describe resources used by device drivers such as IRQ, DMA, and I/O ports. Each of these resources has a particular format. The following buffer describes an I/O resource:

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x0)
{
    Name (RES1, buffer(){
        0x47, 0x01, 0x62, 0x00, 0x62, 0x00, 0x00, 0x01,
        0x47, 0x01, 0x66, 0x00, 0x66, 0x00, 0x00, 0x01,
        0x79, 0x00
    })
}
```

When a driver wants to understand the configuration of devices through ACPI, it calls the AML interpreter for a named object, which may be assigned to a buffer similar to `BUF1`. Programming this by hand in the above example can lead to mistakes because this notation lacks semantic meanings. If device resources need to be described, use the `ResourceTemplate` macro instead. By using the `ResourceTemplate` macro, the buffer can be encoded like so:

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x0)
{
    Name (RES2,
        ResourceTemplate(){
            IO(Decode16, 0x62, 0x62, 0, 1)
            IO(Decode16, 0x66, 0x66, 0, 1)
    })

}
```

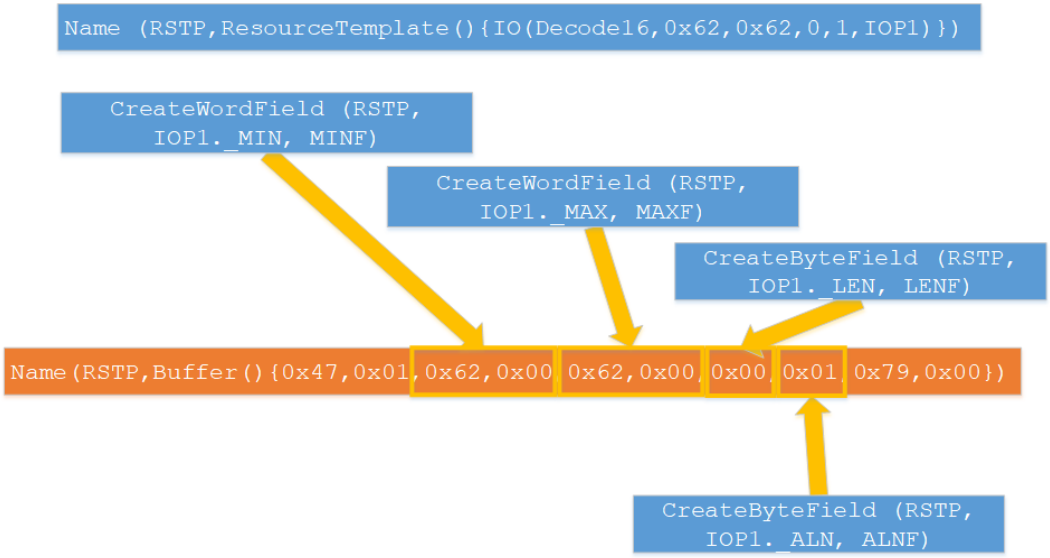
When compiled, RES2 will be translated during compilation to look exactly like RES1. For `RES1` and `RES2`, the translations are depicted in the image below.



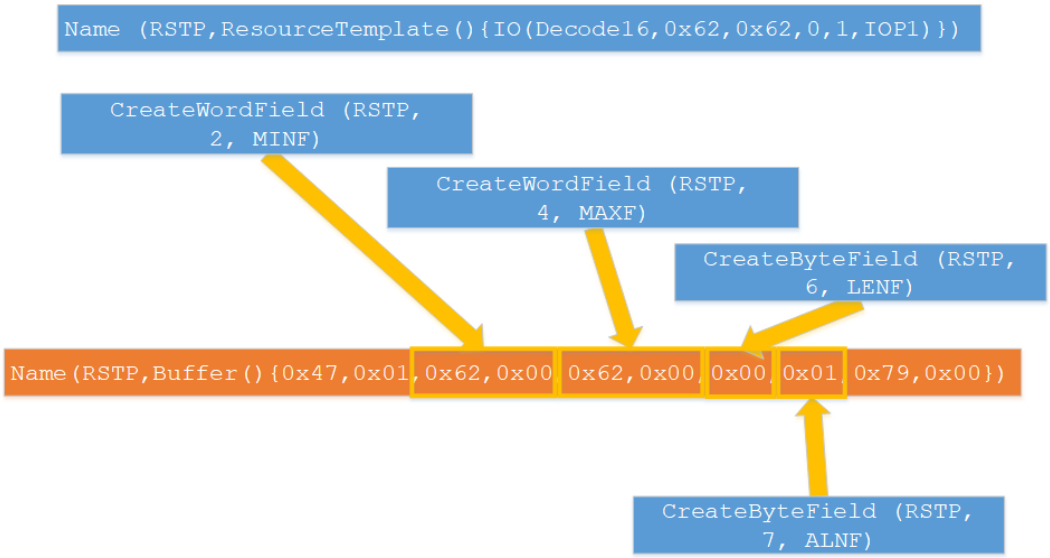
To read or write to elements of RES2, create bit fields that overlay a parameter and write to the bit field. In the above example, the second parameter of the first `IO` macro can be written like this:

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x0)
{
    Name (RES2,
        ResourceTemplate(){
            IO(Decode16, 0x62, 0x62, 0, 1, IOP1)
            IO(Decode16, 0x66, 0x66, 0, 1, IOP2)
    })

    Method (WRRT, arg1)
    {
        /*
         * According to the ACPI specification, the second
         * field of the IO macro is 16 bits. Therefore, use
         * CreateWordField to overlay a region that is 16 bits
         * in length.
         */
        CreateWordField (RES2, \IOP1._MIN, MINF)
        MINF = 0x1234 // This will write 0x1234
                      // to the second field in IOP1.
    }
}
```

As a cautionary note, IOP1 and IOP2 are macros that get transformed during compilation to integer values used to index into the ResourceTemplate. These labels do not get inserted in the ACPI namespace. However, RES2 gets inserted into the namespace, and the contents of IOP1 and IOP2 will be available for use by the driver.



In reality, `IOP1`, `IOP2`, and `_MIN` are macros that translate to integers that indicate an offset into `RES2`.



5.6.1 Exercises

1. Write a method to change `_MAX` of `IOP2` to `0xABCD`.

5.7 Control Method Calling Convention

ASL methods use the "call-by-reference-constant" calling convention. In this calling convention, method arguments are passed as references to reduce overhead in copying data. To reduce aliasing issues, control method arguments are essentially passed as constants and cannot be easily overwritten. One exception to this rule is when an argument contains references. In this case, the object value can be changed by dereferencing.

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x0)
{
    Name (INT1, 0x00)
    Method (CHNG, 1)
    {
```

```
        arg0 = 0x1234
    }
    Method (M001)
    {
        CHNG(INT1)
    }
}
```

In the table above, `M001` passes `INT1` to `CHNG` and `arg0` is incremented. However, the value of `INT1` will remain as `0x00`. Arguments can be overwritten if they are references. In order to access package or buffer elements, the index operator `[]` is needed. After evaluating `M001`, the value of `INT1` will remain 0x01. However, `PKG1` will contain `0x13`, and `BUF1` will contain `0x20`.

```
DefinitionBlock ("", "DSDT", 2, "", "", 0x0)
{
    Name (INT1, 0x01)

    Name (PKG1, Package() {INT1})
    Name (BUF1, Buffer() {0x0E})

    Method (CHNG, 1)
    {
        arg0[0] = 0x12 // overwrite
    }

    Method (M001)
    {
        CHNG (PKG1)
        CHNG (BUF1)
    }
}
```

5.7.1 Exercises

1. Write a method to increment all elements of a buffer or a package containing integers by 5. The buffer or package will be passed as `arg0`.

5.7 Other ASL Operators

There are many other types of ASL operators. They include:

- ASL Compiler Controls
- ACPI Table Management
- Named Object creation
- Synchronization
- Object References
- Integer Arithmetic
- Logical Operators
- Method Execution
- Data Type Manipulations
- Resource Descriptor Macros
- Constants
- Control method objects

To view a full list of ASL operators, consult section 19.5 of the ACPI specification.

6. ASL Programming Tips

6.1 Avoid Declaring Named Objects in Methods

In addition to executable procedures, named objects can be declared inside of methods. This is highly discouraged because any named object that is declared inside a method declaration only exists for the duration of the method execution. The named object gets inserted into the ACPI namespace and is removed from the namespace at the end of the method. This is a waste of computation for the AML interpreter, especially if the entire purpose of declaring a named object is for the OS to interact with it. If a named object is only present for the method execution, the only part of the ASL that can refer to the named object is the method itself. In order to store temporary values, use `local0` through `local7`.

6.2 Integer Size is Determined by ComplianceRevision of the DSDT

ASL integer sizes depend on the ComplianceRevision field of a DSDT. If this is greater than 1, the integer size will be 64 bits in size. Otherwise, integers are 32 bits in size.

Appendix:

ACPICA tools

ACPI Component Architecture (ACPICA) is an operating system independent open source software project that implements the ACPI specification. This project provides tools such as the iASL compiler/disassembler, acpiexec, acpidump, acpihelp, and other tools for firmware developers. These tools are used extensively throughout this tutorial, so take a moment to download and install the tools to do the exercises.

The project source can be downloaded at `https://github.com/acpica`. If you are compiling from source on a unix-like system, the project requires the following tools to build:

- gcc
- flex (or lex)
- bison (or yacc)
- make
- GNU m4

After installing these tools, go to the top-most directory of the ACPICA project and type `make && make install` to build and install the project tools.

On Ubuntu*, use the following sequence of commands in the terminal to download and install ACPICA user space tools:

```
sudo apt install git gcc flex bison make m4
git clone https://github.com/acpica/acpica.git
cd acpica
make && sudo make install
```

On a Windows* system, the project binaries can be downloaded from the ACPICA website here: https://acpica.org/downloads.