The Plan 9 AML Interpreter and ACPI support

Francisco J. Ballesteros
nemo@lsub.org

ABSTRACT

This is a technical note regarding the implementation of the Plan 9 AML interpreter and support for ACPI. It is intended as an aid for those who must change or understand its implementation.

Introduction.

ACPI is an standard for configuration and power management on the PC. In ACPI, the platform (motherboard and devices) provides tables that convey configuration information to the OS.

Besides the static information found in tables, ACPI configuration includes bytecode for a machine language known as AML. The OS is expected to execute AML code both to initialize ACPI and to trigger actions on devices (e.g., to change the power state). Also, some hardware interrupts may require that we should call AML to handle the implied events.

AML is the machine language for an abstract language known as ASL. It turns out that AML is more a representation of ASL than it is a language for a machine. In understanding what the interpreter must do it helps to look at ASL definitions for the operations considered. AML is close to them.

The language operates on objects that live in a tree-shaped namespace, global to AML. The AML ''machine'' does not have a general purpose stack. It operates on the name space and has ''builtin objects'' that represent arguments and local variables. Thus, thanks to the AML specification, a name may represent a method object, or may not; when it does, what follows in the code may be arguments for it. There is no operation for calling a routine. In general, it is not feasible to say if a bytecode is an operation or raw data.

The name space is initialized as a side-effect of loading `DSDT` and `SSDT` tables (and evaluating AML code on them).

Names are four characters, but searching rules for names are not what could be expected. For example, a relative name may be the resolved as the aunt of dot. As another example, names _OS_ and _OS are considered to be the same. The global `acpins` keeps the root of the name space.

A name refers to an object (perhaps none), and besides that, may or may not have children. So, names are both like files and directories, depending on the point of view. This is an example (portion) of a name space:

```
/_SB_/PBTN= dev <nil> pnp 'PNP0C0C' addr 0x0 {
   /_SB_/PBTN/_HID=  0xc0cd041
   /_SB_/PBTN/_PRW= pkg[2] { 0x8,  0x4}
   /_SB_/PBTN/_PSW= method _PSW/1
}
/_SB_/_INI= method _INI/0
```

Here, PBTN is both a device and the root for a (sub)tree of ACPI objects referring to PBTN. Syntax for names in ACPI is actually _SB_.PBTN and not /_SB_/PBTN. This implementation uses the more familiar UNIX syntax (with the required ACPI semantics).

There are different object types including strings, buffers, integers, references to other objects, buffer slices, register (bit) fields, memory (or I/O) regions, etc. An important object type is called a *method* in AML. It's not a ''method'', but a subroutine that may be called. In general, methods create objects in the name space and use them very much like local variables and, therefore, are not reentrant.

Execution of AML operations is performed by using (and modifying) objects in the name space. There are control operations, arithmetic operations, and operations on objects (roughly speaking). There is no operation to call a method. AML is not a language for an abstract machine, but a compact representation of a source language. Thus, evaluating AML is similar to interpreting source (ASL). For example, to call a method, the name is placed into the bytecode and then the arguments. Depending on the context, a name found in the bytecode stream may imply a method call.

Most operations copy objects and may lead to either explicit or implicit type casts, which must follow rules indicated on the spec. Beware that casts are in many cases not what could be expected. For example, obtaining an integer from a string differs from what *strtoul* would do. Buffers may be handled as integers or buffers depending on their size. Converting a buffer to a string and writing it into a register does not produce the same effect than writing the buffer. Etcetera.

To look for further information on ACPI, ASL, and AML it helps to consider:

- *Advanced Configuration and Power Interface Specification. 4.0. June 2009.*
  This is the ACPI specification. Chapter 5 is a description of the various ACPI tables, the namespace, and important objects. Chapter 18 is a description of ASL. Chapter 19 is a description of AML (or, rather, a description of how ASL is encoded into AML). Section 18.2.5 lists the different objects and type conversion rules.
- *OpenBSD ACPI implementation.*
  See /sys/dev/acpi in the system kernel source. Many operating systems use an Intel provide implementation for the interpreter. However, this one is easier to understand and can be used as additional documentation for ACPI and AML.

The AML interpreter is currently work in progress, and can be used as a user program to parse and execute AML code as dumped by a kernel during boot, using a temporary *acpi* device for such purpose.

When ready, the interpreter must execute at *links* time during boot, or anytime before configuring devices. By that time, there are no processes in the kernel.

Implementation

The important files are:

aml.h
: It defines the data structures used by the interpreter.

aml.c
: A driver for the interpreter. Most of it would go when in the kernel. It contains the main program and code to translate the dumps made by the kernel to binary data

that can be interpreted as AML bytecodes.

`amlrt.c`
The run time for the interpreter. Includes helpers and several important entry points to evaluate AML code.

`amlop.c`
Implementation for AML operations.

`amlconv.c`
Implementation for type casts and object copying. This part of the standard is a tangled web and its code is kept appart.

`kernel.c`
Compatibility tools to keep the interpreter code as ready for the kernel as feasible.

`acpi.c`
Contains tools that should be provided by the kernel ACPI driver. They are not part of the interpreter but may be necessary for this program. There is a version for the kernel in `9pc/acpi.c`, intended to replace this file when the interpreter gets into the kernel.

Names and objects

The global `acpins` keeps the root of the name space. This is the data type for a name:

```
struct Aname
{
        char    s[5];           /* NNNN  */
        char*   path;
        Aobj*   o;
        Aname*  parent;
        Aname*  childhd;        /* first child */
        Aname*  childtl;        /* last child */
        Aname*  sibling;        /* pointer to next sibling */
        Atable* table;          /* [ds]sdt table id; for unload */
        Aname*  mnext;          /* in list of per-method binds */
};
```

The list of children starts at `childhd`. The actual name is `s`, but the entire `path` is included as a convenience. The object bound to this name, if any, is kept at `o`.

The data type for an ACPI, or AML, object is this:

```
/*
 * ACPI Aobject.
 */
struct Aobj
{
        Ref;
        int     type;                           /* for this object */
        Aname*  name;                           /* ACPI name if bound or nil */
        Aobj*   next;                           /* in list */
```

```
        union{
                u64int          ival;           /* doubles as handle */
                char*           sval;           /* strings and names */
                Nbuf            buf;            /* byte buffer */
                Nslice          slice;          /* buffer slice (bits) */
                Nreg            reg;            /* memory, i/o, ... region */
                Nfield          field;          /* slice (bits) of object */
                Nref            oref;           /* local, ref, arg */
                Nlist           list;           /* packages */
                Nscope          scope;          /* for parsing */
                Nmethod         method;
                Ndev            dev;
                Npwr            pwr;
                Nproc           proc;
                Nthermal        thermal;
                Nmutex          mutex;
                Nevent          event;
        };
};
```

Objects may be kept in a list of free objects (when free) or in a list of objects contained in a *package* object (an ACPI array). They are reference counted (including as references those from names, from other objects, from the interpreter stack, etc.). Type may be:

```
Onone   =  0,
Ofree,                  /* debug */
Oslice,                 /* buffer field or slice */
Oscope,                 /* scope */
Omethod,                /* method */
Odev,                   /* device */
Oproc,                  /* processor */
Opwr,                   /* pwrrsrc */
Othermal,               /* thermal zone */
Oref,                   /* ref to object */
Olocal,                 /* ref to local object */
Oarg,                   /* ref to arg object */
Ohandle,                /* definition block handle */
Obuf    = 'B',          /* buf */
Oevent  = 'E',          /* sleep/wakeup */
Ofield  = 'F',,         /* register field */
Oint    = 'I',          /* integer */
Oname   = 'N',          /* a name */
Omutex  = 'M',          /* mutex */
Opkg    = 'P',          /* package object (array) */
Oreg    = 'R',          /* data region */
Ostr    = 'S',          /* string */
Odecstr = 'D',          /* like string, but bytes from buffer */
                        /* values are printed in decimal */
Oany    = 'O',          /* any object after resolving names */
```

Onone is uninitialized. Ofree is used while the object is on the free list (for debug checks). Oslice is called a *field* in ACPI, but there are three or four types of fields it is easier to understand this object as a slice of a buffer. We use Ofield for fields that identify bit-slices of other fields or system memory and I/O spaces. Opkg is actually an array.

A slice value (buffer field in ACPI) is defined as

```
/*
 * buffer field. slice of bits in buffer.
 */
struct Nslice
{
        Aobj*   src;            /* source Obuf object */
        uintptr off;            /* starting bit index */
        uintptr len;            /* number of bits */
};
```

We keep offset and length in bits because `Nslice` is used to represent several ACPI field types with bit/byte/... granularity. A general purpose `bitcpy` function in the interpreter is used to move bytes when feasible and bits otherwise.

An address space region (memory, I/O, PCI, etc.) is represented by `Nreg`:

```
struct Nreg
{
        int     spc;            /* io space type */
        u64int  base;           /* address, physical */
        uchar*  p;              /* address, kmapped */
        uintptr len;
        uint    tbdf;           /* pci only */
};
```

All other field types in ACPI are represented by `Nfield`. This represents a slice (bits) of a region or another field and is the main object used to perform I/O.

```
/*
 * bit field for a region, or banked-register bit field, or
 * (index,data) register bit field.
 */
struct Nfield
{
        char*   fname;          /* field name */
        int     accsz;          /* access sz in bits for source*/
        int     locking;        /* needs locking */
        int     update;         /* how to update */
        uintptr off;            /* in bits */
        int     len;            /* in bits */
        Aobj*   reg;            /* region the field is for */
        Aobj*   bank;           /* bank name (for banked field) or nil */
        Aobj*   bankval;        /* bank value (for banked field) */
        Aobj*   idx;            /* index reg. (for index field) or nil */
        Aobj*   idxval;         /* value for index */
        Aobj*   data;           /* data (for index field) or nil */
};
```

To make things uniform, it has bit granularity so we may forget in general if it is a bit or byte aligned field. `Accsz` dictates the number of bits to I/O at a time. `Update` dictate what to do with remaining bits in the source object upon writes to the field (preverve them by reading them before, write as one or write as zero). There are two kinds of fields:

Index fields

A slice of a `data` register available after setting an index register, `idx`, to a value.

Region fields

A slice of a region, perhaps requiring a write to a `bank` register before using.

Scope objects are not ACPI objects. They are used to represent a scope (also introduced by an AML *scope* operation) as an aid in parsing.

```
struct Nscope
{
        char*   name;
        uchar*  pe;             /* end of AML for it */
};
```

A method including information about the number of arguments and the portion of AML implementing the method's body.

```
struct Nmethod
{
        char*   name;
        int     nargs;          /* arity */
        int     isexcl;         /* is serialized? */
        int     synclvl;
        int     trace;          /* 1: on; -1: off; 0: as you were */
        uchar*  ps;             /* aml for this method */
        uchar*  pe;             /* end of aml */
        void    (*f)(Aobj*);    /* C implementation for builtins */
};
```

There are other objects that should be either self-explanatory or uninteresting by now.

Devices

Devices include information about resources for them. They are kept in al list of `Res` items.

```
/*
 * ACPI device as seen by drivers.
 */
struct ACPIdev
{
        char*   name;
        char*   pnpid;          /* pnp id */
        u64int  addr;           /* address */
        Res*    res;            /* current resource settings */
};

struct Ndev
{
        ACPIdev;
        uchar*  pe;
        void    (*handler)(Aobj*, int); /* for events */
        void*   aux;
};
```

A resource is described by this data type:

```
struct Res
{
        Res*    next;           /* in resource list */
        int     type;           /* resource type */
        union{
                struct{
                        u32int  irqs[16];
                        uint    flags;
                        int     idx;    /* index in src */
                        char*   src;    /* where resources come from */
                }irq;
                struct{
                        uint    chans;
                        uint    flags;
                }dma;

                struct{
                        u64int  min;    /* address of the range */
                        u64int  max;    /* address of the range */
                        uint    align;
                        int     isrw;   /* memory only, ro or rw? */
                }io, mem;

                struct{
                        u64int  mask;   /* which bits are decoded */
                        u64int  min;    /* address on the other side */
                        u64int  max;
                        u64int  off;    /* translation adds this */
                        int     idx;    /* index in src */
                        char*   src;    /* where resources come from */
                        int     flags;
                        int     rev;    /* acpi revision id or 0 */
                        u64int  attr;
                }as;
                Gas field;

                /* non ACPI resources; invented by us to keep them here */
                struct{
                        u64int  addr;
                        int     pin;
                        char*   src;
                        int     idx;
                }prt;                   /* pci intr. routing, from _PRT */
        };
};
```

This is a ‘‘compact’’ representation of the ACPI data structures for resource settings, parsed by the implementation. A cleaner structure should be used instead, once we know which pieces of information are really necessary.

There are other objects for processors, thermal zones, mutexes, etc. Just keep in mind that they are not devices in ACPI.

The interpreter

The AML interpreter is a table driven implementation. It is a single thread that loops fetching operation codes and executing operations according to them. Operation codes may be one or two bytes. Arguments depend not only on the operation code, but also on the state of the interpreter.

A table, amlops, provides the implementation for each 1-byte operation code. Another table, amlxops, provides the implementation for, so called, extended operation codes. This is an excerpt:

```
[OpZero]        {opnum,         "zero",         Dry, ""},
[OpOne]         {opnum,         "one",          Dry, ""},
[OpAlias]       {opalias,       "alias",        0, "On"},
[OpName]        {opname,        "name",         0, "nO"},
[OpMid]         {opmid,         "mid",          0, "OIIt"},
```

Each entry is defined by this type:

```
typedef void (*Opx)(void);
/*
 * AML operation
 */
struct Op
{
        Opx     x;              /* implementation */
        char*   name;           /* debug */
        int     flags;          /* call on dry runs? Print { } on dumps? */
        char*   args;           /* arguments, for AML parsing */
        int     ncalls;         /* how many times used */
};
```

The implementation assumes that each operation is responsible for building an object (in general) and may require arguments to perform its task. A string in each entry encodes argument processing so that, after fetching an operation, the pargs function may parse as much AML as needed to build argument objects as dictated by the argument string. For example, opalias requires an evaluated object and a name, as encoded by the string ''On''.

In the argument string, a lowercase refers to an object of a particular type with no evaluation performed on it. An uppercase refers to an object that, after evaluation, has a particular type. For example, ''S'' means to parse an object, evaluate it, and obtain a string object (perhaps by doing a type cast to string). On the other hand, ''n'' means to parse an object that should be a name.

These characters may be used to specify arguments:

```
Obuf    = 'B',          /* buf */
Oevent  = 'E',          /* sleep/wakeup */
Ofield  = 'F',,         /* register field */
Oint    = 'I',          /* integer */
Oname   = 'N',          /* a name */
Omutex  = 'M',          /* mutex */
Opkg    = 'P',          /* package object (array) */
Oreg    = 'R',          /* data region */
Ostr    = 'S',          /* string */
Odecstr = 'D',          /* like string, but bytes from buffer */
                        /* values are printed in decimal */
Oany    = 'O',          /* any object after resolving names */
```

These other characters, quoted from the implementation, may be used but do not correspond to any object type as known by the interpreter (they build objects, but do not represent different object types):

```
/*
 *      - 'l' means packagelen
 *      - 'b' means byte, 'w' word, 'd' d-word,
 *      - 'f' fieldlist
 *      - 's' string
 *      - 'n' name or arg or local
 *      - 't' target (name, perhaps created, dst object, nil)
 *      - 'o' any object, maybe a name.
 * Upper case args are evaluated by pargs to obtain the typed arg.
 * Other args are not evaluated before calling the operation.
 */
```

But for the pointer to the implementation for each operation and the argument string, the only interesting information in the operation table is a flag to see if the operation must be called on dry runs or not and a flag to help pretty-print (sic) AML for debugging.

The main entry point for the interpreter is

```
char*
amleval(uchar *ps, uchar *pe, Aobj **po, Name *n, Atable* table, int flags)
```

which evaluates AML code between pointers `ps` and `ps` using `n` as the namespace dot for the evaluation. It returns an error string and the resulting object in `*po`.

Another function evaluates a single call to a method object (just a subroutine):

```
Aobj*
amlcall(Aobj *mo, Aobj **args, Atable *table, int flags)
```

It accepts arguments for the method call and returns a result object.

Both functions are implemented by calling `pamlblock`, which parses (and evaluates) an AML block. `Pamlblock` is mostly a loop that calls `pamlobj`, which parses (and evaluates) as much AML as needed to build a single object.

Depending on the flags given to the evaluator, it may just dissasemble AML, execute it, print actions as they are performed, and perform various debug checks:

`Edryrun`
: asks for a dry run. Usually combined with other flags that dump information.

`Eidump`
: asks for instruction dumps. This is close to a disassembler.

`Exdump`
: asks for messages about actions performed.

`Escheck`
: asks for full stack checks to ensure that reference counting is correct and to see if objects seem to be correct. This slows down execution a lot, but is useful after making changes to the interpreter, because it is easy to make mistakes regarding object references.

When the interpreter finds a method definition is skips its implementation and only records where the code starts and its length. Method evaluation is performed by using `pamlblock` on that part of the program.

Environments

`Env` represents an evaluation environment at a particular scope. Multiple `Env`s are nested as a result of entering and leaving scopes in the AML program. This happens both to implement actual AML scopes and also to open temporary scopes to save the current environment (e.g., during argument evaluation). All `Env` structures live in the C

stack.

A global variable, amlenv, points to the current (i.e., last, or inner-most) environment. This Env can be considered the state of the abstract machine and is implicit to all AML operations.

```
struct Env
{
        char*   name;           /* debug */
        Env*    prev;           /* in env stack */
        int     ic;             /* instruction code */
        uchar*  ps;             /* start of what's left of program (PC) */
        uchar*  pe;             /* end of program */
        int     lvl;            /* env nesting level (debug) */

        Op*     op;             /* current op */
        uchar*  opp0;           /* operation address 0 */
        Nlist   olist;          /* list of obj in env */
        Aobj*   args[Nargs];    /* arguments for op */
        int     nargs;          /* max number of args used */
        uchar*  argpe;          /* pe resulting from arg 'l' */
        Aobj*   res;            /* result from op */

        Mframe* mp;             /* Method frame pointer */
        Aname*  dot;            /* current acpi name */
        Atable* table;          /* [ds]sdt table id */
        int     flags;
};
```

As said, the implementation looks up an operation code at a time and processes its arguments as indicated by an argument string. Some arguments are simple and are parsed directly. Others are not and may change the environment while they are evaluated. In this case, a nested environment is pushed to protect the environment for the operation (so that its implementation may assume that the environment corresponds to the operation and has not been altered much during argument processing).

When an operation is called:

amlenv->op
: points to the operation being executed (its table entry).

amlenv->ic
: is the instruction code for the operation (the second byte for two-byte operations).

amlenv->ps
: is the program counter (the start of what's left of the program). When the operation includes an argument string, it points past the AML code for the arguments. Upon return, it must point past the last byte of AML code for the operation (including arguments and body, if any). N.B.: the implementation for opwhile violates this assumption for a good reason.

  Operations with implementations that are also used to parse arguments should look at amlenv->ps[0] and not to amlenv->ic to see their IC. That is because the IC is assumed to correspond to the operation itself and not to its arguments.

amlenv->pe
: is the end of the program. Operations may include a length argument that constraints the program size (e.g., to define a portion of code as the *then*-arm of an *if* instruction).

amlenv->opp0
: is the program counter for the current operation (the zero address for its AML

code). This is important to opwhile.

When operations with argument strings are called:

amlenv->ps
: is past the arguments.

amlenv->args
: includes pointers to argument objects for the operation.

amlenv->argpe
: is the end of the program according to the length argument. The operation should not touch AML code past it.

Upon return from an operation:

amlenv->res
: points to the object returned from the operation (perhaps nil).

Arguments and locals are closed when the operation returns. When the environment is terminated (pop) or the next operation executes, amlenv->res is closed. Thus, operations that don't have to return a value may leave an object linked there, so it is collected upon errors.

Environments used to evaluate methods include a pointer to an Mframe.

```
/*
 * Method activation frame.
 */
struct Mframe
{
        Aobj*   args[Nmargs];   /* method arguments */
        int     nargs;
        Aobj*   lcls[Nmlcls];   /* method locals */
        int     nlcls;
        Aobj**  targs;          /* tmp. method args while parsing them */
        Aname*  binds;          /* done by this method call */
};
```

This is just a place to store method arguments and locals, as well as a place to keep the list of names bound by a method call (which must be unbound upon returns). A method call pushes a new Env (in the C stack) that points to a new Mframe (in the C stack) via mp. Further scopes entered inherit the mp pointer from the parent Env.

Error handling

Remember that there are no processes by the time the interpreter runs. Errors are reported using an error stack, similar to that used by processes. This is an example:

```
static void
oppackage(void)
{
        Env env;
        ...
        amlpush(&env, amlenv->op->name, nil, amlenv->argpe);
        if(wasamlerror()){
                amlpop();
                amlerror(nil);  /* pass a string to raise it */
        }
        pamlblock(Keep);
        ...
        popamlerror();
        amlpop();
}
```

Operations `break`, `continue`, and `return` raise their names and rely on this exception mechanism for their implementation.