Fakultät Informatik

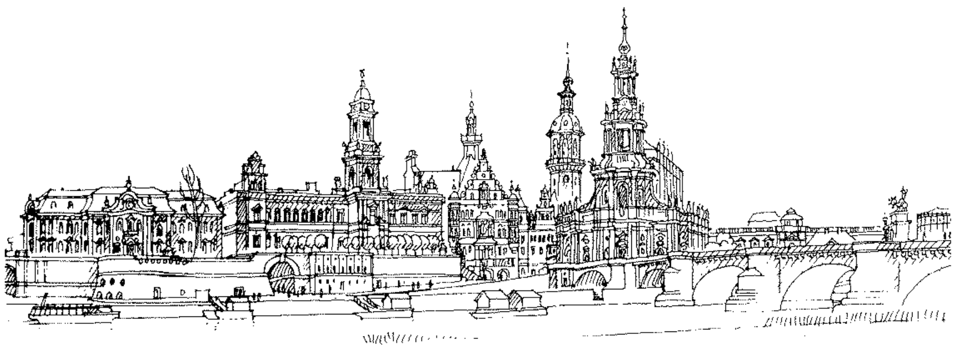
Technische Berichte
Technical Reports
ISSN 1430-211X

TUD-FI09-09 August 2009

Bernhard Kauer

Institut für Systemarchitektur

ATARE: ACPI Tables and Regular Expressions



ATARE: ACPI Tables and Regular Expressions

Bernhard Kauer
Technische Universität Dresden
Dresden, Germany
bk@vmmon.org

ABSTRACT

Operating system drivers often need platform specific knowledge to work correctly. One example might be the routing of the interrupt lines on todays PC platforms. The canonical way to get these information is to rely on ACPI [1], a standard for platform configuration and power management.

In OS our environment where power management is not an issue and a minimal TCB is one of our goals, we are not willing to provide a complete ACPI implementation. Nevertheless ACPI seems to be the only source for the configuration of these platforms.

To bridge the different requirements we found an surprisingly simple heuristic based on regular expressions that can extract information from ACPI tables. Our experiments show that our heuristic can provide the IRQ routing of the platform in all of our test cases with a code size that is approximately two orders of magnitude smaller than a full ACPI interpreter.

1. INTRODUCTION

The ACPI machine language (AML) [1, 2] is a domain specific language for hardware device configuration and power management. As AML is Turing complete and quite powerful, evaluating ACPI tables written in AML is a complex task.

In many cases only a fraction of the information available in AML is actually needed. One example is the routing of PCI interrupt lines [14] to interrupt controller (I/O-APIC [8]) pins, an information needed by drivers to get the right IRQ. This platform specific knowledge is pretty much based on the chipset as well as the board wiring. During boot-time of a system or in an OS without power management, PCI IRQ routing can be the only mandatory information that needs to be extracted from the ACPI tables. Utilizing and maintaining an AML interpreter for this task seems to be disproportional for getting a single number per PCI device.

We therefore searched for alternatives and came up with a radical new approach based on regular expressions that can evaluate AML.

The paper is structured as follows: The next section we describe other approaches to the IRQ routing problem. In the third section we argue why we want to avoid an AML interpreter. After that we explain how we came up with pattern matching in AML and gives an heuristic to extract the IRQ routing out of AML which is much simpler than a full AML interpreter. In the fifth section we evaluate how many cases we can cover with it and analyze the reasons why it is successful. The last section concludes the paper with further work.

2. HOW ARE IRQS ROUTED?

The simplest approach to get the IRQ routing of a PCI device is to read it from the PCI config-space. Unfortunately this works only with the legacy PIC [7] mode. Using them would give up most of the gains from the newer I/O APIC architecture such as reduced IRQ sharing and fine-grained IRQ routing to different processors [10]. Furthermore acknowledging an IRQ can be quite slow on a PIC. Sticking with the PIC mode is therefore not an option for us.

Message Signaled Interrupts (MSI) [3] are the latest alternative to interrupt controllers and dedicated IRQ lines. A PCI device with MSI support can be programmed to directly send messages containing an interrupt vector to the CPU. With MSIs IRQ routing is not an issue anymore as any interrupt vector can be directly specified. Unfortunately MSI support is not available on many devices and even broken on some platforms. Linux 2.6.27.4 for example disables MSI in 24 cases [9]. MSIs will be the future, but an OS can not solely rely on them today.

There are other sources for IRQ routing. One could for example rely on the MP configuration table, as defined in the Multi-Processor Specification [11]. It consists of a much simpler format than AML but it is deprecated since the first ACPI specification was published in 1999. Today PCs are sometimes shipped without an MP table. Another hypothetical option would be to build our own IRQ routing database into the OS. As collecting these information for every motherboard type is a tremendous task, this idea can be easily dismissed.

Name		LOC
Linux 2.6.27.4	drivers/acpi	50295
	include/acpi	7325
FreeBSD head	contrib/dev/acpica	44621
	dev/acpica	12878
ACPICA 20080926	binary: 98.2k	73840

Figure 1: ACPI implementations: Lines of Code

In summary there is no good alternative to ACPI as the authoritative source to get the IRQ routing of a platform. Unfortunately the routing is not contained in static ACPI tables, such as the MADT, which are relatively easy to parse. Instead it is embedded in AML byte-code of the System Description Tables (DSDT and SSDTs). To get this information an OS has to be able to (at least partially) evaluate AML.

3. AVOIDING AN AML INTERPRETER

In the previous section we have shown that ACPI can not be ignored and that evaluating AML is therefore mandatory. In this section we argue that utilizing a fully featured AML interpreter is not a good idea when AML provides only minimal information such as IRQ routing, which can be the case during boot-time or in an OS without power management support.

The current ACPI specification v3.0b [2] has more than 600 pages, defines 13 ACPI tables and references 19 externally defined ACPI tables. It specifies AML (ACPI machine language) as domain specific language for System Definition Tables. Furthermore it contains a corresponding source language (ASL), a hierarchical name space and dozens of AML methods that needs to be provided from the platform vendor. An OS evaluates these methods to get and set the platform configuration and to perform power management tasks.

Table 1 gives lines of code for major parts of the ACPI implementations in Linux and FreeBSD. As both are based on ACPICA, its size is also given as reference. Please note that ACPICA also includes various tools such as an ASL compiler that are not strictly needed for an OS. When looking at the numbers we can understand that the "Linux/ACPI subsystem is large and complex" [4]. In summary more than 50000 lines are needed for a full ACPI implementation.

Interpreting AML has also some security implications. As shown by Heasman [6] implementing a rootkit in AML is possible. Sandboxing the AML interpreter could be one solution to this problem. Windows for example blocks access to some IO-ports [15], not for security but for availability reasons. Secure Sandboxing of ACPI that does not exclude to many platforms is a research challenge on its own.

We conclude that interpreting ACPI tables is a complex problem, that leads to large code and can pose a security threat. We therefore search for a simpler approach.

4. PATTERN MATCHING IN AML

To understand how we came up with using regular expressions for pattern matching on AML and why it works, we have to dig deeper into AML/ASL and how the IRQ routing information is represented in it.

```
Method (_PRT, 0, NotSerialized)
{
    If (GPIC)
    {
        Return (Package (0x01)
        {
            Package (0x04)
            {
                0x0014FFFF,
                0x02,
                0x00,
                0x12
            },
        })
    }
    else
        Return (Package (0x01)
        {
            Package (0x04)
            {
                0x0014FFFF,
                0x02,
                \_SB.PCI0.LPC0.LNKC,
                0x00
            },
        })
}
```

Figure 2: A _PRT method in ASL.

ACPI code is written in ASL (ACPI source language) and later compiled to AML (ACPI machine language) a more compact byte-code representation with the very same features. So even if a compiler is used to convert ASL to AML and a decompiler to convert it back there is a 1:1 correspondence between these two languages.

ASL has many features: It knows named objects, methods that could be called on them and control structures such as `if`, `while` and `return` that are used to implement methods. It supports different types such as strings, variable sized lists called packages and numbers. Different operations could be done on these types. Numbers for example support the usual arithmetic and bit-operations on them. In summary AML/ASL is a Turing-complete domain-specific language specially tailored for platform configuration and power management.

4.1 IRQ Routing

IRQ routing information is returned in ACPI by calling the `_PRT` method of a PCI bridge. This returns a variable sized package of IRQ mappings for devices on its bus. An IRQ mapping is itself a four element package with the following fields:

1. PCI device address
2. IRQ pin (0 - `#IRQA`, 1 - `#IRQB`,...)
3. a name of a PCI IRQ router
4. a GSI (global system interrupt) number if the third field is zero

```
NAMESEG = "[A-Z_][A-Z_0-9]{3}"
NAMERE  = "((\\\\|\\^*)(("+NAMESEG + ")
           |(\x2e" + NAMESEG +  NAMESEG + ")
           |(\x2f.("+NAMESEG+")*)))"
PKGLEN  = "(([\x00-\x3f])
           |([\x40-\x7f].)
           |([\x80-\xbf]..)
           |([\xc0-\xff]...))"
DATA    = "([\x00\x01\xff]
           |([\x0c]....)
           |([\x0b]..)
           |([\x0a].))"
DEVICES = "\\[\x82"+ PKGLEN + NAMERE
METHODS = "[\x14]" + PKGLEN + NAMERE
SCOPES  = "[\x10]" + PKGLEN + NAMERE
IRQMAP  = "[\x12]" + PKGLEN +
          "[\x04]("+ DATA + "{4})"
DEFNAME = "[\x08]" + NAMERE +
           "(" + DATA + "|([\x12]" + PKGLEN + "))"
```

Figure 3: Regular Expressions for AML

Figure 2 shows a slightly modified real-world _PRT method. Depending on the value of the GPIC variable, which distinguish in this example whether the OS runs in I/O-APIC (if) or PIC mode (else), it returns different IRQ mappings. The difference between both cases is obvious: in the I/O-APIC case the IRQ pin is routed to GSI 0x12 whereas in the PIC case it is routed to an IRQ router named LNKC. As we are interested to which input pin of an I/O APIC an IRQ line of the PCI device is connected, we can ignore the second case.

4.2 A Pattern

There is a pattern in pretty much all DSDTs that we can also see here: The IRQ mappings for the I/O APIC do not facilitate an IRQ router instead fixed GSI values are hard coded in the ASL. It should therefore be possible to find a similar pattern in AML.

Our first approach to find this pattern in AML was unsuccessful. To search the _PRT method in the byte-code we tried to exactly parse the AML elements and skip all the unneeded ones. Unfortunately there is not a clear hierarchy between different element types. It is possible that a calculation of some value of a variable is directly followed by a definition of a new method. To find the beginning of the method we had to be able to skip pretty much every different element of AML. This turned to be impractical, so we had to throw away the idea of an accurate AML parsing.

4.3 A Sparsely Encoding

A deeper look in the AML definition revealed that distinct binary ranges where often used for different elements. Figure 3 lists all the regular expressions in Python RE-syntax [12] that we use. The first four ones are just shortcuts for definition reasons, the last five regular expressions in this list are the ones used to do the search.

Name segments for example are always four characters long and start with an uppercase letter or an underscore. The last three chars can also contain digits. Another example would be the data that is found in the four element IRQ mapping packages we are interested in. It consists either of one of

```
fail if there is no _PIC method
for every _PRT method:
   search bridge device which contains this _PRT
   search for _ADR, _BBN, _SEG of the bridge
   look for IRQ mappings contained in the _PRT method
   if nothing found:
       search for references in the _PRT method
       for every reference:
           get corresponding object
           search IRQ mappings in the object
   output _ADR, _BBN, _SEG and IRQ mappings
```

Figure 4: Pseudo-code of a heuristic to search for IRQ routing information

the characters {0,1,255} or it starts with one of {12,11,10} followed by a {4,2,1} byte-wide integer respectively. The pattern called DATA in Figure 3 reflects this example.

AML is densely packed binary data but still contains enough "space" in its encoding. This allows us to apply regular expressions to search for specific AML elements. Furthermore the variable sized objects, such as METHODS or DEVICES directly contain its length. Whether one object is contained in another one is therefore easily decidable.

In summary IRQ routing information for the I/O APIC is often hard coded in a repeating design pattern in ASL. We need only 5 regular expressions to match this pattern in the corresponding AML byte-code.

4.4 Searching for IRQ Mappings

We can use the regular expressions listed in Figure 3 to search for IRQ mappings. A plain search for packages that look like IRQ mappings is not sufficient. We also have to output the PCI bus address. The _ADR, _BBN and _SEG numbers of the bridge device reveal the device address, the bus and segment number respectively.

To avoid false positives we have to make sure that IRQ mappings are returned from a _PRT method. We therefore search for IRQ mappings contained in it.

Not all IRQ mappings can be found this way. ASL writers sometimes put the package of IRQ mappings outside the actual _PRT method and just reference them via a name. To cope with that case we follow the references and search for IRQ mappings in them, too. Our tests have not shown any usage of indirect references, even if ACPI seems to allow this.

Figure 4 lists the pseudo-code of the search. We immediately fail if there is no _PIC method present. The _PIC method is needed to tell ACPI that the OS switched from the default of using the PIC to I/O-APIC mode. The absence of it is a good indicator that the ACPI table is too old to have support for an I/O-APIC.

compile error	108
no I/O-APIC	270
good case	456
IRQ router	17
Sum	851

Figure 5: DSDTs tested

4.5 Implementation

We implemented a prototype of the heuristic in Python. We choose Python instead of C or another low-level language, because of its good rapid prototyping property. Our implementation currently consists of 108 lines of python code.

A low-level C port is currently work in-progress. It will be a stand-alone component executed from the boot loader before the OS runs. It will parse the DSDT and SSDTs and overrides the part of the PCI config space that normally holds the PIC IRQ with the I/O APIC pin number. This interface seems to be much simpler than inventing a new data-structure or reconstructing an old one. The only disadvantage seems to be that the rarely used PCI hotplug will not work.

Implementing regular expression in C can be a lengthy task. A scanner generator such as re2c [5] can ease the implementation a lot. Another solution would be to link against a libregex, trading some performance for simplicity. Regular expression libraries can be quite small, the version in dietlibc for example consists of less than 500 SLOC.

5. EVALUATION

In this section we try to estimate the success probability of our approach in the real-world, explain on what we depend to be successful and illuminate reasons for failure and how we cope with them.

5.1 Measure our Approach

We used 851 real-world DSDTs from a now discontinued DSDT repository [13]. The oldest entries date back to 2003 while the latest ones were added in November 2007. Collecting and testing newer DSDTs is left as a future task.

Figure 5 shows the results. From the original 851 DSDTs 13% or 108 had compile errors since they were submitted in the wrong format such as hex dumps or C-code. Around 32% of the DSDTs where not I/O-APIC aware. They either didn't include a `_PIC` method or they simply ignored it. This left 456 good cases and 17 ones where our heuristic failed. Please note that DSDTs repeat in the database as users were uploading updated versions. A list of the different vendor and version strings showed that from the 456 DSDTs only 190 have a unique vendor and version.

We manually inspected the 17 failed ones and figured out that they relied on IRQ routers that also allow to route IRQs to I/O APICs. This is possible by reusing outputs of the PCI IRQ routers that would normally clash with legacy I/O devices such as the PIT (IRQ 1) or the RTC (IRQ 8) and connect them to I/O APIC inputs that are normally unavailable. Fortunately at boot-time the IRQ routers are configured for PIC mode.

So by using a fallback when the IRQ extraction failed and recover the IRQ router configuration from the PCI config space, we can cope with these 17 cases. Because we are not able to change the routing we are probably left with some shared IRQs and unused I/O APIC pins.

In more than 95% of the cases our heuristic extracts the IRQ routing from AML. A simple fallback results in a quality degradation for the missing cases but not in a failure. In summary our approach is successful in all known cases.

5.2 Successful and Simple

We have seen that our approach is successful and with 5 regular expression also quite simple. In this section we give a couple of reasons that reveals on what our approach depends. They can also be used to judge similar cases.

A repeating design pattern People tend to be lazy. A reason may be that BIOS developers never use these ACPI tables but have to adopt them for every board revision.

Simple hardware IRQ routing is often static, thus hard-coded values are sufficient.

Minimal information We need only a very small number of words (in average 132 bytes) out of a big datastructure (in average 22k), thus 5 regular expressions are enough.

A sparse encoding There is no complicated state-machine for decoding AML. A sparse encoding can be matched with a simple expression.

Known structure sizes All structures have a known size or the size is encoded directly at the beginning. This allows to easily decide which belongs to whom in the hierarchy.

The first point is based on peoples behavior, it is the weakest point in our heuristic. The second point is a hardware property that can change over time if new devices emerge. It is mainly driven by cost reasons of the hardware manufacturer. The last three points were decisions by the designers of the domain specific language. Our use case was surely never intended by them, but we benefit here from language properties that are unlikely to change.

5.3 Failure Handling

Our search is a heuristic and can therefore fail. One reason may be the use of indirect references: a `_PRT` method calls a method that calls another method to return the IRQ mappings. A second reason would be if the list of IRQ mappings is not static, but a template that is filled on call time. A third reason would be if an ACPI table contains superfluous entries, that are never reached in an interpreter due to AML's control structures [1]. Our code would output multiple entries for a single PCI device without any hint which one is the right one. None of these degrees of freedom provided by AML where actually observed in our tests.

[1]Example: `if (0) ...`

A failure of our heuristic is easily detectable and results in DoS of the system: drivers will not get interrupts and the system will probably not boot anymore. This can only happen at defined points in time: either this is a new motherboard or a BIOS update has changed the ACPI tables. A virus can also overwrite the ACPI tables. This would render them unusable for us as well as for a full ACPI interpreter. In summary the effects of a failure is quite limited and no security implications are involved.

One obvious reaction to new cases not handled by our heuristic is to put more knowledge into the regular expression, thereby extracting more context information from AML. In the end this could mean we need to fully understand AML and therefore will be as complex as a full AML interpreter and have not gained anything from this approach. This will surely never happen. This assumption is backed by the fact that 95% of the current cases are happy with a static IRQ routing. So why should an AML writer make the code more complicated as it needs to be?

6. FUTURE WORK AND SUMMARY

There are a couple of things that should be done in the future:

- show that our heuristic can be efficiently implemented in a non-scripting language such as C,
- collect and test newer ACPI tables,
- apply regular expressions to more parts of AML, to extract the I/O-port regions of the legacy serial ports or to evaluate power management functions, and
- research whether other implementations of domain specific languages can also be benefit from regular expressions.

ACPI handling is a mess. There are two extremes of coping with ACPI: either ignoring it or implementing a full featured AML interpreter. We developed a radical new approach that resides between these two. By using regular expressions to evaluate AML we can reduce the complexity of the implementation by approximately two orders of magnitudes, thus making it more maintainable and likely more secure. We implemented a prototype that extracts IRQ routing from ACPI tables and showed that it handles all our test cases. We think that this approach has further applications for AML as well as for other domain specific languages.

APPENDIX

A. REFERENCES

[1] ACPI - Advanced Configuration & Power Interface. URL: `http://acpi.info`.

[2] Advanced Configuration and Power Interface Specification, Revision 3.0b. URL: `http://www.acpi.info/DOWNLOADS/ACPIspec30b.pdf`, 2006.

[3] J. H. Baldwin. PCI Message Signaled Interrupts. URL: `http://people.freebsd.org/~jhb/papers/bsdcan/2007/article/node7.html`, 2007.

[4] L. Brown. ACPI in Linux - Myths and Reality. In *Proceedings of the Linux Symposium, Ottawa, Ontario, Canada*, 2007.

[5] P. Bumbulis and D. D. Cowan. Re2c: a more versatile scanner generator. *ACM Lett. Program. Lang. Syst.*, 2(1-4):70–84, 1993.

[6] J. Heasman. Implementing and Detecting an ACPI Rootkit. In *BlackHat Federal*, January 2006.

[7] Intel. i8259 - Programmable Interrupt Controller (PIC). URL: `http://en.wikipedia.org/wiki/Intel_8259`, 1988.

[8] Intel. I/O Advanced Programmable Interrupt Controller (I/O APIC) Datasheet. URL: `http://www.intel.com/design/chipsets/datashts/290566.htm`, 1996.

[9] Linux 2.6.27.4 PCI quirks. URL: `http://lxr.linux.no/linux+v2.6.27.4/drivers/pci/quirks.c`, 2008.

[10] Microsoft. Key Benefits of the I/O APIC. URL: `http://www.microsoft.com/whdc/archive/IO-APIC.mspx`, 2001.

[11] MultiProcessor Specification. URL: `http://www.intel.com/design/pentium/datashts/242016.htm`, 1997.

[12] Python Library Reference - Regular Expression Syntax. URL: `http://www.python.org/doc/2.5.2/lib/re-syntax.html`, 2008.

[13] Linux/ACPI Project - DSDT repository. URL: `http://acpi.sourceforge.net/dsdt`.

[14] T. Shanley and D. Anderson. *PCI System Architecture*. Addision Wesley, forth edition, 1999.

[15] Windows XP: I/O Ports Blocked from BIOS AML. URL: `http://www.microsoft.com/whdc/archive/BIOSAML.mspx`, 2003.