Erledigt "Du hast ja Alles" - hmmm vielleicht, wenn ich einen Laptop habe.

Beitrag von „Brumbaer“ vom 8. Februar 2018, 16:10

Illuminati
Hintergrundbeleuchtung, ist noch so eine Sache mit der ich mich für gewöhnlich nicht rumschlagen muss. Nach meinen bisherigen Erfahrungen mit dem Miix befürchte ich, dass das eher in die Richtung "Ins Dunkle zu treiben" als in die Richtung "hoch im Licht" geht.

Und wieder beginnt es mit "das geht ganz einfach". Eine SSDT, um ein PNLF Device zu erzeugen, ein Kext in den Other Ordner und gut ist. Meine Reaktion auf solch optimistische Worte ist ein wissendes, aber gequältes Lächeln.

Ich bin bei der Recherche über zwei Kexte gestolpert eins arbeitet über ACPI und eins über den Intelchipsatz, beide benötigen ein PNLF Device und beide gelten als überholt. Es gibt allerdings eine Patch Anleitung von rehabman.
Ich habe die SSDT installiert und zwei Kexte ausprobiert, eins crashte, das Intel basierte funktionierte überraschenderweise.
Dann erinnerte ich mich, dass es in Clover ein AddPNLF Häkchen gibt, also SSDT raus und Häkchen an, geht immer noch.
Dann hab ich das Kext auch noch rausgeschmissen und es geht immer noch.
Dann, und dann, dann war ich verblüfft.
Hintergrundbeleuchtung durch Clover Häkchen. Weil ich gerade solch eine Erleuchtung hatte, gleich noch die Tastenkombination für die Tastaturbeleuchtung ausprobiert. Die Tastaturbeleuchtung ist scheinbar eine reine Tastaturfunktion und geht deshalb ohne Zutun.
Direkt im Anschluss stieß ich einen lauten Schmerzensschrei auf, weil ich mir auf die Kinnlade getreten bin, die hat ja auch nichts auf dem Boden verloren. Frau und Kinder kommen angerannt, vom Schmerzensschrei alarmiert. Aber ich umarme sie nur und wir tanzen ausgelassen einen Ringelrein. Noch schnell die Nachbarn auf ein Glas Schampus eingeladen und die Feuerwerksreste von Sylvester verschossen und den Tag im Kalender ganz dick angestrichen. Alles um das Ereignis gebührend zu feiern. Und euch Zweiflern und Kleingläubigen sei gesagt, es gibt auch Dinge bei Installation von macos auf dem Miix, die ganz leicht von der Hand gehen. Na ja, da es um den Miix geht, sollte ich vielleicht erst mal die Finger nachzählen.

Rückblende

Nach so einem positiven Erlebnis ist es Zeit einmal innezuhalten und an Vergangenes zu denken. Dazu gehört das _LID Problem. Wir erinnern uns, beim Lesen des Klappenstatus klappte der Mac zusammen und nichts ging mehr. Nun hatten wir gerade die DSDT gepatched, weil macos beim Lesen vom 16 Bit Werten aus dem EmbeddedControl Speicherbereich Schwierigkeiten hatte. Vielleicht, aber auch nur vielleicht, ist das Geklapper ja auch so was. Also schauen wir uns die _LID Methode einmal an.

Code

```
1. Method (_LID, 0, NotSerialized) // _LID: Lid Status
2. {
3. If (LEqual (ECRD (RefOf (LSTE)), One))
4. {
5. Return (Zero)
6. }
7. Else
8. {
9. Return (One)
10. }
11. }
```

Alles anzeigen

Die Methode heißt *_LID* und hat 0 Parameter, so weit, so gehöft. Die Methode selbst ist eher übersichtlich.
ECRD macht etwas mit *LSTE* und wenn das Ergebnis 1 ist, wird 0 zurückgegeben ansonsten 1.
Die zwei Rückgabebefehle mit Konstanten und das *else*, sollten keine Problem machen.
Das lässt uns nur die *If*-Abfrage. *ECRD* ist kein Standard AML(ACPI Machine Language) Befehl, muss also eine Methode sein. *ECRD*, klingt wie "EmbeddedControllerReaD" oder "Elf Clowns Rasen Davon".
RefOf ist ein AML Befehl und entspricht in etwa dem Adresse-Operator in C. *RefOf(LSTE)* ist dann sowas wie ein Zeiger auf *LSTE*. *RefOf* wird verwendet, wenn man ein Objekt an eine Methode übergeben und sicher gehen will, dass der Wert des Objektes erst in der Methode ermittelt wird. Das ist wichtig, wenn sich der Wert zwischen Aufruf und Abfrage in der Methode, z.B. durch Optimierungen oder asynchrone Ereignisse, ändern könnte. *RefOf* erzeugt einen fatalen Fehler, wenn es das Objekt nicht gibt. Ein kurzes Suchen in der *DSDT* findet *LSTE*.
LEqual überprüft ob seine beiden Parameter gleich sind.
Wenn es hier irgendwo knallt, dann muss es in der *ECRD* Methode passieren, oder die Jungs haben doch noch einen übergebliebenen Böller gefunden. Schauen wir nach was *ECRD* macht.

Code

```
1. Method (ECRD, 1, Serialized)
2. {
3. Store (DerefOf (Arg0), Local0)
4. Return (Local0)
5. }
```

Weniger ist mehr

Das ist wirklich nicht viel. *DerefOf* behandelt den Parameter als Zeiger auf ein Objekt und bestimmt dessen Inhalt. Mit *Store* wird dieser in einer lokalen Variablen gespeichert und diese zurückgegeben. Somit liest *ECRD* den Inhalt des Objektes auf das sein Parameter zeigt und gibt ihn zurück. Eine Methode um den aktuellen Wert einer Variablen zu bestimmen. Bei einer normalen Variablen würde man das nicht machen, also handelt es sich vermutlich um ein Hardware Register oder etwas Ähnliches, dass sich jederzeit außerhalb der Programm-Kontrolle ändern könnte.
DerefOf klappt zusammen, wenn es das Objekt auf das sein Argument zeigt nicht gibt. Aber *LSTE* gibt es.

Code

```
1. Schnipp
2.
3.
4. OperationRegion (ECF2, EmbeddedControl, Zero, 0xFF)
5. Field (ECF2, ByteAcc, Lock, Preserve)
6. {
7. XXX0, 8,
8. XXX1, 8,
9. XXX2, 8,
10. Offset (0x14),
11. VCMD, 8,
12. VDAT, 8,
13. VSTA, 8,
14. Offset (0x20),
15. RCMD, 8,
16. RCST, 8,
17. Offset (0x60),
18. TSR1, 8,
19. TSR2, 8,
```

20. TSR3, 8,
21. TSI, 4,
22. HYST, 4,
23. TSHT, 8,
24. TSLT, 8,
25. TSSR, 8,
26. CHGR, 16,
27. Offset (0x6A),
28. Offset (0x6B),
29. Offset (0x6C),
30. Offset (0x6D),
31. Offset (0x6E),
32. TSRT, 8,
33. Offset (0x72),
34. CHGT, 8,
35. NPST, 8,
36. PCVL, 8,
37. Offset (0x7F),
38. LSTE, 1,
39. Offset (0x80),
40.
41.
42. Schnapp

Alles anzeigen

Alte Bekannte

LSTE ist ein Feld in einem *Field*-Befehl der Operationregion *ECF2*. Die kennen wir inzwischen so gut, dass ich sie zum Sektumtrunk hätte einladen können. Wichtig daran ist, dass *LSTE* demzufolge im Speicherbereich *EmbeddedControl* liegt und der bei 16 Bit Zugriffen Schwierigkeiten macht.

Auf die Länge kommt es an

Aber *LSTE* ist nur 1 Bit lang und nicht 16. Als Arbeitshypothese behaupte ich, dass Apple nicht nur mit Feldern, die länger als 8 Bit sind Probleme hat, sondern mit allen Feldern, die nicht 8 Bit lang sind.

Wenn dem so wäre, müssten wir den Zugriff auf *LSTE* so ändern, dass ein ganzes Byte, statt nur eines Bits, gelesen wird und die anderen 7 Bits verworfen werden.

Wir könnten nun eine Methode schreiben, die ein Byte liest, so wie wir eine Methode geschrieben haben, die ein Wort (16 Bit) byteweise liest (*RDWD*) - oder wir lesen statt eines Bytes einfach ein Wort und werfen 15 statt 7 Bit weg. Da letzteres weniger Schreibarbeit ist

und wir schon wissen, dass die Methode zum 16 Bit Lesen funktioniert, wählen wir diese.
Wir brauchen für RDWD die Byteadresse an der das erste zu lesende Byte steht. Direkt vor *LSTE* steht *Offset(0x7F)* und schon haben wir die die Byteadresse von *LSTE*.
RDWD(0x7F) gibt einen 16 Bit Wert zurück indem *LSTE* enthalten ist. LSTE hat eine Länge von einem Bit und seine Deklaration folgt direkt hinter einem *Offset* Befehl. *LSTE* ist also das erste Bit (Bit 0) in dem Byte an der Adresse 0x7F. Wir lesen 16 Bit der Adresse 0x7F, aber da *LSTE* das erste Bit an dieser Adresse ist, ist es auch das erste Bit im Wort.

Maskenball
Um den Wert einer bestimmten Bitkombination aus einem Wert zu ermitteln, verwendet man ein Verfahren namens Maskierung. Die Maske entspricht der Binärzahl die man erhält, wenn man die interessanten Bits auf 1 setzt. Wir haben 16 Bit und uns interessiert Bit 0. Wie bei anderen Zahlen stehen links die höherwertigen Stellen, Bit 0 ist also ganz rechts 0b0000000000000001. Wandelt man das in eine Dezimalzahl erhält man 1.
Beim Maskieren führt man eine *Und*-Verknüpfung zwischen Maske und Wert aus. Alle Bits, die in der Maske eine 0 haben, werden zu Nullen und die anderen Bits nehmen den selben Zustand wie im Ausgangswert an.
And (RDWD(0x7F), 1) liest die 16Bit aus dem Controller und löscht alle Bits außer dem einen (die Maske für Bit 0 ist 1), das uns interessiert. Das Ergebnis des *And*-Befehls ist 0 oder 1, ganz so als ob wir nur das eine Bit gelesen hätten.

Wir ändern unsere _LID Methode in

Code

```
1. Method (_LID, 0, NotSerialized) // _LID: Lid Status
2. {
3. If (LEqual (And (RDWD(0x7F), 1), One))
4. {
5. Return (Zero)
6. }
7. Else
8. {
9. Return (One)
10. }
11. }
```

Alles anzeigen

DSDT.aml geändert, gespeichert und neugestartet. Und ... vergessen den Patch in der *Config.plist* abzuschalten ... Auf ein Neues, Neustart, im Clover Menü, Optionen, *ACPI, Patch*

abgeschaltet und .... Tadaah, bootet, kein Crash, kein Crash, kein Crash.
Klappe schließen und der Bildschirm wird dunkel. Kein Sekt mehr da, so ein Pech, Böller sind auch alle, also nur ruhige gelassene Heiterkeit statt ausgelassener Feierstimmung.
Ach ja, da wir wissen, dass es funktioniert, können wir den eben erwähnten Patch in Clover abschalten oder entfernen.

Irgendwas ist immer

Und irgendwo zwischen Verblüffung und heiterer Gelassenheit stelle ich fest, dass die NVME SSD Schwierigkeiten macht und manchmal hängen Maus und Tastatur beim Systemstart. System auf USB Medium hat scheinbar keine Probleme, wenn denn Maus und Tastatur gehen.
Habe ich wohl was an der DSDT verbastelt. Also alte und neue DSDT verglichen.

Alles zu seiner Zeit

DSDTs und *SSDTs* liegen in zwei Formaten vor. Als *aml* oder als *dsl* Datei. *Aml* Dateien enthalten ausführbaren, maschinenlesbaren Code. *Dsl* Dateien enthalten nicht ausführbaren, menschenlesbaren Code. Wenn man eine Datei im *dsl* Format im patched Ordner ablegt, passiert nichts, denn sie enthält keinen ausführbaren Code.
Wenn man eine *aml* Datei in *MaciAsl* öffnet, erzeugt es intern eine *dsl* Datei, zeigt sie an und lässt sie uns editieren und beim Speichern erzeugt sie aus der internen *dsl* wieder eine *aml* Datei. Das erweckt den Eindruck, dass eine *aml* Datei und eine *dsl* Datei für Menschen gleich leicht zu lesen seien. Dem ist nicht so.
Wenn man zwei *DSDTs* oder *SSDTs* vergleicht, sollte man das mit den *dsl* Versionen tun, denn was Unterschiede in den *aml* Versionen bedeuten, erkennt nur der Geübte. Deshalb vor dem Vergleich die *aml* Dateien mit *MaciAsl* in *dsl* Dateien umwandeln. *Dsl* Dateien sind normale Textdateien man kann somit die Text-Vergleichs-App seiner Wahl verwenden.
Wenn man keine hat, kann man *FileMerge* verwenden. *FileMerge* findet man in *Xcode* im *XCode Menü* unter *Open Developer Tool*. *FileMerge* dient dem zusammenfügen von Dateien, zeigt aber auch die Unterschiede an.
Natürlich habe ich erwartet, die von Hand gemachten Änderungen zu sehen. Aber da gibt es auch auch unerwartete Änderungen.

Theorie ...

Irgendwo im *BIOS* ist eine *DSDT* gespeichert und diese wird dann an *Clover* und von dort an *macos* übergeben. Ändert man was an den *BIOS* Einstellungen, wird eine neue DSDT erstellt. Speichert man eine eigene *DSDT* im patched Ordner, ersetzt *Clover* die *BIOS-DSDT* durch die eigene. Das ist kein Problem, solange die Änderung einer BIOS Einstellung nicht eine Änderung

in der *DSDT* bewirkt. Denn diese Änderung wird nicht automatisch in die eigene *DSDT* übernommen. Dann passen verwendete *DSDT* und BIOS Einstellungen nicht mehr zueinander.

... und Praxis

Das ist das Konzept. Die Realisation kann im Detail anders aussehen. Z.B. spricht nichts dagegen, dass das BIOS die *DSDT* bei jedem Start komplett neu erstellt. Was bleibt ist das Problem, dass wenn sich etwas in der vom BIOS bereitgestellten *DSDT* ändert, die Änderung nicht in der gepatchten *DSDT* wider gespiegelt wird.

Hier haben wir solch einen Fall. die Adresse der Operationregion *GNVS* unterscheidet sich. D.h. sie kann sich ändern und dann stimmen die Adressen aus der *BIOS DSDT,* die sagt wo *GNVS* tatsächlich gespeichert ist, und der gepatchten *DSDT*, die sagt wo *GNVS* gespeichert wurde als die Kopie der *DSDT* gemacht wurde, nicht mehr überein und die Zugriffe auf *GNVS* liefern Müll.

Die Schreiber von Clover wissen um das Problem und deshalb gibt es die Option *FixRegions* in den *ACPI Fixes*.

"Haaaahh, Haaaahh ! Nimm das du Schurke !", denke ich mir und setze die Option. Neustart, Crash. So leicht lassen sich Schurken wohl nicht aufhalten.

Plan B

Das Problem lässt sich lösen indem man die *DSDT* nicht patched, sondern alle Änderungen in einer neuen *SSDT* zusammenfasst und dort von der *BIOS DSDT* aufrufen lässt.

Aber um in der *DSDT* etwas in der *SSDT* aufrufen zu können, muss ich die *DSDT* patchen, "Katze, Schwanz, und so".

Die Lösung sind Clover DSDT Patches. Diese werden auf die aktuelle *DSDT* angewandt. Ohne *DSDT.aml* in *patched* Ordner wird das die zum BIOS passende sein. Mit *Clover* nennt man dann die Methoden um, so dass sie nicht mehr in der *DSDT* gefunden werden. Dann woanders gesucht und hoffentlich in unserer *SSDT* gefunden werden.

Klappe, die dritte

Nehmen wir als Beispiel die *_LID* Routine.

Wir packen sie in eine *SSDT*.

Das Gerüst einer *SSDT* sieht wie folgt aus:

Code

```
1. DefinitionBlock ("", "SSDT", 2, "VonMir", "Brumbaer", 0x00000001)
2. {
3. }
```

MEINER und *Brumbaer* sind zwei Strings zur Identifikation des Autors und der Tabelle.

Der Autor darf sechs Zeichen lang sein, der Tabellenname acht.

Da Brumbaer 8 Buchstaben lang ist, habe ich ihn als Tabellennamen verwendet. Kein guter

Stil. Aber solange man nicht zwei *SSDTs* mit dem Tabellennamen *Brumbaer* anlegt, geht das. Die 1 am Ende ist die Versionsnummer, die 2 davor schaltet in den 64Bit Mode.

Kopiert man die *_LID* Routine in die neue *SSDT* rein, erhält man:

Code

```
1. DefinitionBlock ("", "SSDT", 2, "VonMir", "Brumbaer", 0x00000001)
2. {
3. Method (_LID, 0, NotSerialized) // _LID: Lid Status
4. {
5. If (LEqual (And (RDWD (0x7F), One), One))
6. {
7. Return (Zero)
8. }
9. Else
10. {
11. Return (One)
12. }
13. }
14. }
```

Alles anzeigen

Alles ein Frage des Kontextes

Wenn man von Herrn Meier redet, weiß keiner welcher Herr Meier gemeint ist. Es sei denn man erwähnt Herrn Meier in einem bestimmten Kontext, wie in einer Geschichte über das Büro. Oder man man gibt mehr Details: Herr Meier im Büro, Herr Meier, wohnhaft in ....

So geht es uns mit unserer *_LID* Methode. Die *_LID* Routine gehört "in" den momentanen Kontext auf Englisch Scope. Wenn wir wollen, dass die *_LID* Routine zu unserem *LID0* Gerät gehört, müssen wir entweder den Kontext dorthin verschieben oder *_LID* mit Wohnort, PLZ, Straße und Hausnummer angeben. Wir entschliessen uns den Kontext, den Scope, zu verschieben. Der Befehl dazu heißt *Scope*, Zufälle gibt's. Der *Scope*-Befehl hat einen Parameter, der angibt wohin der *Scope* verlagert werden soll. *Scope(LID0)* klingt plausibel, ist aber nur korrekt, wenn der *Scope* vorher schon auf dem Objekt stand zudem *LID0* gehört. Um das wasserdicht zu machen, geben wir als Parameter für den *Scope*-Befehl, die komplette Adresse inklusive Plz. und Hausnummer an. Allerdings müssen wir sie erst finden.

In der Seitenleiste zeigt *MaciAsl* einen Objekt-Baum. Wählt man ein Objekt (*_LID*) an, wird es im Objekt-Baum markiert. Wenn man nun im Objekt-Baum nach oben wandert und die einzelnen Stufen mit Punkten zu einem Namen verknüpft bekommt man:

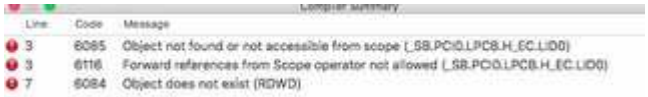
_SB.PCI0.LPCB.H_EC.LID0._LID
_LID ist die Methode und wir suchen das Objekt zu dem sie gehört bzw. die neue gehören soll. Also ist das Objekt *_SB.PCI0.LPCB.H_EC.LID0*

Code

```
1. DefinitionBlock ("", "SSDT", 2, "VonMir", "Brumbaer", 0x00000001)
2. {
3.
4.
5. Scope (_SB.PCI0.LPCB.H_EC.LID0)
6. {
7. Method (_LID, 0, NotSerialized) // _LID: Lid Status
8. {
9. If (LEqual (And (RDWD (0x7F), One), One))
10. {
11. Return (Zero)
12. }
13. Else
14. {
15. Return (One)
16. }
17. }
18. }
19.
20.
21. }
```

Alles anzeigen

Sofortiges Kompilieren führt zu sofortigen Fehlermeldungen.



_SB.PCI0.LPCB.H_EC.LID0 ist kein Standardobjekt laut ACPI Spezifikation, und deshalb kennt der Compiler es nicht.
Für die Methode *RDWD* gilt das Gleiche.

Code

```
1. DefinitionBlock ("", "SSDT", 2, "MEINER", "Brumbaer", 0x00000001)
2. {
3. External (_SB_.PCI0.LPCB.H_EC.LID0, DeviceObj)
4. External (_SB_.PCI0.LPCB.H_EC.RDWD, MethodObj)
5.
6.
7. Scope (_SB.PCI0.LPCB.H_EC.LID0)
8. {
9. Method (_LID, 0, NotSerialized) // _LID: Lid Status
10. {
11. If (LEqual (And (RDWD (0x7F), One), One))
12. {
13. Return (Zero)
14. }
15. Else
16. {
17. Return (One)
18. }
19. }
20. }
21. }
```

Alles anzeigen

Die erste *External*-Anweisung sagt dem Compiler, dass das Objekt *_SB_.PCI0.LPCB.H_EC.LID0* ein Gerät (Device) ist und dass es irgendwo anders definiert wurde.
Die zweite *External*-Anweisung sagt dem Compiler, dass das Objekt *_SB_.PCI0.LPCB.H_EC.RDWD* eine Methode ist und dass sie irgendwo anders definiert wurde.
Der Pfad vor dem *RDWD* (*_SB_.PCI0.LPCB.H_EC*) ist komplett in dem Pfad des Scope-Befehls (*_SB.PCI0.LPCB.H_EC.LID0*) enthalten, deshalb genügt es *RDWD* in der *_LID* Routine zu schreiben.
Wäre das nicht der Fall hätte man

Code

```
1. Schnipp
2. If (LEqual (And (_SB_.PCI0.LPCB.H_EC.RDWD (0x7F), One), One))
3. Schnapp
```

schreiben müssen. Über Scope ließe sich mehr sagen, aber ich verweise interessierte an die

ACPI Spezifikation.
Jetzt müssen wir noch dafür sorgen, dass unsere neue *_LID* Methode auch verwendet wird.

Raus aus die Kartoffeln, rin in die Kartoffeln

In der *DSDT* ist ja schon eine *_LID* Methode und die muss weg, damit unsere neuere, bessere und schönere Methode verwendet wird. Deshalb verwende wir einen Clover DSDT Patch, um die *_LID* Routine in der *DSDT* umzubenennen, denn *Clover DSDT Patches* können nicht löschen.
Den Patch haben wir im dritten Artikel gemacht und in diesem Artikel weiter oben, wieder rausgeworfen. Jetzt holen wir ihn wieder rein.

Für mich sieht ein _LID aus, wie das andere

Die *_LID* Routine gibt es nur einmal. Was mache ich denn, wenn es eine Routine mehrmals gibt, wie *_DSM* oder *_CRS* ?
Es gibt in der *Clover DSDT Patches* Tabelle eine Spalte *TgtBridge*. Dort kann man eine Kennung eines Gerätes eintragen auf das der Patch begrenzt werden soll. Auch wieder als Hexzahlen. *TgtBridge* ähnelt dem Scope-Befehl. Dummerweise ist die Funktionalität eingeschränkt, da das nur funktioniert, wenn das zu ersetzende Etwas innerhalb des *Device*-Befehls der *TgtBridge* steht. Leider werden *Scope*-Befehle ignoriert.

Bei den Battery Patches bietet es sich an die gesamten *_BST* und *_BIF* Methoden auszulagern. Wichtig ist immer, das man nichts auslagert, was sich wechselnde *OperationRegion*-Befehle enthält.
Und siehe da, alles funktioniert noch und die NVME und Maus/Tastatur-Probleme beim Start sind weg.

Eine Folge hab ich noch.