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

Beitrag von „Brumbaer“ vom 6. Februar 2018, 22:51

Das Leben geht weiter
Dumdidum war gestern. Das Oberwasser ist zurückgegangen.
Jetzt, da ich eine Netzwerkverbindung habe, kann ich die Bildschirmfreigabe verwenden.
Ich mache das gerne, denn ich arbeite am liebsten an meinem großen Rechner.
Der Bildschirm ist da wo ich ihn gut sehen kann, die Tastatur ist do, wo ich gut tippen kann und das Trackpad ist da, verd. .. wo ist mein Trackpad ?
Wiedergefunden, in einem ordentlichen Haushalt kommt halt nichts weg.
Außerdem lassen sich mit der Bildschirmfreigabe mal schnell Dateien kopieren. In diesem speziellen Falle hat die Bildschirmfreigabe den zusätzlichen Vorteil, dass Tastatur und Maus beim ferngesteuerten Rechner nicht per USB angebunden sind. Nach einem Wake funktionieren Tastatur und Maus nicht mehr, obwohl die Geräte noch in der IORegistry stehen. Das externe USB Laufwerk geht interessanterweise noch, BT scheint auch nicht zu gehen. Also mal schnell die Kennung der USB Ports auf extern gestellt und die Gleichstellung von internen und externen USB Anschlüssen belegt.
Das böse Erwachen lässt sich auch nicht durch das Dunkle Erwachen beeinflussen.
Hibernatemode, sollte nichts ändern, aber man kann's ja mal versuchen. Man versucht's und es ändert nichts.
Der Unterschied zwischen in- und externen Gerät ist, dass das externe Gerät ein eigenes Netzteil hat und nicht am internen Akku hängt. Ich will jetzt nicht hängen bleiben, weder am Akku, noch einem Sleep/Wake Problem, also zur Seite geschoben.

Problemkind
Ich habe auf mehr als zehn Motherboards macos installiert und noch nie solch eine Anhäufung von Problemen erlebt, sollte irgendetwas bei der Installation auf dem Miix einfach nur so funktionieren, mache ich ein dickes Kreuz in den Kalender.

(Duracell)Hase oder Meister Lampe ?
Batterie oder Hintergrundbeleuchtung ? Hintergrundbeleuchtung sollte einfacher sein und ich bin für jedes Erfolgserlebnis dankbar. Aber man bekommt so selten, was man sich wünscht, also Batterieverwaltung.

Nichts einfacher als das

"Da legst du das Kext vom Rehabman in den Other Ordner und fertig", habe ich verstanden, ist ja ganz einfach. Nach dem Kext geggoogelt, gedownloaded und in den Other Ordner gepackt, Neustart, Energie sparen öffnen. Ladezustand 0% und ein clicken auf *Batteriestatus in der Menüleiste anzeigen* setzt den Haken nicht. Sieht nach Brandschutzklasse B1 aus, selbstlöschend.
Ich merke schon, das wird heute kein Spass.

Wer lesen kann, ist klar im Vorteil
Rehabmans Artikel gelesen und da steht irgendwo, dass man die DSDT patchen muss, wenn Felder in der Operationregion des Embedded Controllers länger als 8 Bit sind.
Also einen Blick in die DSDT werfen.

Darin Steht Dieser Text

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. ECWR, 8,
41. XX10, 8,
42. XX11, 16,
43. B1DC, 16,
44. B1FV, 16,
45. B1FC, 16,
46. XX15, 16,
47. B1ST, 8,
48. B1CR, 16,
49. B1RC, 16,
50. B1VT, 16,
51. BPCN, 8,
52. Offset (0xC0),
53. MGI0, 8,
54.
55.
56. Schnapp

Alles anzeigen

Treiber greifen auf die Register von Schaltkreisen (Hardware-Register) zu. Die Register sehen nach außen hin aus wie normale Speicherzellen. Aber tatsächlich spiegeln die Bits und Bytes Schaltzustände des Schaltkreises wieder und/oder lösen Vorgänge und Abläufe aus statt sich nur Werte zu merken. Diese Register stehen für gewöhnlich an festen Adressen. So hat jede PCIe Karte einen für ihre Zwecke reservierten Speicherbereich. In diesem gibt es einen festen Bereich für die PCIe Konfigurationsregister. in diesem sind die ersten Register für die VendorId und ProductId vorgesehen.
Auch einige BIOS-Variablen stehen an festen Speicherstellen.

Der Befehl *OperationRegion* erzählt dem BIOS von solchen Speicherbereichen und ermöglicht dadurch den Zugriff auf Systemvariable und Hardware-Register. Wir suchen den *OperationRegion* Befehl, der den Zugriff auf den Speicherbereich *EmbeddedControl* ermöglicht, weil Rehabman schreibt, dass es um die geht. *EmbeddedControl* ist einer von 9 in der ACPI Spezifikation definierten Standardspeicherbereichen. Ein weiterer dieser Bereiche ist *PCI_Config* , der Zugriff auf die erwähnten Konfigurationsregister erlaubt.
Ich kann die Lektüre der ACPI Spezifikation nur jedem empfehlen. 1036 Seiten purer Poesie, vollgepackt mit spannender Action und herzzerreißenden Dramen, ein Muß für jeden, jeden .... pffffffff, ich komme später darauf zurück.
Denen, deren Herz der Spezifikation nicht standhalten würde, sei gesagt, dass der Befehl am Anfang eine Operationregion mit dem Namen ECF2 definiert, die die 255 Bytes ab Adresse 0 im Speicherbereich des EmbeddedControllers belegt.

Ordnung ist das halbe Leben

ECF2 ist einfach nur ein Speicherbereich von 255 Bytes. Das macht den Zugriff auf Register, Funktionen und Schaltzustände zwar möglich, aber nicht einfach.
Mit dem *Field*-Befehl, kommen Ordnung und Übersichtlichkeit in die Sache. Mit dem Befehl weisst man einzelnen Speicherzellen bzw. Bitbereichen Namen zu, bevorzugt solche, die die Funktion beschreiben.
Ein Eintrag im *Field*-Befehl besteht aus dem Register Namen und der Länge des Registers in Bits. Das nächste Register beginnt and dem Bit, das auf das letzte des vorhergehenden Registers folgt. Statt eines Namens kann dort auch ein *Offset*-Befehl stehen. In diesem Fall beginnt das nächste Register bei Bit 0 des Bytes, das so viele Bytes vom Anfang des Speicherbereichs entfernt ist, wie der Wert in Klammern angibt.

Die erwähnten Konfigurationsregister, die jeweils 16 Bit lang sind, würden wie folgt definiert und abgefragt.

Code

```
1. // Ein Bereich von 64 Byte am Anfang des PCI Config Space.
2. // Die Länge ist nicht wirklich wichtig, solange die Register rein passen.
3. // Laut PCI Spezifikation ist der "normale" Config-Space 64 Byte lang
4. OperationRegion (CONF, PCI_Config, Zero, 64)
5. Field (CONF, ByteAcc, Lock, Preserve)
6. {
7. VNID, 16, // Vendor Id 16 Bit lang
8. PRID, 16, // Product Id 16 Bit lang
9. }
10.
11.
```

```
12. Method (GTVE, 0, NotSerialized) // Methode gibt den Inhalt des Registers mit der VendorId zurück
13. {
14. return (VNID)
15. }
```

Alles anzeigen

Lange Kerls

Uns interessieren, die Register in der Miix DSDT, die länger als 8 Bit sind. Das sind in der DSDT des Miix 9 Stück. Von denen interessieren uns nur die, die auch benutzt werden. Im Suchfeld von MaciAsl, kann man sehen, wie oft das Suchwort gefunden wurde. Wenn da eine 1 steht, kann man das Register ignorieren. Jedes der 9 Register einmal suchen und es bleiben 6 übrig.
AML kennt nur wenige "Schreib-Befehle". Store ist einer, wie der Name schon sagt. Store (denWert, nachHier) - speichert den Wert des ersten Parameters im zweiten. Mathematische Operationen können mit zwei oder drei Parametern - Add (das, zudem, speichereHier) - aufgerufen werden. Werden sie mit drei Parametern aufgerufen, so wird das Ergebnis der Operation im letzten Parameter gespeichert. Schauen wir uns die Stellen an, an denen die Register verwendet werden, stellen wir fest, dass die Register in der DSDT immer nur gelesen werden. Alle langen Kerls haben 16 Bit Länge und werden immer nur gelesen. Das macht das Leben einfacher, denn wir müssen uns nur mit Lesebefehlen von jeweils 2 Byte Länge beschäftigen.

Divide et impera

Nun kann Apples ACPIInterpreter, wohl an dieser Stelle nicht mit Bitlängen über 8 arbeiten. Also muss man jeden 16 Bit Zugriff in zwei 8 Bit Zugriffe aufteilen und das Ergebnis zu einem 16 Bit-Wert zusammenfügen. Rehabman zeigt eine Möglichkeit, die mir aber nicht liegt. Ich will nicht so viel tippen und jedes Register in zwei zu zerlegen ist umständlich und dann beim Lesen jedesmal beide Teilvariablen anzugeben auch. Details findet man in Rehabman's Hilfethread zum Thema Batterieanzeige.

Vitamin C

Ich verwende eine bei C gebräuchliche Methode. Ich lege ein Byte Array über den Speicherbereich und greife darüber auf einzelne Bytes zu. Beim Lesen einer Variable muss ich nur den ersten Offset angeben und ich muss die Original Felddefinition nicht ändern bzw. eine neue anlegen.

Die Methode, um ein 16 Bit Wort zu lesen, soll RDWD heißen, und hat einen Parameter. den Offset des ersten Bytes vom Beginn der Operationregion:

Code

```
1. Method (RDWD, 1, NotSerialized) // definiere Methode RDBY die einen Parameter hat.
2. {
3. OperationRegion (BYAR, EmbeddedControl, Arg0, 2) // Lege eine OperationRegion an, die im EmbeddedControl Speicherbereich an der durch den Parameter
4. // angegebenen Adresse liegt und 2 Byte lang ist.
5. Field (BYAR, ByteAcc, NoLock, Preserve) // der Speicherbereich wird in Low und High Byte unterteilt
6. {
7. BLOW, 8,
8. BHGH, 8,
9. }
10.
11.
12. Name (TEMP, Buffer (0x02) {}) // Lege einen 2 Byte Buffer an
13. Store (BLOW, Index (TEMP, Zero)) // Kopiere das Low-Byte des Speicherbereichs in das erste Byte des Buffers
14. Store (BHGH, Index (TEMP, One)) // Kopiere das High-Byte des Speicherbereichs in das zweite Byte des Buffers
15. Return (TEMP) // Gib den Buffer aus, der dann von der aufrufenden Methode als 16 Bit Wert interpretiert wird.
16. }
```

Alles anzeigen

Die RDWD-Methode schreibt man in der DSDT, direkt hinter den Field Befehl für die Operationregion ECF2.

Abstand halten

Jetzt braucht's nur noch die Offsets der Register. Das ist einfach: Der Offset-Befehl direkt vor ECWR sagt uns, dass ECWR bei Byte 0x80 steht. XX10 ist 8 Bit, ein Byte, weiter also bei 0x81. XX11 wieder 8 Bit weiter, also bei 0x82. XX11 ist 16 Bit lang, somit ist B1DC 2 Byte weiter, bei 0x84.
Die Liste der Felder mit ihren Abständen (Offsets)

Code

```
1. Schnipp
2.
3.
4. Offset (0x80),
```

5. ECWR, 8, // 0x80
6. XX10, 8, // 0x81
7. XX11, 16, // 0x82
8. B1DC, 16, // 0x84
9. B1FV, 16, // 0x86
10. B1FC, 16, // 0x88
11. XX15, 16, // 0x8A
12. B1ST, 8, // 0x8C
13. B1CR, 16, // 0x8D
14. B1RC, 16, // 0x8F
15. B1VT, 16, // 0x91
16. BPCN, 8, // 0x93
17. Offset (0xC0),
18. MGI0, 8, // 0xC0
19.
20.
21. Schnapp

Alles anzeigen

Nun verwenden wir Suchen und Ersetzen um jeden Aufruf von B1DC, außer dem im Field Befehl, durch RDWD(0x84). Entsprechend geht man bei den anderen Registern vor.

Bei der Gelegenheit sehen wir, dass die Aufrufe im Gerät BAT0 erfolgen.

Das ist dann wohl unsere Batterie. BAT0 könnte natürlich auch für Fledermaus 0 stehen, aber wenn mein Leben davon abhängen würde, würde ich eher auf Batterie tippen.
Warum die Felder B1XX heißen, wenn die Batterie Nummer 0 ist, ist eines der vielen Lenovo Geheimnisse. Es gibt noch mehr BAT Geräte, aber offensichtlich sind sie nur Platzhalter.

Beim Durchsuchen der SSDTs findet man in der SSDT-8 Zugriffe auf einige der Register. Also muss man auch in der SSDT-8 die Zugriffe durch RDWDs ersetzen. Das passiert ebenfalls mit suchen und Ersetzen. Allerdings muss man die Methode noch zu den Externals Deklarationen hinzufügen, damit MaciASL weiß, dass es sie gibt..

Code

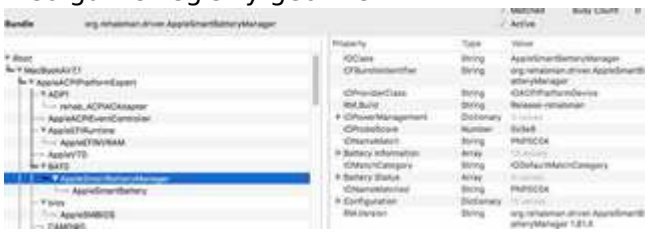
1. Schnipp
2.
3. External (_SB_.PCI0.I2C1.TPL1, DeviceObj) // (from opcode)
4. External (_SB_.PCI0.LPCB.H_EC, DeviceObj) // (from opcode)
5. External (_SB_.PCI0.LPCB.H_EC.RDWD, MethodObj)
6. External (_SB_.PCI0.LPCB.H_EC.B1CI, UnknownObj) // (from opcode)
7. External (_SB_.PCI0.LPCB.H_EC.B1DC, UnknownObj) // (from opcode)
8. External (_SB_.PCI0.LPCB.H_EC.B1DI, UnknownObj) // (from opcode)
9.
10. Schnapp

Die DSDT im patched Ordner speichern und Neustarten.

Close, but no cigar

Die gute Nachricht ist, dass sich das Batteriesymbol in der Menüleiste anzeigen lässt, die schlechte ist, dass der Ladezustand immer 0% ist. Netzteil abgeklemmt, Ladezustand immer noch 0%, hätte ja auch negativ werden können.

Also gut IORegistry geöffnet.



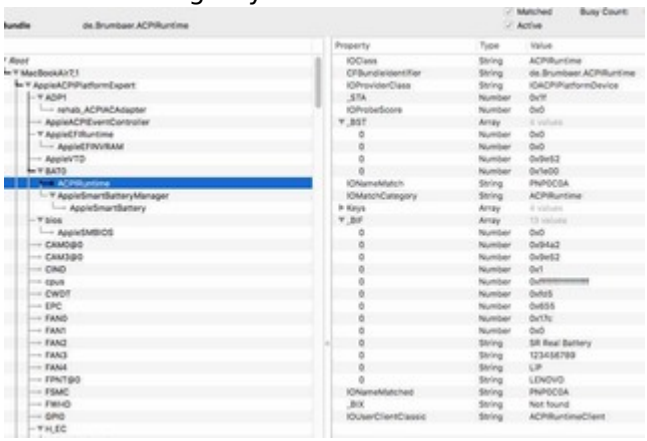
Wir sehen Rehabmans AppleSmatBatteryManager wird geladen. Das Gerät ist vom Typ PNP0C0A. Ein kurzer Blick in die ACPI Spezifikation zeigt, dass es sich um eine Control-Method-Battery handelt und keine Smart-Battery wie der Treibername vermuten lässt. Da der Treiber allerdings für Geräte des Types PNP0C0A gedacht ist (siehe Info.plist des Batterie-Kextes), scheint der Batterietyp der gewünschte zu sein.
Die Spezifikation sagt uns, dass ein Control-Method-Battery Gerät die Methoden _BST und _BIF

oder _BIX zum Auslesen der Batteriewerte verwendet.

Ver_BIXT und zugenäht

Also ein Kext programmiert, das die Werte eines ACPI Gerätes abfragen kann. Das Teil heißt ACPIRuntime, weil ich schon ein EFIRuntime geschrieben habe, das unter anderem EFI Variablen ausliest.

ACPIRuntime schreibt die gelesenen Werte als Property in die IORegistry und man kann diese mit dem IORegistryEditor anzuschauen.



Offensichtlich funktionieren _BST und _BIF, _BIX hingegen gibt's nicht.

Ich könnte der DSDT eine _BIX Routine hinzufügen, oder erst einmal einen Blick in die Quellen des AppleSmatBatteryManager werfen.

Sucht man in den Quellen nach _BIX findet man eine ganze Menge, darunter folgendes:

Code

```
1. Schnipp
2.
3.
4. /*****************************************************************************
5. * AppleSmartBatteryManager::getBatteryBIX
6. * Call DSDT _BIX method to return ACPI 4.x battery info
7. *****************************************************************************/
8.
9.
10. IOReturn AppleSmartBatteryManager::getBatteryBIX(void)
11. {
```

```
12. DebugLog("getBatteryBIX called\n");
13.
14.
15. Schnapp
```

Alles anzeigen

Das ist offensichtlich die Methode, die die _BIX Methode der Batterie liest. Suchen wir nach getBatteryBIX finden wir nur einen Aufruf der Methode, der Rest gehört zu deren Definition.

Code

```
1. Schnipp
2.
3.
4. fProvider->getBatterySTA();
5. if (fBatteryPresent)
6. {
7. if (fUseBatteryExtendedInformation)
8. fProvider->getBatteryBIX();
9. else
10. fProvider->getBatteryBIF();
11. if (fUseBatteryExtraInformation)
12. fProvider->getBatteryBBIX();
13. fProvider->getBatteryBST();
14. }
15.
16. Schnapp
```

Alles anzeigen

Man sieht, dass getBatteryBIX nur aufgerufen wird, wenn fUseBatteryExtendedInformation wahr ist. Ansonsten wird _BIF verwendet. Also nach fUseBatteryExtendedInformation suchen und wir finden:

Code

```
1. Schnipp
2.
3.
```

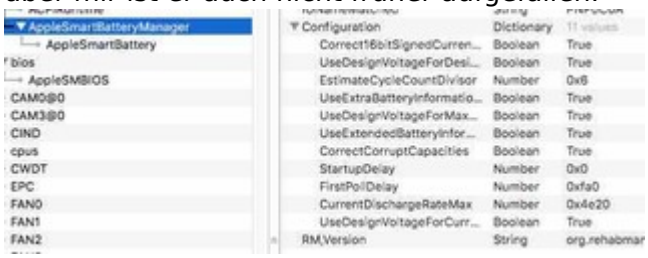
4. fUseBatteryExtendedInformation = useExtendedInformation->isTrue();
5. if (fUseBatteryExtendedInformation && kIOReturnSuccess != fProvider->validateBatteryBIX())
6. fUseBatteryExtendedInformation = false;
7.
8. Schnapp

Der Wert hängt von einer anderen Variablen ab und von validateBatteryBIX. Nur anhand des Namens, kann man vermuten, dass in der Methode überprüft wird ob _BIX vernünftige Werte liefert. In unserem Falle wird _BIX das nicht tun, es existiert ja nicht. Der Batterietreiber wird in unserem Falle _BIF verwenden, da validateBatteryBIX false liefern wird und deshalb fUseBatteryExtendedInformation ebenfalls false ist. Das fehlende _BIX is somit nicht unser Problem.

Spannung

IORegistry in der linken und die ACPI Spec in der rechten, überprüfe ich ob die Werte in _BST und _BIF Sinn machen. Tun sie. Die Design Voltage ist zwar als unbekannt markiert, aber das ist laut Spezifikation erlaubt.

Dem aufmerksamen Betrachter mag oben der Eintrag Configuration im AppleSmatBatteryManager aufgefallen sein. Ich bin versucht "ein bisschen spät" zu sagen, aber mir ist er auch nicht früher aufgefallen.



UseDesignVoltageFor... springt sogleich ins Auge. Also noch mal in den Programmcode geschaut und festgestellt, dass das Kext für die Laufzeitberechnung auf die Design Voltage zurückgreift. Es merkt aber nicht, dass der Design Voltage Wert auf unbekannt steht und schon wird mit -1 multipliziert statt mit 7 irgendwas Volt, es sei denn man lässt das Kext die DesignVoltage ignorieren und stattdessen die aktuelle Batterie Spannung verwenden..

Also gut in der Info.plist unter Configuration die beiden Uses... auf NO gesetzt.

Neustart und siehe da: 100%. Dabei fühle ich mich inzwischen als sei meine fast Batterie leer.

Dann kommt die Hintergrundbeleuchtung eben morgen dran.

PS.
Man könnte statt die beiden Uses... zu ändern, natürlich auch in der _BIF Methode einen Wert für die Design Spannung vorgeben. 0x1E00, würde sich anbieten, denn das ist die in _BST angegebene momentane Batteriespannung.
Man könnte auch AppleSmatBatteryManager so ändern, dass es eine validateDesignVoltageMethode gäbe, mit deren Hilfe dieUses... automatisch auf false gesetzt werden würden, falls die DesignVoltage nicht definiert ist.