[Guide] Using Clover to "hotpatch" ACPI

Introduction

Patching ACPI is always necessary to enable (near) full functionality when installing OS X on non-Apple hardware.

There is a complete guide here:
http://www.tonymacx86.com/threads/guide-patching-laptop-dsdt-ssdts.152573/

That guide uses what is known as "static patching". In order to inject patched ACPI files, we extract native ACPI, disassemble them, make changes, then recompile and place the files in ACPI/patched, so that Clover injects the patched ACPI instead of native ACPI. With the techniques detailed in this guide, the changes can be made directly to the ACPI binaries provided by BIOS, skipping the extract, disassembly, and recompilation steps.

You should have a solid understanding of static ACPI patching before attempting to hotpatch. You should also have an understanding of the ACPI spec, binary patching, programming, and ACPI concepts. Don't expect step-by-step and spoon feeding in this thread.

Clover mechanisms

Clover provides a few methods for accomplishing ACPI hotpatch:

- config.plist/ACPI/DSDT/Fixes

- config.plist/ACPI/DSDT/Patches
- ability to inject additional SSDTs

DSDT/Fixes provide fixed function ACPI patching. Each "Fix" can do a particular kind of patching that can be used instead of typical patching you might do with MaciASL and static patching. For example, "IRQ Fix" can be accomplished with "FixHPET" "FixIPIC" "FixRTC" and "FixTMR". As an other example, "Fix _WAK Arg0 v2" can be accomplished with "FixWAK". You can read the Clover wiki for more information on each patch. Most of the time, there are not many DSDT "Fixes" needed for basic functionality. DSDT "Fixes" are useful for implementing patches that are difficult or impossible to implement with ACPI/DSDT/Patches or additional SSDTs.

DSDT/Patches allows for binary search and replace by Clover. Clover loads the native ACPI files, applies the patches specified in ACPI/Patches using binary search/replace, then injects the patched ACPI. You need to have an understanding of the binary AML format. It is fully documented in the ACPI spec.

ACPI namespace is built by merging the DSDT and SSDTs at load time. By placing additional SSDTs into ACPI/patched, we can essentially add code to this ACPI set. Since many OS X patches involve adding properties to ioreg with a _DSM method, it is often adequate to simply add an SSDT which contains the additional _DSM method instead of patching the native ACPI files. A perfect example you're already familiar with is the SSDT.aml that is generated by Pike's ssdtPRgen.sh.

In some cases, more than one mechanism must be used to accomplish a single goal. For example, you might use binary patching to disable or rename components in the native ACPI set, and then replace it with additional SSDTs.

Renaming ACPI objects

Since OS X can depend on specific ACPI object names used by Macs, a common patch is to rename an object in the native ACPI set. For example, most PC laptops use GFX0 for the integrated Intel GPU object (Intel HD Graphics). In OS X, power management for Intel graphics is not enabled unless this device is named IGPU. Using static patching, we apply "Rename IGPU to GFX0" in order to rename this object. The patch must be applied to the DSDT and all SSDTs that reference it.

With hotpatch, we can rename GFX0 to IGPU using a simple Clover patch in ACPI/DSDT/Patches. Such patches apply to DSDT and all native SSDTs (DSDT/Patches do not apply to SSDTs that are added via ACPI/patched). The patch for the rename would be:

Comment: Rename GFX0 to IGPU
Find: <4746 5830>
Replace: <4947 5055>

The hex values in Find and Replaces are the ASCII codes for GFX0 and IGPU, respectively.

Note:

Code:

```
u430:~ RehabMan$ echo -n GFX0|xxd
0000000: 4746 5830                                GFX0
u430:~ RehabMan$ echo -n IGPU|xxd
0000000: 4947 5055                                IGPU
```

There are number of common renames, and most are in the config.plist that are part of my Clover/hoptpatch project:

https://github.com/RehabMan/OS-X-Clover-Laptop-Config/tree/master/hotpatch

In fact, the hotpatch SSDTs that are part of the same project depend on the renames being implemented.

Common renames:
GFX0 -> IGPU
SAT0 -> SATA
EHC1 -> EH01
EHC2 -> EH02
XHCI -> XHC
HECI -> IMEI
MEI -> IMEI
LPC -> LPCB
HDAS -> HDEF
AZAL -> HDEF

Note: All ACPI identifiers are 4 characters. Shorter names are padded with underscore. So, for example, XHC is represented in the AML binary as XHC_, EC would be EC__, EC0 would be EC_, MEI would be MEI_, etc.

Removing methods

It is very difficult to remove ACPI objects, (methods, names, devices, etc) using Clover binary patches. Commonly, we must add _DSM methods to inject properties that describe various hardware properties. But added _DSM methods can conflict with existing _DSM methods that may already be in the native ACPI files. With static patching, "Remove _DSM methods" would be used.

Since it is difficult to remove the methods, but we don't want the native methods to conflict with new _DSM methods that are added, the fix is to rename the native methods to something else.

So... again, we use a simple rename patch:

Comment: Rename _DSM to XDSM
Find: <5f44534d>
Replace: <5844534d>

Sometimes, you might rename an object to effectively disable it so it does not cause problems. For example, my Intel DH67GD DSDT defines an APSS object. If this object is left in the DSDT it interferes with power management (causes panic). I use a rename from APSS -> APXX. Because AppleIntelCPUPowerManagement is looking for APSS, it does not cause a problem once renamed to APXX.

Redirect and Replace

In some cases, we would like to replace code to change the behavior. For this, we can rename the object and provide an alternate implementation in an SSDT.

A common fix is spoofing the ACPI code in DSDT and SSDTs such that it behaves as if a certain version of Windows is the ACPI host. When static patching, we might use "OS Check Fix (Windows 8)". When applied, it changes code from:

Code:

```
If (_OSI("Windows 2012"))
```

To:

Code:

```
If (LOr(_OSI("Darwin"),_OSI("Windows 2012"))
```

Since the _OSI implementation in OS X only responds to "Darwin" the code is changed so that this specific _OSI check also accomodates "Darwin".

With hotpatching, the opposite approach is taken. Instead of changing the code using _OSI, we change the code so it calls a different method that emulates the _OSI implementation that would be in the Windows ACPI host.

This technique relies on two of the techniques... a patch to change all calls from _OSI to XOSI... and an implementation of XOSI that emulates what Windows would do for a certain Windows version.

First, changing the code to call XOSI instead of _OSI:

Comment: Change _OSI to XOSI
Find: <5f4f 5349>
Replace: <584f 5349>

The hex codes above should be no mystery (they are ASCII for _OSI and XOSI, respectively).

Now the code mentioned above, after patching by Clover, will read:

Code:

```
If (XOSI("Windows 2012"))
```

Now we need an SSDT that implements XOSI. You will find such an implementation in the repository (SSDT-XOSI.dsl).

Note that without the SSDT that implements the XOSI method, the (patched) calls to XOSI would cause an ACPI abort (ACPI abort causes execution of the ACPI method to be terminated immediately with an error). Don't use the _OSI->XOSI patch without the XOSI method.

Rename and Replace

A second pattern, similar to "Redirect and Replace" is "Rename and Replace". In this case, instead of changing all the call sites, we change the method definition such that the method is named something different than it was originally, but leave the original method name at the call sites. This allows the method that is the target of the calls to be replaced.

For example, it is very common for USB devices to cause "instant wake". As a work around, wake on USB can be disabled. Most laptops don't have a BIOS option for it, so instead the _PRW methods that control this feature are patched.

For example, the native _SB.PCI0.EHC1._PRW method might read:

Code:

```
            Method (_PRW, 0, NotSerialized)  // _PRW: Power Resources for Wake
            {
                Return (GPRW (0x6D, 0x03))
            }
```

In order to patch it so USB devices on EHCI#1 cannot cause wake, it would be changed:

Code:

```
            Method (_PRW, 0, NotSerialized)  // _PRW: Power Resources for Wake
            {
                Return (GPRW (0x6D, 0))
            }
```

Usually, there are several such call sites to GPRW that need to be changed (also, keep in mind not all ACPI sets use the specific name GPRW). Instead of trying to patch all the call sites as above, we can instead patch the method definition of GPRW instead:

With the original code:

Code:

```
Method (GPRW, 2, NotSerialized)
{
   ...
}
```

If we change it to:

Code:

```
Method (XPRW, 2, NotSerialized)
{
   ...
}
```

Since you don't want to change any call sites, the patch must be constructed such that it affects only the method itself and not the call sites. According to ACPI spec, a method definition starts with bytecode 14, is followed by the method size, the method name, argument count/flags. You can use the "-l" option in iasl to generate a mixed listing of your ACPI file. For example, the Lenovo u430 GPRW method mixed listing:

Code:

```
    4323:            Method (GPRW, 2, NotSerialized)
```

```
00003F95:  14 45 08 47 50 52 57 02     ".E.GPRW."

    4324:          {
    4325:              Store (Arg0, Index (PRWP, Zero))

00003F9D:  70 68 ..................    "ph"
00003F9F:  88 50 52 57 50 00 00 ...    ".PRWP.."
```

We could do a find replace using the method header bytes:
Find: <14 45 08 47 50 52 57 02>
Replace: <14 45 08 58 50 52 57 02>

But what happens if the method differs slightly between different versions of BIOS or models that are similar, but not exactly the same? In that case, the byte following the 14 will change due to the change in the method length.

My thought is that the beginning of the method body is less likely to be different than the total length of the method, so it helps to add a few extra bytes from the body of the method to the find/replace specification:

Find: <47 50 52 57 02 70 68>
Replace: <58 50 52 57 02 70 68>

The number of follow-on bytes that you use depends on how many you need to make the find/replace affect only the method definition. You can verify by looking at the native AML binary in a hex editor such as Hex Fiend (it is a nice hex editor and is also open source).

Note: Although you could search just for the method name + arg count/flags, it is possible that the same pattern will find a call site to the method which you don't want to change. In the case of the u430 that wasn't the case, so I was able to find/replace with just the method name+flags.

Find: <47505257 02>
Replace: <58505257 02>

In the case of the ProBook UPRW, it was necessary to use the follow-on bytes that are part of the method body:

Find: <55505257 0a7012>
Replace: <58505257 0a7012>

Now any code that calls GPRW (or UPRW in the ProBook example) will not call the implementation in XPRW because the name doesn't match. The original XPRW is now unreachable code. Which means the GPRW implementation can be changed for our purpose:

Code:

```
Method(GPRW, 2)
{
    If (0x6d == Arg0) { Return(Package() { 0x6d, 0, }) }
    External(\XPRW, MethodObj)
    Return(XPRW(Arg0, Arg1))
}
```

Explaining the code: For any call to GPRW with the first argument set to 0x6d (the GPE we are trying to disable for wake), instead of returning what the original GPRW would, we return a package with 0x6d and 0 (which disables wake). And for other GPE values, the code simply calls the original GPRW method now named XPRW.

Another simple case is patching EC query methods to fix the brightness keys. A simple rename of the _Qxx methods involved to XQxx, and a new definition of the method with the original name is all that is needed.

For example, in the HP Envy Haswell repo:

Code:

```
        // _Q13 called on brightness/mirror display key
        Method (_Q13, 0, Serialized)  // _Qxx: EC Query
        {
            External(\HKNO, FieldUnitObj)
            Store(HKNO, Local0)
            If (LEqual(Local0,7))
            {
                // Brightness Down
                Notify(\_SB.PCI0.LPCB.PS2K, 0x0405)
            }
            If (LEqual(Local0,8))
            {
                // Brightness Up
                Notify(\_SB.PCI0.LPCB.PS2K, 0x0406)
            }
            If (LEqual(Local0,4))
            {
                // Mirror toggle
                Notify(\_SB.PCI0.LPCB.PS2K, 0x046e)
            }
        }
```

And the associated patch:
Comment: change Method(_Q13,0,S) to XQ13
Find: <5f513133 08>
Replace: <58513133 08>

This same "Rename and Replace" mechanism can be used in cases that are much more complex than this. For example, it is typically used to patch battery methods, which need to be patched to avoid access to multibyte EC fields.

Tips for complex Rename and Replace

As you probably already know, patching for battery status (multibyte EC fields) can be very complex and can involve a lot of code changes to many methods.

This section will detail some of the techniques and procedures used for battery patching.

It is advisable to patch for battery without using hotpatch first. After you get it working, then attempt hotpatch. Also, the difference between the code not patched for battery and the code patched for battery is very helpful. You can use a tool like 'diffmerge' to compare each. This is especially true if there is already a static battery patch for your laptop in my laptop repository.

General procedure:
- start with native ACPI
- patch for battery status using static patching (verify it works)
- use diffmerge to compare the unpatched code with patched code
- for each method that is different, implement the "Rename and Replace" pattern
- for the EC fields, create another EC OperationRegion (use a name that is different from the original) and Field definition as a sort of "overlay" which contains only the EC fields you need to patch
- to create the EC overlay, you can use the patched Field/OperationRegion in the patched DSDT, then eliminate unpatched fields
- use External to allow the replacement methods in the SSDT to access the fields defined elsewhere in the ACPI set (usually DSDT)
- let the compiler point out where you need to use External
- watch out for symbols with duplicate names in different scopes

An example is provided in post #2 of this thread.

Code value patching

Consider the case of "Fix Mutex with non-zero SyncLevel". This patch finds all Mutex objects and replaces the SyncLevel with 0. We use this patch since OS X does not support Mutex debugging correctly and aborts on any Acquire with a Mutex that has a non-zero SyncLevel.

As an example, the u430 has Mutexes delcared as such:

Code:

```
        Mutex (MSMI, 0x07)
```

To make it compatible with OS X it must be changed:

Code:

```
        Mutex (MSMI, 0)
```

The ACPI spec defines how a Mutex object is encoded in the AML, but it can be helpful to look at a mixed disassembly of a small ACPI file:

Code:

```
DefinitionBlock ("", "DSDT", 2, "test", "test", 0)
{
    Mutex(ABCD, 7)
}
```

The iasl compiler can create a mixed listing file with the "-l" option.

If we compile the above file with: iasl -l test.dsl, test.lst contains:

Code:

```
       1:  DefinitionBlock ("", "DSDT", 2, "test", "test", 0)

00000000:  44 53 44 54 2B 00 00 00     "DSDT+..."
00000008:  02 36 74 65 73 74 00 00     ".6test.."
00000010:  74 65 73 74 00 00 00 00     "test...."
00000018:  00 00 00 00 49 4E 54 4C     "....INTL"
00000020:  10 04 16 20 ...........     "... "

       2:  {
       3:      Mutex(ABCD, 7)

00000024:  5B 01 41 42 43 44 07 ...    "[.ABCD."
       4:  }
```

As you can see, the Mutex(ABCD, 7), is encoded as <5B 01 41 42 43 44 07>

It is now very easy to construct a patch for it:

Comment: Change Mutex(ABCD,7) to Mutex(ABCD,0)
Find: <5B 01 41 42 43 44 07>
Replace: <5B 01 41 42 43 44 00>

Note that with recent versions of Clover, all Mutex objects with non-zero SyncLevel in DSDT can be fixed with the "FixMutex" option.

Clover ACPI configuration

With static patching, DropOem=true is used and patched DSDT and SSDTs are added to ACPI/patched. With hotpatch, instead use DropOem=false, and only add-on SSDTs are placed in ACPI/patched.

It is important to note that config.plist/ACPI/patches are applied only to native SSDTs, and not the SSDTs in ACPI/patched. This means that if you are renaming objects using config.plist, the add-on SSDTs must refer to the new names, not the old names. Unlike SSDTs in ACPI/patched, binary patches in ACPI/Patches *do apply* to DSDT.aml that might be in ACPI/patched. Keep this in mind if you're using a combination of static and hotpatching.

Also, with static patching, SortedOrder is used to specify the order of SSDTs in ACPI/patched. With hotpatch it is not strictly necessary as it is possible to construct the code in each SSDT such that the code is not order dependent. Especially if you place all add-on code in a single SSDT such as many of my laptop repo examples. Unless your add-on SSDTs are order dependent, you do not have to name each one in SortedOrder.

It is also not necessary to choose "numbered names" for each SSDT. Instead you can use meaningful names, such as "SSDT-USB.aml", SSDT-XOSI.aml", etc. Using numbers instead of meaningul names will just confuse you. Don't do it.

Troubleshooting

You can use patchmatic to look at your complete ACPI set as injected by Clover after patching. By runnning 'patchmatic -extract', patchmatic will write all injected DSDT.aml and SSDT*.aml in the order they were injected by Clover. You can disassemble them with 'iasl -da -dl *.aml'. If iasl shows errors with the disassembly (for example duplicate symbols), it is likley OS X is also rejecting the conflicting SSDTs.

If you're a novice with this technique, it is a good idea to implement one patch at a time, and slowly build it up to a full set of working patches + SSDTs. Trying to do all at once can make it difficult to locate your mistake.

Problem Reporting

Download patchmatic: https://bitbucket.org/RehabMan/os-x-maciasl-patchmatic/downloads/RehabMan-patchmatic-2015-0107.zip
Extract the 'patchmatic' binary from the ZIP. Copy it to /usr/bin, such that you have the binary at /usr/bin/patchmatic.

In terminal,

Code:

```
if [ -d ~/Downloads/RehabMan ]; then rm -R ~/Downloads/RehabMan; fi
mkdir ~/Downloads/RehabMan
cd ~/Downloads/RehabMan
patchmatic -extract
```

Note: It is easier if you use copy/paste instead of typing the commands manually.

Attach contents of Downloads/RehabMan directory (as ZIP).

Also, attach ioreg: http://www.tonymacx86.com/audio/58368-guide-how-make-copy-ioreg.html. Please, use the IORegistryExplorer v2.1 attached to the post! DO NOT reply with an ioreg from any other version of IORegistryExplorer.app.

And output from:

Code:

```
kextstat|grep -y acpiplat
kextstat|grep -y appleintelcpu
kextstat|grep -y applelpc
```

Also, attach EFI/Clover folder (press F4 at main Clover screen before collecting). Please eliminate 'themes' directory, especially if you have an overabundance of themes installed. Provide only EFI/Clover, not the entire EFI folder.

Also post output of:

Code:

```
sudo touch /System/Library/Extensions && sudo kextcache -u /
```