Enabling Docking Station Support for the Linux Kernel

Is Harder Than You Would Think

Kristen Carlson Accardi
Open Source Technology Center, Intel Corporation
kristen.c.accardi@intel.com

Abstract

Full docking station support has been a feature long absent from the Linux kernel—for good reason. From ACPI to PCI, full docking station support required modifications to multiple subsystems in the kernel, building on code that was designed for server hot-plug features rather than laptops with docking stations. This paper will present an overview of the work we have done to implement docking station support in the kernel as well as a summary of the technical challenges faced along the way.

We will first define what it means to dock and undock. Then, we will discuss a few variations of docking station implementations, both from a hardware and firmware perspective. Finally, we will delve into the guts of the software implementation in Linux—and show how adding docking station support is really harder than you would think.

1 Introduction and Motivation

It's no secret that people have not been clamoring for docking station support. Most people do not consider docking stations essential, and indeed some feel they are completely unnecessary. However, as laptops become thinner and lighter, more vendors are seeking to replace functionality that used to be built into the laptop with a docking station. Commonly, docking stations will provide additional USB ports, PCI slots, and sometimes extra features like built in media card readers or Ethernet ports. Most vendors seem to be marketing them as space saving devices, and as an improved user experience for mobile users who do not wish to manage a lot of peripheral devices.

We embarked on the docking station project for a few reasons. Firstly, we knew there were a few members of the Linux community out there who actually did use docking stations. These people would hopefully post to the hotplug PCI mailing lists every once in a while, wondering if some round of I/O Hotplug patches would enable hot docking to work. Secondly, there was a need to be able to implement a couple scenarios that dock stations provide a convenient test case for. Many dock stations are actually Peer-to-peer bridges with a set of buses and devices located behind the bridge. Hot add with devices with P2P bridges on them had always been hard for us to test correctly due to lack of devices. Also, the ACPI _EJD method is commonly used in AML code for docking stations, but this method can also be applied to any hotpluggable tree of devices. Finally, we felt that with the expanding product offerings for dock stations, filling this feature gap would eventually become important.

2 Docking Basics

There are three types of docking that are defined.

- **Cold Docking/Undocking** Laptop is booted attached to the dock station. Laptop is powered off prior to removal from the dock station. This has always been supported by Linux. The devices on the dock station are enumerated as if they are part of the laptop.
- **Warm Docking/Undocking** Laptop is booted either docked or undocked. System is placed into a suspend state, and then either docked or undocked. This may be supported by Linux, assuming that your laptop actually suspends. It depends really on whether a driver's resume routine will rescan for new devices or not.
- **Hot Docking/Undocking** Laptop is booted outside the dock station. Laptop is then inserted into the dock station while completely powered and operating. This has recently had limited support, but only with a platform specific ACPI driver. Hotplugging new devices on the dock station has never been supported.

Docking is controlled by ACPI. ACPI defines a dock as an object containing a method called _DCK. An example dock device definition is shown in Figure 1.

_DCK is what ACPI calls a "control method". Not only does it tell the OS that this ACPI object is a dock, it also is used to control the isolation logic on the dock connector.

```
Device (DOCK1) {
        Name(_ADR, ...)
        Method(_EJ0, 0) {...}
        Method(_DCK, 1) {...}
}
```

Figure 1: Example DSDT that defines a Dock Device

When the user places their system into the docking station, the OS will be notified with an interrupt, and the platform will send a Device Check notify. The notify will be sent to a notify handler and then that handler is responsible for calling the _DCK control method with the proper arguments to engage the dock connector.

_DCK as defined in the ACPI specification is shown in Figure 2. Assuming that _DCK returned successfully, the OS must now re-enumerate all enumerable buses (PCI) and also all the other devices that may not be on enumerable buses that are on the dock.

Undocking is just like docking, only in reverse. When the user hits the release button on the docking station, the OS is notified with an eject request. The notify handler must first execute _DCK(0) to release the docking connector, and then should execute the _EJ0 method after removing all the devices that are on the docking station from the OS.

The _DCK method is not only responsible for engaging the dock connector, it seems to also be a convenient place for system manufacturers to do device initialization. This is all implementation dependent. I have seen _DCK methods that do things such as programming a USB host controller to detect the USB hub on the dock station, issuing resets for PCI devices, and even attempting to modify PCI config space to assign new bus numbers[1] to the dock bridge.

[1]Highly unacceptable behavior

This control method is located in the device object that represents the docking station (that is, the device object with all the _EJx control methods for the docking station). The presence of _DCK indicates to the OS that the device is really a docking station.

_DCK also controls the isolation logic on the docking connector. This allows an OS to prepare for docking before the bus is activated and devices appear on the bus [1].

Arguments:

Arg0

1 Dock (that is, remove isolation from connector)

0 Undock (isolate from connector)

Return Code:

1 if successful, 0 if failed.

Figure 2: _DCK method as defined in the ACPI Specification

The only way to know for sure what the _DCK method does is to disassemble the DSDT.

3 Driver Design Considerations

There are platform specific drivers in the ACPI tree. The `ibm_acpi` drier had previously implemented a limited type of docking station support that would only work on certain ibm laptops. Essentially, this driver would hard code the device name of the dock to find the dock, and then would execute the _DCK method without rescanning any of the buses or inserting any of the non-enumerable devices. It suffers from being platform specific, which is not ideal. We wanted to make a generic solution that would work for most platforms.

We originally assumed that all dock stations were the same: a dock bridge would be located on the dock station, which was a P2P bridge, and all devices would be located behind the P2P bridge. The IBM ThinkPad Dock II is an example of this type of implementation, shown in Figure 4. The same driver (`acpiphp`) that could hotplug any device that had a P2P bridge

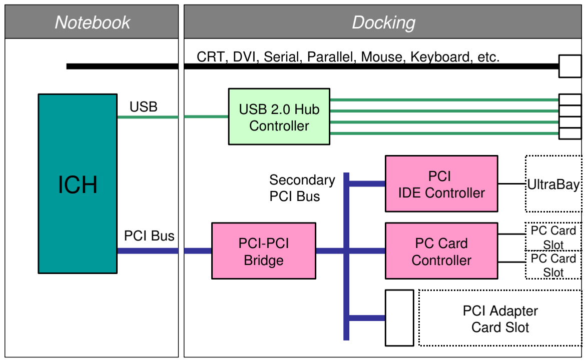

Figure 4: The IBM ThinkPad Dock II ©2006, Noritoshi Yoshiyama, Lenovo Japan, Ltd—Used by Permission

on it could be used to hotplug the dock station devices, with the minor addition of needing to execute the _DCK method prior to scanning for new devices.

These were bad assumptions.

3.1 Variations in dock device definitions

The dock device definition for a few IBM ThinkPads that I had available is shown in Figure 5. The physical device is a P2P bridge.

```
IBM_HANDLE(dock, root, "\\_SB.GDCK",    /* X30, X31, X40 */
           "\\_SB.PCI0.DOCK",    /* 600e/x,770e,...,X20-21 */
           "\\_SB.PCI0.PCI1.DOCK",      /* all others */
           "\\_SB.PCI.ISA.SLCE",        /* 570 */
    );
```

Figure 3: Defining a dock station in ibm_acpi.c

It appears to fit the ACPI definition of a standard PCI hotplug slot, in that it exists under the scope of PCI0, it has an _ADR function, and it is ejectable (has an _EJ0). It contains the _DCK method, indication that it is a docking station as well. This was our original view of the docking station.

```
T20,T30,T41, T42 look like this:
Device (PCI0)
    Device (DOCK)
    {
        Name (_ADR, 0x00040000)
        Method (_BDN, 0, NotSerialized)
        Name (_PRT, Package (0x06)
        Method (_STA, 0, NotSerialized)
        Method (_DCK, 1, NotSerialized)
        Method (_EJ0, ...)
```

Figure 5: IBM T20, T30, T41, T42 DSDT

Unfortunately for us, not all dock stations are the same. Sometimes system manufactures create a "virtual" device to represent the dock. It simply calls methods under the "real" dock bridge. In this case, the `acpiphp` driver will not recognize the GDCK device as an ejectable slot because it has no _ADR. In addition, it will not recognize the "real" dock device as an ejectable PCI slot because _EJ0 is not defined under the scope of the Dock(), but instead under the virtual device GDCK. An example of this type of DSDT is shown in Figure 6. There are also dock stations that do not utilize a P2P bridge for PCI devices, such as the Lenovo ThinkPad Advanced Dock Station, shown in Figure 7. In addition, there are dock stations that do not have any PCI devices on them at all. This made using the ACPI PCI hotplug driver a bit nonsensical. However, the normal ACPI driver model also didn't work, because ACPI drivers will only load if a device exists. However, we decided to move the implementation from the PCI hotplug driver into ACPI, because there really was nowhere else to put it.

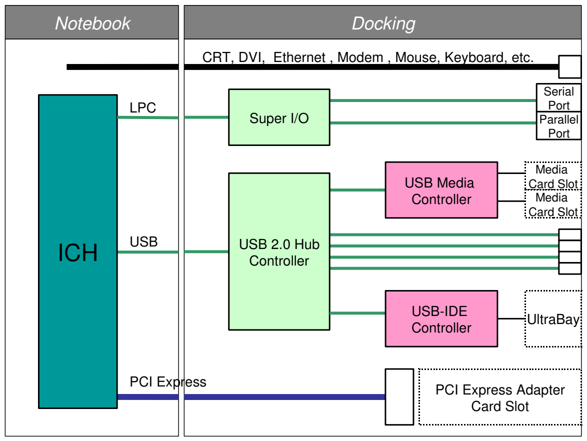

Figure 7: The Lenovo ThinkPad Advanced Dock Station
©2006, Noritoshi Yoshiyama, Lenovo Japan, Ltd—Used by Permission

In order to decouple the dock functionality from the hotplug functionality, the dock driver needs to allow other drivers to be notified upon a dock event, and also to register individual hotplug notification routines. This way, the dock

```
Scope (_SB)
    Device(GDCK)
        Method (_DCK, 1, NotSerialized)
        {
            Store (0x00, Local0)
            If (LEqual (GGID (), 0x03))
            {
                Store (\_SB.PCI0.LPC.EC.SDCK (Arg0), Local0)
            }
            If (LEqual (GGID (), 0x00))
            {
                Store (\_SB.PCI0.PCI1.DOCK.DDCK (Arg0), Local0)
            }
            Return (Local0)
        }
        Method (_EJ0, 1, NotSerialized)
            ...
    Device (PCI1)
        Device (DOCK)
        {
            Name (_ADR, 0x00030000)
            Name (_S3D, 0x02)
            Name (_PRT, Package (0x06)
```

Figure 6: Alternative dock definition

driver can just handle the dock notifications from ACPI, and individual subsystems/drivers can handle how to hotplug new devices. In the case of PCI, `acpiphp` can still handle the device insertion, but it will not be used if there are no PCI devices on the dock station.

4 Driver Implementation Details

The driver is located in `drivers/acpi/dock.c`. It makes a few external functions available to drivers who are interested in dock events.

4.1 External Functions

`int is_dock_device(acpi_handle handle)`
: This function will check to see if an ACPI device referenced by handle is a dock device. This means that the device either is a dock station, or a device on the dock station.

`int register_dock_notifier(struct notifier_block *nb)`
: Sign up for dock notifications. If a driver is interested in being notified when a dock event occurs, it can send in a `notifier_block` and be called right after _DCK has been executed, but before any devices have been hotplugged.

`int unregister_dock_notifier(struct notifier_block *nb)`
: Remove a driver's `notifier_block`.

`acpi_status register_hotplug_dock_device (acpi_handle, acpi_notify_handler, void *)`
: Pass an ACPI notify handler to the dock driver, to be called when a dock event has occurred. This allows drivers such as `acpiphp` which need to re-enumerate buses after a dock event to register their own routine to handle this activity.

`acpi_status unregister_hotplug_dock_device(acpi_handle handle)`
: Remove a notify handler from the dock station's hotplug list.

4.2 Driver Init

At init time, the dock driver walks the ACPI namespace, looking for devices which have defined a _DCK method.

```
/* look for a dock station */
acpi_walk_namespace(
ACPI_TYPE_DEVICE,
ACPI_ROOT_OBJECT, ACPI_UINT32_MAX,
find_dock, &num, NULL);
```

If we find a dock station, then we create a private data structure to hold a list of devices dependent on the dock station, and also hotplug notify blocks.

We can detect devices dependent on the dock by walking the namespace looking for _EJD methods. _EJD is another method defined by ACPI, that is associated with devices that have a dependency on other devices. From the spec:

> This object is used to specify the name of a device on which the device, under which this object is declared, is dependent. This object is primarily used to support docking stations. Before the device indicated by _EJD is ejected, OSPM will prepare the dependent device (in other words, the device under which this object is declared) for removal [1].

So, to translate, all devices that are behind a dock bridge should have an _EJD method defined in them that names the dock.

Drivers or subsystems can register for dock notifications if they control a device dependent on the dock station. Drivers use the `is_dock_device()` function to determine if they are a device on a dock station. This allows for re-enumeration of the subsystem after a dock event if it is necessary. In the case of PCI devices, the `acpiphp` driver is modified to detect not only ejectable PCI slots, but also PCI dock bridges or hotpluggable PCI devices. If it does find one of these devices, then it will request that the dock driver notify `acpiphp` whenever a dock event occurs. When a system docks, the `acpiphp` driver will treat the event like any other PCI hotplug event, and rescan the appropriate bus to see if new devices have been added.

4.3 Dock Events

At driver init time, the dock driver registers an ACPI event handler with the ACPI subsystem. When a dock event occurs, the dock driver event handler will be called. A dock is a `ACPI_NOTIFY_BUS_CHECK` event type. First, the event handler will make sure that we are not already in the middle of docking. This check is needed, because I found on some dock stations/laptop combos that false dock events were being generated by the system—probably due to a faulty physical connection. We ignore these false events. It is also necessary to ensure that the dock station is actually present before performing the _DCK operation. This is accomplished by the `dock_present()` function. `dock_present()` just executes the ACPI _STA method. _STA will report whether or not the device is present.

```
if (!dock_in_progress(ds) &&
dock_present(ds)) {
```

`begin_dock()` just sets some state bits to indicate that we are now in the middle of handling a dock event.

```
begin_dock(ds);
```

dock() will execute the _DCK method with the proper arguments.

```
dock(ds);
```

We confirm that the device is still present and functioning after the _DCK method.

```
if (!dock_present(ds)) {
    printk(KERN_ERR PREFIX
"Unable to dock!\n");
        break;
}
```

We notify all drivers who have registered with the `register_dock_notifier()` function. This allows drivers to do anything that they want prior to handling a hotplug notification. This can be important if _DCK does something that needs to be undone. For example, on the IBM T41, the _DCK method will clear the secondary bus number for the parent of the dock bridge[2]. This makes it a bit hard for `acpiphp` to scan buses looking for new devices. `acpiphp` can register a function that is called by the `notifier_call_chain` that will clean up this mistake prior to calling the hotplug notification function.

[2]also highly unacceptable

```
notifier_call_chain(
&dock_notifier_list, event,
NULL);
```

Drivers or subsystems that need to be notified so that devices can be hotplugged can register a hotplug notification function with the dock driver by using the `register_hotplug_dock_device()` function. `hotplug_devices()` just walks the list of hotplug notification routines and calls each one of them in the order that it was received.

```
hotplug_devices(ds, event);
```

We clear the dock state bits to indicate that we are finished docking.

```
complete_dock(ds);
```

Now we alert userspace that a dock event has occurred. This event should be sent to the acpid program. If a userspace program is ever written or modified to care about dock events, they can use acpid to get those events.

```
if (acpi_bus_get_device(
ds→handle, &device))
    acpi_bus_generate_event(
device, event, 0);
```

Undocking is mostly just the reverse of docking. An undock is a `ACPI_NOTIFY_EJECT_REQUEST` type. Once again, we must not be in the middle of handling a dock event, and the dock device must be present in order to handle the eject request properly.

```
if (!dock_in_progress(ds) &&
dock_present(ds)) {
```

Because undocking may remove the `acpi_device` structure that we need to send dock events to userspace, we send our undock notification to the acpid prior to actually executing _DCK.

```
if (acpi_bus_get_device(
ds→handle, &device))
    acpi_bus_generate_event(
device, event, 0);
```

We also must call all the hotplug routines to notify them of the eject request. This is important to do prior to executing _DCK, since _DCK will release the physical connection and may make it impossible for clean removal of some devices. Finally, we can call `undock()`, which simply executes the _DCK method with the proper arguments.

```
hotplug_devices(ds, event);
undock(ds);
```

The ACPI spec requires that all dock stations (i.e. objects which define _DCK) also define an _EJ0 routine. This must be called after _DCK in order to properly undock. What this routine actually does is system dependent.

```
eject_dock(ds);
```

At this point, a call to _STA should indicate that the dock device is not present.

```
if (dock_present(ds))
    printk(KERN_ERR PREFIX
"Unable to undock!\n");
```

The design of the driver was intentionally kept strictly to handling dock events. For this reason, this is the only thing of interest that this driver does.

5 Conclusions

Dock stations make excellent test cases for hotplug related kernel code. Attempting to hotplug a device which can be a PCI bridge with a tree of devices under it exposed some interesting problems that apply to other devices besides dock stations. Right now we require the use of the pci=assign-buses parameter, mainly because the BIOS may not reserve enough bus numbers for us to insert a new dock bridge and other buses behind it. I found a couple problems with how bus numbers are assigned during my work which required patches to the PCI core. In many ways the problems that are faced with implementing hot dock are directly applicable to hotplugging on servers. Therefore, it is valuable work to continue, even if only 3 people in the world still use a docking station. We do believe that docking station usage will rise as system vendors create more compelling uses for them.

Dock station hardware implementations can really vary. It's very common to have a P2P bridge located on the dock station, with a tree of devices underneath it, however, it isn't the only implementation. Because of this, it's important to handle docking separately from any hotplug activity, so that all the intelligence for hotplug can be handled by the individual subsystems or drivers rather than in one gigantic dock driver. I have only implemented changes to allow one driver to hotplug after a dock, but more drivers or subsystems may be modified in the future.

I have very limited testing done at this point, and every time a new person tries the dock patches, the design must be modified to handle yet another hardware implementation. As usage increases, I expect that the implementation described in this paper will evolve to something which hopefully allows more and more laptop docking stations to "just work" with Linux.

References

[1] *Advanced Configuration and Power*

Interface specification. www.acpi.info, 3.0a edition.

[2] *PCI Hot Plug specification*. www.pcisig.com, 1.1 edition.

[3] *PCI Local Bus specification*. www.pcisig.com, 3.0 edition.

[4] *PCI-to-PCI Bridge specification*. www.pcisig.com, 1.2 edition.

[5] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, http://lwn.net/Kernel/LDD3/.

Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada