Multiple Passes of the FreeBSD Device Tree

John H. Baldwin
New York, NY 10005
jhb@FreeBSD.org, http://people.FreeBSD.org/~jhb

Abstract

The existing device driver framework in FreeBSD works fairly well for many tasks. However, there are a few problems that are not easily solved with the current design. These problems include having "real" device drivers for low-level hardware such as clocks and interrupt controllers, proper resource discovery and management, and allowing most drivers to always probe and attach in an environment where interrupts are enabled. I propose extending the device driver framework to support multiple passes over the device tree during boot. This would allow certain classes of drivers to be attached earlier and perform boot-time setup before other drivers are probed and attached. This in turn can be used to develop solutions to the earlier list of problems.

1 Introduction

One of the most basic tasks of an operating system is managing hardware resources. This includes directly controlling the hardware and providing interfaces to the hardware for application software. Logical functions of hardware resources are commonly called devices, and the software that controls a specific device is called a device driver. Devices may also use lower-level resources to interface to the rest of the system (examples include I/O ports, a memory-mapped I/O window, or interrupt signals). In some cases a device may provide lower-level resources for use by other devices. An example of this would be an interrupt controller which provides interrupt lines to other devices.

There is a wide assortment of hardware devices which require a correspondingly wide assortment of device drivers. To make it easier to write device drivers, operating systems typically provide a device driver framework. This framework might organize devices into a structure and/or name space. It also usually provides abstractions for managing the lower-level resources used by a device.

At its inception, FreeBSD inherited the "old-config" device driver framework from 4.2BSD [2]. This device driver framework suffered from the limitation that the knowledge of every device in a system was compiled into the kernel. Changing the hardware in a system required recompiling the kernel to update its configuration table or to add and remove drivers.

Hardware, however, was becoming more dynamic and intelligent. Some newer buses such as PCI and ISA Plug'n'Play provided mechanisms to enumerate devices on the bus. Other buses such as PCCard also supported adding and remove devices at run-time.

Although FreeBSD had extended "old-config" to add device entries for self-identifying buses such as PCI, new device drivers could not always be loaded into the system after boot. Instead, each bus driver had to provide its own infrastructure to manage this. The PCI bus driver did not support it at all. The PCCard bus driver did support loading drivers, however, it required a userland daemon to assign drivers to devices. This was problematic as the knowledge about which devices a driver supported was not contained in the driver itself but required patching a separate global config file.

In FreeBSD 3.0, a new device driver framework called "new-bus" [1] was developed as

part of the port to the Alpha architecture. This framework was ported to the i386 architecture in FreeBSD 4.0 and has been used on all FreeBSD architecture ports since. One of the key differences between "new-bus" and "old-config" is that with "new-bus" the kernel no longer contains a compiled-in list of static devices. Instead, devices on non-self-enumerating buses such as ISA are described by meta-data that is separate from the kernel. Secondly, in "new-bus" devices are organized into a hierarchical tree. Each node in the tree is represented by an object of type `device_t`. This allows "new-bus" to better handle dynamic devices.

Prior to "new-bus", each bus supported by FreeBSD was statically compiled into the the kernel. In the `configure` function called during boot, the kernel would call a function to probe all the devices for a specific type of bus. The PCI probe routine was responsible for walking the entire PCI device tree, for example. It attached drivers to PCI devices by scanning each PCI bus sequentially. After the PCI probe had finished, the ISA probe routine was called, etc.

In "new-bus", any buses or bridges in the system are treated as device objects in addition to tree leaves. For example, in "new-bus" each PCI bus device is responsible for scanning all of its child devices. Also, bridge devices create appropriate child bus devices. For many device related requests such as resource allocation, requests by a device are passed up the device tree hierarchy. This provides a simple way for buses and bridges to operate on requests from children device. For example, routing PCI INTx interrupts for devices behind one or more PCI-PCI bridges can require routing interrupts across each PCI-PCI bridge in different ways [3]. A top-down approach such as in "old-config" would require that the PCI support code know all the possible methods for routing an INTx interrupt as well as which method to use for a given bridge. In "new-bus", however, different PCI-PCI bridge drivers implement different routing methods and the bridge driver is only responsible for routing interrupts across itself. This allows a routing request to walk up the tree using different routing methods via different bridge drivers until it is satisfied.

2 Some Tricky Problems

One of the design goals of "new-bus" is that all devices in the system are probed and attached to drivers in a single pass of the device tree during boot. Rather than having the device driver support code automatically probe new devices as they are added, "new-bus" requires bus drivers (i.e. any device that creates children devices) to explicitly probe and attach any devices it creates during its attach routine. While this single-pass system has worked relatively well so far, there are some specific problems that it does not easily solve.

2.1 Low-level Hardware

Many hardware devices provide resources used by other hardware devices. For example, a bridge may provide ranges of address space for devices on a bus, or interrupt controllers may map device interrupt line assertions to CPU interrupt events. A device driver depends on these resources being available and usable while it is managing a device. Currently, these low-level devices are not always managed within the "new-bus" framework.

Some devices such as bridges are always probed before the devices that depend on them. Those devices are easily managed as "new-bus" devices. Other devices are not always probed first, however.

On x86 machines the 8259A interrupt controllers are generally enumerated as devices on the ISA bus that is the child of a PCI-ISA bridge. The PCI-ISA bridge is normally near the logical "end" of its parent PCI bus. It is very common for PCI devices that need interrupt resources to probe and attach before the ISA bus in these systems. The x86 platforms work around this by setting up interrupt controllers in platform-specific code that does not use "new-bus" devices. This code uses machine-dependent APIs to setup interrupt handlers and access I/O resources. Later during the "new-bus" device probe, dummy device drivers attach to the 8259A interrupt controller devices.

The ACPI bus driver uses a hack to probe certain devices earlier than others. To provide this, the ACPI bus driver knows the PnP IDs of specific "special" devices and inserts those devices earlier in its list of children devices to ensure they are probed before other devices. However, this is hacky as special knowledge about children devices belongs in the device drivers for those devices, not in the parent bus.

2.2 Resource Discovery and Management

One of the tasks of a device driver framework is to distribute low-level resources among devices. Some devices, such as non-PnP ISA devices, have a fixed set of hard-coded resources. For these devices, resource management consists of reserving those resources so that other devices do not try to use them. Other devices are more complex. They have fully configurable resources that may be assigned by system software. An example of this class of device is a PCI device.

Currently, FreeBSD relies on the system firmware to initialize most of the resources for PCI devices. This has improved in recent years as FreeBSD can now route PCI interrupts and it can assign resources to a device if the parent bus has available resources of the requested type. However, there are still some cases that are not handled.

One specific case involves allocating resources for devices behind a PCI-PCI bridge. Currently, if a device requests a memory or I/O port resource and there is not available room in the parent bridge's existing resource windows, then FreeBSD does not try to grow the resource window. There are some simple cases that could be solved in the current system. First, if the bridge has no window at all then an arbitrary resource range can be allocated from the parent. Second, if the bridge is able to extend its existing window it could use that extension to satisfy the original request. However, in many cases the request to extend the window will fail because the adjacent resource range is already assigned to another device. Moving resources for a device with an active device driver would be error prone. One way to handle this would be to walk the PCI device tree before any devices have started using resources to determine the requirements of all the devices and bridges. Appropriate ranges of resources could then be assigned to each PCI bus behind a PCI-PCI bridge possibly adjusting the ranges set by the firmware if needed.

2.3 Boot vs Non-boot Device Probing

Device drivers may attach to devices in two different environments. The first environment is the single pass of the device tree during boot. The second environment is after the system has booted and is fully initialized. This latter environment is used by drivers that are loaded via kldload(8) [5] after boot.

The boot time environment has several restrictions. Prior to FreeBSD 5.0, interrupts were not enabled until later in the boot process, so any drivers that needed to perform more complicated actions using interrupts had to defer that work. Even though interrupts are now enabled during this pass in recent versions of FreeBSD, the thread scheduler is not sufficiently initialized to allow threads to sleep while waiting for an interrupt. In addition, only one CPU is active and performing work during this time.

As a consequence of these restrictions, device drivers that need to perform tasks requiring interrupts when attaching to hardware must handle deferring this work. Although the config_intrconfighook(9) [4] API provides this functionality in a way that works in both environments, it still requires extra work in the driver to use. Having a more unified environment could simplify device drivers.

One of the requirements of the thread scheduler is the ability to provide time outs on sleep requests. This requires some sort of interrupt to fire when a time out request expires. Some platforms provide this via an interrupt from a timer device. For these platforms, the thread scheduler cannot be started

until the timer device has been attached.

3 Multiple-Pass Proposal

To aid with solving these problems, I propose extending the "new-bus" framework to perform multiple passes over the device tree during boot. Device driver attachments would now be assigned a "pass" value. A driver is only invoked to probe devices once the system-wide pass level is greater than or equal to the attachment's pass level. A driver's pass level is tied to a specific attachment. If a driver attaches to multiple buses, then it may have different pass levels for each attachment. This allows for stronger ordering of drivers with respect to each other across the device tree. It also allows for the kernel to perform work other than attaching drivers with a partially-attached tree.

The current implementation defers probing the majority of drivers until the final pass during boot. These drivers require no code changes. Drivers that wish to probe during an earlier pass do require changes.

3.1 Changes to "new-bus" Infrastructure

The changes to "new-bus" itself to support multiple passes are relatively minor. A few places have to be adjusted to ignore drivers whose pass level is greater than the current system-wide pass level. A facility for raising the pass level and triggering scans of the device tree for new pass levels is also required.

3.1.1 Special Handling of Driver Methods

A few places where "new-bus" invokes device driver methods must skip drivers with a pass level greater than the current system pass level. Most device driver methods are only invoked once a driver has probed and attached to a device. If device drivers are prevented from probing devices too early, then those methods do not need special handling. In fact, the only methods which must be handled specially are `DEVICE_IDENTIFY` and `DEVICE_PROBE`. Specifically, `bus_generic_probe` only invokes `DEVICE_IDENTIFY` for driver attachments whose pass level is less than or equal to the current pass. Similarly, `device_probe` only tries driver attachments whose pass level is less than or equal to the current pass. This has the effect that device drivers may always use `bus_generic_probe` and `bus_generic_attach` to attach children devices as they do now.

One bus method also requires special treatment. The `BUS_PROBE_NOMATCH` method is called for devices that are not probed by any drivers during boot. Most devices will not be probed by any drivers during early passes, however. This would result in many spurious calls of `BUS_PROBE_NOMATCH`. The solution is to change `device_probe` to only invoke this method during the final pass.

3.1.2 Managing Pass Levels

Most of the changes to "new-bus" provide management of pass levels. These changes include tracking the passes used by drivers, raising the pass level, and rescanning the device tree.

One of the goals of this implementation is to support a dynamic set of sparse pass levels. The interface should be similar to the subsystem levels used with `SYSINIT()` [6]. Adding a new driver with a new pass level to the system should cause "new-bus" to rescan the tree for that pass.

A simple implementation would be to rescan the tree for every possible pass level during boot from zero to `INT_MAX`. However, it is expected that there will normally be very few active pass levels, so the vast majority of these tree scans would be wasted effort. Instead, a new list of active pass levels is maintained. The list is sorted by pass level and contains one driver per pass level. When a new driver is registered with the system dur-

ing boot, that driver is added to the list if it uses a previously-unused pass level. This provides a quick and easy way to enumerate the pass levels in use.

The current system pass level is stored in a new global variable `bus_current_pass`. The system pass level can be raised to a specific value by calling the new `bus_set_pass` function. The requested pass level does not have to be used by any drivers, but lowering the pass level is not permitted.

The `bus_set_pass` function may invoke multiple scans of the tree during a single call. It walks the list of active pass levels until it either hits the end of the list or encounters a pass level higher than the requested level. Any pass levels less than the current system pass level are skipped. The remaining pass levels each trigger a separate scan of the device tree.

Rescanning the device tree is implemented by a new bus method `BUS_NEW_PASS`. The `bus_set_pass` function invokes this method on the `root_bus` device each time the pass level is raised. A default implementation is provided by the new `bus_generic_new_pass` function. It first walks all the driver attachments for the current device. If any of the attachments use the new pass level, then their `DEVICE_IDENTIFY` method is invoked. After this is completed, it walks the list of children devices. If a child device has an attached driver, then the driver's `BUS_NEW_PASS` method is invoked. Otherwise, the device is reprobed. This allows drivers that were made eligible for probing by the new pass to now probe the device.

3.2 Writing an Early Pass Driver

In a system with multiple passes of the device tree, the majority of existing drivers will only probe devices during the final pass. These drivers do not need any modifications. Drivers that do wish to probe devices during an earlier pass do require modifications, however.

All early drivers are required to indicate the earliest pass at which they are eligible to probe devices. This is accomplished by a new `EARLY_DRIVER_MODULE` macro. This macro is similar to the existing `DRIVER_MODULE` macro. It simply adds a new argument to specify the pass level of the driver attachment. For some drivers this is the only modification needed.

Bus drivers may need additional changes to their attach routines. Specifically, many of the tasks bus drivers currently preform in their attach routines may need to be deferred until a specific pass has completed. For example, PCI bus drivers should not attempt to route interrupts for child devices until after the pass which adds interrupt controller drivers is completed. Buses should also not assign resources to devices until system resource drivers have attached and reserved resources that other devices should not use.

One possible method for addressing this is for bus drivers to provide a custom method for `BUS_NEW_PASS`. This custom method would perform actions dependant on an earlier pass level the first time the pass level is raised to a greater level. This would be a bit clunky and require bus drivers to keep track of which initialization steps had already been performed, however. It is also not very intuitive that one cannot simply "hook" into an existing pass level. Instead, the bus driver needs to be certain that an entire pass has completed before it performs actions that depend on that pass. This requires the actions to be deferred until the next pass level instead.

Another approach would be to add specific event notification methods to the bus interface. For example, a new `BUS_ASSIGN_RESOURCES` method would be invoked at the top-level when the system was ready for buses to assign resources to child devices. One downside of this method is that bus drivers would be required to explicitly pass the notifications down to continue the tree scans. This could be somewhat mitigated by providing default implementations similar to `bus_generic_new_pass`.

If a bus driver can be attached after boot, then any changes made to its attach routine will need to take this into account. It can do this by conditionally performing tasks such as

resource allocation in its attach routine based on the current system pass level.

Finally, if a bus driver uses hints to enumerate children devices it may need to defer adding hinted children. If all of the bus's child devices are enumerated by hints, then no changes are needed. However, if the bus supports a mixture of self-enumerated devices and hint-enumerated devices and allows self-enumerated devices to claim hint devices via `bus_hint_device_unit`, then special care must be taken to not add hint-enumerated devices until after all of the self-enumerated devices have been probed during the final pass. The only driver that currently has to deal with this is the ISA bus driver on the amd64 and i386 platforms.

3.3 Pass Levels

The multiple pass system is designed to easily allow new pass levels to be added. At a minimum it requires two pass levels to be present, the initial pass level used to attach the `root_bus` device and the default pass level used by most drivers. However, for the system to be useful additional pass levels must be used. A list of possible pass levels in increasing order follows. Each level is named by a constant present in `<sys/bus.h>`.

The `BUS_PASS_ROOT` pass level is level 0 and is reserved as a marker for the `root_bus` device. The root bus driver does not probe and attach normally, so it does not have an actual pass number assigned. Instead, "newbus" creates the device and assigns the driver manually to provide a starting point for the device tree. Pass level 0 is similarly special in that it is the initial system pass level. No drivers should use this pass level.

The `BUS_PASS_BUS` pass level is used by bus and bridge drivers. These drivers are responsible for populating the device tree. Note that other early drivers need device nodes to probe and attach to, so this is a prerequisite for other early drivers.

The `BUS_PASS_CPU` pass level is used to create devices for CPUs. Note that devices that attach to CPUs such as cpufreq(4) drivers may probe later. This is simply a continuation of the previous level to fully enumerate the device tree.

The `BUS_PASS_RESOURCE` pass level is used by drivers that need to probe before resources are assigned to devices. System resource drivers would attach at this pass. Once this pass is complete, bus drivers may assign non-interrupt resources to devices.

The `BUS_PASS_INTERRUPT` pass level is used by drivers that provide interrupt support services. Interrupt controllers and PCI interrupt routers would attach during this pass. Once this pass is complete, bus drivers may assign interrupt resources to devices.

The `BUS_PASS_TIMER` pass level is used by drivers that implement any timers needed to drive the thread scheduler. Clock drivers would attach during this pass.

The `BUS_PASS_SCHEDULER` pass level is used to indicate the point at which the thread scheduler is started. Any other devices not already probed that are required for the thread scheduler would attach during this pass. Once this pass is complete, the thread scheduler could be started.

Finally, the `BUS_PASS_DEFAULT` pass level is the final pass level. It has a value of `INT_MAX`. All devices not already probed by an earlier pass would attach during this pass if a suitable driver is available.

4 Tricky Problems Revisited

On its own multiple passes of the device tree does not solve any problems. However, it does provide a framework that can be used to solve the problems described earlier. A possible solution to each problem using multiple passes is outlined below.

4.1 Low-level Hardware

Of the problems listed, drivers for low-level hardware is probably the simplest to solve. With the multiple pass framework, the software to manage these devices can move from platform-specific start-up code into platform-specific early drivers. For example, interrupt controllers on x86 platforms can probe as "new-bus" devices that properly reserve all their associated resources.

In the case of the ACPI bus driver, the hack to order specific devices based on device IDs can be removed. Instead, the drivers for the various "special" devices can become early drivers. The pass levels of these drivers can be used to guarantee ordering.

4.2 Resource Discovery and Management

In the previous discussion of resource discovery and management, a solution for PCI bus hierarchies was proposed. This solution was to walk the device tree allocating resources as necessary before attaching drivers. One of the complications with this is that this requires recursing into buses behind PCI-PCI bridges and maintaining state for each bus, etc. This can be accomplished more easily if each PCI-PCI bridge and PCI bus are managed by a device driver when this process is performed. In that case, each driver is only responsible for managing its own state.

A way to achieve this using early drivers is to allow PCI bridge and bus drivers to probe as `BUS_PASS_BUS` devices which enumerate all child devices during the initial attach. However, resources would not be allocated to child devices until a later point such as after the `BUS_PASS_RESOURCE` pass has completed. PCI bus drivers would provide methods to enumerate the range of resources required for child devices. The implementation of this method would include special handling for PCI-PCI bridges that called down into the child PCI bus to determine its resource requirements and use that to program the decoding windows on the parent side of the PCI-PCI bridge.

4.3 Boot vs Non-boot Device Probing

One of the largest differences between boot and non-boot device probing is the lack of thread scheduling during boot. One of the things the thread scheduler requires to run are working timers to support sleep timeouts. Changing the drivers for timer devices to be early devices that probe as `BUS_PASS_TIMER` would allow these devices to probe before most other drivers. After the `BUS_PASS_SCHEDULER` pass, the thread scheduler could be started. All drivers probed after that would then probe in an environment similar to the post-boot environment.

5 Conclusion

Multiple passes of the device tree provides a framework that can be used to solve several problems. The current design does not require any modification to the majority of device drivers and seeks to minimize the requirements of early device drivers. In fact, most of the changes to early drivers result from the need to solve other problems such as improved resource management.

6 Acknowledgments

Thanks to Attilio Rao for reviewing this paper. Thanks also to the several folks who have worked on the "new-bus" infrastructure including (but not limited to) Matthew Dodd, Warner Losh, Doug Rabson, Peter Wemm, and Garrett Wollman.

7 Availability

Currently the multiple pass support code is present in the

`//depot/projects/multipass/...` branch in the FreeBSD Perforce depot. Due to the nature of kobj, this framework could be merged back to stable branches with a small ABI compatibility stub for the `driver_module_handler` function.

This document is available at `http://www.FreeBSD.org/~jhb/papers/bsdcan_09`.

References

[1] *NEWBUS*, FreeBSD Architecture Handbook, `http://www.FreeBSD.org/doc/en/books/arch-handbook`

[2] Marshall Kirk McKusick, George V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*, Pearson Education, Inc. (2005) p. 281.

[3] John Baldwin, "PCI Interrupts for x86 Machines under FreeBSD", Proceedings of the BSDCan Conference (2008).

[4] *config_intrhook*, FreeBSD Kernel Developer's Manual, `http://www.FreeBSD.org/cgi/man.cgi`

[5] *kldload*, FreeBSD System Manager's Manual, `http://www.FreeBSD.org/cgi/man.cgi`

[6] *The SYSINIT Framework*, FreeBSD Architecture Handbook, `http://www.FreeBSD.org/doc/en/books/arch-handbook`