USENIX Association

Proceedings of BSDCon '03

San Mateo, CA, USA
September 8–12, 2003

devd – A device configuration daemon

M. Warner Losh
Timing Solutions, Inc
Boulder, Co
imp@bsdimp.com

Abstract

Hot–pluggable bus technologies have proliferated, rendering traditional boot time configuration of devices via an `/etc/rc` script insufficient for many user's needs. Most implementations of hot–plug technologies have provided a means to address these deficiencies, yet their solutions tend to be confined to only that technology. The goal of FreeBSD's devd(8) is to provide a uniform framework by which interesting events relating to hot–plugging can be handled. devd provides a regular framework for these technologies to have user–land configuration commands run in a generic, extensible way. The implementation encountered a number of issues which are instructive to explore. devd only responds to events that the kernel generates and does not participate in interactions with the kernel that would block another thread of execution. At the present time, devd supports executing arbitrary commands when a driver attaches to the tree, when it detaches and when a bus detects an unknown device attached to that bus.

1 Problem Statement

When Unix was originally designed, all the devices that the machine had were hardwired into the kernel. Over time mechanisms evolved to allow users to configure which devices their kernels should have. As technology improved, hardware became hot–pluggable and self identifying. Traditional Unix kernels are awkward to use on a machine where devices dynamically change after the kernel has booted. A number of hot–plug technologies have been brought to market over the past few years: 16–bit and 32–bit PC Card[PC Card], IEEE 1394–1995[Firewire], HotPlug PCI[PCIHotPlug], USB 1.0 and 2.0[USB], SCSI, ATA, SATA, and others.

Traditional Unix systems could only configure network interfaces, or otherwise bring devices into a useful state during the boot process through `/etc/rc` or similar mechanisms. Since many hot–pluggable devices do not exist at boot, some other mechanism was needed to bring these devices to a useful state.

Typically, hot–plug technologies have been integrated in an ah–hoc way. For example, the PC Card daemon `pccardd` in FreeBSD handles only 16–bit PC Cards. The USB daemon `usbd` in FreeBSD and NetBSD handles only USB devices. Each of these daemons does its job well, but is specialized to its particular technology. These daemons have low reusability for new technologies. Each new technology needs a new daemon to be written. There's presently no daemon to handle firewire devices on either FreeBSD or NetBSD, for example. The proliferation of daemons takes up additional resources, and wastes programming effort for each new technology.

A related problem is that sometimes devices are attached to busses for which no driver exists in the kernel. Bus technologies that support hot–plugging of devices typically have some sort of so called "Plug and Play" identifiers that can be used to intelligently select the driver. In FreeBSD there is only limited support for loading device drivers when unknown devices are present. Only `usbd` supports loading drivers when unknown hardware is plugged in, and nothing supports augmenting drivers for unknown devices encountered during the boot phase.

Many of these busses also provide "location" information to uniquely identify multiple instances of the same kind of card. While not necessarily used by most devd users, this location information is exposed in a uniform way. This uniformity allows users to configure their system based on where a device is, rather than what order it happens to probe

in. Prior to the devd efforts, FreeBSD could only find the location of a device for most busses via bus specific methods that often proved unreliable.

devd was conceived to try to solve these problems. devd will be used in this paper to refer to all parts of the system, even though the devd daemon is only part of the solution.

2 Prior Art

A number of ad–hoc solutions to these problems have existed over the years. Each one of them solved a small slice of the problem set. My search for prior art didn't uncover any truly generic method for all drivers in a system.

2.1 FreeBSD pccardd

The original PC Card implementation in FreeBSD, sometimes called OLDCARD, has a split user-land kernel implementation. The bare minimum of functions that were required in the kernel were implemented there, but the bulk of the PC Card configuration procedures were implemented in a daemon called `pccardd`. The kernel would tell this daemon of card insertions. The daemon would then parse the CIS information from the card, chose a driver and its resources based on a configuration file and tell the kernel which device to attach. When that was done, it would optionally run additional commands. It did similar things when a card was ejected. A companion program pccardc offered some additional control to the system, such as dumping a CIS for debugging purposes, or powering off a card.

This system worked fairly well when it was written. In those days, most of the hardware was still ISA based and everything was hardwired at a particular address. As systems became more complicated, problems arose with resource conflicts. In addition, as bus technologies changed from edge triggered interrupts to level triggered interrupts, the split between user–land and the kernel was found to be suboptimal because some of the things that the user–land daemon was doing would cause interrupt storms because the kernel didn't expect to cooperate with `pccardd` for those operations.

2.2 NetBSD usbd

NetBSD's `usbd` provided a way for a user-land process to react to events in the usb system and to force a traversal of the usb tree. NetBSD moved this functionality into a kernel thread. FreeBSD expanded its `usbd` similarly to FreeBSD's `pccardd` to allow for aribtrary commands to be run on attach and detach. The underlying usb driver provided hooks for additional events that were never used by `usbd`. In addition, all the usb configuration data comes from the kernel, rather than from parsing the data like `pccardd` does.

2.3 Windows

While Windows doesn't exactly support running a command when a device attaches or detaches from the device tree, it does define a similar C API. Windows solves the problem of which driver to load and attach differently. In the windows world, a configuration file associates a driver to a plug and play ID and the device manager, not the device driver, decides which devices a driver services.

2.4 MAC OS/X

OS/X facilities are similar to Windows. OS/X uses XML based config files to determine which device driver is attached to a given device. I am unaware of command execution when a driver attaches or detaches.

3 Design Overview

3.1 Unix Philosophy

The Unix philosophy has been to keep things as simple as possible. Each tool does one small job, and does it every well. Complex problems are solved by stringing together simple tools that solve orthogonal problems. Much like cat(1) does not try to implement paging functionality, devd does not try to dictate policy to the kernel. devd is reactionary: it only responds to events that have already happened. It

does not participate in the decision making process of kernel device configuration.

3.2 Big Picture

Let's look at a typical case to illustrate how the process works. A wireless card is inserted into a CardBus slot. The CardBus bridge notices this has happened, and causes all drivers with CardBus attachments to be probed. One of them claims the card, and then its attach routine is called. After a successful attach, an event is sent to the devd daemon. The daemon inspects the event and decides that it is configured to start `dhclient` and does so.

All the data flow is one way. Some event happens, which causes a device driver to attach. That act causes an event to be sent to devd, which does something. devd only reacts to events that have happened in the past. devd does not participate in the bidding on a device. The kernel will never block waiting for data from devd.

3.3 Kernel Driver

FreeBSD stores the kernel device driver tree in a generic format for all devices in the system. Since manipulation of the tree is confined to a few well known routines, the first part of devd is a driver that hooks into this code. This driver queues this information and delivers it to user–land via the `/dev/devctl` device. Each message is one line of text. The messages have a well defined meta-format so messages not known to a particular client may easily be discarded without loss of synchronization in the protocol stream.

3.4 User–land Daemon

The user–land daemon, called devd, processes the events from the kernel and dispatches them based on devd's configuration file `/etc/devd.conf`. This dispatching is regular expression based. Any information that the kernel provides about the event can be matched. Actions are arbitrary Unix commands.

4 Implementation Details

4.1 Kernel portion

The kernel portion of devd is implemented in `/usr/src/sys/kern/subr_bus.c`. Hooks in the attach, detach and nomatch portions of the NEWBUS code queue events, if `devctl` is not disabled. The queue depth is unlimited, but it is cleared when `devctl` is disabled early in the boot process for those systems that choose not to use devd's features. Look for the functions `devadded()`, `devremoved()` and `devnomatch()` for details. Since `devctl` hooks in at such a low level of FreeBSD's NEWBUS system, all busses in the system are automatically supported at a rudimentary level.

The interface to the kernel is through the `/dev/devctl` device driver. The devd daemon opens this device, and reads events from the queue using the standard read(2) calls. Each event is on a line by itself, in ascii format. In theory, this interface would support a devd-like program written in Perl, Ruby or some other programming language. `devctl` also supports ioctls for attaching and detaching devices, but those ioctls are beyond the scope of this paper. `devctl` presently supports only one reader, but future versions will have a cloning driver that will allow multiple readers as people write status programs that wish to listen to the event stream from the kernel.

4.2 Bus driver

While full support of all the devd functionality requires bus drivers to implement several functions, minimal support for devd only requires bug free operation of the underlying bus.

The most basic requirement is that busses must provide NEWBUS attachments to their children. Busses may provide additional information about location and identifying characteristics of the device. Finally, they may also support bus rescanning when a new driver which has attachments on that bus is loaded.

Most of the busses in the system do provide NEWBUS attachments for each of their children. Since it mostly predates the NEWBUS system, CAM

doesn't use NEWBUS children for the SCSI and ATAPI devices it finds. Since the usb stack was ported from NetBSD, not all of its devices are NEWBUS devices. For these busses, the support for them from devd is limited. All the other busses in the tree that have been examined use full NEWBUS children for all the devices attached to the bus.

While all the busses using NEWBUS for their children are automatically supported at a basic level due to devctl's hooks inside of newcard, each bus can enhance its support for devd in three ways. First, it can implement a location kobj function, bus_child_location_str(), which devctl uses to tell devd where a given driver has attached to the bus. Second, the bus can report identification information with the bus_child_pnpinfo_str() method. Finally, to support automatic loading of drivers for unknown devices, the bus needs to support rescanning unattached devices when a new driver is loaded.

Location information is a bus specific way of locating the device, possibly uniquely. Many busses have a notion of a slot number where the card with the function(s) resides. Other busses have a notion of where the device lives within a tree of objects. The information returned by the bus_child_location_str, method can therefore be used by the user locate definitively the device.

Information about the nature of the device may also be supplied. This information is similar to the plug and play strings that Windows generates and uses in its device configuration. Busses wishing to implement this are required to implement the bus_child_pnpinfo_str kobj method. This method generates strings which are presented to devd by way of devctl. These string aren't strictly required, as devctl will simply pass less information about the device up to devd when they aren't present. When they are present, they can be used to select the driver to load that supports the hardware.

Finally, bus drivers that wish to fully support devd's functionality must also implement effective reprobing on module loading. NEWBUS provides hooks necessary to implement this feature, and most busses already support it. However, some busses attach a generic device to those devices that do not otherwise match. In those cases, the bus will need to detach that device driver from those children that are generically matched in order to allow the newly loaded driver a chance to bid on the device.

4.3 devd

devd dispatches commands based how a kernel event matches information from a configuration file. devd reads this configuration file on startup. This config file may contain a list of directories from which to read additional files. These additional files may contain anything that's in the main directory, and are only scanned once at startup. Directories are only scanned once, even if they are listed multiple times.

After devd reads it config file, it begins to process events. Since devd is a daemon, it will place itself into the background. There are two modes of operation for this. The first mode, the default, waits until all the events in the queue have been read and their actions dispatched. This is used during boot to eliminate a race later in the boot process between devd configuring a device and rc scripts using those results. The second mode of operation causes devd to become a daemon immediately after reading its config file. This mode can be used to optimize boot time when no such dependency exists.

There are three kinds of events that devctl produces for devd to consume. When a bus driver attaches a driver to a device on that bus, an attach event is generated. When a driver detaches from a device, the detach event is generated. When a bus driver detects a device on the bus no driver claims, a nomatch event is generated.

Attach and detach events happen after the fact. The driver is already attached or detached before the events happen. The command(s) that are run in a matching action must take this into account. This especially means that it is not possible to run any commands before the driver detaches. In controlled situations, this may be undesirable behavior, but when a card is ejected, say, from a PC Card slot that hardware is gone and there is no control possible. In addition, a fundamental design decision of devd was to have it be reactionary only. Having the kernel stall on devd before detaching the device was deemed undesirable. Two way communication is much harder than one way communication.

A nomatch event is generated when a driver cannot be matched to a device. This may happen if the driver for that device is not in the kernel, or if no such driver exists. The typical response for a nomatch event is to either load a driver (in the former

case), or to ignore the event (in the latter). Again, devd is reactionary. It does not participate in the bidding process for the device. However, FreeBSD's NEWBUS system allows devd to be reactionary, while still allowing the device to ultimately attach. When the driver is loaded, the bus driver(s) for that driver will rescan all of their unattached children. Any new driver(s) to the system will be eligible to attach to the unknown device on this rescan. Another implication of this is that if a few different drivers could attach to the device, all can be loaded and the right one will win the bidding. While the "Plug and Play" information is typically sufficient to select only a unique driver to load, in some rare cases it only selectes one of two drivers.

5 Configuring devd

All of the details about devd's config file can be found in the devd.conf(5) manual page on FreeBSD[devd.conf]. Briefly, the config file is similar to configuration files for such software packages as bind and dhcpd. A number of different sections are used to control devd's behavior. A section exists for global configuration information. In addition a number of sections control the actions to take on certain events. Finally, a number of different comment styles are provided.

5.1 Options Section

The options section contains different options for devd's operation. Configuration options are specified as the option name, followed by one or more words or strings as appropriate for the option. While many options exist, only two will be discussed here.

The **directory** option takes one string. This string specifies a directory to read additional files from. All the files in the listed directory are merged into the configuration of devd, as if their contents had been appended to the original devd.conf file. The directory option may be specified many times; however devd only scans any given directory once. This slightly arcane mechanism was designed to allow for packages to participate in devd with a simple addition of a file on install and a simple unlink on removal.

The **set** option takes two parameters. The first parameter is the name of the variable to set. The second variable is the string to set it to. devd sets a number of its own variables during event processing, discussed below, and the set command augments those variables.

5.2 Attach, Detach and Nomatch Section

Each of these sections is given a weight. For each action type, the sections are sorted in decreasing order. Each section consists of zero or more matching directives, and zero or more actions. When processing an event, the sections are scanned in their sorted order. The first one whose matching directives all match is considered the best match. Only the section with the best match has its actions executed. All actions will be executed, reguardless of their exit code.

Each **match** directive contains two strings. The first string is the keyword to match against. The second string is the regular expression to match against. Both of these strings have devd variables expanded before their use. Each device event contains a number of key value pairs that are the plug and play information and location information the parent device provides. This is discussed in the section on event generation above.

`device-name "foo"` is a shorthand for `match "device-name" "foo"`. It is nothing more than syntactic sugar.

The **action** directives have one string. This is the string to execute. Like the **match** directive, it too has its devd variables expanded before the strings are used.

All three types of sections have identical syntax. However, the **attach** section is run only on attach events from devctl; the **detach** section on detach events; and the **nomatch** section on nomatch events.

6 A Few Examples

A few examples will illustrate how devd works from end to end. Each section will contain an excerpt from a configuration file, a description of the problem and solution and a walk through of all the events that happen in the system.

6.1 Using a Wireless CardBus Card

In this example, we have just purchased a brand new Atheros wireless card, supported by the `ath` driver. Assume that the `ath` driver is compiled into the kernel. Let us further suppose that if this machine has a atheros card on a pci bus, we want some other mechanism to configure the card. Figure 1 shows the relevant portion of the devd configuration file for this example.

```
attach 10 {
match "bus" "cardbus[0-9]+";
device-name "ath[0-9]+";
action "/etc/wlan $device-name start"
};
detach 10 {
match "bus" "cardbus[0-9]+";
device-name "ath[0-9]+";
action "/etc/wlan $device-name stop"
};
```

Figure 1: Simple wireless configuration file.

In this example, the following events happen:

1. The user inserts an Atheros wireless card into one of the CardBus slots
2. The CardBus bridge notices the card has been inserted
3. The CardBus bus attaches the ath driver to the card
4. /dev/devctl generates an attach event similar to the following:

   ```
   +ath0 at slot=0 function=0 on cardbus1
   ```

5. devd reads this event
6. devd sets "device-name" equal to "ath0", "slot" to "0", "function" to "0" and "bus" to "cardbus1".
7. determines that the above attach clause is the best match
8. devd expands the strings and executes "/etc/wlan ath0 start"
9. start-wlan-card configures the ath0 interface and the user goes wireless
10. time passes
11. the user ejects the atheros card
12. the CardBus bridge notices that the card is gone and the ath0 driver detaches from cardbus0
13. /dev/devctl generates an attach event similar to the following:

    ```
    -ath0 at slot=0 function=0 on cardbus1
    ```

14. devd reads the event, set variables and matches the above detach clause.
15. devd executes "/etc/wlan ath0 stop"

6.2 Automatic Driver Loading

In this example, a driver for the ALi M7101 Power Management Controller called `apmc`. In this example, the system is booting an encounters an unknown device. Figure 2 shows the relevant portion of the devd configuration file for this example.

```
attach 10 {
match "bus" "pci[0-9]+";
device-name "apmc[0-9]+";
action "/etc/powermon $device-name start"
};
detach 10 {
match "bus" "pci[0-9]+";
device-name "apmc[0-9]+";
action "/etc/powermon $device-name stop"
};
nomatch 10 {
match "bus" "pci[0-9]+";
match "vendor" "0x10b9";
match "device" "0x7101";
action "kldload apmc"
};
```

Figure 2: Unknown device configuration file.

In this example, the following events happen:

1. The system boots and pci bus 2 is scanning its children
2. No driver accepts the device in slot 17
3. /dev/devctl generates a nomatch event similar to the following (line breaks have been inserted for clarity):

```
? vendor=0x10b9 device=0x7101
  subvendor=0x1265 subdevice=0x7101
  class=0x068000 at slot=17
  function=0 on pci2
```

4. eventually devd starts and reads events
5. devd reads the nomatch event, sets the variables, and picks the nomatch clause above.
6. devd executes "kldload apmc"
7. The pci bus rescans its unattached children because a new pci driver has been loaded
8. the `apmc` driver attaches to the device in slot 17
9. /dev/devctl generates an attach event similar to the following:

```
+apmc0 at slot=17 function=0 on pci2
```

10. devd reads this event, sets the variables and executes the event.
11. devd expands strings and executes "/etc/powermon apmc0 start"

7 Lessons Learned

A number of problems were encountered during the development and deployment of devd. This section details those lessons, as well as the solutions that were arrived at for them.

7.1 To queue or not to queue

The initial implementation of devctl didn't queue events if there was no daemon running. There was a problem with this. Devd would have then ignored removable devices present at boot. In this case the rc system could take over, but it is optimized to static devices, so some functionality that devd offers would be lost. In addition, devd would have seen detach events when/if the card was removed without a corresponding insertion event.

The solution was to always queue the events. This allowed devd to receive the attach events for everything in the system and to process them for each device. It also allowed devd load drivers for unknown devices. There was a problem with this solution also. It assumes that devd will run relatively early in the boot process to reclaim the memory used by `devctl`'s queue. However, the user can choose not to run devd, or devd could have a bug and exit. In these cases, unbounded event queue would consume kernel resources indefinitely.

The solution to the problems introduced by the first solution was to add a sysctl that would enable or disable devctl's queuing of events. This sysctl would also free any events in the queue, freeing their memory. This solution works well for most cases, and does exactly what it says it does. It was a simple matter to wire this sysctl to a startup script parameter to turn devctl off in the kernel. Since devd was disabled by default, this solved the resource utilization issue.

One problem encountered quickly during prototyping with this solution was that users thought devd was broken. The solution was to have devd automatically re-enable `devctl` when it was disabled on startup. This solved the foot shooting issue without introducing more problems.

From the original queueing question, I found and corrected four issues. It makes one think of the rabbits in Australia.

7.2 Ordering

The next problem with devd was one of ordering. I naively thought that running devd as early as possible would be the best possible thing to do. The thinking was that devd could be used to configure any sort of device, and one wants to do that as early in the boot process as possible. However, there were problems with this. Since the environment that exists early in the boot process is severely limited, the file systems that devd can access to configure the devices is limited to those files in the root filesystem. This environment is a fairly restricted one.

Many users require more features to configure their devices. Many scripts use languages in /usr/local, such as Perl or Ruby. A number of desirable binaries are in /usr/bin or /usr/sbin, such as awk or wicontrol. Placing things last is too late, since it precludes configuring network cards early enough to be used by daemons, etc. In the end, starting devd just before the rc.d scripts mounted the remote file systems. In this way, devd users could maximize its use of the local resources available, while at the same time maximizing the amount of the startup process that can utilize the dynamically configured devices.

Since both devd and /etc/rc.d are both used to configure the kernel devices from user-land, some conflicts arose. This configuration typically is forming ifconfig commands for network devices, mounting remote file systems, loading firmware in some devices and so forth. Since devd treats all devices the same, both dynamic and static, some method for making sure that devices weren't configured multiple times was needed. This issue was resolved by using the 'legacy' configuration methods first. The devd scripts have checks to make sure that devices aren't already configured before doing its configuration.

7.3 Racing in the background

In the prototype versions, devd became a daemon very early in the process, so as to not hold up the boot while it read and processed its config file. There was a subtle problem with this approach that took a while to understand. If there were dependencies on devd having configured a device in the rest of the startup code, devd might not have configured it by the time it was needed because devd was running in the background, asynchronously to the rest of the boot process. For example, if a network interface were configured and then an NFS filesystem mounted over that interface, devd may or may not be able to configure the card's network address before the nfs mount started attempted. This lead to a mount failure. If this configuration is done with soemthing like `dhclient`, the configuration time is variable and possibly significant. To address this problem, devd processes all of the outstanding events from `/dev/devctl` before becomes a daemon. This stalls the boot process until all the devices are configured, eliminating the race. Since some users do not desire this stalling, devd can also optionally become a daemon early in the process to not stall the rest of the boot process.

8 Acknowledgments

I would like to thank Poul–Henning Kamp, John Baldwin, Peter Wemm and Scott Long for reviewing early designs and providing valuable insight into the problems.

9 Future Work

devd is a good start. However, it is a very simple tool. It provides the basic configuration stuff. Additional information is necessary to keep the full state of a system. This is needed for those people that only want to run sshd, for example, when they have a network card installed. FreeBSD's driver configuration is in such a state that it is not easy to produce tables of "Plug and Play" information that easily could translate into devd config files for dynamic loading of device drivers from a minimal kernel.

Power management events would offer a good future expansion of devd's event structure. Suspend and resume events are currently handled by `apmd` for APM. However, ACPI is replacing APM and it has no equivalent to `apmd`. Having yet another daemon to process events seems wasteful, so it may be profitable to explore expanding devd to encompass these events. Network interface events, such as carrier detect or carrier loss, may also be fertile ground for future expansion. It is not clear if a separate daemon is needed for the network events, or if programs like `dhclient` could handle those events in a smarter manner.

10 Availability

The software described in this paper have been integrated into FreeBSD 5.1–RELEASE. FreeBSD is available free of charge for download from **http://www.freebsd.org/** in many different forms. Improvements to devd will be incorporated into future versions of FreeBSD.

References

[devd.conf] **/usr/share/man/man5/devd.conf.5**

[Firewire] **http://developer.apple.com/firewire/platform.html**

[FreeBSD] **http://www.freebsd.org/**

[Linux–HotPlug] **http://sourceforge.net/projects/linux–hotplug/**

[NetBSD] **http://www.netbsd.org/**

[OpenBSD] **http://www.openbsd.org/**

[PC Card] *PC Card Standard, Release 7.0*, PCMCIA, (February 1999). **http://www.pcmcia.org**

[PCIHotPlug] *PCI Hot–Plug Specification, Revision 1.0*, PCI Sig, (October 6, 1997). **http://www.pcisig.com**

[USB] **http://www.usb.org/developers/docs.html**