Kernel Korner

Hot Plug

Greg describes the new framework in the Linux kernel for supporting USB and other hot-pluggable devices.

by Greg Kroah-Hartman

Hot-pluggable devices have been created to solve a number of user needs. On laptop computers, PCMCIA devices were designed to allow the user to swap cards while the computer was still running. This allowed people to change network adaptors, memory cards and even disk drives without shutting down the machine.

The success of this led to the creation of the USB and IEEE1394 (FireWire) buses. These designs allow for peripherals to be attached and removed at any point. They also were created to try to move systems away from the ISA bus to a full Plug-and-Play-type system.

From the operating system's point of view, there are many problems with hot plugging devices. In the past, the operating system only had to search for the various devices connected to it on power-up, and once seen, the device would never go away. From the view of the device driver, it never expects to have the hardware that it is trying to control disappear. But with hot-pluggable devices, all of this changes.

Now the operating system has to have a mechanism that constantly detects if a new device appears. This usually is done by a bus-specific manager. This manager handles the scanning for new devices and recognizes this disappearance. It must be able to create system resources for the new device and pass control off to a specific driver. The device driver for a hot-pluggable device has to be able to recover gracefully when the hardware is removed and be able to bind itself to new hardware at any moment. Not only does the kernel need to know when devices are removed or added, but the user also should be notified when this happens. Other kinds of kernel events, such as the creation of network devices or the insertion of a laptop into a docking station, also would be useful for the user to know about.

This article describes the new framework in the Linux kernel for supporting USB and other hot-pluggable devices. It covers how the past implementation of PCMCIA loaded its drivers and the problems of that system. It presents the current method of loading USB and PCI drivers, and how this same framework can handle other kinds of user configuration issues easily.

The Past

Linux has had support for PCMCIA since 1995. In order for the PCMCIA core to be able to load drivers when a new device was inserted, it had a user-space program called cardmgr. The cardmgr program would receive notification from the kernel's PCMCIA core when a device had been inserted or removed and use that information to load or unload the proper driver for that card. It used a configuration file located at /etc/pcmcia/config to determine which driver should be used for which card. This configuration file needed to be kept up to date with which driver supported which card, or ranges of cards, and has grown to be over 1,500 lines long. Whenever a driver author added support for a new device, they had to modify two different files to enable the device to work properly.

As the USB core code became mature, the group realized that it also needed something like the PCMCIA system to be able to load and unload drivers dynamically when devices were inserted and removed. The
group also noted that since USB and PCMCIA both needed this system, and that other kernel hot-plug subsystems also would use such a system, a generic hot-plug core would be useful. David Brownell posted an initial patch to the kernel (marc.theaimsgroup.com/?l=linux-usb-devel&m=96334011602320), enabling it to call out to a user-space program called /sbin/hotplug. This patch eventually was accepted, and other subsystems were modified to take advantage of it.

Let the Computer Do It Itself

All USB and PCI devices contain an identifier that describes either what kind of functions they support (like a USB audio or USB mass storage device), or if they do not support a class specification, they contain a unique vendor and product identifier. PCMCIA devices also contain these same kind of identifiers.

These identifiers are known by the PCI and USB kernel drivers, as they need to know which kind of devices they work properly for. The USB and PCI kernel drivers register with the kernel a list of the different types of devices that they support. This list is used to determine which driver will control which devices.

The kernel knows when and what kind of devices are inserted or removed from the system through the device bus core code (USB, FireWire, PCI, etc.). It can send this information to the user.

Taking these three pieces together (devices tell the computer what they are, drivers know what devices they support and the kernel knows what is going on) provides us with a solution to let the computer automatically load the proper driver whenever a new device is inserted.

/sbin/hotplug

The kernel hot-plug core provides a method for the kernel to notify user space that something has happened. The CONFIG_HOTPLUG configuration item needs to be selected for this code to be enabled. The notification happens when the kernel calls the executable listed in the global variable hotplug_path. When the kernel starts, hotplug_path is set to /sbin/hotplug, but the user can modify the value at /proc/sys/kernel/hotplug to change this. The kernel function call_usermodehelper() executes /sbin/hotplug.

As of kernel 2.4.14, the /sbin/hotplug method is being used by the PCI, USB, IEEE1394 and Network core subsystems. As time goes on, more subsystems will be converted to use it. Patches are available for the PnP-BIOS (notification when a laptop is inserted and removed from a docking station), Hot-Plug CPU, SCSI and IDE kernel subsystems. These are expected to be merged into the main kernel over time.

When /sbin/hotplug is called, different environment variables are set, depending on what action has just occurred.

PCI

PCI devices call /sbin/hotplug with the following arguments:

```bash
argv [0] = hotplug_path
argv [1] = "pci"
argv [2] = 0
```

and the system environment is set to the following:

```bash
HOME=/
PATH=/sbin:/bin:/usr/sbin:/usr/bin
PCI_CLASS=class_code
PCI_ID/vendor:device
PCI_SUBSYS_ID=subsystem_vendor:subsystem_device
```
PCI SLOT_NAME=slot_name
ACTION=action

The action setting is `add` or `remove` depending on whether the device is being inserted or removed from the system. The class_code, vendor, subsystem_vendor, subsystem_device and slot_name environment settings represent the numerical values for the PCI device's information.

**USB**

USB devices call /sbin/hotplug with the following arguments:

```
argv [0] = hotplug_path
argv [1] = "usb"
argv [2] = 0
```

and the system environment is set to the following:

```
HOME=/
PATH=/sbin:/bin:/usr/sbin:/usr/bin
ACTION=action
PRODUCT=idVendor/idProduct/bcdDevice
TYPE=device_class/device_subclass/device_protocol
```

The action setting is `add` or `remove` depending on whether the device is being inserted or removed from the system, and idVendor, idProduct, bcdDevice, device_class, device_subclass and device_protocol are filled in with the information from the USB device's descriptors.

If the USB device's deviceClass is 0 then the environment variable INTERFACE is set to:

```
INTERFACE=class/subclass/protocol
```

This is because USB has a much more complex model for device configuration than PCI does.

If the USB subsystem is compiled with the usbdevfs filesystem enabled, the following environment variables also are set:

```
DEVFS=/proc/bus/usb
DEVICE=/proc/bus/usb/bus_number/device_number
```

where bus_number and device_number are set to the bus number and device number that this specific USB device is assigned.

**Network**

The network core code calls /sbin/hotplug whenever a network device is registered or unregistered with the network subsystem, and /sbin/hotplug is called with the following arguments when called from the network core:

```
argv [0] = hotplug_path
argv [1] = "net"
argv [2] = 0
```

and the system environment is set to the following:

```
HOME=/
PATH=/sbin:/bin:/usr/sbin:/usr/bin
INTERFACE=interface
ACTION=action
```
The action setting is `register` or `unregister` depending on what happened in the network core, and interface is the name of the interface that just had the action applied to itself.

**CPU**

The Hot-Plug CPU patch (available at sourceforge.net/projects/lhcs) calls /sbin/hotplug after a CPU is removed or added to the system, and /sbin/hotplug is called with the following arguments:

```bash
argv [0] = hotplug_path
argv [1] = "cpu"
argv [2] = 0
```

and the system environment is set to the following:

```
HOME=/
PATH=/sbin:/bin:/usr/sbin:/usr/bin
CPU=cpu_number
ACTION=action
```

The action setting is `add` or `remove` depending on what happened to the CPU, and cpu_number is the number of the CPU that just had the action applied to itself.

**Examples**

The /sbin/hotplug script can be a very simple script if you only want it to control a small number of devices. For example, if you have a USB mouse and wish to load and unload the kernel driver whenever the mouse is inserted or removed, the following script, located at /sbin/hotplug, would be sufficient:

```bash
#!/bin/sh
if [ "$1" = "usb" ]; then
    if [ "$INTERFACE" = "3/1/2" ]; then
        if [ "$ACTION" = "add" ]; then
            modprobe usbmouse
        else
            rmmod usbmouse
        fi
    fi
fi
```

Or if you want to run ColdSync (www.ooblick.com/software/coldsync) automatically when you connect your USB HandSpring Visor to the computer, the following script located at /sbin/hotplug would work well:

```bash
#!/bin/sh
USER=gregkh
if [ "$1" = "usb" ]; then
    if [ "$PRODUCT" = "82d/100/0" ]; then
        if [ "$ACTION" = "add" ]; then
            modprobe visor
            su $USER -c "/usr/bin/coldsync"
        else
            rmmod visor
        fi
    fi
fi
```

If you want to make sure that your network devices always come up connected to the proper Ethernet card, the following /sbin/hotplug script, contributed by Sukadev Bhattiprolu, can do this:

```bash
#!/bin/sh
if [ "$1" = "network" ]; then
```
if [ "$ACTION" = "register" ]; then
    nameif -r $INTERFACE -c /etc/mactab
fi

Listing 1 shows a more complex example that can handle automatically loading and unloading modules for three different USB devices.

Listing 1. Script to Load and Unload Modules Automatically for Three Different USB Devices

Need for Automation

The previous small example shows the limitations of being forced to enter in all of the different device IDs manually, product IDs and such in order to keep a /sbin/hotplug script up to date with all of the different devices that the kernel knows about. Instead, it would be better for the kernel itself to specify the different types of devices that it supports in such a way that any user-space tools could read them. Thus was born a macro called MODULE_DEVICE_TABLE() that is used by all USB and PCI drivers. This macro describes which devices each specific driver can support. At compilation time, the build process extracts this information out of the driver and builds a table. The table is called modules.pcimap and modules.usbmap for all PCI and USB devices, respectively, and exists in the directory /lib/modules/kernel_version/.

For example, the following code snippet from drivers/net/eepro100.c:

```c
static struct pci_device_id eepro100_pci_tbl[] __devinitdata = {
    { PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82557, PCI_ANY_ID, PCI_ANY_ID, },
    { PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82562ET, PCI_ANY_ID, PCI_ANY_ID, },
    { PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82559ER, PCI_ANY_ID, PCI_ANY_ID, },
    { PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_ID1029, PCI_ANY_ID, PCI_ANY_ID, },
    { PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_ID1030, PCI_ANY_ID, PCI_ANY_ID, },
    { PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801BA_7, PCI_ANY_ID, PCI_ANY_ID, },
    { 0, }
};
MODULE_DEVICE_TABLE(pci, eepro100_pci_tbl);
```

causes these lines to be added to the modules.pcimap file:

```
<table>
<thead>
<tr>
<th>Vendor</th>
<th>Product</th>
<th>Domain</th>
<th>System</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00008086</td>
<td>0x00001229</td>
<td>0xffffffff</td>
<td>0xffffffff</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x00008086</td>
<td>0x00001029</td>
<td>0xffffffff</td>
<td>0xffffffff</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x00008086</td>
<td>0x00001030</td>
<td>0xffffffff</td>
<td>0xffffffff</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x00008086</td>
<td>0x00002449</td>
<td>0xffffffff</td>
<td>0xffffffff</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>
```

As the example shows, a PCI device can be specified by any of the same parameters that are passed to the /sbin/hotplug program.
A USB device can specify that it can accept only specific devices such as this example from drivers/usb/mdc800.c:

```c
static struct usb_device_id mdc800_table [] = {
    { USB_DEVICE(MDC800_VENDOR_ID, MDC800_PRODUCT_ID) },
    { } /* Terminating entry */
};
MODULE_DEVICE_TABLE(usb, mdc800_table);
```

which causes the following line to be added to the modules.usbmap file:

```
mdc800 0x0003 0x055f 0xa800 0x0000 0x00 0x00 0x00 0x00 0x00 0x00 0x00000000
```

Or it can specify that it accepts any device that matches a specific USB class code, as in this example from drivers/usb/printer.c:

```c
static struct usb_device_id usblp_ids [] = {
    { USB_INTERFACE_INFO(USB_CLASS_PRINTER, 1, 1) },
    { USB_INTERFACE_INFO(USB_CLASS_PRINTER, 1, 2) },
    { USB_INTERFACE_INFO(USB_CLASS_PRINTER, 1, 3) },
    { } /* Terminating entry */
};
MODULE_DEVICE_TABLE(usb, usblp_ids);
```

which causes the following lines to be added to the modules.usbmap file:

```
p1 0x0380 0x0000 0x0000 0x0000 0x0000 0x00 0x00 0x00 0x07 0x01 0x01 0x00000000
p2 0x0380 0x0000 0x0000 0x0000 0x0000 0x00 0x00 0x00 0x07 0x01 0x02 0x00000000
p3 0x0380 0x0000 0x0000 0x0000 0x0000 0x00 0x00 0x00 0x07 0x01 0x03 0x00000000
```

Again these USB examples show that the information in the modules.usbmap file matches the information provided to /sbin/hotplug by the kernel, enabling /sbin/hotplug to determine which driver to load without relying on a hand-generated table, as PCMCIA does.

**Preprocessor Abuse**

The macro MODULE_DEVICE_TABLE automatically creates two variables. For the example: `MODULE_DEVICE_TABLE (usb, usblp_ids);` the variables __module_usb_device_size and __module_usb_device_table are created and placed into the read-only data section and the initialized data section of the module, respectively. The variable __module_usb_device_size contains the value of the size of the struct usb_id structure, and __module_usb_device_table points to the usblp_ids structure. The usblp_ids variable is an array of usb_id structures with a terminating NULL structure at the end of the list.

When the depmod program is run, as part of the kernel installation process, it goes through every module looking for the symbol __module_usb_device_size to be present in the compiled module. If it finds it, it copies the data pointed to by the __module_usb_device_table symbol into a structure, extracts all of the information and writes it out to the modules.usbmap file, which is located in the module root directory. It does the same thing while looking for the __module_pci_device_size in creating the modules.pcinumap file.

With the kernel module information exported to the files modules.usbmap and modules.pcinumap, our version of /sbin/hotplug can look like Listing 2 [available at ftp.ssc.com/pub/lj/listings/issue96/5604.tgz]. This example only tests for a match of the USB product ID and vendor IDs. The Linux-Hotplug Project has
created a set of scripts that covers all of the different subsystems that can call /sbin/hotplug. This enables drivers to be loaded automatically when new devices are inserted into the systems. It also starts up network services when network devices are seen. These scripts are released under the GPL and are available at linux-hotplug.sourceforge.net. Almost all major Linux distributions are currently shipping this package, so it is probably already on your machine.

The Future

The current /sbin/hotplug subsystem needs to be incorporated into other kernel systems, as they develop hot-plug capability. SCSI, IDE and other systems all have hot-plug patches available for kernel support but need to have script support, kernel macro support and modutils depmod support added in order to provide the user with a consistent experience.

As the kernel boots, and discovers new devices, it tries to spawn /sbin/hotplug, but since user space has not been initialized yet, it cannot run. This means that any USB or PCI devices that are needed at boot time need to be compiled into the kernel or exist in an initrd RAM disk image as a module. Sometime during the 2.5 development process, the initrd RAM disk image will be converted to contain an entire small user-space tree. This will allow /sbin/hotplug to be run during the boot process and load modules dynamically. Some links describing this disk image idea are: lwn.net/2001/0712/kernel.php3, marc.theaimsgroup.com/?l=acpi4linux&m=99705696732868, marc.theaimsgroup.com/?l=linux-kernel&m=99436439232254 and marc.theaimsgroup.com/?l=linux-kernel&m=99436253707952.

Because of the small space requirements of this RAM disk image, the dietHotplug program has been written. It is an implementation of the Linux-Hotplug bash scripts in C and does not require modules.*map files when the program runs. The executable size of the entire dietHotplug program is one-fifth of the size of the original modules.*map files themselves. The small size is due to the use of dietLibc (found at www.fefe.de/dietlibc) and other space-saving techniques. dietHotplug will undergo more development as the 2.5 kernel requirements are more fully known. dietHotplug can be downloaded from the Linux-Hotplug site.

Acknowledgements

I would like to thank David Brownell who wrote the original /sbin/hotplug kernel patch and most of the Linux Hotplug scripts. Without his persistence, Linux would not have this user-friendly feature. I also would like to acknowledge the entire Linux USB development team, who have provided a solid kernel subsystem in a relatively short amount of time.

Keith Owens wrote the supporting code in the depmod utility and has endured constant changes to the format of the MODULE_DEVICE_TABLE() USB structure.

The other developers on the linux-hotplug-devel mailing list who have helped with their patches and feedback on the hot-plug scripts also deserve recognition, along with the wonderful Linux distribution-specific support that Debian, Red Hat and Mandrake have provided.

This article was based upon a paper and presentation that I gave at the 2001 Ottawa Linux Symposium.

Greg Kroah-Hartman is currently the Linux USB and PCI Hotplug kernel maintainer. He works for IBM, doing various Linux kernel-related things and can be reached at greg@kroah.com.