www.photocase.de

Bash-based hardware detection for PCI and USB

DISCOVERY SCRIPTS

If you need fast answers for what's inside, you can use a Bash script to obtain an inventory of hardware on your Linux system.

BY MIRKO DÖLLE

If you operate outside of the mainstream distributions or compile your own Linux system, whether an embedded Linux, a rescue system, or just a do-it-yourself distro, you need to solve the issue of finding the right kernel modules for your collection of hardware. Armed with some background knowledge, you can use almost any programming language to create a system for PCI or USB hardware detection. In this article, I'll show you a method for obtaining information about devices on your Linux system with a Bash script. Using *sysfs* or *proc*, you can find all the information you need about the manufacturer, device, and device class just by asking the kernel.

Listing 1: Evaluating PCI IDs

```
01 if [ -e /sys/bus/pci/devices ]; then
02   PCIDevices=/sys/bus/pci/devices/*
03   Method="sysfs"
04 elif [ -e /proc/bus/pci ]; then
05   PCIDevices=/proc/bus/pci/??/*
06   Method="proc"
07 fi
08
09 for device in $PCIDevices; do
10   if [ "$Method" = "sysfs" ]; then
11     read Vendor < ${device}/vendor
12     Vendor=${Vendor:2:4}
13     read Device < ${device}/device
14     Device=${Device:2:4}
15     read Class < ${device}/class
16     Class=${Class:2:4}
17   elif [ "$Method" = "proc" ]; then
18     Vendor=`hexdump -s 0 -n 2 -e '1/2 "%04x"' $device`
19     Device=`hexdump -s 2 -n 2 -e '1/2 "%04x"' $device`
20     Class=`hexdump -s 10 -n 2 -e '1/2 "%04x"' $device`
21   fi
```

Kernel Interfaces

Hardware detection is easiest if your kernel supports *sysfs*. In contrast to the *proc* interface, the *sysfs* interface does not require you to evoke external applications or process binary files to obtain to the vendor and devices IDs. This said, *proc* will give you the same information, which means hardware detection will work equally well no matter which kernel interface you decide to use.

sysfs uses a symbolic link with the PCI ID for each PCI device below */sys/bus/pci/devices*; the link points to the appropriate device directory below */sys/devices/pci**. Access via */proc/bus/pci/devices* is simpler, as this is where all PCI devices for all PCI buses reside; in contrast, */sys/devices* has a separate directory with the devices for each PCI bus. The device directory has the data required for hardware detection: *vendor* gives you the vendor ID in hexadecimal format, *device* the product ID, and *class* the device class based on [3]. Listing 1 is an excerpt from the Bash script *pcidetect*, which you will find at the Linux Magazine Website [1]. Lines 11 through 16 of Listing 1 parse the files required to identify your hardware from the *sysfs* structure.

The device pseudo-files sorted by the PCI controller are listed below */proc/bus/pci*. Each device file contains a full set of data in a compact format; *sysfs* presents this information individually. Lines 18 through 20 in Listing 1 contain the part of the PCI hardware detection script from [1] that identifies the hardware vendors, the product IDs, and the class codes.

The PCI ID Database

In order to output the vendor and device names in clear text, the hardware detection script references the PCI ID database at [2], following the same approach as the Linux kernel. The PCI database is easy to parse; Listing 2 shows you the relevant section from the hardware detection script.

The *PCIIDCMD* variable contains the command for parsing the PCI ID database, which is either a simple *cat* against the file or a call to *wget*. The database is easy to parse; each vendor or device entry occupies a single line with tab-separated fields. To process a complete line, line 1 of Listing 2 sets the separator variable, *IFS*, to *$'\n'*, thus separating the *for* loop at the line ends. Line 3 writes a tabulator to the *IFS* variable in order to separate the fields, and line 4 handles the separation.

The query in line 8, which ascertains whether column four contains a zero, is for quality assurance. Anyone can add to the PCI database, and new entries are identified by a 1 in column 4. This is changed to a zero for verified entries. The vendor ID for vendor entries, or the combined vendor and device IDs for device entries, are stored by lines 9 and 13 along with the description.

Listing 2: Parsing the PCI ID database

```
01 IFS="${Newline}"
02 for z in `eval ${PCIIDCMD}`;
   do
03   IFS="${Tab}"
04   set -- $z
05
06   case "$1" in
07     v)
08       if [ "$4" = "0" ]; then
09         declare v${2}=$3
10       fi
11       ;;
12     d)
13       declare d${2}=$3
14       ;;
15   esac
16 done
```

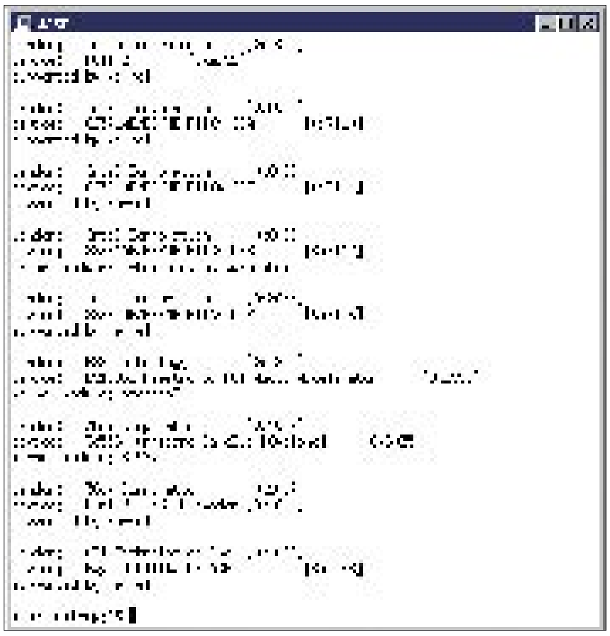
Figure 1: Automatic hardware detection is no rocket science. The kernel gives you all the device information you need, and a short script will handle the process. The detailed vendor and device information is available in the PCI ID database.

Variable Variable Names

For Bash, storing the PCI ID database is a major challenge. Both the vendor and the device IDs are 16-bit values, but Bash only supports 16-bit indexes for arrays, and unidimensional arrays. There is no support for hashes. Variable variable names are the solution to this problem. As lines 9 and 13 show, the names can be assign as follows:

```
declare prefix${name}=$value
```

Variables for vendor entries are assigned a *v* for "vendor" as a variable name prefix, followed by the vendor ID as a four-digit hexadecimal number; device entries start with a *d*, followed by the vendor and device IDs. This approach gives you an advantage over *grep*-based approaches in that you only need to parse the PCI ID database once for each device. On the downside, the memory requirement is enormous at about 8 to 9 Mbytes; additionally, the computer can take up to a minute to run the script. There are a number of ways of optimizing the resource demands.

Identifying Modules

Using an approach similar to the approach we used to parse the PCI ID database, the *modules.pcimap* file for the current kernel is now parsed. The file contains a list of all kernel modules and the devices they support in the form of the vendor and device IDs, and the class code for generic drivers such as sound or Firewire controllers. Listing 3 shows you the code segment that parses *modules.pcimap* and stores the results in variable variable names, just like with the PCI ID database.

The functionality provided by Listing 3 is the same as the functionality provided

Listing 3: Identifying Kernel Modules

```
01 IFS="${Newline}"
02 for z in `cat $PCIMAP`; do
03   IFS=" "
04   set -- $z
05
06   if [ "$2" = "0xffffffff" -a
07        "$3" = "0xffffffff" ];
   then
08     id="c${6:4:4}"
09   elif [ "$3" = "0xffffffff"
   ]; then
10     id="m${2:6}"
11   else
12     id="m${2:6}${3:6}"
13   fi
14
15   if [ -z "${!id}" ]; then
16     declare ${id}="$1"
17   else
18     declare
   ${id}="${!id}${Tab}${1}"
19   fi
20 done
```

Listing 4: Parsing System.map

```
IFS="${Newline}"
for z in `cat $SYSTEMMAP`; do
  IFS="${Tab} "
  set -- $z

  if [ "${3:0:12}" = "__devicestr_" ]; then
    IFS="_"
    set -- $3
    declare k${4}=1
  fi
done
```

by Listing 2 with the exception that the individual data files are separated by blanks rather than tabs. The evaluation of the class code in line 6 of Listing 3, if the vendor and device IDs have a value of -1, is another special function for the script. Line 8 provides a responsible approach to handling generic vendor drivers.

The query in line 14 contains a construction that is rarely seen, *${!id}*. This construction is the return function for the value of a variable with a variable name – if the curly braces are followed by an exclamation mark, Bash interprets any following characters up to the closing bracket as variable values that should be used as the variable names for the whole expression. If a value of *3c59x* is stored in the *id* variable, *${!id}* returns the content of the variable *3c59x*; in other words *${!id}* is written as *${3c59x}*.

Inside the Kernel

To discover which PCI devices lack driver support, the hardware detection script still needs to discover which drivers have been built into the kernel. To discover which drivers are built in, the script uses the *System.map* file: each driver adds a symbol to the kernel based on the following pattern:

```
__devicestr_vendor-/device-id
```

Listing 4 shows the code segment that parses *System.map* and stores the results in variables with variable names. The only major difference from Listings 2 and 3 is, the symbol entries in Listing 4 are separated by underscores and just a 1 is stored for each supported device.

Now that we have parsed and stored the vendor and device IDs, the kernel modules and their responsibilities, and the devices with direct kernel support, all we need to do is supply user output (Figure 1) in Listing 5 (which follows on from Listing 1).

In lines 1 through 6 of Listing 5, the variable names are concatenated to identify the vendor and device IDs, the required modules, generic modules, kernel support, and modules for each device class; the remainder of Listing 5 is nothing special, as it just uses the variable variable names to reference the variables with the kernel modules or device descriptions, and then outputs the results. Line 10 might look slightly more complex at first glance; this is where the script checks if there is an entry for the device in the regular kernel modules, the vendor specific modules, or whole device classes.

USB and Firewire

Hardware detection for USB devices does not require major changes to the PCI script. With the exception of the far more complex formating of the USB ID list, which makes the information more difficult to separate, the USB detection script from [1] simply parses different pseudo-files from */sys* and */proc*. Even Firewire hardware detection follows the same pattern, although this article does not offer a script specifically designed for Firewire detection.

A simple listing of supported devices and their kernel modules is the first step towards gaining a better understanding of a router or embedded Linux system. With a few minor changes, such as replacing the references to the PCI and USB ID databases and automatically running *modprobe* instead of outputting the kernel modules, the hardware detection script could easily be used as a start script to load all required kernel modules on booting a system. ■

Listing 5: Output

```
01   v="v${Vendor}"
02   d="d${Vendor}${Device}"
03   m="m${Vendor}${Device}"
04   g="m${Vendor}"
05   k="k${Vendor}${Device}"
06   c="c${Class}"
07
08   echo "Vendor:   ${!v}${Tab}[0x${Vendor}]"
09   echo "Device:        ${!d}${Tab}[0x${Device}]"
10   if [ -n "${!m}" -o -n "${!g}" -o -n "${!c}" ]; then
11     set -- ${!m} ${!g} ${!c}
12     if [ "$#" -gt "1" ]; then
13       echo "Kernel modules: $*"
14     else
15       echo "Kernel module: $1"
16     fi
17   elif [ -n "${!k}" ]; then
18     echo "Supported by kernel"
19   else
20     echo "Not supported"
21   fi
22   echo
23 done
```

INFO

[1] Hardware detection scripts for PCI and USB: *http://www.linux-magazine.com/Magazine/Downloads/60/detection*

[2] PCI ID database: *http://pciids.sf.net/pci.db*

[3] PCI header file with vendor, device and class definitions: *http://www.pcidatabase.com/pci_c_header.php*

THE AUTHOR

Mirko Dölle is the head of our Hardware Competence Center, and as such, he tests everything he can get his hands on – even if the lid is nailed down. On his leisure time, Mirko is the developer of the Ro-Resc miniature rescue distribution and the co-author of the LinVDR distribution. On the weekend he makes the old alchemists' dream come real, turning gold into lead...