Towards a small, efficient Linux hardware inventory system

Martin Schwenke

IBM OzLabs Linux Technology Center

<martins@au.ibm.com> · <martin@meltin.net>

Abstract

lsvpd is a serviceability package written for Linux running on IBM pSeries® systems, although it also runs on other architectures with at least a Linux® 2.6 kernel. Developed as a look-alike for the AIX® tool of the same name, lsvpd includes a hardware inventory database, scanning components that populate the database, and a variety of query tools. The focus is on listing Vital Product Data (VPD) for components as part of hardware service calls - in such cases knowledge of model numbers, engineering versions of circuit boards, and microcode versions can be critical in determining the the correct fix for a problem. Use of a database, rather than querying hardware directly is an important requirement since it may not be possible to get required information from a faulty component.

Certain problems with the current version of lsvpd can't be reliably fixed without proper hotplug support. The current version of lsvpd for Linux is a combination of bash scripts and "helper programs" written in C. Therefore, due to the large number of processes that are spawned, it is unlikely to cope well with a large number of hotplug events in a short amount of time. A decision has been made to "migrate" the current implementation to C - the plan is to make the helper programs incrementally larger so they eventually replace the entire bash implementation. One feature of the current bash implementation is run-time loadable modules that can provide additional functionality depending on available system features, such as sysfs. Implementing this as elegantly in C will be interesting.

This paper discusses the above in a number of contexts, including:

- The need for a small, efficient hardware inventory system for Linux. Other systems, such as OpenPegasus, also do hardware inventory, and could use a lower-level implementation if it were available and suitable.
- The use of a hardware inventory system as part of a persistent device naming system. This would involve generalising something like the `scsi_id` program, but would use general VPD values for many different types of devices. A prototype has been demonstrated using the current implementation of lsvpd.
- The level of success migrating the current implementation to C. Some part of the migration will be complete by the time the paper is due.
- The trend towards writing system utilities in scripting languages. As a serviceability tool, lsvpd may be required to help diagnose problems when only a partial system is available - there may be no /usr filesystem.

1 Introduction

lsvpd is a serviceability package for Linux that implements a hardware inventory system. It is a (partial) reimplementation of the AIX `lsvpd` command and some other related commands. The target platform for Linux lsvpd is IBM pSeries servers, but it also runs on other Linux systems running a 2.6 kernel.

There are a number of reasons why hardware inventory systems are useful, including:

Hardware detection: Traditionally, Linux has used hardware detection systems to facilitate automatic selection of device driver kernel modules. Examples include Red Hat's kudzu[1] and SUSE'S hwinfo.

System management: System management tools, such as WBEM [2] (Web-Based Enterprise Management), require a reasonable knowledge of the system's hardware to be able to perform management tasks. In particular, WBEM includes some explicit hardware inventory elements in its core schema.

Serviceability: To be able to adequately service a system it is important to know details of the hardware that comprises it. For serviceability, such details might include a part number, serial number or physical location. It is also preferable to have information stored in a database, instead of needing to query hardware at run-time, since it may not be possible to retrieve information from a faulty component.

This paper focuses on hardware inventory systems for the last category. While a system that supports serviceability requirements may be able to help support hardware detection and system management, it may have extra requirements not needed for those two categories. For example, the author spent some time looking at OpenPegasus[3] (a WBEM system), trying to figure out if it handles hardware inventory. There were no clear conclusions: there's over 33MB of source code with no obvious hardware inventory component(s), suggesting there might be a bit too much infrastructure to build something that will fit into an initramfs (see Section 1.1)! However, a future version of lsvpd will offer a library with a C API to allow access to its database so that higher-level tools can be built on top of lsvpd.

1.1 Small Is Good

One of the requirements of lsvpd is for it to be as small as possible. Ideally, the system should be able to run from an initramfs, or other limited boot-time environment. The current version has an installed size of less than 750KB.[4] However, the current system is written in a mixture of `bash` scripts and C helper programs so it doesn't scale well on large systems, particularly those that

[1] `http://rhlinux.redhat.com/kudzu/`
[2] `http://www.dmtf.org/standards/wbem`
[3] `http://openpegasus.org/`
[4] However, lsvpd may require significantly more filesystem space for its database.

may have a lot of hotplug activity. In fact, due to this, only limited hotplug capabilities have been written so far, and even these remain unused.

lsvpd is currently being rewritten in C, although possibly only for Linux 2.6. This will make it small and fast, at least compared to the current version. Comments on the progress of the rewrite can be found in Section 3.

An earlier, more primitive version was written in Perl, but sophisticated scripting languages such as Perl are rarely found in initramfs or similar environments. More about that in Section 4.

1.2 Digging For Details

Hardware is varied and complex. lsvpd uses a wide variety of techniques to find out detailed information about different types of hardware. Luckily, most adapters conform to one of the PCI standards (for example [4]), which describe some standard ways of retrieving VPD from PCI adapters. SCSI devices tend to have a lot of information available via SCSI inquiry commands, and the structure of this information tends to be fairly uniform within the product lines of particular vendors, with some information available under Linux 2.6 via sysfs. SCSI inquiries are implemented in Linux via the generic SCSI `ioctl(2)`interface. IDE devices don't have much information in sysfs — procfs still tends to be used — but the `HDIO_GET_IDENTITY` `ioctl(2)` manages to provide some useful information. On IBM pSeries systems, and others that use Open Firmware, the device-tree is also a useful source of information.

However, from there it gets murky. Different device drivers support *ad hoc* methods of exposing various tidbits of information. A flexible system for providing hooks (or even 'plug-ins'?) needs to be provided for retrieving this information in the most general possible way. Methods that are entirely driver-specific are best avoided, since they will almost certainly become difficult to maintain.

1.3 Example Output

```
*DS PCI-X Dual Channel Ultra320 SCSI RAID Adapter
*AX scsi3
*PN 97P3960
*FN 97P3960
*SN YL10C3306827
*MN 000C
*EC 0
*RM 0309002d
*Z0 5703
*Z1 1
*YL U7879.001.11C543F-P1-C5-T1
```

Figure 1: An example of VPD

An excerpt from an example of output from the `lsvpd` command is shown in Figure 1. Some of the important fields are summarised below:

DS: Description. Usually displayed as the first item.

AX: Operating system name. Actually 'AIX name'!

PN: Part Number. Knowing the history of a part may help to explain certain types of faults.

FN: FRU (Field Replaceable Unit) number. This is a generalisation of a model number, representing a family of models that are interchangeable. For example, an old model may no longer be available, but there may be a newer model with the same FRU number that can be used as a replacement.

SN: Serial Number. It is useful to confirm the serial number of a component before removing it, when possible.

RM: Alterable ROM Level. In this case, the firmware version of the card, which may need to be updated.

YL: Physical Location. All fields beginning with 'Y' are system specific fields and, in this case, this field is particular to IBM pSeries systems. This example can be read (from right-to-left) as port 1, on card connector 5, on planar (or backplane) 1, in unit `U7879.001.11C543F` (which may be in a separate drawer or rack to the main part of the system).

Figure 2 shows an example, of the same VPD as it is listed by the more user friendly `lscfg` command. This type of output is more likely to be used by field engineers, whereas output from `lsvpd` is more likely to be used by upstream serviceability applications.

```
scsi3            U7879.001.11C543F-P1-C5-T1
                                    PCI-X Dual Channel Ultra320 SCSI RAID
                                    Adapter

      Part Number.................97P3960
      FRU Number.................97P3960
      Serial Number..............YL10C3306827
      Manufacture ID.............000C
      EC Level...................0
      Alterable ROM Level.........0309002d
      Device Specific.(Z0)........5703
      Device Specific.(Z1)........1
      Device Specific.(YL)........U7879.001.11C543F-P1-C5-T1
```

Figure 2: An example of VPD, as listed by the `lscfg` command

1.4 Other Information

Various aspects of lsvpd for Linux have been previously discussed in [1, 2, 3].

2 VPD-based Persistent Device Naming

The device naming system udev[5] can be used with Linux 2.6 to provide a configurable, dynamic `/dev` directory. Normal udev rules can only make use of sysfs properties in the naming process. Although it would be possible to put all VPD in sysfs by adding VPD retrieval code to the kernel, this would be inflexible and would amount to kernel bloat. However, due to good design, udev has a call-out facility - an arbitrary program can be executed by a device naming rule and can, therefore, use arbitrary information to assist with name generation.

One such existing program is `scsi_id`. This program attempts to find either a world wide name or a serial number in a device's SCSI inquiry data and, if possible, prints a unique identifier for the device. The unique identifier can then be used in a udev rule to identify the device, allowing a preconfigured name to be assigned. Further, after all desired SCSI devices have been configured, it is possible to use a simple script that invokes `scsi_id` on each SCSI device to generate rules that assign a simple, sequentially assigned name to each device. This provides for a simple form of persistent SCSI device naming.

This approach can be extended to arbitrary devices using a hardware inventory system like lsvpd and has been previously described in [3]. Here's a suggested sequence of hotplug events that could occur when a device is added:

1. `lsvpd-hotplug` retrieves VPD for the device and stores it in lsvpd's database.
2. udev calls out to `lsvpd-namedev`, which uses its own rules to determine whether particular combinations of VPD elements for the device have a corresponding known device name. If so, that name is returned. If not, a new name is generated from a sequence and the new name is stored in `lsvpd-namedev`'s naming database.
3. `lsvpd-hotplug-name` adds the name to the VPD entry in lsvpd's database.

In such a system, the main VPD database is dynamic and does not need to be remembered between system boots for naming purposes. Although it is still useful to maintain old copies of the database for serviceability reasons, device name management is separated in an additional layer of logic.

When lsvpd is reimplemented to be smaller and more efficient than the current version (see Section 3), a sample device naming system will eventually be included.

3 Rewriting lsvpd

As mentioned in Section 1.1, the current version of lsvpd is written in `bash`. It uses helper programs to do things that `bash` can't do, such as performing SCSI inquiries and parsing binary data. Some of these helpers are part of the lsvpd packages and others are from third party packages (such as sg3_utils). The current system functions quite well. A main script `update-lsvpd-db` is used to collect VPD at boot time. This script may also be run at any time by the user.

[5] `http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html`

Although many steps have been taken to minimise the number of unnecessary sub-processes created by `update-lsvpd-db`, it is still too inefficient for use in a busy hotplug environment. Therefore, the package is being rewritten in C. The goal is to have a database initialisation program and hotplug programs that do all of their work efficiently, each within a single process.

The following sections discuss various issues that have arisen during the early stages of the rewrite. Note that although the discussion generally refers to lsvpd, most attention is being paid to reimplementing `update-lsvpd-db` rather than the query programs.

3.1 Dynamically Configurable Modules

The `bash` implementation of lsvpd uses several directories of 'modules' that are loaded at run-time, depending on a condition that is checked at the top of the module. This allows conditional code execution, including setting variables and defining functions, which supports a form of run-time configuration. Thus, the current system can determine available system features at run-time, rather than at packaging or installation time, perhaps allowing improved operation when a system is upgraded. For example, on IBM pSeries systems `lsvpd` performs a lot better when sysfs is present, so if a system is upgraded from Linux 2.4 to 2.6 `lsvpd` will automatically become more efficient.

This is implemented in `bash` in a very obvious way. All of the modules in a configuration directory are sourced, and each checks some initial condition, like the example in Figure 3.

```
[ -n "$sysfs_dir" ] || return 0

list_devices_functions="sysfs_list_devices"

sysfs_list_devices ()
{
    ...
}
```

Figure 3: An example of dynamic configuration in `bash`

This initially looked like it would be quite difficult to do in C. However,after learning about the possibility of using *ELF sections*, which can be manipulated via some GCC extensions and used for this sort of initialisation, things looked much more straightforward. Each module defines an initialisation function, which. by convention. is called `init()`, and a pointer to this function is added to a special section using a macro called `INIT`. An example can be seen in Figure 4. As will be seen in Section 3.2, `init()` functions can also support function overriding in a fairly obvious way.

The `INIT` macro, which looks a bit like 'line noise', is shown in Figure 5. The main job of the `INIT` macro is to place a function pointer into the `init_call`

```
static void
init(void)
{
        if (NULL != lsvpd_sysfs_dir) {
                device_listing_functions_clear();
                ...
        }
}

INIT(init);
```

Figure 4: An example of dynamic configuration in C

ELF section. This is an *ad hoc* section, so a function similar to `call_inits` must be called at the beginning of `main()` to call each function pointer in the section.

```
#define INIT(fn) static initcall_t __initcall_##fn \
__attribute__((__unused__)) \
__attribute__((__section__("init_call"))) = &fn

static inline void call_inits (void)
{
        extern initcall_t __start_init_call[], __stop_init_call[];
        initcall_t *p;
        for (p = __start_init_call; p < __stop_init_call; p++)
                (*p)();
}
```

Figure 5: `INIT` macro and calling code in C

One nice feature of this method is that the linker (`ld`) combines the sections in link order, so the order that object files are passed to the linker determines the order the initialisation functions are called in. That is, no explicit code is required to ensure the initialisation functions are called in the correct order — the logic appears in only one place: the `Makefile`.

Two other methods, both involving ELF sections were examined with a view to simplifying things.

- Use of the `.init_array` section, where function pointers in the section are automatically called before `main()`. This would mean no `CALL_INITS` function would be needed. However, handling of the `.init_array` section appears to be a new feature, requiring a relatively new version of libc. Therefore, this is not (yet) portable enough.
- Use of the `constructor` attribute to mark functions that should be called before `main()`. This seems to be portable, but a simple experiment showed the call order is not necessarily well defined.

3.2 Function Overriding and Clever Tricks

Overriding functions in `bash` is trivial: simply define a new function with the same name as an old one. In C this is more difficult, but the solution is still reasonably obvious: call functions via pointers, and set the pointers in the initialisation code. This works well.

To cut down the number of obvious conditional statements in the `bash` implementation, a limited form of object-orientation, termed *function multiplexing*, has been implemented. It works like this...

A SCSI device is added, so the following function call occurs:

```
device_add scsi 0:0:8:0
```

If `device_add_scsi` is a function, it is called with the single argument `0:0:8:0`. Otherwise, if `device_add_DEFAULT` is a function, it is called with both of the original arguments. Otherwise, no function is called. This is implemented using the code shown in Figure 6.

```
multiplex ()
{
    local func="$1" ; shift
    local type="$1" ; shift

    local f="${func}_${type}"
    local t=$(type -t "$f")
    if [ "$t" = "function" ] ; then
        "$f" "$@"
    else
        f="${func}_DEFAULT"
        local t=$(type -t "$f")
        if [ "$t" = "function" ] ; then
            "$f" "$type" "$@"
        else
            : "No function defined for \"${func}\" \"${type}\""
        fi
    fi
}

make_multiplexed ()
{
    eval "$1 () { multiplex $1 \"\$@\" ; }"
}

make_multiplexed device_add
```

Figure 6: Function multiplexing in bash

This is somewhat trickier in C. What's more, in `bash` it is possible for a module to undefine one or more specific functions, such as `device_add_scsi`, and provide a more general definition, such as `device_add_DEFAULT`, to replace it. An initial thought was to use an array of functions to implement each multiplexed function. However, this would have meant that some decisions, such as the size of the array and allocation of array indices, would need to be centralised. Also, to make things type-safe (unlike the `bash` implementation) macros would need to be used in unnatural and ugly ways.

To simplify the C implementation a decision was made to group functions to make the implementation more manageable. For example, functions relating to devices would be grouped, allowing the relevant pointers to be put into a structure. The structure also includes an identifier indicating the type of functions defined there — for example, SCSI. These type identifiers are currently implemented as strings, since this is quite intuitive, the strings are naturally unique and comparisons are quite cheap (since failure to match usually occurs at the first character). However, if this turns out to be a source of inefficiency, pointers to the identifier strings, rather than the strings themselves, could be compared. This would be a relatively small change.

The only real negative aspect of this implementation is that the top-level multiplexed function actually needs to be explicitly defined. This could probably be done with macros, but it wouldn't be pretty.

Examples of the C implementation are not given here. To be useful they would take up too much space!

3.3 Infrastructure

So, how well is the reimplementation proceeding? OK, but more slowly than first planned. The main problem is *infrastructure*, and there are two categories of infrastructure of interest: language infrastructure and project infrastructure.

C doesn't provide much infrastructure, especially when compared to sophisticated scripting languages. There's libc, but it provides minimal string handling and requires more attention be paid to memory management than to solving the problem you're working on. There are also 'higher level' libraries like glib, but they tend to be quite idiomatic — you tend to do some weird stuff to get the job done. To write good quality C code you need to bring some good infrastructure with you... or you need to write it.

Project infrastructure is important too. In the `bash` version of lsvpd, many new features can be added in minutes. This is because the project has built up a lot of domain-specific infrastructure. To get started on the C version, quite a lot of this infrastructure needs to be duplicated.

Even so, the original reimplementation plan was to incrementally increase the number and size of C helper programs, gradually reduce the amount of `bash` code and then magically combine the C code into one amorphous lump. This turned out to be quite a naïve plan. The main problem is that many of the code paths call functions that have alternate implementations based on orthogonal conditions. For example, the multiplexed `device_add` functions call multiplexed

`device_add_hook` functions. The implementation of `device_add` depends on the type of device being added, but the implementation of `device_add_hook` depends on whether `/proc/device-tree` and sysfs exist. Therefore, to implement IDE device handling, general code needs to be written relating to devices, sysfs and device-trees — unless one wishes to contemplate `bash` code executing C code executing `bash` code — and that's most of the device-handling code. So, the plan became to (simply!) replace all the device handling code. However, this also requires a little bit of adapter handling code, since devices are attached to adapters.

The reimplementation is proceeding, but it is happening slower than originally planned...

3.3.1 Linux 2.6 Only?

One way of speeding up the reimplementation process is to only implement the parts relevant to Linux 2.6. Although this is most of the system, it does save rewriting a non-trivial amount of code. Supporting Linux 2.4 systems would simply be a matter of executing (say) `update-lsvpd-db24` on systems where sysfs is not being used. This also has the advantage of using existing, working code rather than introducing new bugs. However, the obvious disadvantage is that, in the future, some bugs will need to be fixed in two places.

The obvious compromise is to *initially* reimplement the parts of the system to support Linux 2.6 and then reconsider reimplementing support for Linux 2.4.

4 Scripting System Tools

There is a strong trend towards writing system tools in scripting languages, such Perl and Python. This might not be a good idea in all cases where it is being done.

The original Linux implementation of lsvpd was written in Perl. This worked well, since Perl is feature-rich enough to do everything required without resorting to external programs. However, given the desire to be able to use lsvpd in a limited environment, such as an initramfs, a sophisticated scripting environment like Perl would not always be available. Also, anyone wishing to build higher level tools on lsvpd would inherit the requirement for Perl. In general, using a modern scripting language introduces non-trivial run-time dependencies. Compilers for these scripting languages exist, but aren't mature enough to be used for real systems. The choice of `bash` has served lsvpd reasonably well. However, even though it is the default Linux shell, `bash` is not likely to be available in an initramfs. However, the choice of `bash` has also introduced the need for helper programs written in C. Soon, ongoing lsvpd development will abandon the use of scripting languages altogether.

5 Conclusions

The Linux lsvpd serviceability package has been through a few major changes. It is currently a useful tool, but could be better. Therefore, it is being rewritten to be more efficient and to have less run-time dependencies, although the rewrite is proceeding more slowly than first envisioned. The new, improved lsvpd should be good enough to support persistent device naming on large systems. Scripting languages should not be used to implement systems like lsvpd, since this limits their use in certain contexts.

Thanks...

The IBM OzLabs team — an amazing group of people to work with — and a host of other IBMers.

Legal Statement

References

[1] Martin Schwenke.
My computer is bigger than yours!
Linux.conf.au 2003 <http://linux.org.au/conf/2003/>, January 2003.
Paper and slides also available from
<http://meltin.net/people/martin/publications/bigger.html>.

[2] Martin Schwenke.
Linux hardware inventory: Current reality, future possibilities.
AUUG 2003 - Open Standards, Open Source, Open Computing. <http://auug.org.au/events/2003/auug2003/>, September 2003.
Also available from
<http://meltin.net/people/martin/publications/linuxhwinv.html>.

[3] Martin Schwenke.
Using Vital Product Data For Persistent Device Naming.
AUUG 2004 - Who Are You? <http://auug.org.au/events/2004/auug2004/>.
September 2004. No paper written, but slides are available from:
<http://meltin.net/people/martin/publications/devnamevpd.html>.

[4] PCI Special Interest Group.
PCI Local Bus Specification. Release 2.2.
December 18, 1998.