Many hackers share the dream to contribute to the core, to the heart of the Open Source software: the Linux kernel. While the kernel's importance is hard to measure – after all, what would we do without a compiler, without a libc implementation? – it is definitely one of the most renowned components, if not the defining component of any Linux-based device. There are, by now, quite many paths to contribute to the Linux kernel,¹ how to learn how to contribute, and how to get mentoring, support and assistance in getting the first patches merged.² But where to start coding?

Some of the worst patches to submit, at the moment, are “whitespace cleanups” or “spelling fixes”. On the one hand, they most often cannot do any damage at all: such whitespace and the spelling of comments are ignored by the compiler anyway.³ On the other hand, though, their benefit-cost ratio is quite low – subsystem maintainers have other things to worry about than how to spell, and to correct the spelling in still comprehensible comments. So where to start instead? One excellent option is to look at issues kernel developers really care about: “ugly” code: code which is hard to read, code which is hard to maintain, and code which does not reflect the Linux way of doing things. One of the best ways for submitting patches is to look at such “ugly” code, change it to make it easier to read, make it easier to maintain and make it behave “the Linux way”. And, as I will try to show you in this presentation, it is not all that hard.

The PCMCIA subsystem – the past⁴

In the dark ages of the Linux kernel versions 2.0 and 2.2, getting a laptop to actually work under Linux was a rather stressful experience. The kernel lacked lots of concepts of how to deal with “hot pluggable” devices, with power management⁵ – and it lacked support for the extension cards called PCMCIA³ or

---

¹ Besides coding, probably the three most valuable ways to contribute to the Linux kernel are testing, reporting and managing bugs, e.g. in the Linux kernel bugzilla database, and by writing, updating or reviewing Documentation.

² Three excellent starting points are the kernelnewbies project at http://www.kernelnwebies.org/, the kernel-mentors mailing list at http://selenic.com/mailman/listinfo/kernel-mentors and the kernel janitors project at http://janitor.kernelnwebies.org/. These and all following links were last checked on 24 March 2010.

³ Of course, fixing the spelling of user-visible messages (such as error reporting), is most valuable.

⁴ In the following, whenever I refer to kernel source files before 2.6.12-rc1, any git IDs specified refer to Thomas Gleixner’s Linux history git tree available at http://git.kernel.org/?p=linux/kernel/git/tglx/history.git;a=summary.

⁵ Besides coding, probably the three most valuable ways to contribute to the Linux kernel are testing, reporting and managing bugs, e.g. in the Linux kernel bugzilla database, and by writing, updating or reviewing Documentation.
CardBus cards. Dave Hinds entered the field, and he wrote a self-contained, separate package of modules, userspace tools and userspace drivers called “pcmcia-cs”. This package was, in many regards, far ahead of its time - it allowed for the suspension and resume of devices; it allowed for hot-plugging, dynamic addition of device IDs - a concept PCI learned of in kernel 2.5 –; and it made PCMCIA devices work under Linux.

Having such modules which talk closely with the hardware, set up and use IRQs outside the Linux kernel seemed to be a bad idea, and therefore large parts of this package were integrated unchanged into the 2.4 kernel versions. Nowadays, it would happen by it first being merged into the “staging” directory, which I will talk about later. But back then, it was simply merged, and seemingly forgotten. Many curses could be heard in the source code of subsystems interacting with PCMCIA, even more curses could be read on the mailing lists about the state of the PCMCIA subsystem, and it not working the Linux way – despite formally being part of the Linux kernel. Let me show you five examples, as things were when kernel 2.5.0 was released in 2002:

**Interaction between User- and Kernelspace**

Back then, a PCMCIA card only worked if a userspace helper, a daemon called “pccardd”, was running. The kernel-level subsystem and the user-space helper communicated over an ioctl, and shared responsibilities for a whole lot of actions, for example binding PCMCIA drivers to PCMCIA devices. What happened if you insert a PCMCIA card? Let me simplify it a bit:

1. The kernel woke up the userspace thread “pccardd”, which then read about the insertion event.

2. “pccardd” used the ioctl to determine some information about the PCMCIA card – what are the card ID and product ID?

3. “pccardd” then tried to determine which driver is capable of driving this card, by looking this up in an database typically located in /etc/pcmcia/config.opts.

Much has happened since my talk at LinuxTag 2005, current trends in Linux kernel power management. A good overview is provided by Patrick Bellasi, Linux Power Management Architecture, ftp://ftp.elet.polimi.it/users/Patrick.Bellasi/Teaching/Embedded%20Systems/02_LinuxPowerManagement.pdf.

---

1 PCMCIA stands for **Personal Computer Memory Card International Association**. PCMCIA cards are 16-bit devices, comparable to small-size ISA-devices

1 CardBus cards are 32-bit devices, comparable to small-size PCI devies. In fact, the Linux kernel considers them to be PCI devices, so they do appear in „lspci“ instead of „lspcmcia“.


3 Specifically, 2.5.70, starting with patch a1c036675493e56f11ef17677dcda1b3bbdf38b3 by Matt Domsch.
4. If “pccardd” found something, it asked the kernel – again through the ioctl – to bind this driver to the PCMCIA device.

What are some of the problems resulting from such an approach? First, you need userspace to actually run to get a PCMCIA card to work. There are some embedded systems out there hardly running userspace at all; furthermore, it's quite tricky – and it used to be way more tricky back in 2.4 – to enable userspace so early that a PCMCIA-based root filesystem could be used. Second, you need to keep userspace and kernelspace compatible – something the graphics subsystem has many problems with, even up to today. Third, you need to coordinate the user- and kernelspace subsytems so that both know which driver is bound to which device; this involves additional overhead.

**Lesson 1**: Keep your userspace and kernel interfaces simple. A stable, backwards- or forwards-compatible interface (API) is hard to get right – so if it is avoidable, try not to create one at all, but either use existing ones, or keep it all in userspace, or kernelspace.

**Coding Style and Typedefs**

Coding style – where to put brackets, where to put tabs and how to indent – is surely some matter of taste. Nonetheless, having the entire kernel in a standard coding style makes reading and reviewing code written by others much easier. Insofar as the PCMCIA subsystem used four whitespaces instead of one tab for indent, that's a small issue - what counts more are the excessive use of “typedefs”. “Typedefs” hide information what an object really is, and such hiding is generally frowned upon in kernelspace. Therefore, do not re-define well-defined types, such as:

```c
typedef u_char    cisdata_t;3
typedef u_short   ioaddr_t;4
```

and, more importantly, do not re-define structs by typedefs:

```c
typedef struct dev_link_t {
    dev_node_t *dev;
    u_int state, open;
    ...
}
```

---

1 See Jake Edge, Nouveau and interface compatibility, LWN, 10 March 2010, http://lwn.net/Articles/377953/.

2 See Documentation/CodingStyle in the most current available kernel 2.6.

3 Defined in include/pcmcia/cs_types.h.

4 Defined in include/pcmcia/cs_types.h. This typedef turned out to be quite problematic: I/O addresses can easily be higher than what fits in an unsigned short. Changing „ioaddr_t“ was impossible, though, as it was part of the ioctl API.

5 Defined in include/pcmcia/ds.h.
Also, use the standard error codes, not something like this:

```c
#define CS_SUCCESS 0x00
```

**Lesson 2**: When coding for the Linux kernel, adhere to the Linux kernel CodingStyle and do not use typedefs unless there is a compelling reason to do so. Use `./scripts/checkpatch.pl` to check for most of the obvious CodingStyle issues.

### Magic Numbers, Version Checking

Back then, the kernel module “PC Card Driver Services”\(^2\) contained the following check in its init function:

```c
pcmcia_get_card_services_info(&serv);
if (serv.Revision != CS_RELEASE_CODE) {
    printk(KERN_NOTICE "ds: Card Services release does not match!\n");
    return -1;
}
```

The kernel module “PC Card Services – core services”\(^3\) contained the following function:

```c
int pcmcia_get_card_services_info(servinfo_t *info)
{
    ...
    info->Revision = CS_RELEASE_CODE;
    ...
    return CS_SUCCESS;
} /* get_card_services_info */
```

However, the Linux kernel already contains some checks to verify only binary compatible modules are loaded. Furthermore, it is generally frowned upon to use modules built for different kernels. In contrast, version and compatibility checks are essential on the userspace-kernelspace interface.

Another issue are “magic numbers”. Some consider it a good idea to “protect kernel data structures with magic numbers. This allows you to check at run time whether (a) a structure has been clobbered, or (b) you've passed the wrong structure to a routine.”\(^4\) However, in my experience, magic numbers are overrated: One realizes quite soon that such things happen, if you a) have

---

1 Defined in include/pcmcia/cs.h.

2 The core of this module is provided by drivers/pcmcia/ds.c. In 2.6, it is renamed to “pcmcia”.

3 The core of this module is provided by drivers/pcmcia/cs.c. In 2.6, it is renamed to “pcmcia-core”.

4 See Documentation/magic-numbers.txt as of kernel 2.6.33.
control over the users of data structures – that is, other kernel modules – and
b) if you have sufficient instrumentation – such as dev_dbg() or printk calls –
and sane checks, e.g. for NULL pointer dereferences, in place. Consider one
data structure which was only accessed and referenced in the “PC Card Driver
Services” module:

typedef struct user_info_t {
    u_int user_magic;
    int event_head, event_tail;
    event_t event[MAX_EVENTS];
    struct user_info_t *next;
} user_info_t;

That is just a singly-linked list, with all accesses and references in less than
1.000 lines of code. If one is not able to properly access and reference one
data structure in such a limited area, he should not touch kernel data
structures anyway.

Lesson 3: Trust other kernel code, distrust userspace. For intra-kernel
communication and data structures, there usually is no need for “version
checking” and “magic numbers”; for communication with userspace these
techniques are quite essential.

Not Invented Here (“NIH”)

I'm not sure which subsystem was developed first, therefore it may not be a
ture “not invented here” case how the binding of a driver to a device happened
for PCMCIA devices compared to PCI devices. Nonetheless, even if good open-
sourced ideas were developed by others, you should keep an eye on them and
think about re-using them for your use case.

Here is how PCMCIA “probed” device-driver combinations in 2.5.0:

1. First, a “central event handler” of the “PC Card Services – core services”
module is informed about a card insertion event.

2. The “central event handler” uses a function called send_event() to inform
all “clients” registered with this socket.

3. By this method, the “PC Card Driver Services” event handler is informed
about the card insertion event.

4. The userspace-kernel-interaction (see above at I. 1.) takes place; “PC
Card Driver Services” now knows which driver to bind.

5. “PC Card Driver Services” calls the drivers “attach” callback.

6. The “attach” callback registers itself with the “PC Card Driver Services”
event handler and returns.

1 Defined in drivers/pcmcia/ds.c.
7. The “PC Card Driver Services” event handler calls the driver event handler.

8. The driver event handler enables and activates the hardware.

PCI used and uses a clearly less error-prone and simpler approach:

1. The PCI core is informed about a new PCI device.

2. It iterates over a doubly-linked list of all loaded drivers, and checks whether the driver's ID table matches the IDs of the PCI device.

3. It calls the “probe” callback of the PCI device, which enables and activates the hardware.

**Lesson 4**: Look at other Linux kernel subsystems or drivers which do similar things. Learn from there, use abstractions and methods they use, and do not try to reinvent the wheel.

**Reference Counting, Locking**

The original PCMCIA code hardly cared about locking at all: it focused on uniprocessor systems. Things did not break apart on SMP systems – or with kernel preemption enabled – because most, but not all, calls to the hardware and accesses to data structures were done by just one thread. Later on, kernel preemption and SMP systems gained more users, and some unrelated changes to the PCMCIA subsystem meant that two or more threads might wish to access the hardware at the same time. All this lead to strange NULL pointer dereferences, non-working hardware and even some lockups, until the locking was fixed.

**Lesson 5**: Think about the lifetime of data objects from the beginning. Even if code is first meant to run only on uniprocessor systems, implement locking to avoid concurrent access to the hardware and to data structures – all hardware moves to multithread capabilities eventually.

**The PCMCIA subsystem – the present**

During the last extensive development cycle – the 2.5. kernel versions – first, valuable steps had been taken to materially integrate the PCMCIA subsystem, and to make it work “the Linux way”. These efforts still have not been completed, though: partly due to lack of developer time, partly due to lack of interest, partly due to systemic issues to the Linux kernel. But let me first show you some examples of how some issues pointed out above could be “fixed”:
Interaction between User- and Kernelspace

The entire driver to device-matching is now done similarly to the PCI subsystem, by looking at an array of device IDs. Therefore, there is no need for such an extensive ioctl previously required between the kernel- and userspace parts of the PCMCIA subsystem. To inform userspace about the state of things, sysfs is used. Sysfs is based around the concept of a device model, in which the hierarchy of devices is represented by directories, and all attributes which describe a device shall only contain one value.

Coding Style and Typedefs

Gradually, the core of the PCMCIA subsystem has been updated to adhere to the Linux kernel coding style. This is made easier by tools such as “lindent”; for most cases, a properly-configured editor is sufficient to update the coding style source code. The PCMCIA device drivers have not all been corrected yet, though.

Deprecating typedefs or custom error codes is merely a bit harder. For example, in one patch I replaced the definition

```c
#define CS_OUT_OF_RESOURCE   0x20
```

with the generic

```c
#define CS_OUT_OF_RESOURCE   -ENOMEM
```

Later on, CS_OUT_OF_RESOURCE could be replaced tree-wide with -ENOMEM, which did not change the compiled source at all.¹

Magic Numbers, Version Checking

In-kernel magic number checks and version checking were removed. Instead, they're only kept as far as userspace is concerned. Quoting a patch by Christoph Hellwig:²

```plaintext
[PCMCIA] kill remaining cardservices version checking

We know we have the right version because we were compiled in the same kernel tree..
```

Not Invented Here ("NIH")

In contrast to the PCMCIA subsystem lagging behind the improvements seen in

¹ See my patch 1168386aa7d850ead2ae135d5a7949a592c6e9a0 .

² See his patch c8635fbc6390fd16a22837a3bda7ab78c2948185 .
other subsystems in the past; we had the inverse problem during 2.5 and 2.6 quite a couple of times: when updating the PCMCIA subsystem to act like a “normal” Linux kernel subsystem, some newly added, generic methods were utilized, such as the device model's class device feature.\footnote{See patch d9dfb4aca2945af791cc1e1ce5f1fb89492ca08b by Greg Kroah-Hartman and, for another front-running issue, patch 4356d73d028ad0726cfa31ad30c5d28fcd98795 by David Brownell.} Sometimes, those features still contained bugs, which meant that PCMCIA was actually the frontrunner to fix some issues, which would have hindered the use of the same generic feature for other subsystems. For each generic feature, there needs to be one subsystem to be on the “bleeding side” of the “bleeding edge” – and for some features, it was the task of the PCMCIA subsystem to be there.

**Reference Counting, Locking**

Last but not least, reference counting and locking was fixed – the latter in an extensive patch series also containing object-based documentation: which data structure – not: which code – is protected by which lock. Even though the locking validation done by “lockdep” is an invaluable help in asserting locking is not leading to deadlocks, it cannot prove lock correctness. Therefore, it is of utmost importance to design the locking as simple but as effective as necessary. Somewhen, I considered it a good idea to use a “read-write-semaphore” to lock one linked list – a linked list which is hardly ever accessed. That was a bad over-optimization, which did not really focus on the true issue at stake: a comprehensive, coherent and understandable locking scheme.

**Side effect: smaller code!**

Even though lots of Documentation was added, even though the PCMCIA subsystem gained quite some features, most patchsets regarding the cleanup of code actually remove many lines from the Linux kernel. This relates less to the PCMCIA core but more to the PCMCIA device drivers, as many generalizations and abstractions were added to the core and removed from the drivers. Still, from 2.5.0 to 2.6.35 the PCMCIA core code only increased from around 5.000 to 7.000 lines. In the same timeframe, however, the size of a typical PCMCIA network device driver decreased by 100 lines or almost 10 %.

**Development Process - Lessons Learned**

Besides the code-related issues mentioned above, five lessons relating to the development process could be learned.

**Lesson 6**: Do not try to merge “experimental code”, or code which still needs cleanups. The earlier you write good code, the less effort it is in the whole term.

Of course, “proof of concept” code – which explains things often more clearly
than words can do – is an invaluable tool during the development process. However, a developer should consider proof of concept code as such, and throw most of it away when writing the code “for real”. In order to avoid double work, it might even be better to write proper code – and not just “proof of concept” code from the beginning.

A most useful tool started in 2008 is the “drivers/staging” directory in the Linux kernel.¹ In there, drivers and subsystems are merged first if their code is still experimental, hard to read, hard to maintain, or which does not reflect the Linux way of doing things. Once they have been cleaned up properly, they may be moved to other parts of the Linux kernel source code. There are a couple of advantages of using this staging directory: exposure to both testers, reviewers and other hackers, who might want to join efforts to clean up code. However, using the staging directory is not an excuse for bad code or bad practices: even if it is in stable, the interaction with userspace should at least be forward- or backward-compatible, it still needs to be licensed properly, and there must be an ongoing effort to work on the code – else it will be removed from the Linux kernel again, such as happened with the Android drivers in 2.6.33.

Lesson 7: Use drivers/staging, follow the Linux kernel development model. Instead of fighting it, make use of it. The staging directory, the -mm tree and, to a lesser extent, subsystem trees do not only offer a basis for code review, feature testing and integration testing – does the code actually compile on all architectures?

Lesson 8: All code contains bugs. It's just a question how fast most of the worst bugs can be found. Therefore, having access to at least some hardware helps, but what counts more is a broad and active tester base.

Lesson 9: If things break, it's always your fault!

In quite many cases, whenever a PCMCIA card no longer works with a new kernel, the PCMCIA subsystem is held responsible. That is a sensible approach from an user's perspective. However, quite often the cause for the bug actually lies in different subsystems. And often, there simply are insufficient resources available to follow all different subsystems for possible ill side-effects. Then, once the bug is there, users and fellow developers require you to take a proactive approach to help tracking the bug down, or even to fix it – and not merely to point fingers.

Lesson 10: There is a whole lot to learn when cleaning up a driver or even a whole subsystem: about the technical aspects of Linux kernel developments; about development process issues – and how to improve them or utilize them even for other projects – and also about socio-cultural issues – how to interact with hardware developers, with fellow kernel developers, and, last but not least, with users. It is a truly rewarding experience, and so I can only encourage everyone interested in contributing to the Linux kernel: pick a broken subsystem, pick a broken driver, and clean it up!