



Kernel Coding the Upstream Way

Tim Bird

Principal Software Engineer

Sony Electronics

Abstract

The Linux kernel has a number of unwritten idioms and design patterns that are used for optimal development and code efficiency. This talk seeks to illustrate these coding patterns, and help developers implement Linux kernel code in the "upstream way".

I will describe some of the kernel idioms for error handling, use of gotos, structures with embedded data, use of function pointers, when to use ifdefs, inlines and macros, structure initialization, and more. Each pattern will be explained, with examples, and the rationale for each pattern will be provided.

The purpose is to allow a developer to write code that is not only efficient, making best use of kernel interfaces, but also to make the code easier to maintain and easier to be contribute upstream, with less correction required for acceptance into mainline.

Agenda

- Kernel development priorities
- Coding Idioms
- Conclusions
- Resources

Kernel Development Priorities

Kernel development priorities

- Correctness
 - Do things work properly? Are they secure?
- Performance
 - Are things as fast and small as possible?
 - Is resource availability for user-space maximized?
- Maintainability
 - Can things be maintained? Does it require super-human effort?

Upstream priorities

- Correctness
- Performance
- Maintainability

- These are interrelated:
 - Maintainable code has less bugs over time, and therefore leads to correctness
 - Is kind of the rock-paper-scissors of priorities:
 - correctness is more important than performance
 - performance is more important than maintainability
 - maintainability leads to correctness, and so maintainability is important

Coding Idioms


Coding Idiom Categories

- Error handling
 - NULL pointers, gotos, return values
- Structures and pointers
 - Function pointers, container_of, embedded anchor
- Code efficiency
 - Inlines, macros, do while (0), likely
- Maintainability
 - Ifdefs, printks, case statements

NULL Pointers

- Check your pointers for NULL on return from allocation
 - Check in sub-routines if needed, but...
 - Don't check for NULL if you can validate that NULL is not possible (ie it is checked for elsewhere)
 - However, be mindful that code changes over time, and if refactoring could easily introduce a code path where the NULL is not checked, add a check.
- Don't use `BUG_ON(ptr==NULL)`
 - Hardware should catch bug on NULL pointer dereference
 - Error looks like this: "BUG: unable to handle kernel NULL pointer dereference at xxx"
- Freeing NULL pointers is OK
 - Don't do: 'if (ptr) kfree(ptr)' – just kfree() the ptr
- Semantic checkers can now find many NULL dereferences
 - See `scripts/coccinelle/null/deref_null.cocci` for one example

NULL pointers notes

- NULL is not 0, and 0 is not NULL
 - They have the same value but not the same type (even if the compiler thinks so)
 - Don't do this: `char * p = 0;`
- It is preferred to check for NULL with:
 - `if (!ptr) ...`
 - rather than: `if (ptr == NULL)`
 - stats: 

count of NULL checks after `kmalloc()`:
766 using `'if (ptr == NULL)'`
5128 using `'if (!ptr)'`
1186 without a NULL check within 4 lines

- Resources
 - NULL v. zero
 - <https://lwn.net/Articles/93574/>
 - Fun with NULL pointers, part 2
 - <https://lwn.net/Articles/342420/>

Gotos

- Normally, goto statements are considered bad
- But, the kernel uses gotos for error unwinding
- Structure of code should be:
 - Allocate resources (with gotos on errors)
 - Perform operations
 - Return success
 - Deallocate resources in reverse order, with labels
 - Return failure

Generic Goto code structure example

```
some_function() {  
    do A;  
    if (error)  
        goto out_undo_a;  
    do B;  
    if (error)  
        goto out_undo_b;  
    return 0;  
out_undo_b:  
    undo B;  
out_undo_a:  
    undo A;  
    return error;  
}
```

Actual example: xfs_open_devices()

```
STATIC int xfs_open_devices(struct xfs_mount *mp)
{ ...
if (mp->m_logname) {
    error = xfs_blkdev_get(mp, mp->m_logname, &logdev);
    if (error)
        goto out;
    dax_logdev = fs_dax_get_by_bdev(logdev);
}
if (mp->m_rtname) {
    error = xfs_blkdev_get(mp, mp->m_rtname, &rtdev);
    if (error)
        goto out_close_logdev;
...
error = -ENOMEM;
mp->m_ddev_targp = xfs_alloc_buftarg(mp, ddev, dax_ddev);
if (!mp->m_ddev_targp)
    goto out_close_rtdev;
...
return 0;
out_close_rtdev:
    xfs_blkdev_put(rtdev);
    fs_put_dax(dax_rtdev);
out_close_logdev:
    if (logdev && logdev != ddev) {
        xfs_blkdev_put(logdev);
        fs_put_dax(dax_logdev);
    }
out:
    fs_put_dax(dax_ddev);
    return error;
}
```

success return

error return

'Success' part of routine

```
STATIC int xfs_open_devices(struct xfs_mount *mp)
{ ...
if (mp->m_logname) {
    error = xfs_blkdev_get(mp, mp->m_logname, &logdev);
    if (error)
        goto out;
    dax_logdev = fs_dax_get_by_bdev(logdev);
}
if (mp->m_rtname) {
    error = xfs_blkdev_get(mp, mp->m_rtname, &rtdev);
    if (error)
        goto out_close_logdev;
...
error = -ENOMEM;
mp->m_ddev_targp = xfs_alloc_buftarg(mp, ddev, dax_ddev);
if (!mp->m_ddev_targp)
    goto out_close_rtdev;
....
if (logdev && logdev != ddev) {
    mp->m_logdev_targp = xfs_alloc_buftarg(mp, logdev, dax_logdev);
    if (!mp->m_logdev_targp)
        goto out_free_rtdev_targ;
} else {
    mp->m_logdev_targp = mp->m_ddev_targp;
}
return 0;
```

Example Details

Error handling part of routine

```
out_free_rtdev_targ:
    if (mp->m_rtdev_targp)
        xfs_free_buftarg(mp->m_rtdev_targp);
...
out_close_rtdev:
    xfs_blkdev_put(rtdev);
    fs_put_dax(dax_rtdev);
out_close_logdev:
    if (logdev && logdev != ddev) {
        xfs_blkdev_put(logdev);
        fs_put_dax(dax_logdev);
    }
out:
    fs_put_dax(dax_ddev);
    return error;
}
```

'Success' part of routine

```

STATIC int xfs_open_devices(struct xfs_mount *mp)
{ ...
if (mp->m_logname) {
    error = xfs_blkdev_get(mp, mp->m_logname, &logdev);
    if (error)
        goto out;
    dax_logdev = fs_dax_get_by_bdev(logdev);
}
if (mp->m_rtname) {
    error = xfs_blkdev_get(mp, mp->m_rtname, &rtdev);
    if (error)
        goto out_close_logdev;
...
    error = -ENOMEM;
    mp->m_ddev_targp = xfs_alloc_buftarg(mp, ddev, dax_ddev);
    if (!mp->m_ddev_targp)
        goto out_close_rtdev;
....
    if (logdev && logdev != ddev) {
        mp->m_logdev_targp = xfs_alloc_buftarg(mp, logdev, dax_logdev);
        if (!mp->m_logdev_targp)
            goto out_free_rtdev_targ;
    } else {
        mp->m_logdev_targp = mp->m_ddev_targp;
    }
}
return 0;

```

Example Details

Note how error is set up ahead of time

Error handling part of routine

```

out_free_rtdev_targ:
    if (mp->m_rtdev_targp)
        xfs_free_buftarg(mp->m_rtdev_targp);
...
out_close_rtdev:
    xfs_blkdev_put(rtdev);
    fs_put_dax(dax_rtdev);
out_close_logdev:
    if (logdev && logdev != ddev) {
        xfs_blkdev_put(logdev);
        fs_put_dax(dax_logdev);
    }
out:
    fs_put_dax(dax_ddev);
    return error;
}


```

Goto error-handling notes

- Set up 'error' (or 'err' or 'ret') variable ahead of time, if needed
- Success return is in middle of routine
- Error handling code is out of the main flow
 - Error paths (unlikely code) are out of the function cache footprint
 - Success case has less jumps
- Use "out_" prefix on your goto labels
 - People also use 'err_' prefix, but "out:" or out_xxx:" is more common
 - Preference is "out_", but follow the style for your subsystem
 - Try to have rest of label be descriptive of undo operation
 - e.g. _free, _release, _unlock, etc.

Resources for goto

Popular goto labels:

- Are more than 16K “out:” or “out_xxx:” labels in kernel
- Most common err and out labels: 

```
50 out_close:
62 out_fail:
73 out_release:
79 out_unmap:
81 error_out:
82 out_error:
155 out_put:
551 out_err:
687 err_out:
707 out_free:
1030 out_unlock:
2891 err:
14874 out:
```


• Resources

- <https://koblents.com/Ches/Links/Month-Mar-2013/20-Using-Goto-in-Linux-Kernel-Code/>
 - Great discussion with Linus about using gotos

Return values in the kernel

- Success = return 0;
- Failure = return -<errno>;
- Exception: functions that are a predicate
 - Should return 0 for false and 1 for true
 - eg. `pci_dev_present()`
- For functions that return a pointer:
 - Return one of: **valid pointer**, **NULL**, or **ERR_PTR(-<errno>)**
 - Caller can check with `IS_ERR(ptr)`, and convert back to `errno` with `PTR_ERR(ptr)`

Return values

- Rationale:
 - Return codes are limited to a single value (integer or pointer)
 - Have to overload things a bit
 - Strong conventions keep everyone sane
- Error numbers defined in `include/uapi/asm-generic/errno.h`
 - Please find the right one for your subsystem/device/driver/condition, etc.
 - Look at related drivers
 - Don't make up your own error codes
- `errno`s are part of the user API, and must be chosen with care
 - Don't break user apps with a weird `errno`
- Some popular `errno`s: 

<i># of uses per errno</i>	
4575	EBUSY
4918	EOPNOTSUPP
6560	EFAULT
7459	EIO
11785	ENODEV
27050	ENOMEM
58204	EINVAL

Return values for pointers

- `ERR_PTR()`, `IS_ERR()`, `PTR_ERR()` defined in `include/linux/err.h`

- How to use:

in function returning a pointer

```
If (some error condition)
    return ERR_PTR(-<errno>);
```

in caller (receiving a pointer)

```
If (IS_ERR(ptr))
    return PTR_ERR(ptr);
```

or

```
If (IS_ERR(ptr)) {
    pr_err("explanation of failure: %ld", PTR_ERR(ptr));
    return PTR_ERR(ptr);
}
```

- Should use `IS_ERR()` before using `PTR_ERR()`
 - Exception: can use `PTR_ERR()` without `IS_ERR()` if only one `errno` is possible
 - example: `if (PTR_ERR(ptr) == -ENOENT) { ...`
- Just remember that `NULL` is not the only error return value

Function pointers

- Kernel has its own brand of Object Oriented Programming
 - Extensive use of function pointers in structures
- Instead of using C++ virtual tables, kernel uses structs with function pointers
 - Some structs have a mix of data and functions
 - e.g. struct `device_driver` – with 7 function pointers
 - Some are purely for holding functions, and a pointer to them is included in other structures
 - e.g. struct `file_operations` – with about 30 function pointers (depending on config)

Example: struct file_operations

Definition in include/linux/fs.h

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    ...
}
```

Declaration and usage in drivers/char/rtc.c

```
static const struct file_operations rtc_fops = {
    .owner    = THIS_MODULE,
    .llseek  = no_llseek,
    .read    = rtc_read,
    .open    = rtc_open,
    .release = rtc_release,
    ...
};

static struct miscdevice rtc_dev = {
    .minor    = RTC_MINOR,
    .name     = "rtc",
    .fops     = &rtc_fops,
};
```

Assigning function pointers


- Assign statically using C tagged structure initialization

- Ex:

```
static const struct file_operations xillybus_fops = {  
    .owner    = THIS_MODULE,  
    .read     = xillybus_read,  
    .write    = xillybus_write,  
    .open     = xillybus_open,  
    .flush    = xillybus_flush,  
    ...  
};
```

- Relies on compiler to default unnamed structure elements to 0 (NULL)
 - “object methods” are defined at compile time
 - It’s uncommon to change function pointers at runtime (after initialization)

Function pointer conventions

- Name for a structure of function pointers:
 - struct xxx_operations
 - Frequently used operations structs: 
- Name for a declaration of the struct:
 - xxx_[x]ops
- Name for the pointer (member in another struct):
 - [x]ops
- Function names have format:
 - <prefix>_<opname>
 - e.g. xillybus_read, xillybus_write

```
51 media_entity_operations
54 configfs_group_operations
54 inode_operations
55 tty_operations
59 smp_operations
60 block_device_operations
65 configfs_item_operations
74 super_operations
98 address_space_operations
131 vm_operations_struct
160 v4l2_file_operations
186 ata_port_operations
218 pernet_operations
325 seq_operations
1606 file_operations
```


Function pointer rationale

- Function pointers allow for polymorphism (subtyping)
 - Caller doesn't know type of object
- More straightforward, and faster than virtual tables
 - Less runtime overhead (no vtable lookup)
 - '_operations' struct is essentially a hand-managed virtual table
- Transparent – you can see from the source what's happening
- Downside to function indirection is reduction in compiler optimization (no inlining), but that's almost impossible with polymorphism anyway

Function pointer resources

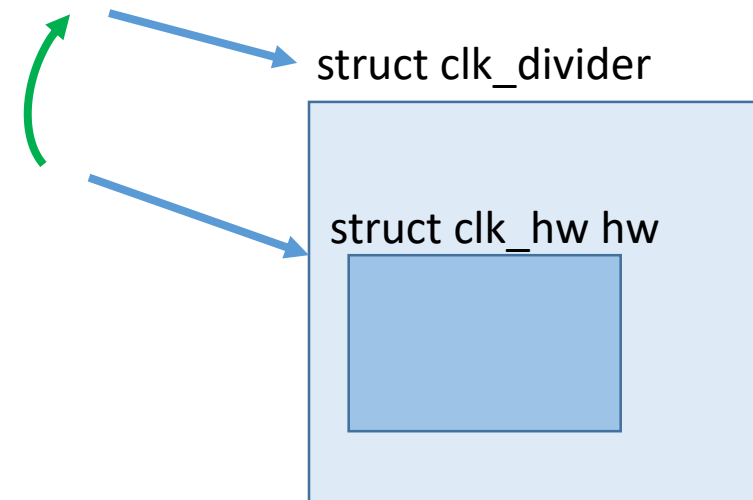
- Object-oriented design patterns in the kernel – part 1, by Neil Brown
 - <https://lwn.net/Articles/444910/>

Container_of

- `container_of` is a macro that converts a pointer to a member of a structure into a pointer to the containing structure
- Lots of internal kernel routines pass around pointers to data objects that are embedded in larger structures
- Here's a picture:

macro 'to_clk_divider()' uses 'container_of' to convert from pointer to `clk_hw` member to enclosing `struct clk_divider`

```
#define to_clk_divider(_hw) container_of(_hw, struct clk_divider, hw)
```



Container_of() defined

- Defined in include/linux/kernel.h

```
#define container_of(ptr, type, member) ({
    void *__mptr = (void *) (ptr);
    BUILD_BUG_ON_MSG(!__same_type(*(ptr), ((type *)0)->member) &&
        !__same_type(*(ptr), void),
        "pointer type mismatch in container_of()");
    ((type *) (__mptr - offsetof(type, member))); })
```

- offsetof() is in include/linux/stddef.h

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

- Reading these will help you go to sleep at night

container_of() explained

- BUILD_BUG_ON_MSG() checks that ptr has the same type as the item in the structure it's supposed to be pointing to
 - Causes an error at compile time, without any overhead at runtime
 - Shows how to do type checking with a macro
- Without the BUILD_BUG_ON(), the routine becomes:

```
#define container_of(ptr, type, member) ({ \
    void *__mptr = (void *) (ptr); \
    ((type *) (__mptr - offsetof(type, member))); })
```

- Notes:
 - The first line is just C casting for the next line's pointer arithmetic
 - offsetof() generates a compile-time constant with the offset of the member item inside the structure (type)
 - The only operation actually performed at runtime is the minus
 - (note: kernel macros have a way of evaporating into very efficient implementations)

container_of() explained

- BUILD_BUG_ON_MSG() checks that ptr has the same type as the item in the structure it's supposed to be pointing to
 - Causes an error at compile time, without any overhead at runtime
 - Shows how to do type checking with a macro
- Without the BUILD_BUG_ON(), the routine becomes:

```
#define container_of(ptr, type, member) ({           \  
    void *__mptr = (void *) (ptr); \  
    ((type *) (__mptr - offsetof(type, member))); })
```

- Notes:
 - The first line is just C casting for the next line's pointer arithmetic
 - offsetof() generates a compile-time constant with the offset of the member item inside the structure (type)
 - The only operation actually performed at runtime is the minus
 - (note: kernel macros have a way of evaporating into very efficient implementations)

container_of resources

- See http://www.kroah.com/log/linux/container_of.html
- <https://stackoverflow.com/questions/15832301/understanding-container-of-macro-in-the-linux-kernel>

Embedded anchor

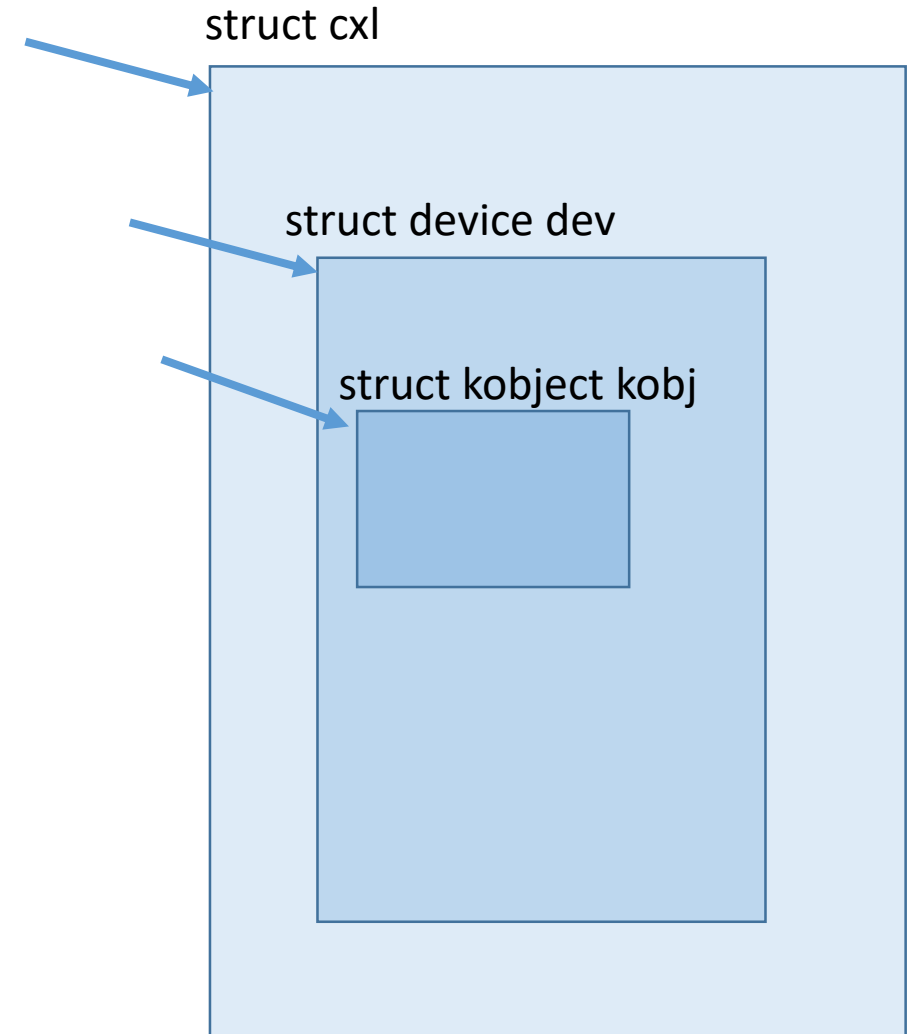
- An object is embedded into another one, and the access to parent object is gained using the pointer to the member
- Is often used with container data structures (lists, hashes, trees, etc.)
 - Can allow objects to be in containers (lists, hashes, etc.)
 - Instead of container pointing to object, container points to anchor inside object
 - Can have as many anchors in the object as needed
- Examples of moving from anchor to object:
 - `interface_to_usbdev` (`include/linux/usb.h`)
 - `kobj_to_dev` (`include/linux/device.h`)
 - `list_entry` (`include/linux/list.h`)

Embedded anchor (cont.)

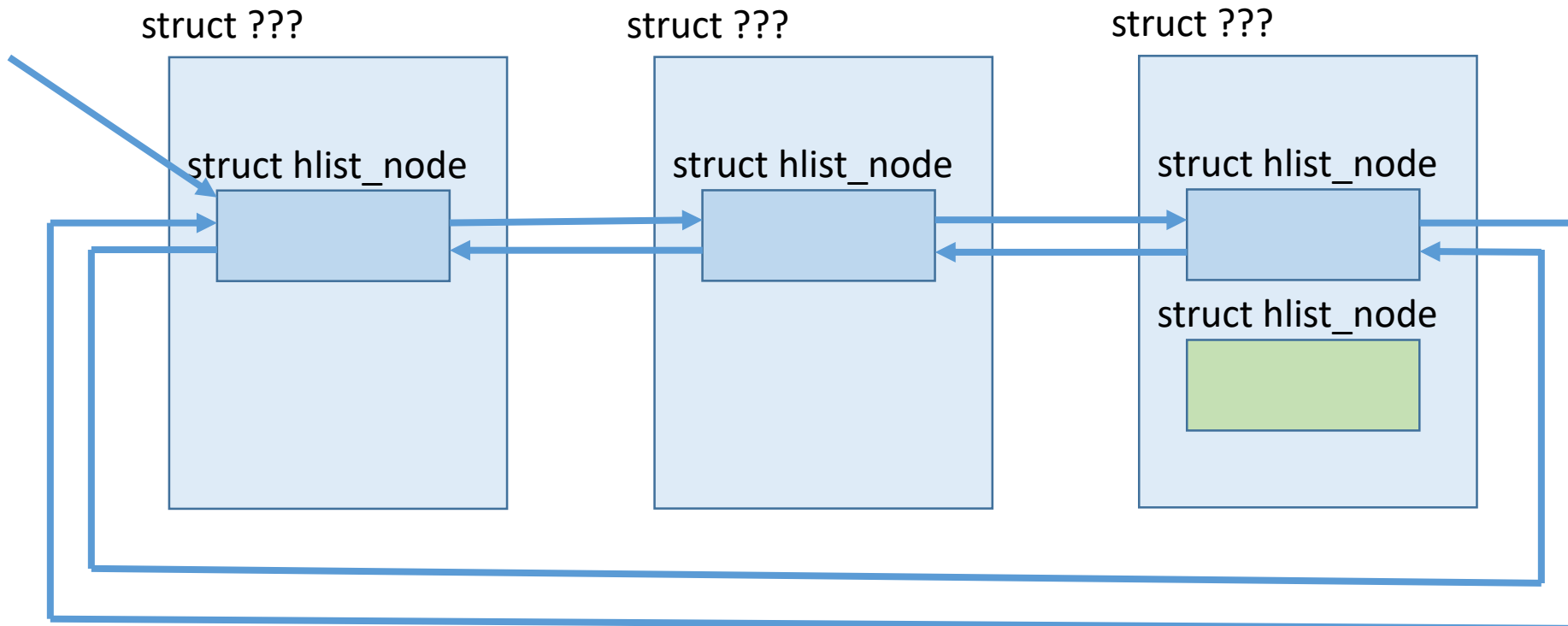
- Uses `containerof()` to back up from anchor to object
- Is used for many different collection types:
 - doubly linked list, singly linked, red-black trees, hashtables
 - Some data types use the same “anchor” structure (e.g. hashtables and lists)
- Reverse of what we normally think of for collections
 - Normally - list struct has separate nodes that point to objects
 - Linux kernel - object struct contains collection node struct (the anchor)
- Also used for things besides collections
 - `struct kobject`
 - `struct dev`
 - many, many more

Embedded anchor diagram

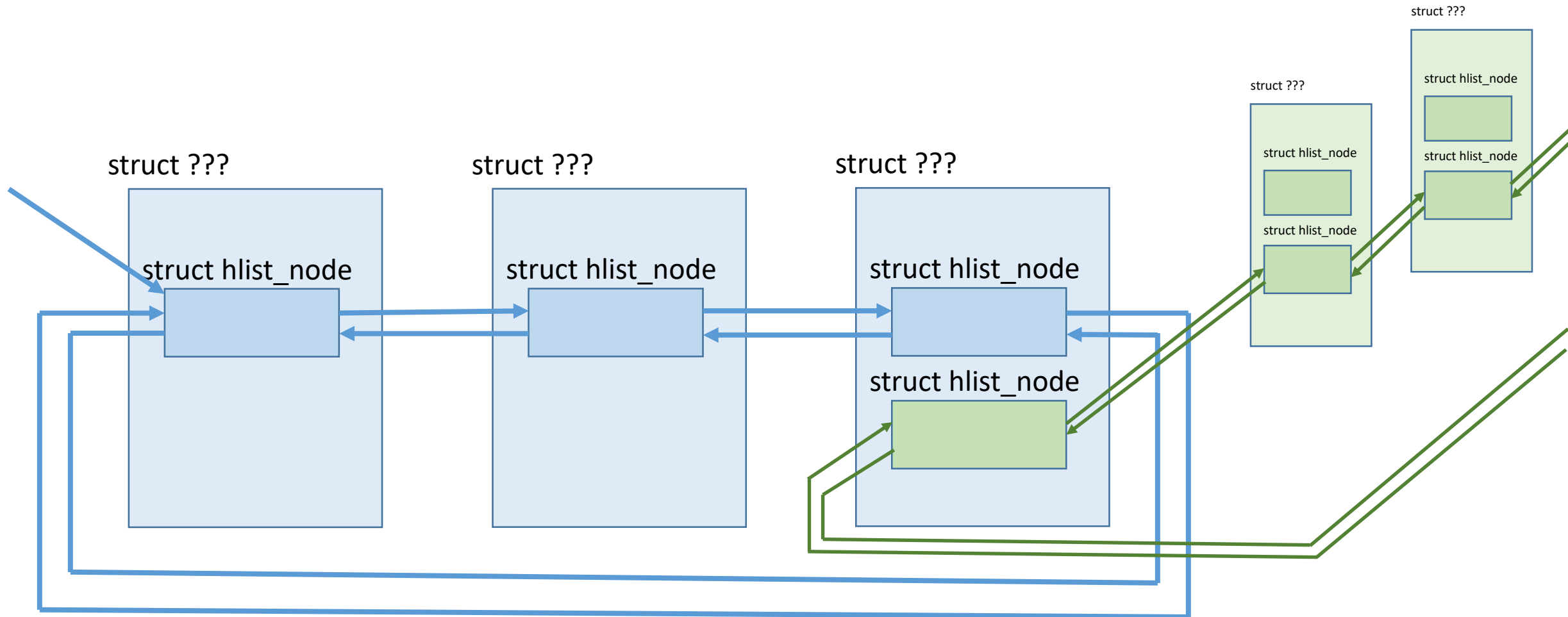
- Example of multiple embedded structures
 - struct cxl, in drivers/misc/cxl/cxl.h



Embedded anchor in linked lists



Items on multiple lists



Embedded anchor Notes

- Rationale
 - Allows for data structure code to be independent of node type
 - Does not require separate allocation of data structure node
 - Allows passing around the pointer to the anchor, instead of the object
 - This allows deferring referencing the object, until needed
- Resources:
 - Linux kernel design patterns – part 2
 - <https://lwn.net/Articles/336255/>

Inlines

- Use inlines instead of macros, where possible
 - You get type checking of parameters and less side effects
- Otherwise, explicit use of inlines is discouraged
 - That is, don't use inlines for 'normal' code
 - gcc will usually do a better job than you at figuring out when to inline
 - You don't know what architecture, number of registers, etc. your code will compile for
 - Yes, there are already over 80,000 inlines in the kernel
 - and about 1200 uses of `__always_inline`
- Guidelines for when to use inlines
 - When you really KNOW you want the code inline
 - ex: `__cmpxchg` is a single instruction – it should be inlined
 - For extremely short code sequences
 - For stubs that the compiler will optimize away
 - e.g. function stubs in header `#ifdefs`

Inline Resources

- Resources:
 - Documentation/process/coding-style.rst (section 15)
 - Documentation/process/4.Coding.rst (Inline functions)
 - Drawing the line on inline
 - <https://lwn.net/Articles/166172/>
 - Who is the best inliner of all?
 - <https://lwn.net/Articles/314848/>

Macros

- Avoiding side effects
- do ... while (0)
- likely/unlikely

Avoiding side effects in macros

- Expressions passed in to macros may be evaluated multiple times, if you are not careful
 - classic example: `#define max(x, y) (x >y ? x : y)`
 - If used with `'max(n++, m)'`, n may get incremented twice (but only sometimes!)
- Solutions:
 - Use inline functions instead of macros
 - Use temp variables in your macros
 - Find an existing kernel macro that does what you need
 - There are LOTS
- For fun, look at definition of `'max(x, y)'` in `include/linux/kernel.h`

See how easy this is? ...

from include/linux/kernel.h

```
#define __typecheck(x, y) \
    (!! (sizeof((typeof(x) *)1) == (typeof(y) *)1)))
#define __is_constexpr(x) \
    (sizeof(int) == sizeof(*(8 ? ((void *) ((long)(x) * 0l)) : (int *)8)))
#define __no_side_effects(x, y) \
    (__is_constexpr(x) && __is_constexpr(y))
#define __safe_cmp(x, y) \
    (__typecheck(x, y) && __no_side_effects(x, y))
#define __cmp(x, y, op) ((x) op (y) ? (x) : (y))
#define __cmp_once(x, y, unique_x, unique_y, op) ({ \
    typeof(x) unique_x = (x); \
    typeof(y) unique_y = (y); \
    __cmp(unique_x, unique_y, op); })
#define __careful_cmp(x, y, op) \
    __builtin_choose_expr(__safe_cmp(x, y), \
        __cmp(x, y, op), \
        __cmp_once(x, y, __UNIQUE_ID(__x), __UNIQUE_ID(__y), op))


#define max(x, y) __careful_cmp(x, y, >)
```

do ... while (0)

- How to structure macros that work no matter what context they are used in

- do ... while(0) macros

- Example: in header:



```
#define perform_foo do { \  
    bar(); \  
    baz(); \  
} while (0)
```

- All statements in the block are performed exactly once
- Rationale:
 - Allows multi-statement macros to work within if or while blocks, regardless of the use of braces, or semi-colon placement

do ... while(0) (cont.)

- Examples of usage:

```
do perform_foo;  
while (some condition);
```

```
do {  
    perform_foo;  
    other_stuff;  
} while (some condition);
```

```
if (some condition)  
    perform_foo;
```

```
if (some condition) {  
    perform_foo;  
    other_stuff;  
}
```

If not wrapped in a do ... while (0),
this one becomes:

```
if (some condition)  
    bar();  
    baz();
```

which will fail to execute baz() due
to missing braces

- Resources:

- See <https://www.quora.com/What-is-the-purpose-of-using-do-while-0-in-macros>

* *<vent>'if' without braces is the thing I disagree with most in kernel coding style </vent>*

Likely/unlikely

- likely/unlikely macros can be used to indicate probability of a conditional, in an if statement
 - There are about 23000 in the kernel now (not counting macros)
- Are used to annotate the code to allow better optimization by gcc
 - Usually mean the compiler will avoid a jump for the most likely branch, which helps to avoid pipeline stalls
- Should not be used if the likeliness is affected at runtime by workload
 - Let the CPU branch predictor handle those cases
 - In weird, really high-performance cases, use “static keys”
 - See [Documentation/static-keys.txt](#)
- Is often used in Linux for error paths
 - It keeps error code out of the hot paths

Likely/Unlikely (cont.)

- Some developers recommend against them
 - But not all CPUs have branch predictors (especially low-end embedded)
- Guideline for when to use:
 - Only use if you are confident the taken/not taken ratio of the branch is greater than (10:1)
 - Developers are notoriously bad at estimating these ratios
- Resources:
 - How likely should likely() be?
 - <https://lwn.net/Articles/70473/>
 - Likely unlikely()s
 - <https://lwn.net/Articles/420019/>

ifdef

- Don't put #ifdefs in C code
- Compile out entire functions, instead of parts of functions
- Convert conditional code to inline functions in header files
 - Use stubs in the #else case
- Use IS_ENABLED()

Example: drivers/usb/hid_core.c

- Wrong way:

```
static void hid_process_event(struct hid_device *hid, struct hid_field *field,
                             struct hid_usage *usage, __s32 value)
{
    hid_dump_input(usage, value);
    ...
#ifdef CONFIG_USB_HIDDEV
    if (hid->claimed & HID_CLAIMED_HIDDEV)
        hiddev_hid_event(hid, usage->hid, value);
#endif
    ...
}
```


Example: drivers/usb/hid_core.c

- Right way:

- hiddev.h

```
#ifdef CONFIG_USB_HIDDEV
    extern void hiddev_hid_event(struct hid_device *, unsigned int usage, int value);
#else
    static inline void hiddev_hid_event(struct hid_device *hid, unsigned int usage, int value) {}
#endif
```

- hid_core.c

```
static void hid_process_event(struct hid_device *hid, struct hid_field *field,
                             struct hid_usage *usage, __s32 value)
{
    hid_dump_input(usage, value);
    ...
    if (hid->claimed & HID_CLAIMED_HIDDEV)
        hiddev_hid_event(hid, usage->hid, value);
    ...
}
```

ifdef notes

- Rationale:
 - Code without #ifdefs is much more readable
 - There's no performance penalty for using an inline empty function
 - Compiler will omit the entire conditional statement when the code inside is empty
 - That is, the 'if' goes away automatically if it's block is empty
- Notes:
 - It may require refactoring your code
 - When debugging, need to remember that some of the code is not actually there in the binary
 - Use a function trace or debugger to see what's going on
 - There's a special macro for testing CONFIG variables

IS_ENABLED conditional for CONFIG_ variable

- Use `IS_ENABLED(CONFIG_FOO)` instead of `#ifdef CONFIG_FOO...`
 - Defined in `include/linux/kconfig.h`
 - `IS_ENABLED()` reads like C code although most of it is C pre-processing
- Use `'#if IS_ENABLED()'` for things like structure definitions
- Use `'if (IS_ENABLED())'` as a statement in C code
- Example:
 - `drivers/usb/core/hcd.c`

```
void usb_hcd_unmap_urb_setup_for_dma(...)
{
    if (IS_ENABLED(CONFIG_HAS_DMA) &&
        (urb->transfer_flags & URB_SETUP_MAP_SINGLE))
        dma_unmap_single(hcd->self.sysdev,
            ...);
    else if (urb->transfer_flags & URB_SETUP_MAP_LOCAL)
        hcd_free_coherent(urb->dev->bus,
            ....
```

ifdef Resources

- See [Documentation/process/coding_style.rst](#)
 - Conditional compilation (item 21)

printks

- Don't add more printks to startup
 - Yes, we know your initialization is important, but startup messages are already long enough
 - Don't think about what users of your code want to see
 - Think of what the millions of people who don't use your code don't want to see
- Preferred to use `pr_<level>`:
 - `pr_emerg`, `pr_alert`, `pr_crit`, `pr_err`, `pr_warn`, `pr_notice`, `pr_info`, `pr_debug`
- Do use sub-system-specific printks, if they are available
 - `dev_printk`, `netdev_printk`, `v4l2_printk`, etc.
 - Use of the `xxx_<level>` versions is preferred (e.g. `dev_err`)
- But don't go nuts and make your own new printk wrappers
 - There are hundreds already


Printks aplenty...

- Here are some popular printks:



```
20 sr_printk
22 scli_early_printk
23 sbridge_printk
25 edac_mc_chipset_printk
27 uvc_printk
30 err_printk
35 dbgp_printk
39 no_printk
40 sd_printk
40 srm_printk
41 st_printk
44 bpf_trace_printk
49 starget_printk
49 trace_printk
54 dbg_printk
55 netdev_printk
57 bpf_printk
59 iscsi_conn_printk
61 vdbg_printk
62 DEBC_printk
64 apic_printk
125 edac_printk
129 ecryptfs_printk
149 arc_printk
155 netif_printk
168 scmd_printk
217 asd_printk
306 sdev_printk
310 dev_printk
379 ql4_printk
410 shost_printk
600 snd_printk
634 pm8001_printk
2297 TP_printk
```

Switch statement fall-through

- Work was recently completed to analyze fall-through for case blocks
 - Annotations were added to cases that correctly fall through to next case
 - ie are missing a 'break' statement at the end of the case
 - Several bugs were fixed where the fall-through was unintentional
 - We can now turn on gcc `-Wimplicit-fallthrough` without a warning storm, and catch unintentional implicit fallthroughs in new code (to catch possible bugs)
- If you do a fall-through, annotate it
 - Use the comment: `/* fall through */`, like so: 
 - Other strings are supported, but please don't proliferate weirdness
 - e.g Don't do: `/* FALL THRU */` or `/* lint -fallthrough */`
- Resource:
 - An end to implicit fall-throughs in the kernel
 - <https://lwn.net/Articles/794944/>

```
...
case 2:
    do stuff ;
    /* fall through */
case 3:
...

```

Conclusions

Conclusions

- Kernel coding is an exercise in balance
 - Take into consideration other developers
 - Take into consideration balance between correctness, performance and maintainability
 - Keep in mind that maintainers are scarce and overworked
- It is usually best to do things the same way as everyone else
 - If your code looks different, figure out why

Conclusions (cont.)

- Most of these idioms are about performance and maintainability
- The kernel is very fast – let's keep it that way
 - Lots of very advanced algorithms (e.g. RCU, xarrays, locking)
 - Very conscious of impacts on cache, lock contention
- A LOT of maintainability comes from consistency
 - But the consistency is for YOUR maintainer's subsystem, not necessarily across the entire kernel
 - If your subsystem maintainer tells you to do things a certain way,
Do it that way

Overall resources

- Greg KH talk on codingstyle:
 - http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/index.html
- Documentation/process/coding-style.rst
- “What Every Programmer Should Know About Memory”, by Ulrich Drepper
 - <https://akkadia.org/drepper/cpumemory.pdf>
- “Linux Device Drivers, Third Edition”
 - <https://lwn.net/Kernel/LDD3/>
- lwn.net kernel pages:
 - https://lwn.net/Kernel/Index/#Coding_style
 - https://lwn.net/Kernel/Index/#Development_model-Patterns

More resources

- Excellent series of articles by Neil Brown:
 - Linux kernel design patterns – part 1
 - <https://lwn.net/Articles/336224/>
 - Linux kernel design patterns – part 2
 - <https://lwn.net/Articles/336255/>
 - Linux kernel design patterns – part 3
 - <https://lwn.net/Articles/336262/>
 - Object oriented design patterns in the kernel – part 1
 - <https://lwn.net/Articles/444910/>
 - Object-oriented design patterns in the kernel – part 2
 - <https://lwn.net/Articles/446317/>

Thanks!

You can e-mail me directly: tim.bird@sony.com

SONY[®]

SONY

Stuff I may or may not use

- Following is additional material that I may use if the talk runs short
- I left it here for anyone reading these slides after the event, in case it's found useful

Coding anti-Idioms

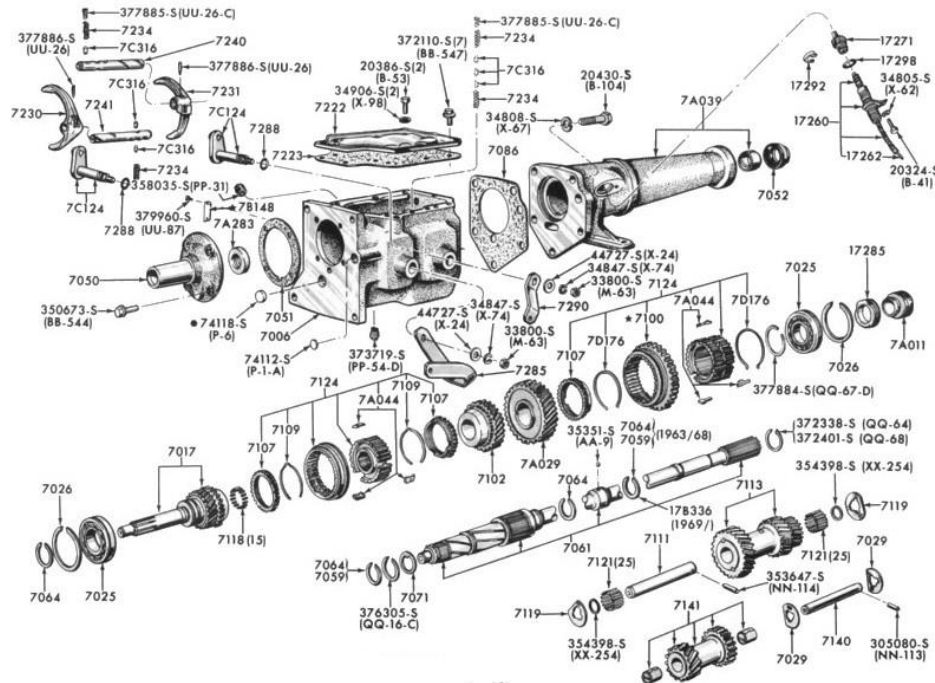
- Support for out-of-tree code
- Big code drops
- Compatibility layers
- Breaking user space
 - Never, ever, ever break user space!!

Support for out-of-tree code

- Out-of-tree code can't be maintained by the public
- Support inside the kernel for out-of-tree code can't be tested, supported or maintained
- Hybrid drivers (part in and part out of the kernel) are usually rejected
- If things are being put into the kernel over time, use staging

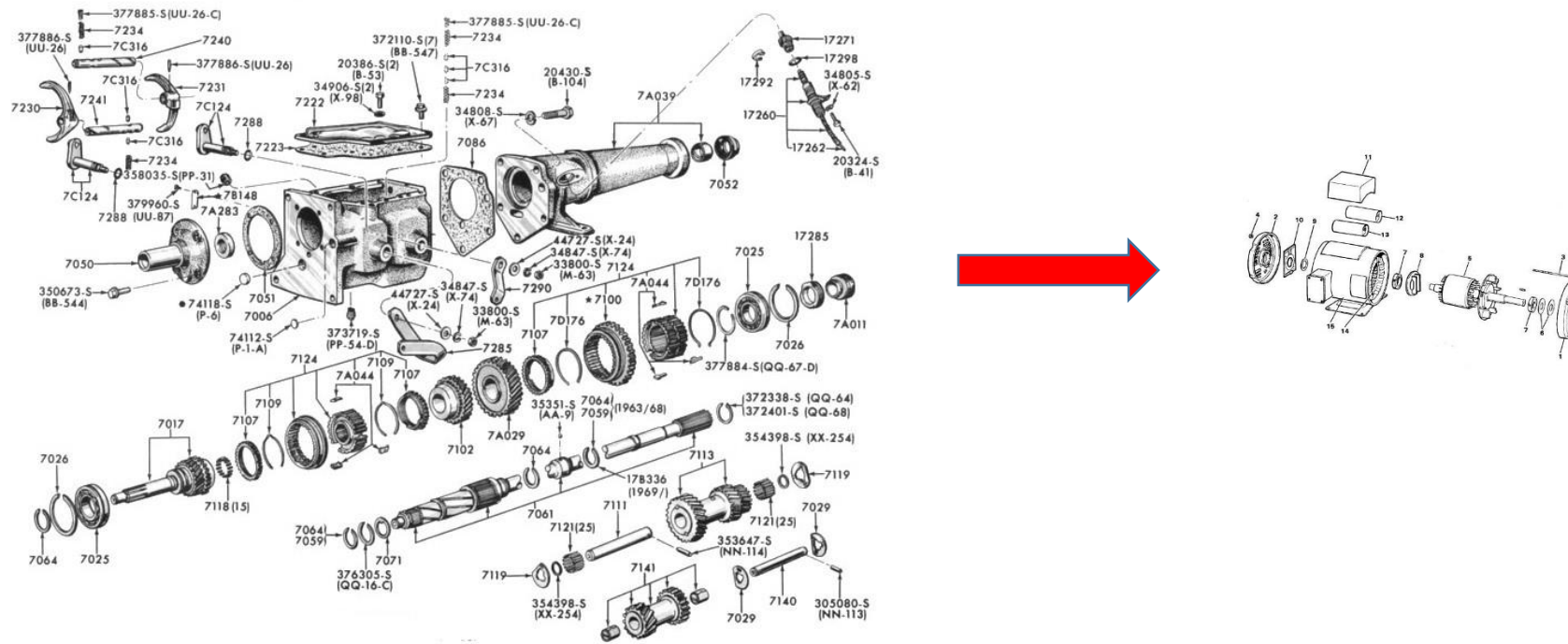
Big code drop

- Sometimes, developers work on a large patch in isolation
- Wait until end of development cycle to contribute
- Attempt to contribute, but find that community wants something else



Big code drop

- Developers often asked to refactor contributions into smaller items
 - Integrate with and use kernel features better
- I have seen code reduced 5-fold in size during refactoring, while keeping same functionality



Releasing late in development

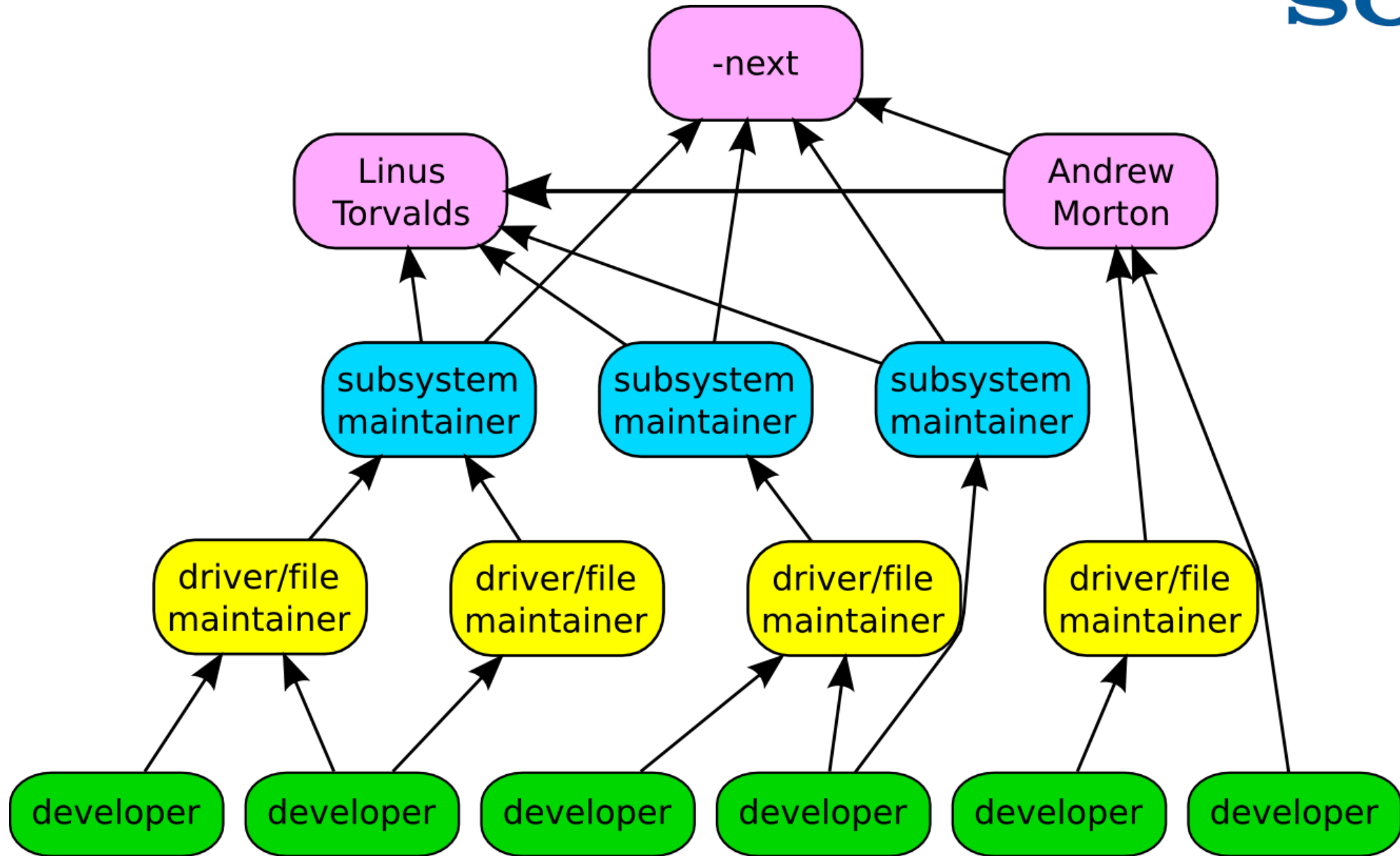
- This is a very common error – everyone has done this!
- You should submit your idea, design and implementation early
 - This is often difficult
- Should view community as co-developers of the code
- Early input (during development) can avoid large patch rework later

Compatibility layers

- A lot of companies don't want to write drivers for multiple operating systems
- Solution (for them) is to write a compatibility layer
 - Macros and functions which hide the difference between multiple Oses
- These layers add inefficiencies to the code, and can't be maintained by Linux kernel developers
 - Requires knowledge of other OS
- While it makes sense from the perspective of the vendor, it doesn't work for the community as a whole
- Items that are part of or contribute to compatibility layers are not accepted.

Breaking user space

- Kernel exists to mediate access to shared machine resources to the user processes
 - cpus, memory, devices, power, storage, I/O
- User-space processes are the reason people use computers, not the kernel
 - Even if they are buggy and wrong
- Backwards compatibility is a huge contributor to the success of Linux
- Doesn't mean to accept *any* behavior in user space
 - Rule is: if it worked before, it should continue working



Make your maintainer happy

- Don't just solve your problem...

Solve the
maintainer's problem

- What maintainers want:
 - The software can be used by more people
 - It's not harder to maintain
 - New stuff should NOT break existing users
- The older the project, or the more complex it is, the harder to maintain these attributes
 - Parts of the kernel are very difficult to contribute to

General Principles for working with maintainers

- Maintainability
- Generalization vs. specialization

Maintainability

- Maintainer wants to control complexity of code
 - Makes maintainers life easier
- Maintainability is more important than features
- This means:
 - Quick and dirty hacks are almost never accepted
 - Sometimes, a new feature will require refactoring the code in a major way

Puppies!

- To a maintainer, your patch might look like a puppy
- Sometimes people love to receive puppies
 - They are cute and fun!
- Sometimes people don't want a puppy
 - They require ongoing maintenance that grows over time
- If it looks like you'll stay and take care of the patch (puppy), the maintainer is more likely to accept it



url=<https://www.flickr.com/photos/27587002@N07/5170590074/>

Photo by Jonathan Kriz, license=[CC-by-2.0](https://creativecommons.org/licenses/by/2.0/)

Generalization vs. specialization

- Goal is to serve as many people as possible
- Code that generalizes the software is preferred over code that specializes the software
 - Example: change definitions of registers to support a new piece of hardware vs add new abstraction layer to support whole category of new hardware
- This means:
 - If you introduce a feature, it should support other use cases and not just your own
 - This can be difficult to determine
 - Communicate with other community members to see what their needs are

Generalization vs. specialization



- Modular
- Interchangeable
- Reusable



- Custom
- Specific
- Fit-for-purpose

Don't submit low-quality or specialized code

- Low-quality
 - Workarounds and quick hacks
- Specialized code
 - Handles only the specific use case for that market
 - Attitude that code is “throwaway”, or that code is “good enough” for one release
 - Assumption that code does not need to be reused
 - Not generalized for other use cases
- Code written in different style than upstream

Thanks!

You can e-mail me directly: tim.bird@sony.com

SONY[®]

SONY