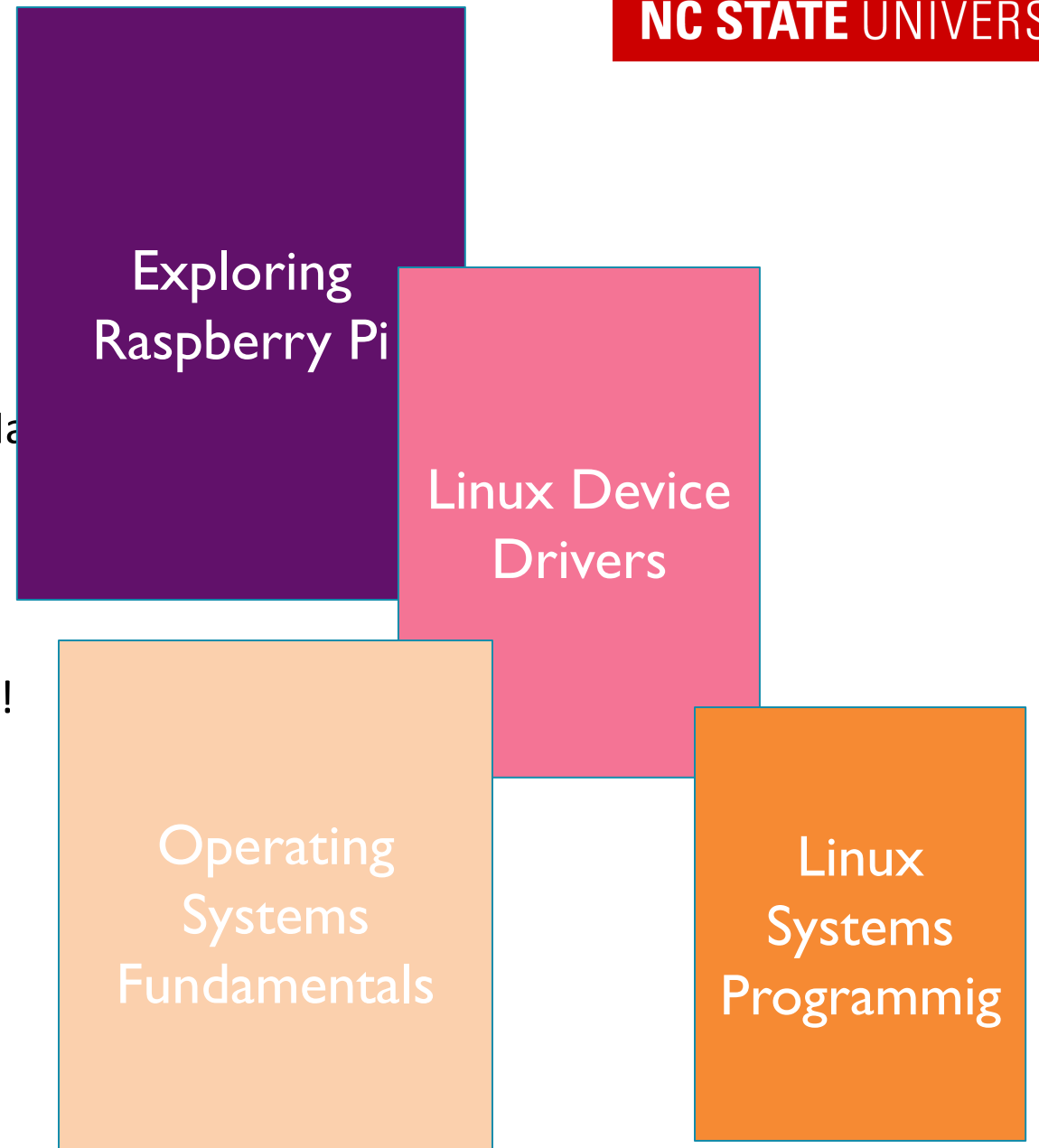
# Loadable Kernel Modules and Device Drivers

v0.99

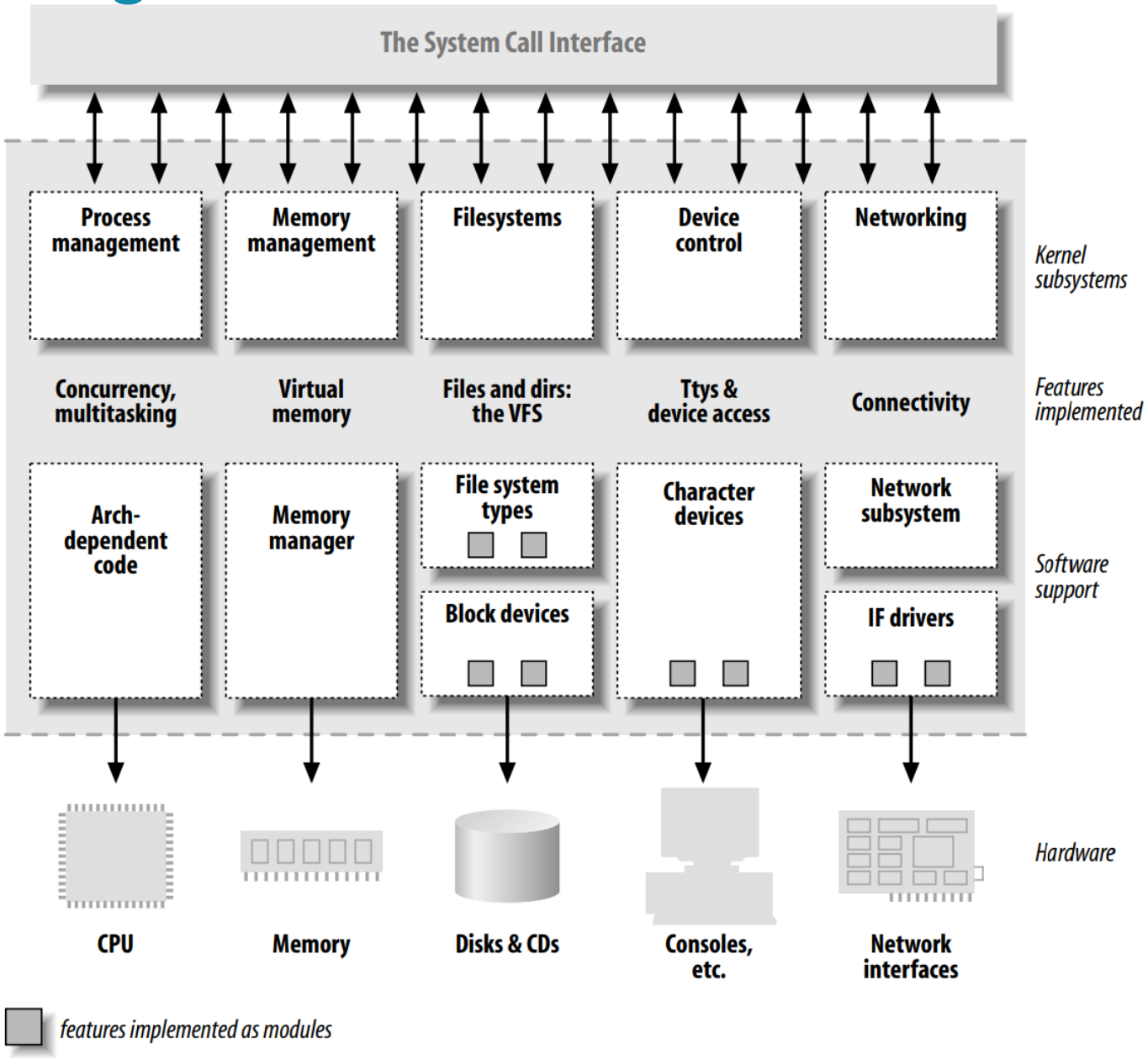
# Overview

- Background (and diagram sources)
  - **Linux Kernel Programming 2** (TBD)
  - **Linux Device Drivers** (Corbet, Rubini and Kroah-Ha
    - Chapters 1-3
  - **Exploring Raspberry Pi** (Molloy)
    - Chapter 16 in textbook
    - Slides highlight *some* points – read the chapter!
  - **Operating Systems Fundamentals**
- Kernel modules
- Four ERPI example programs
  - Module basics
  - Interrupts and GPIO
  - sysfs interface
  - Kernel thread



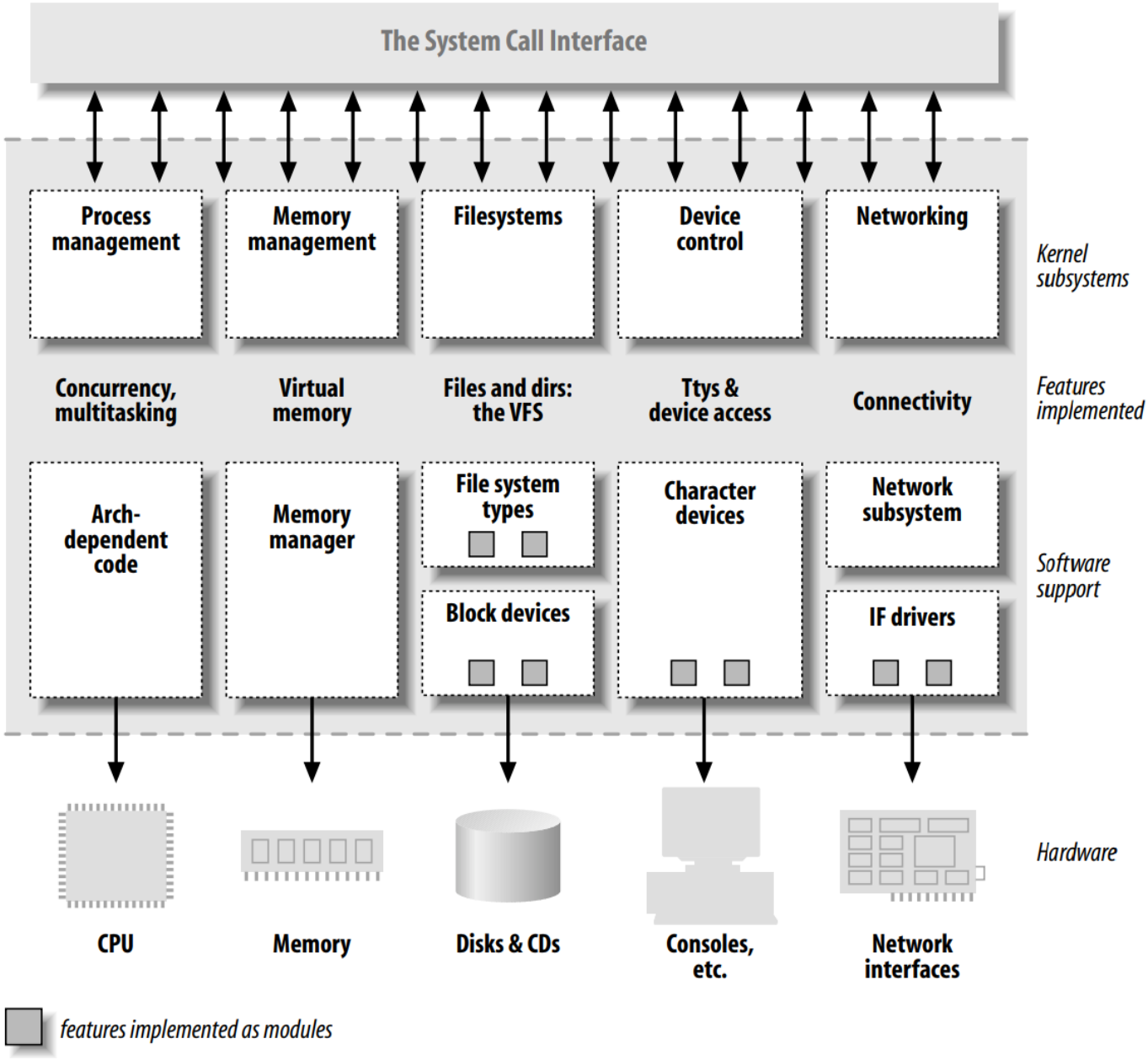
# Keep User Code from Misbehaving

- For safety and modularity, only kernel can access critical features (part of "kernel space")
  - Special CPU instructions
  - CPU interrupt system (including ISRs)
  - CPU control registers
  - Peripherals
- User code can't (part of "user space")
  - Must use **system call interface** to ask kernel to do the work (and it may refuse)
  - Can't access memory space of other processes (user or kernel)

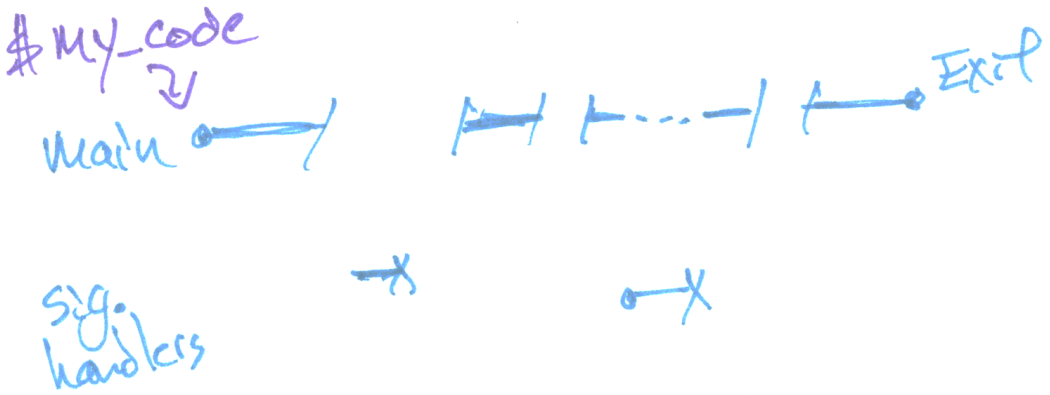


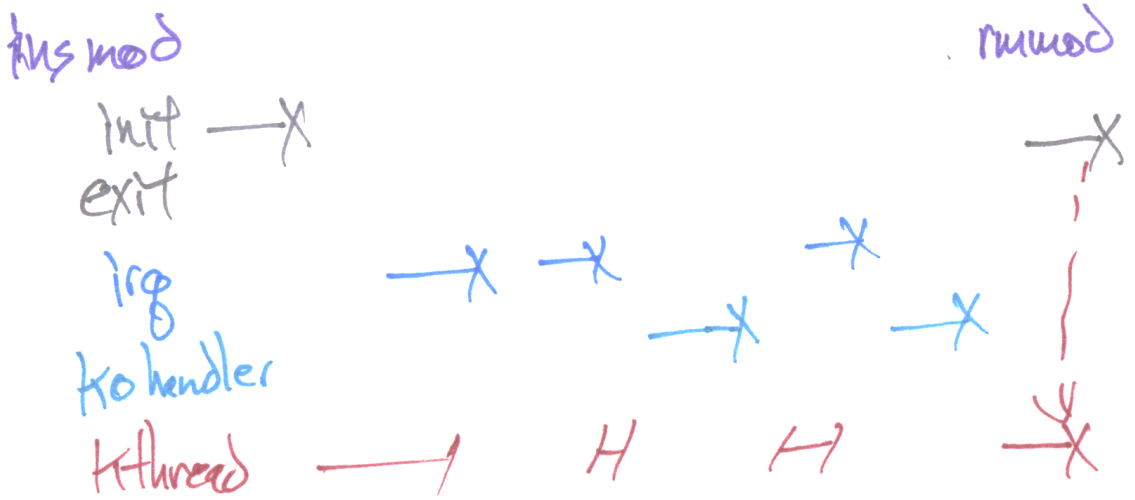
# Changing the Kernel

- Can add/remove features to/from kernel at run-time by using **loadable kernel modules**
  - Modules are dynamically linked (addresses resolved at run-time)
  - Don't need to recompile kernel to do this
- Use **insmod** and **rmmod** to insert and remove modules
  - Must have rights – root/superuser
- Device drivers (and other things) are implemented with modules
- Modules can provide system call interface to user space programs



# Code Execution Models

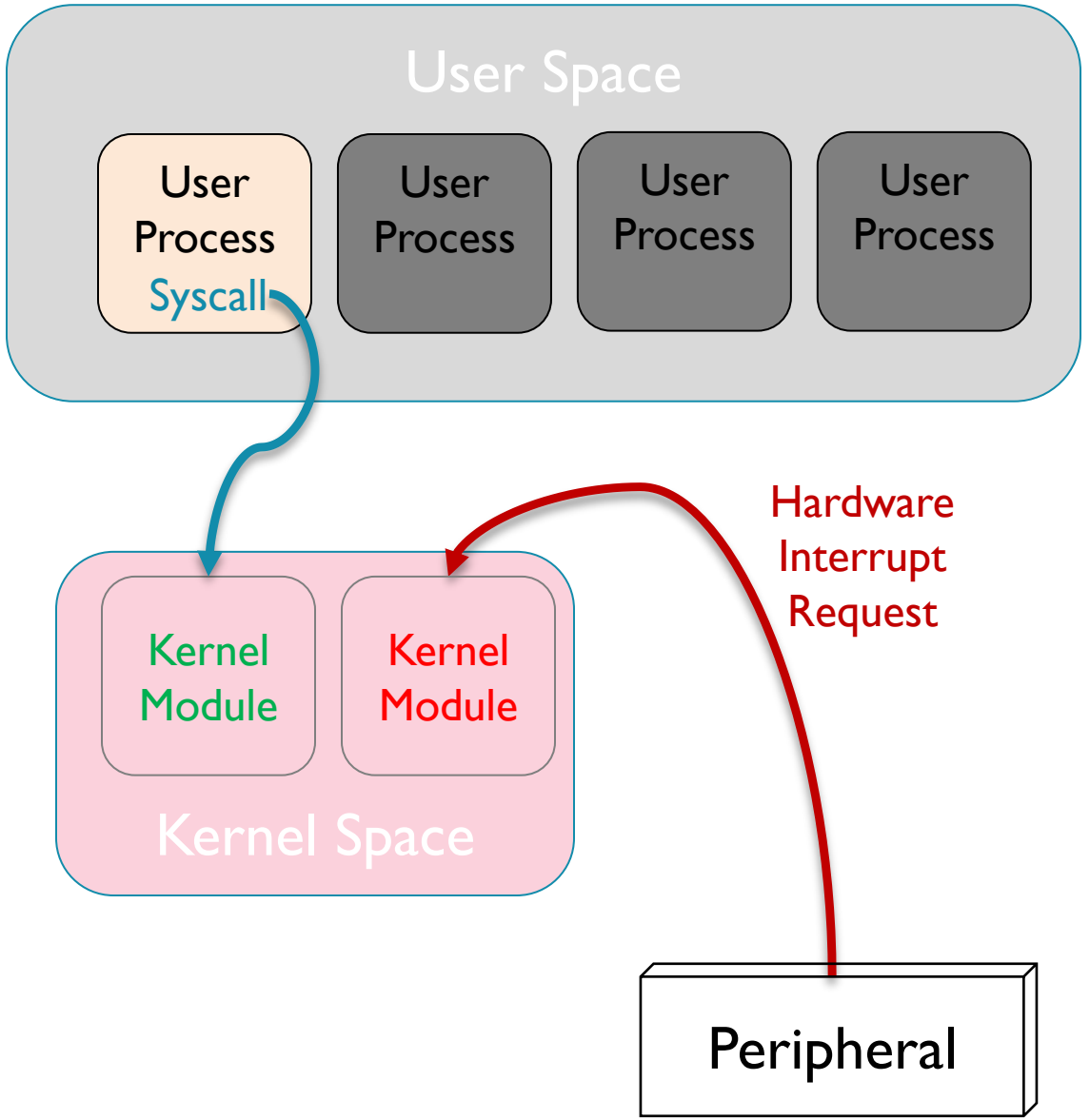



- Program – usually more sequential execution
  - Structure
    - Sequential thread(s) which don't exit until end of thread X, but may block /
    - Possibly with a few event handlers/callbacks added, typically run-to-completion X
  - Behavior
    - Loader allocates memory, loads program with shared library modules
    - Program runs, starting at main()
    - Program completes, OS frees allocated memory
- Kernel module – event-triggered execution (usually)
  - Structure
    - Multiple handler/callback functions, each runs to completion X per event
    - Possibly a sequential thread (or more) added, typically runs as long as module is loaded
  - Behavior
    - On loading, initialization code runs and completes
    - Events trigger handlers to execute
    - On unloading, exit code runs

# Module Execution Triggers

- What triggers a module to run?
  - Module management operations: Loading with insmod, unloading with rmmod
  - System call (software interrupt) from process
  - Hardware interrupt from peripheral



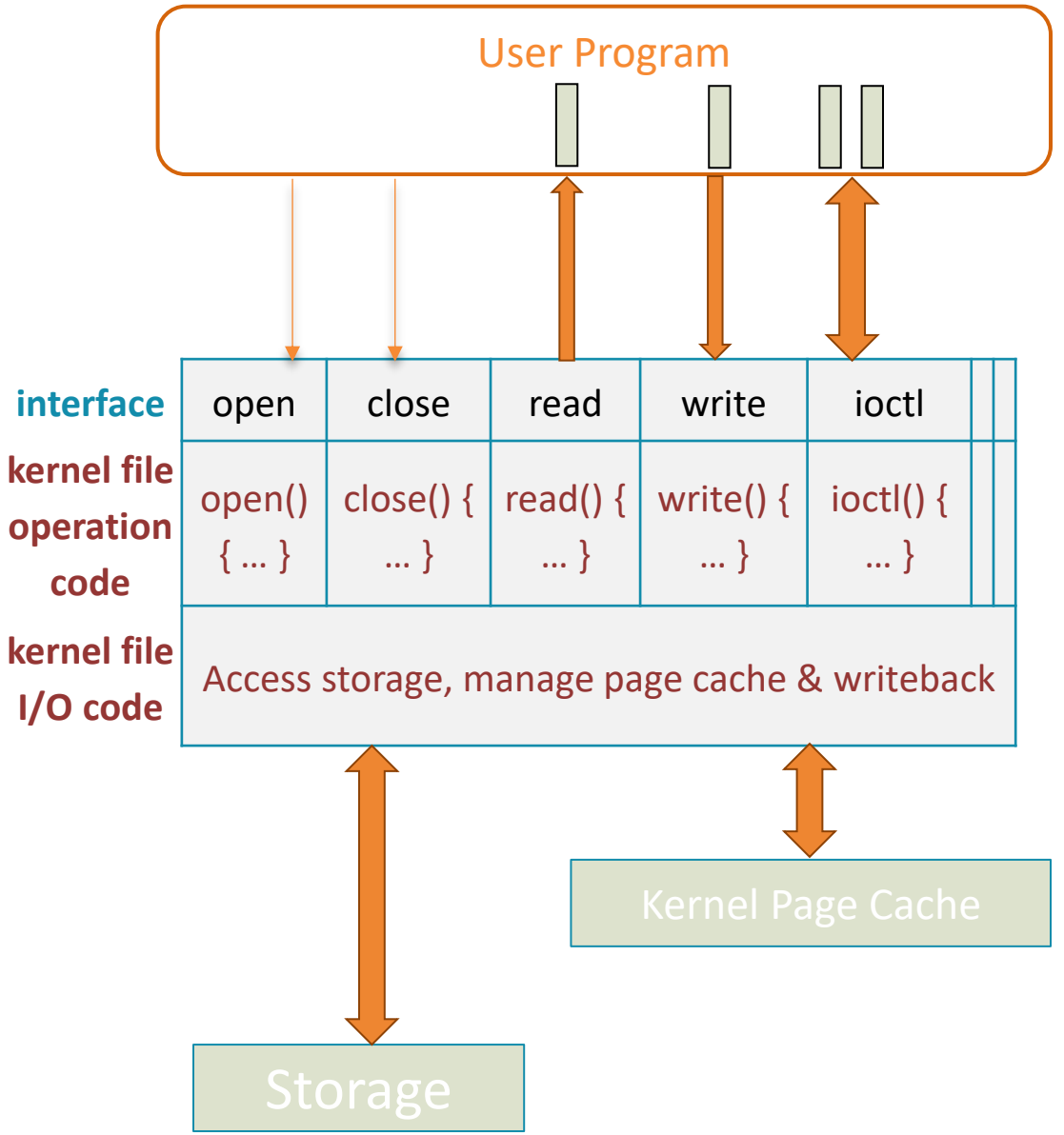
# User-Kernel Communication Pathways

## See Chapter 2 of **LKP2** (Billimoria)

- Possible communication methods
  - Use virtual file system interface
    - Use “fops” (struct file_operations) to define custom callback functions to read or write data
  - sysfs: proper way to interface with user space
    - “One value per sysfs file” rule – difficult to scale up, so good fit for only some applications
  - debugfs: no rules
  - procfs: Don’t use. Exists for in-kernel use

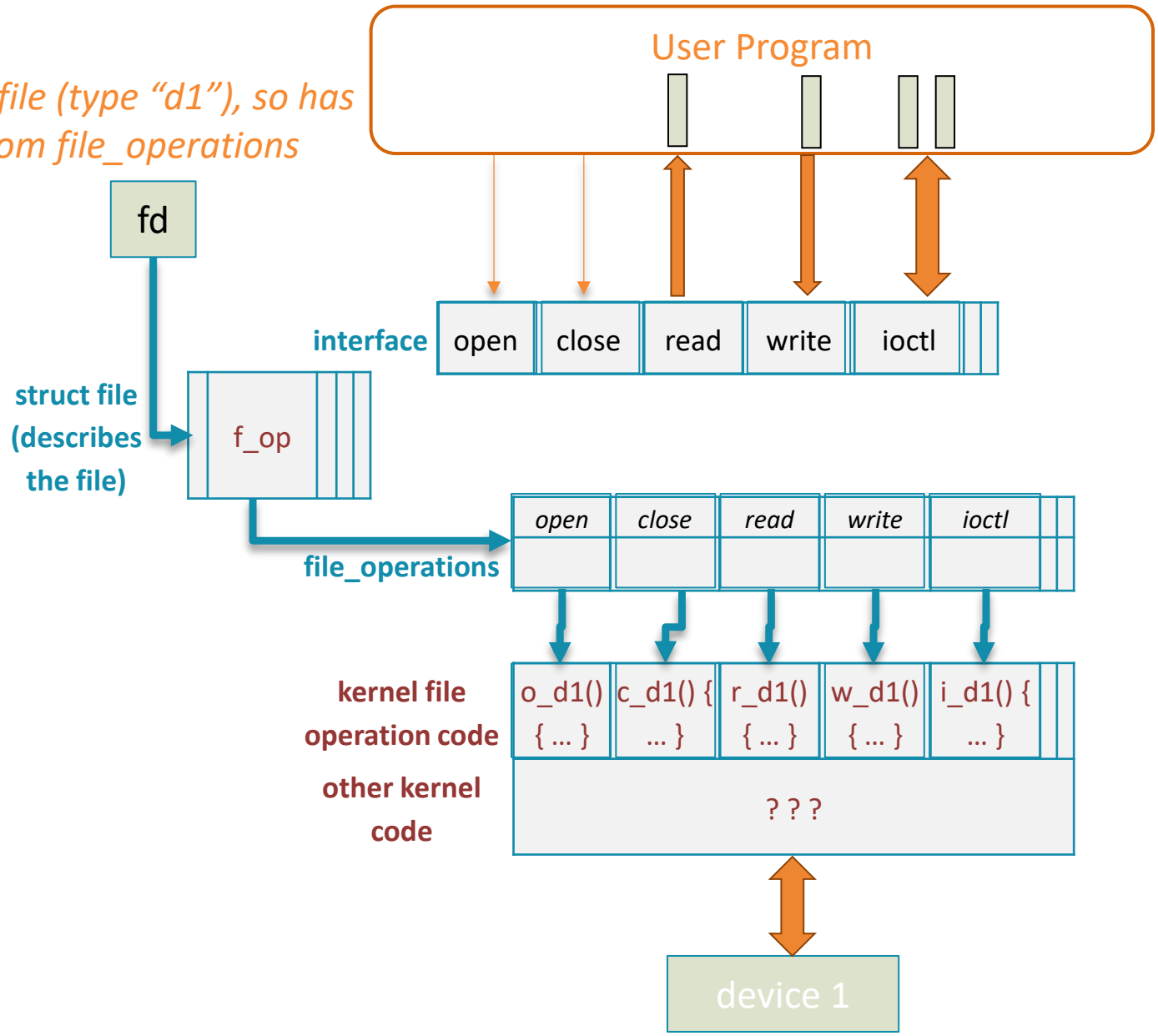
# File System Interface Review

- Kernel file system code offers standard operations on files (common file model)
  - open, close, read, write, llseek, ioctl, poll, flush, etc.
- User program requests operations with system calls
- When operating on a regular* file, kernel code will access storage or page cache
  - * Regular file: linear array of bytes in storage (or cached), called “byte stream”
- Other (“special”) file types represent **kernel objects**
  - Device files
    - Block: Data is **array** of bytes, may access in **random** order
    - Char: Data is **queue** of bytes, **must** access in **sequential** order
  - IPC (inter-process communication): Named pipes, sockets



# Virtual File System

- Access special file types with virtual file system
  - Retarget file operations to do other work based on device type
- Call to **open** creates a **file** struct describing open file
  - File descriptor fd indicates which **file** struct refers to the file
- **file** struct contains pointer f_op to translation table
  - From type of syscall (write?) to function which performs that file operation
- Opening a regular file uses default functions for file operations
- Special files use different file operations
- For device drivers, we override file operations as needed



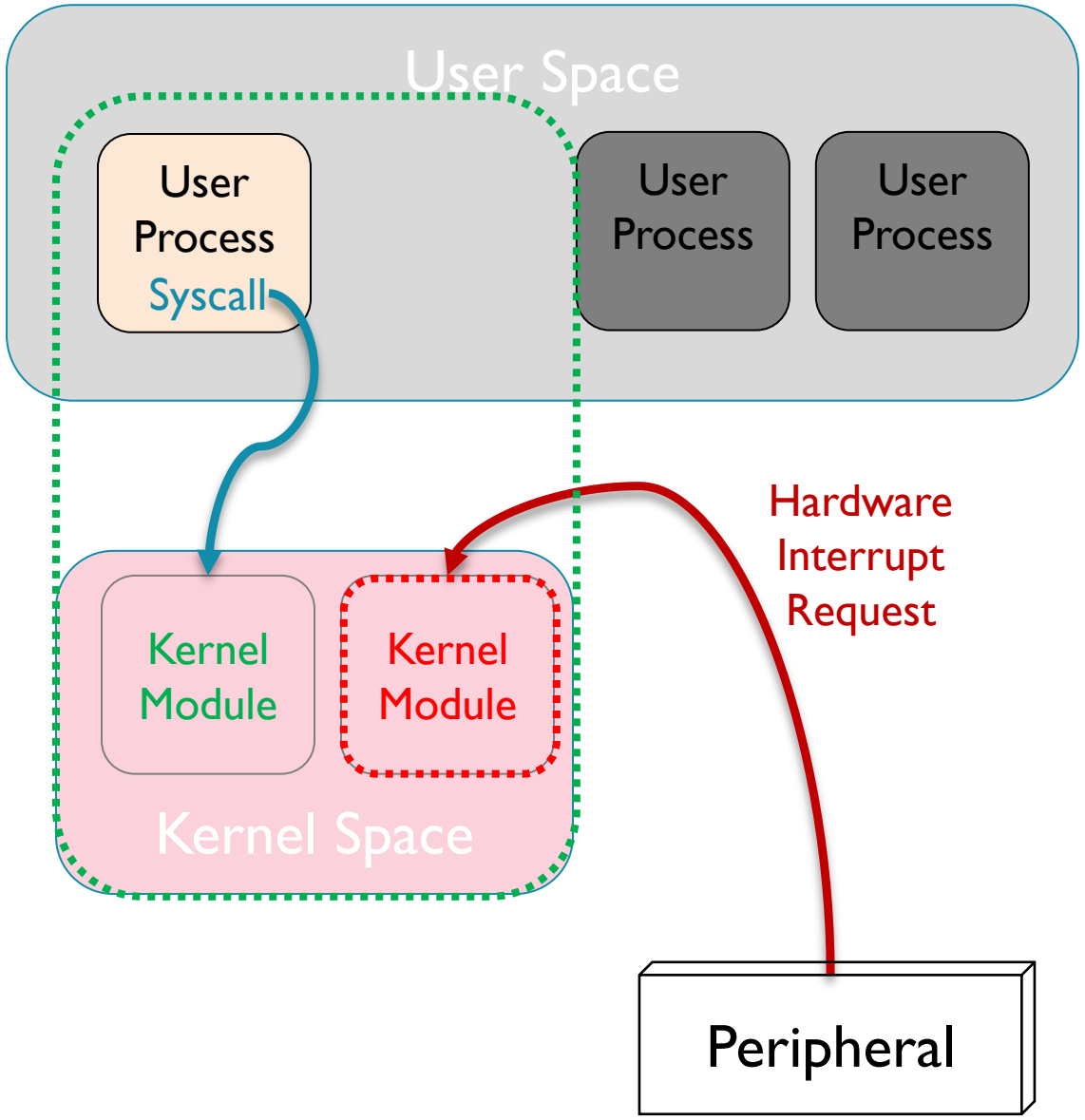
# Details of Mapping File Operations to Code

- file_operations structure maps operations to code
  - See LKP2 pp. 22-28, LDD pp. 49-53
  - struct is defined in in linux/fs.h
    - less /lib/modules/`uname -r`/source/include/linux/fs.h
- Fields in file_operations
  - Owner module
  - Entry for each possible operation
    - Holds pointer to method function (or null if not implemented)

```
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 *);
        int (*iterate) (struct file *, struct dir_context *);
        int (*iterate_shared) (struct file *, struct dir_context *);
        __poll_t (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        unsigned long mmap_supported_flags;
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *,
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
```

# Which Address Spaces are Accessible Depends on Trigger

| What triggers module to run? | What memory can module access? |
|---|---|
| System call (software interrupt) from process | Does work for the calling process, so module has access to address spaces of kernel and calling process |
| Hardware interrupt from peripheral | Not associated with any process, so have access only to kernel address space |



# Copying Data between User and Kernel Spaces

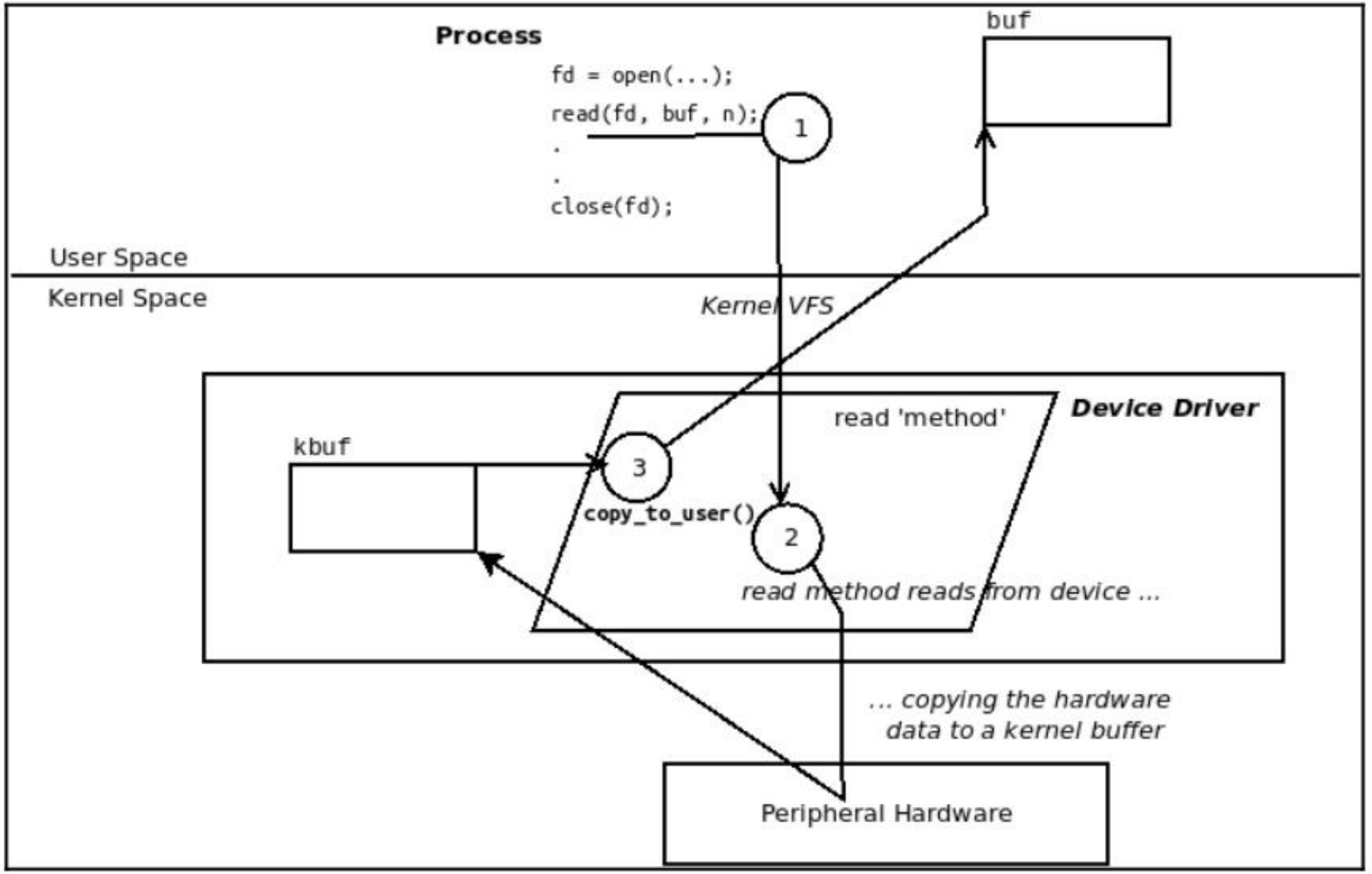



Figure 1.7 – Read: copy_to_user(): copying data from the hardware to a kernel buffer and from there to a user space buffer

# Module Basics

- Linked only to kernel code (not user libraries), so…
  - **Most header files should not be #included**
  - **Can only use functions in kernel space**
  - **Can't use printf. Use printk instead (no floating-point support)**
- Required Functions
  - insmod command runs …_init() function
  - rmmod command runs …_exit() function
- Code execution model is event-driven
  - … until we get to kernel threads

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

# SKIP: Coverage of Device Drivers in Textbooks

| | LKP2 | | OSF | | ERPi | | | |
|---|---|---|---|---|---|---|---|---|
| Chapter | 1 | | 2 | | | | | |
| Feature | miscdrv | miscdrv_rdwr | | Hello 16-1 | gpio_test 16-3 | button 16-5 | led 16-6 | |
| _init | y | | y | y | y | y | y | |
| fops | y | | y | | | | | |
| misc_[de]register | y | | ? | | | | | |
| _exit | y | | y | y | y | y | y | |
| open_, close_ | y | | | | | | | |
| read_, write_ | y | | y | | | | | |
| context | | y | | | | | | |
| copy_to_user, copy_from_user | | y | | | | | | |
| | | | | | | | | |
| GPIO | | | | | y | y | | |
| ISR | | | | | y | | | |
| kobject sysfs interface | | | | | | y | y | |
| Kernel Thread | | | | | | | y | |

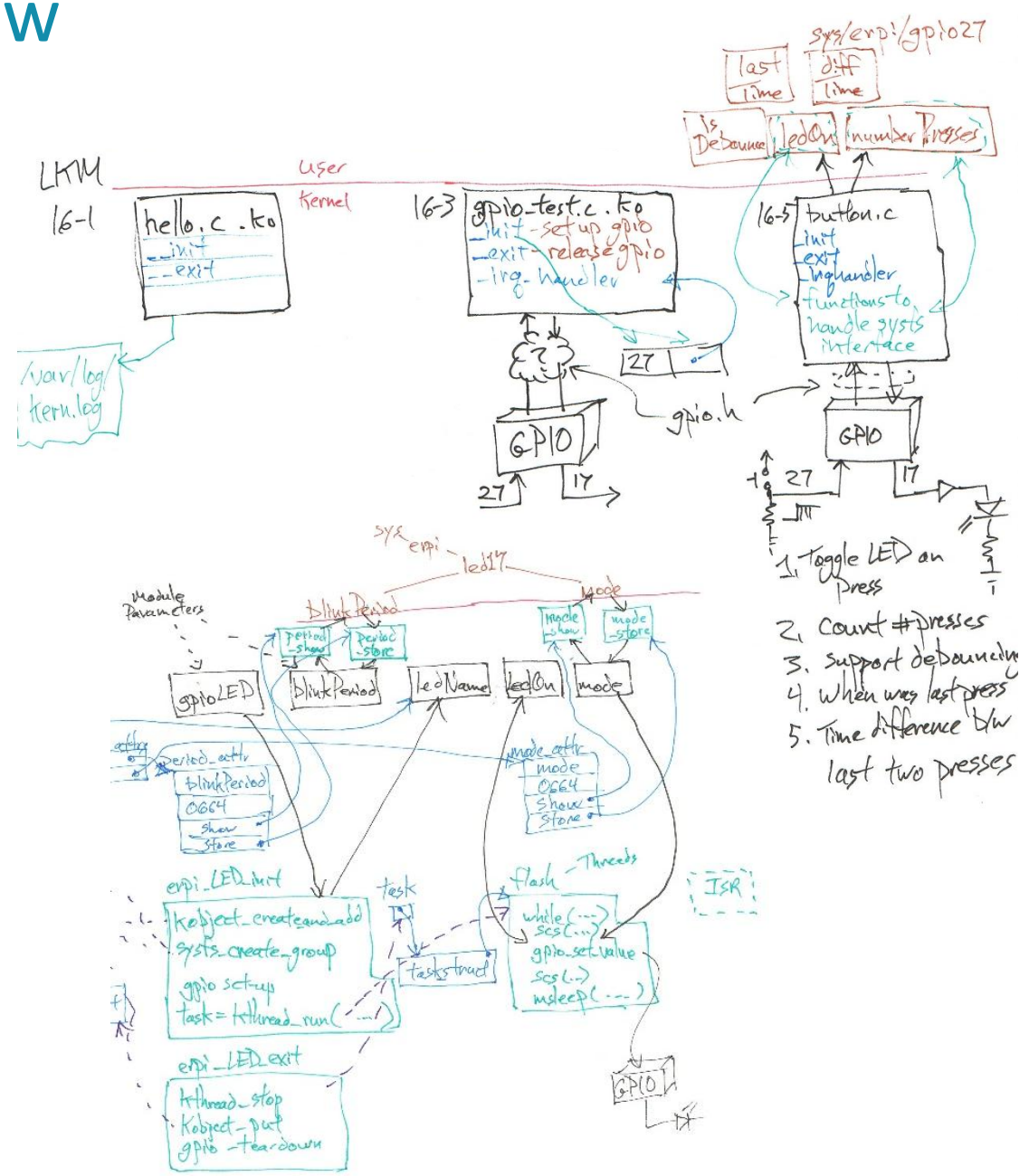
# OSF Example

**Listing 8.2.2: Example device driver**

```
1   #include <linux/cdev.h>
2   #include <linux/errno.h>
3   #include <linux/fs.h>
4   #include <linux/init.h>
5   #include <linux/kernel.h>
6   #include <linux/module.h>
7   #include <linux/uaccess.h>
8
9   MODULE_LICENSE("GPL");
10  MODULE_DESCRIPTION("Example char device driver");
11  MODULE_VERSION("0.42");
12
13  static const char *fortytwo = "*";
14
15  static ssize_t device_file_read(struct file *file_ptr,
16           char __user *user_buffer,
17           size_t count,
18           loff_t *position) {
19    int i = count;
20    while (i--)
21      if( copy_to_user(user_buffer, fortytwo, 1) != 0 )
22        return -EFAULT;
23    return count;
24  }
25
26  static struct file_operations driver_fops = {
27    .owner  = THIS_MODULE,
28    .read   = device_file_read,
29  };
30
31  static int device_file_major_number = 0;
32  static const char device_name[] = "The-Meaning-Of-Life";
33
34  static int register_device(void) {
35    int result = 0;
36    result = register_chrdev(0, device_name, &driver_fops);
37    if( result < 0 ) {
38      printk(KERN_WARNING "The-Meaning-Of-Life: "
39            "unable to register character device, error code %i", result);
40      return result;
41    }
42    device_file_major_number = result;
43    return 0;
44  }
45
46  static void unregister_device(void) {
47    if(device_file_major_number != 0)
48      unregister_chrdev(device_file_major_number, device_name);
49  }
50
51  static int simple_driver_init(void) {
52    int result = register_device();
53    return result;
54  }
55
56  static void simple_driver_exit(void) {
57    unregister_device();
58  }
59
60  module_init(simple_driver_init);
61  module_exit(simple_driver_exit);
```

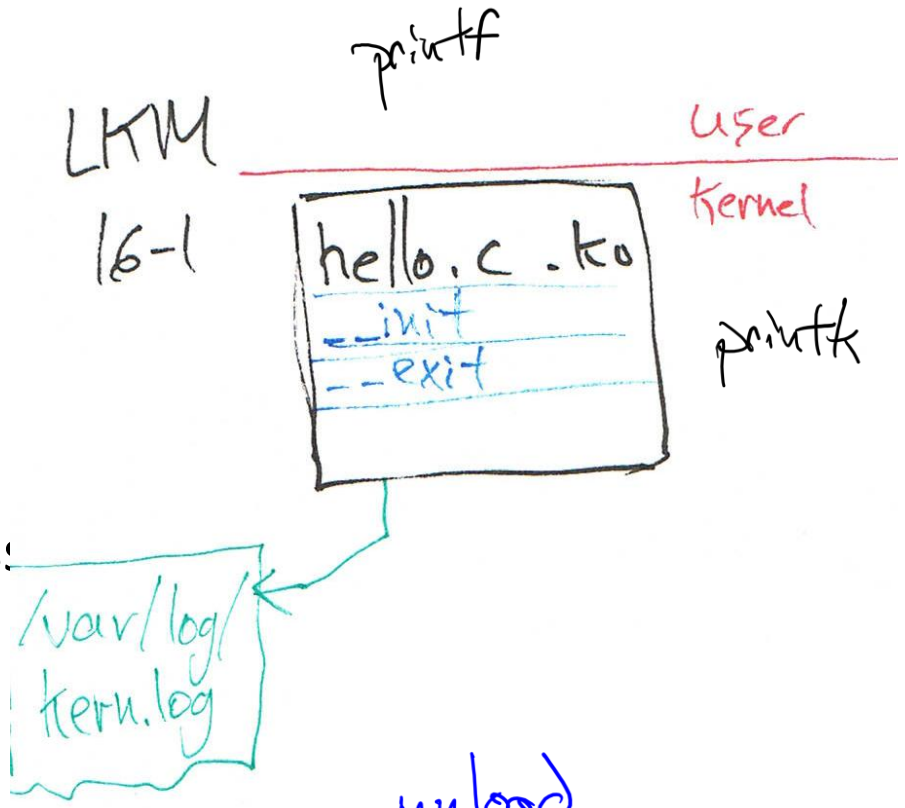
# ERPi Device Driver Module Overview

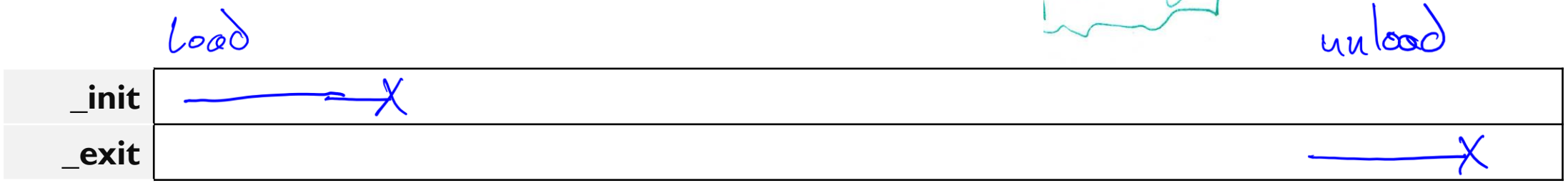
- Example 1: hello (listing 16-1)
  - **Module loading and unloading**
  - **Messages**
- Example 2: gpio_test (listing 16-3)
  - Toggles output (LED) on rising input edge
  - + **Kernel GPIO output for LED, input for switch**
    - + Uses legacy gpio_* interface (deprecated, now should use descriptor-based gpiod_ functions
  - + **Interrupt service routine for switch**
- Example 3: button (listing 16-5)
  - sysfs-enabled GPIO driver: nodes in file system for:
    - LED state, number of switch presses, debouncing information, Time of last press, time between presses
  - + **kobject interface to sysfs**
- Example 4: led (listing 16-6)
  - Independent LED flasher
  - Uses kobject sysfs interface to LED driver (mode, blinkPeriod)
  - + **kernel thread**



# Example 1: hello

- Two functions
  - erpi_hello_init
  - erpi_hello_exit
- Can take “name” parameter
- Logs information to /var/log/kern.log
- No printf, floating-point math, GPIO, user-space acces




- BES/GPIO-In/LKM_ISR/hello/hello.c

# Kernel Objects and Module Initialization

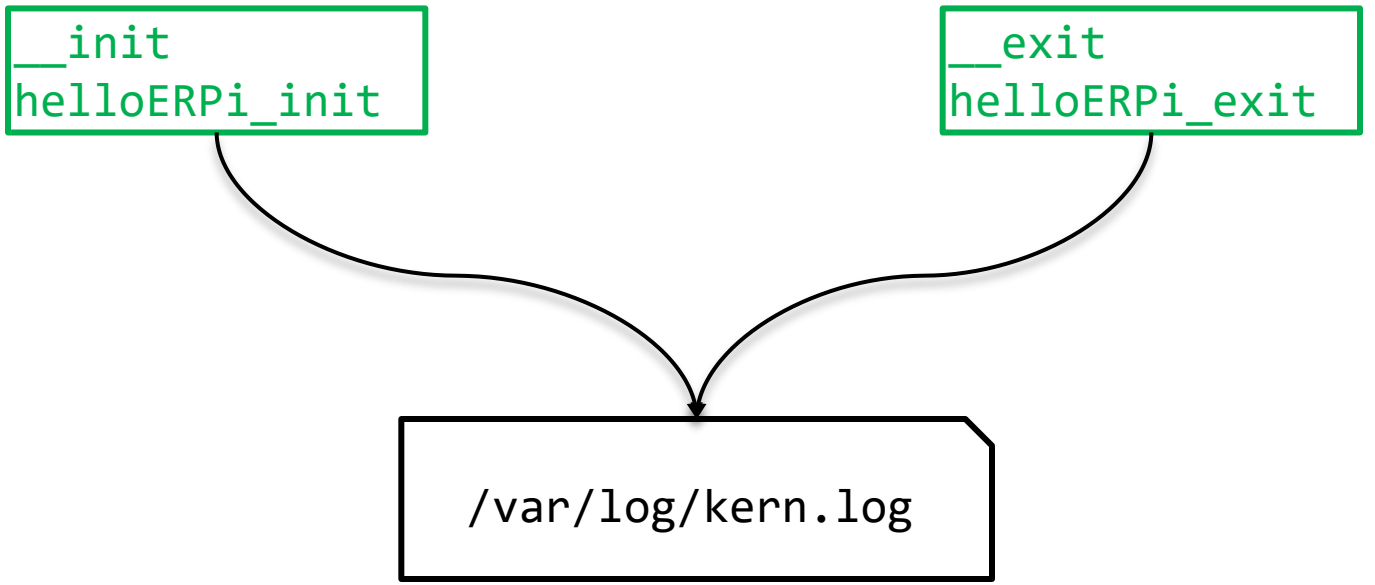
- Module initialization
  - Ask kernel to create and add a kobject
- Header files at /include/linux/(kobject.h, sysfs.h)
- kobject struct fields
  - Name
  - Type of object, affects creation and destruction
  - Sysfs directory entry
  - Parent kobject
  - Set information (if in kset of kobjects)
  - Reference count – used to manage cleanup


```
/** @brief The LKM initialization function */
static int __init erpi_button_init(void){

   // create the kobject sysfs entry at /sys/erpi
   erpi_kobj = kobject_create_and_add("erpi", kernel_kobj->parent);
```

# Example 1: Details

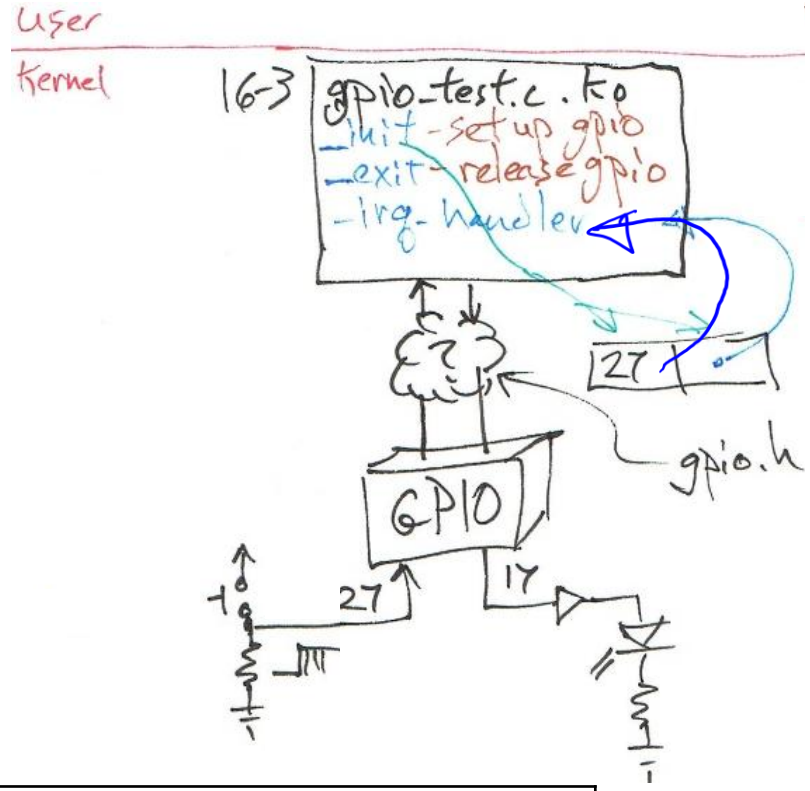
User Space

Kernel Space



# Example 2: gpio_test

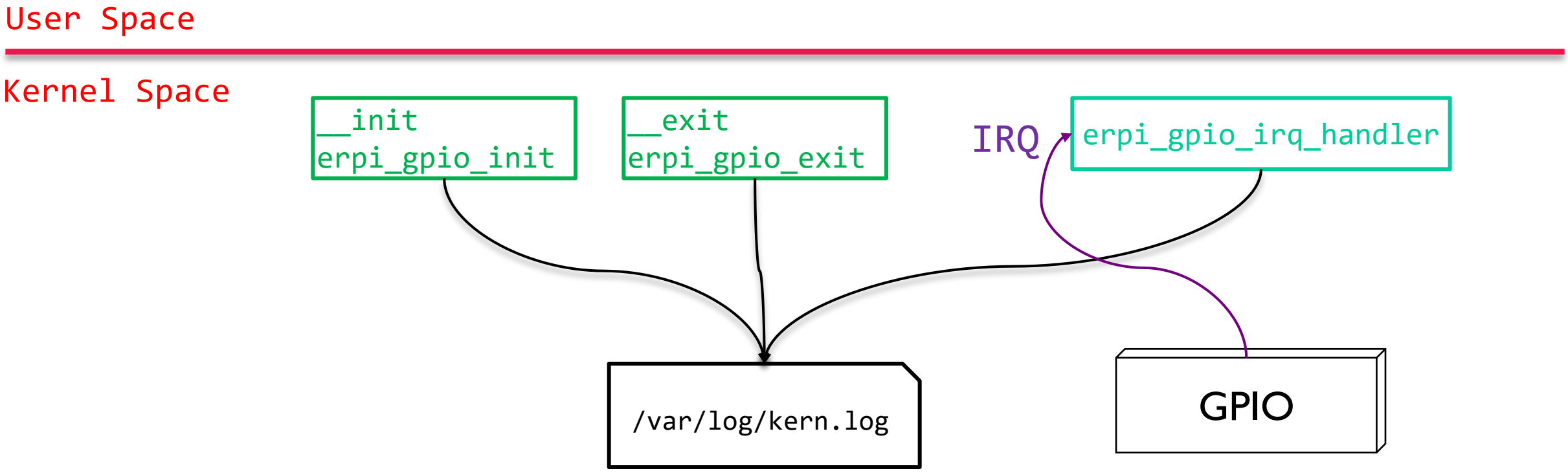
- Changes output (LED) on rising input edge
- Uses kernel-space GPIO interface (linux/gpio.h)
- Three functions
  - _init
  - _exit
  - _irq_handler



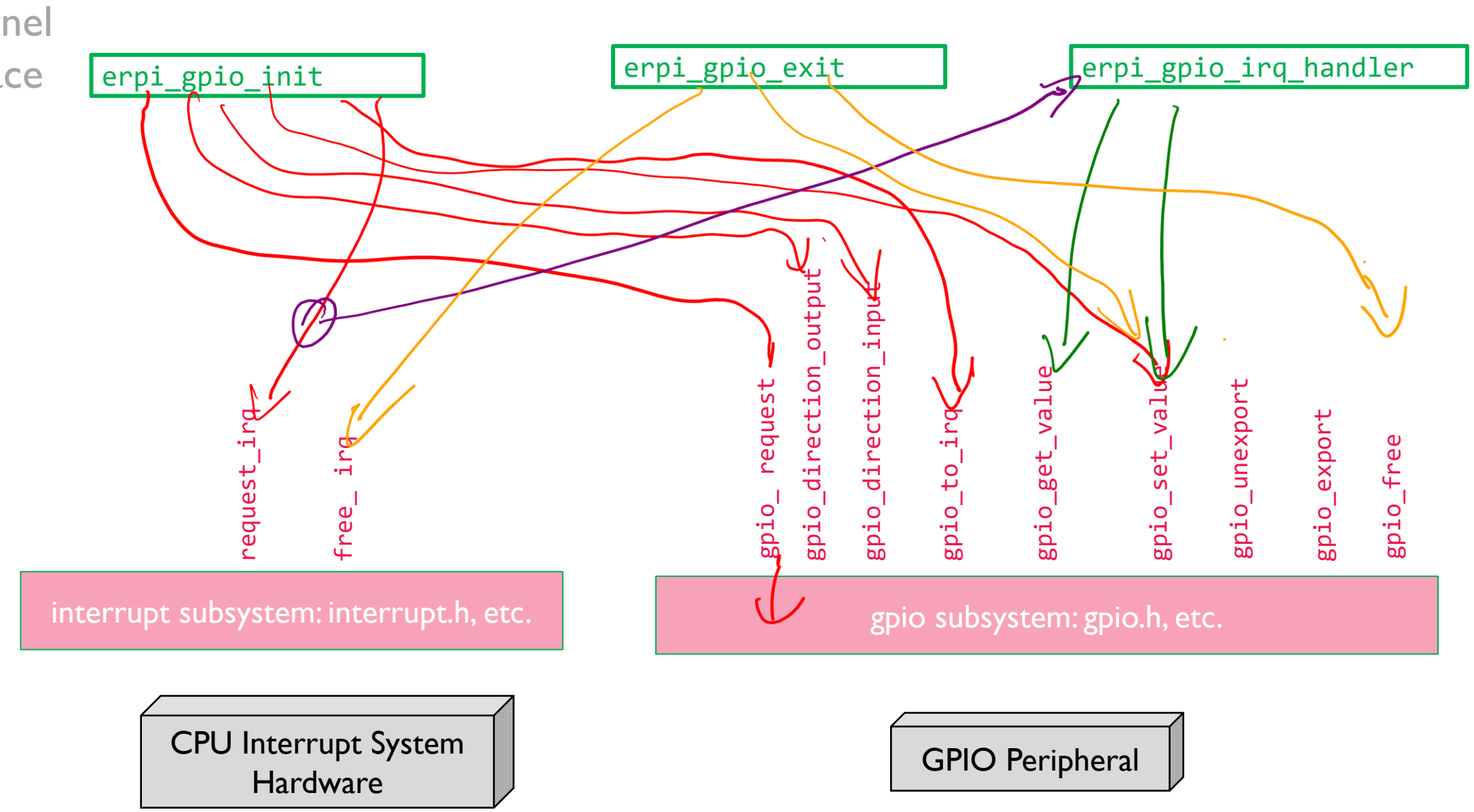


- BES/GPIO-In/LKM_ISR/gpio/gpio_test.c

# Example 2: Details



# Example 2: ERPi

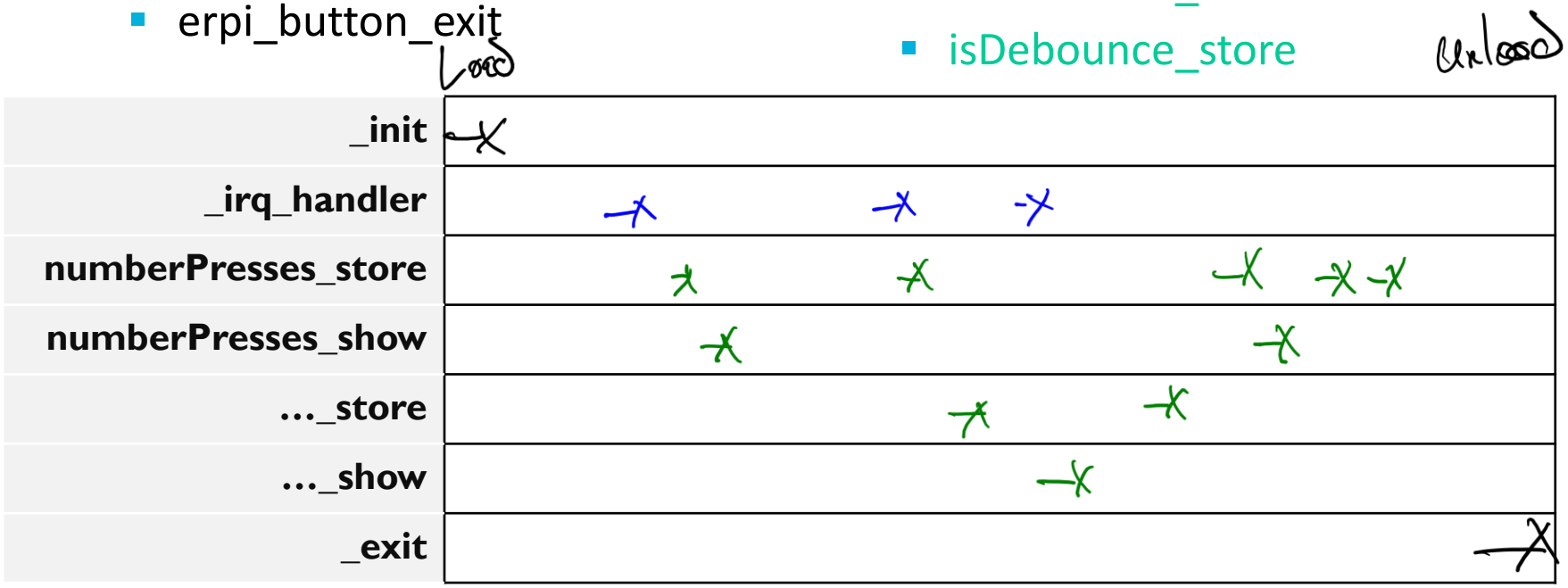


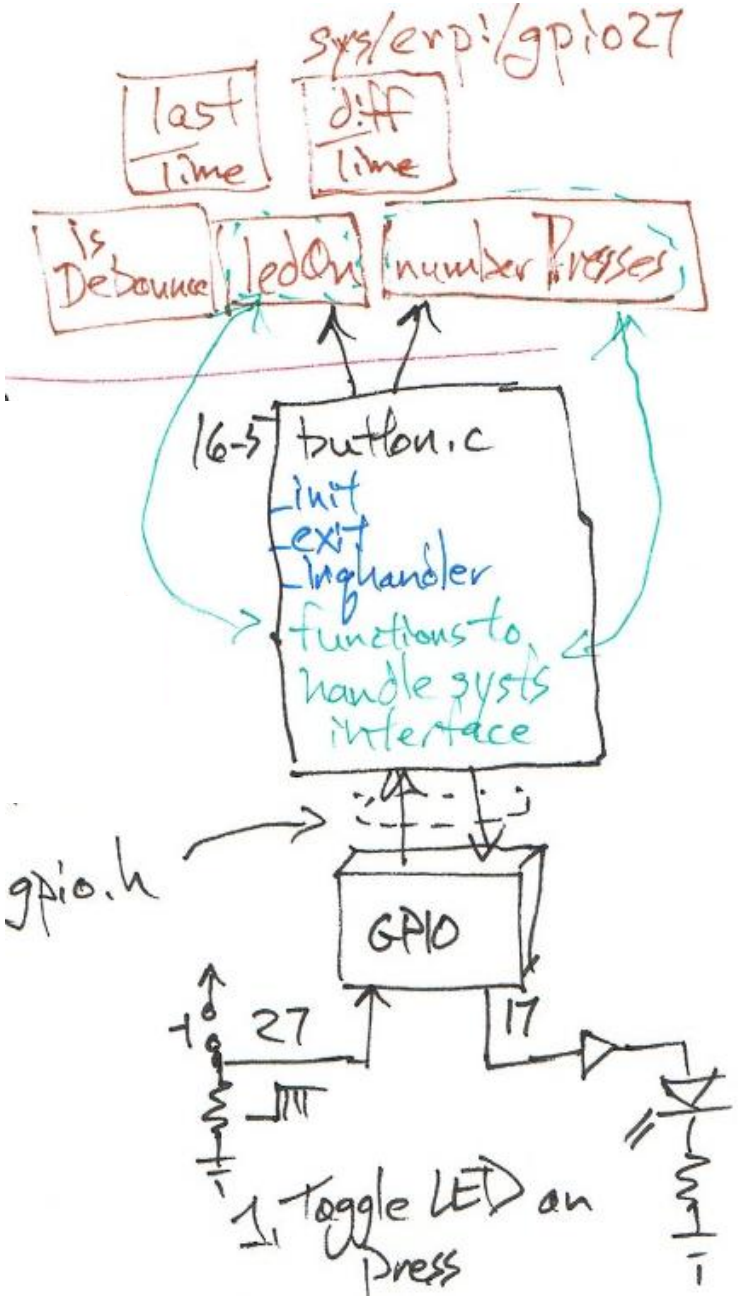
# Example of Module with Descriptor-Based GPIO Access

```
// https://github.com/Johannes4Linux/Linux_Driver_Tutorial/tree/main/03_gpioctrl
#include <linux/module.h>
#include <linux/init.h>
#include <linux/gpio/consumer.h>

static struct gpio_desc *led, *button;

#define IO_LED 21
#define IO_BUTTON 20
#define IO_BASE 512

static int __init my_init(void)
{
   int status;

   led = gpio_to_desc(IO_LED + IO_BASE);
   if (!led) {
      printk("gpioctrl - Error getting pin %d\n", IO_LED);
      return -ENODEV;
   }
   button = gpio_to_desc(IO_BUTTON + IO_BASE);
   if (!button) {
      printk("gpioctrl - Error getting pin %d\n", IO_BUTTON);
      return -ENODEV;
   }
   status = gpiod_direction_output(led, 0);
   if (status) {
      printk("gpioctrl - Error setting pin %d to output\n", IO_LED);
      return status;
   }
   status = gpiod_direction_input(button);
   if (status) {
      printk("gpioctrl - Error setting pin %d to input\n",
      IO_BUTTON);
      return status;
   }
   gpiod_set_value(led, 1);
   printk("gpioctrl - Button is %spressed\n",
      gpiod_get_value(button) ? "" : "not ");
   return 0;
}

static void __exit my_exit(void)
{
   gpiod_set_value(led, 0);
}

module_init(my_init);
module_exit(my_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Johannes 4Linux");
MODULE_DESCRIPTION("An example for using GPIOs without the device
tree");
```

# Example 3: button

- sysfs-enabled GPIO driver toggles LED with debounced switch, monitors switch times
- Uses kobject interface
- Many functions
  - erpi_button_init
  - erpi_button_exit
  - erpi_gpio_irq_handler
  - numberPresses_store
  - numberPresses_show
  - ledOn_show
  - lastTime_show
  - diffTime_show
  - isDebounce_show
  - isDebounce_store

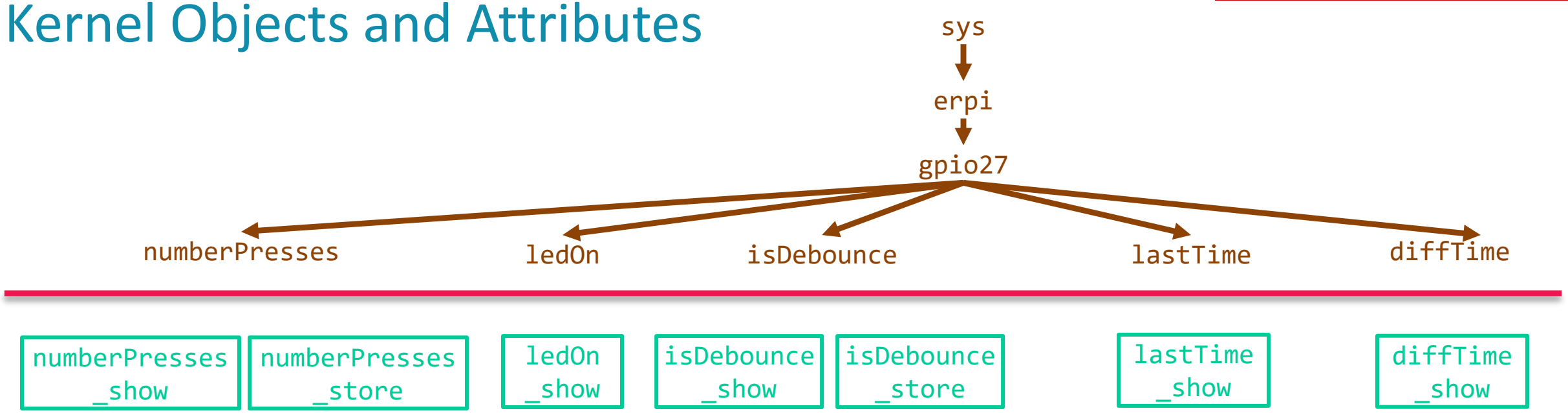





BES/GPIO-In/LKM_ISR/button/button.c

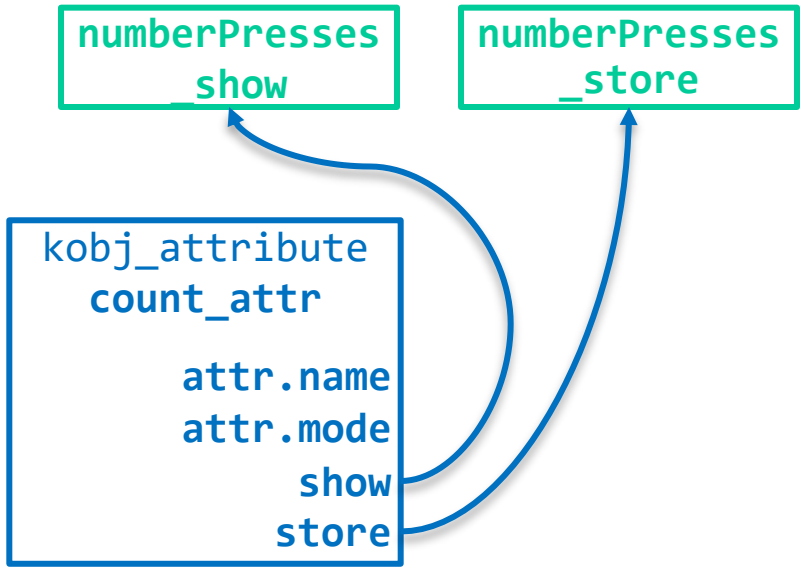
# Kernel Objects and Attributes




- How to make something available through sysfs filesystem?
  - Use kobject abstraction and its interface
  - kobject visible as folder in sysfs: gpio27
  - kobject can have attributes visible as files in that folder: numberPresses, etc
  - Need to link attributes with kobject
  - Need to link read, write functions with each attribute
    - Code to “read” attribute (show function)
    - Code to “write” attribute (store function)

# Attributes

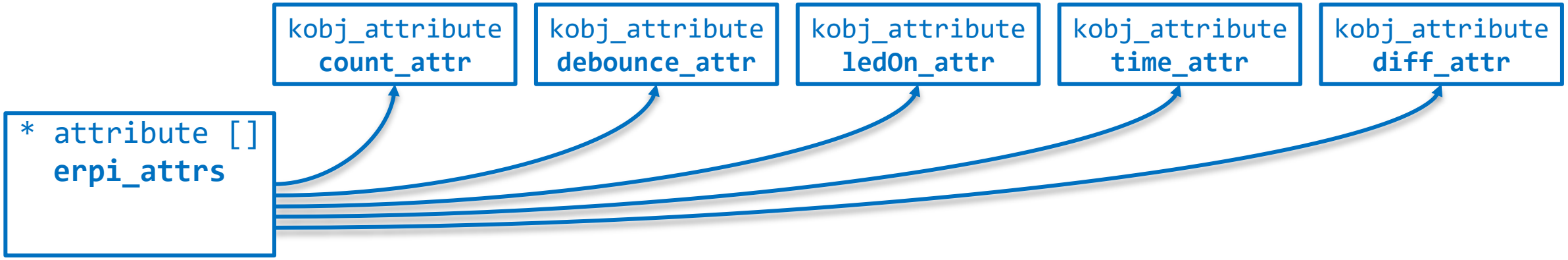
numberPresses




```
/**  Use these helper macros to define the name and access levels of the
 * kobj_attributes. The kobj_attribute has an attribute attr (name and mode),
 * show and store function pointers. The count variable is associated with
 * the numberPresses variable and it is to be exposed with mode 0664 using
 * the numberPresses_show and numberPresses_store functions above
 */
static struct kobj_attribute count_attr = __ATTR(numberPresses, 0664, numberPresses_show, numberPresses_store);
```

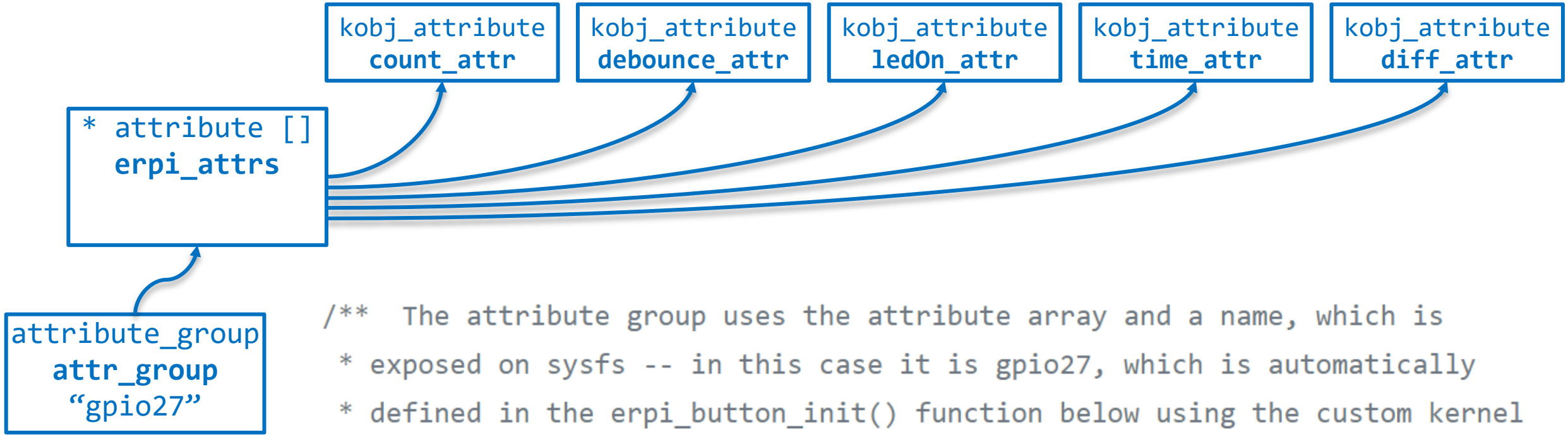
- Attribute show and store functions
  - Arguments: * kobject, * kobj_attribute, character array buffer for data
  - Return value: number of bytes read or written
- Use kobj_attribute macros in sysfs.h to initialize struct
  - __ATTR(): Read/write
  - __ATTR_RO(): Read-only
  - __ATTR_WO(): Write-only

# Gathering Attributes



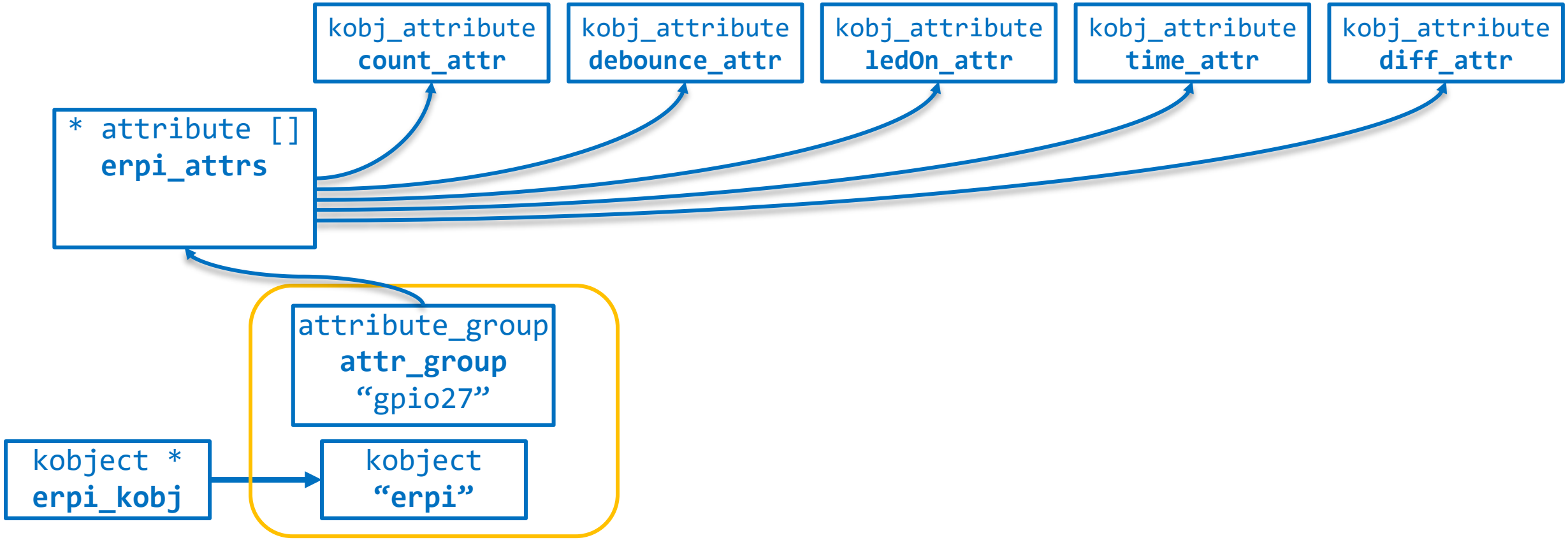

```
static struct attribute *erpi_attrs[] = {
        &count_attr.attr,          // the number of button presses
        &ledon_attr.attr,          // is the LED on or off?
        &time_attr.attr,           // button press time in HH:MM:SS:NNNNNNNNN
        &diff_attr.attr,           // time difference between last two presses
        &debounce_attr.attr,       // is debounce state true or false
        NULL,
};
```

# Attribute Group



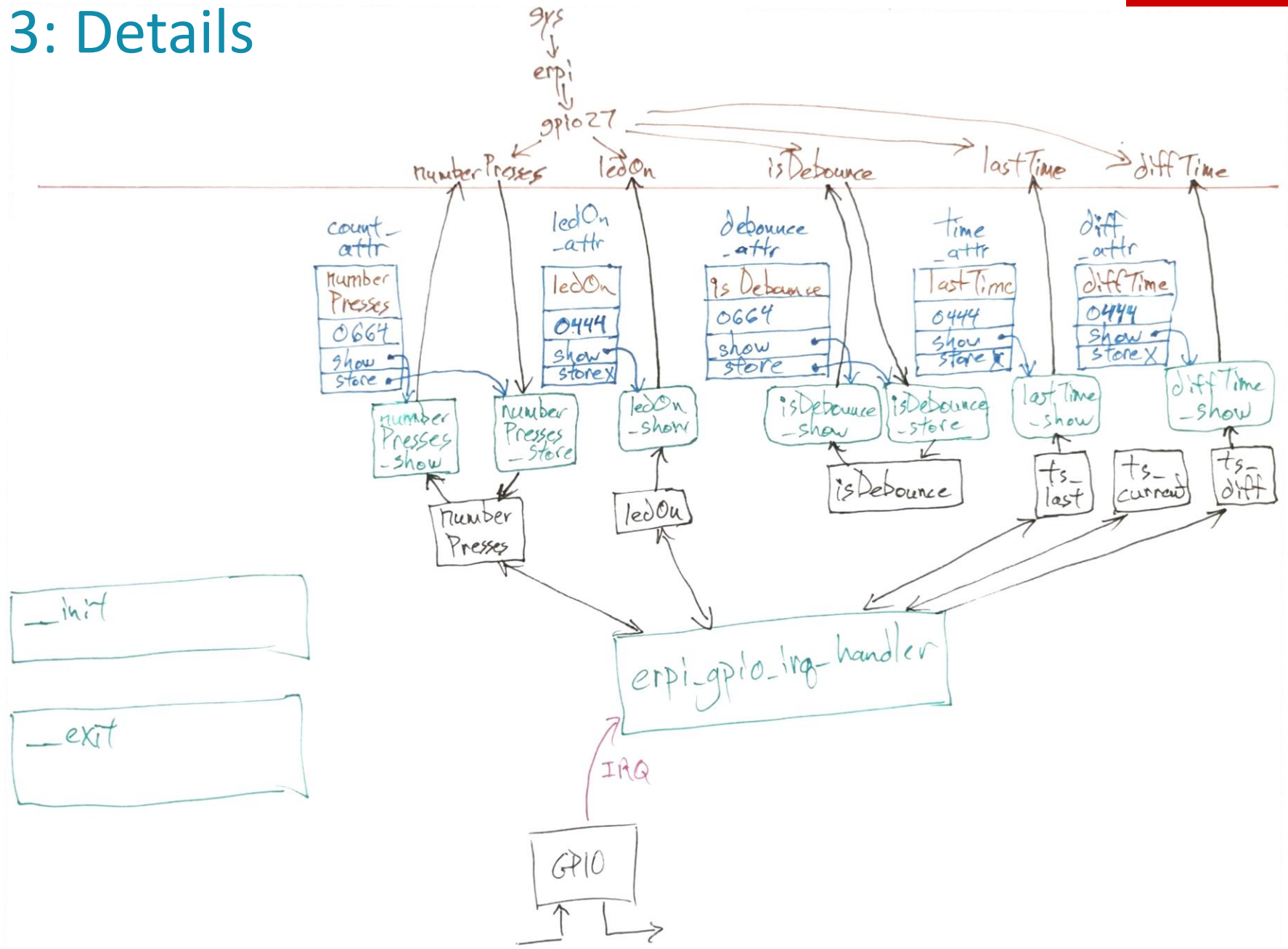

```
/**  The attribute group uses the attribute array and a name, which is
 * exposed on sysfs -- in this case it is gpio27, which is automatically
 * defined in the erpi_button_init() function below using the custom kernel
 * parameter that can be passed when the module is loaded.
 */
static struct attribute_group attr_group = {
        .name  = gpioName,          // the name generated in erpi_button_init()
        .attrs = erpi_attrs,        // the attributes array defined just above
};
```

# Attribute Connections

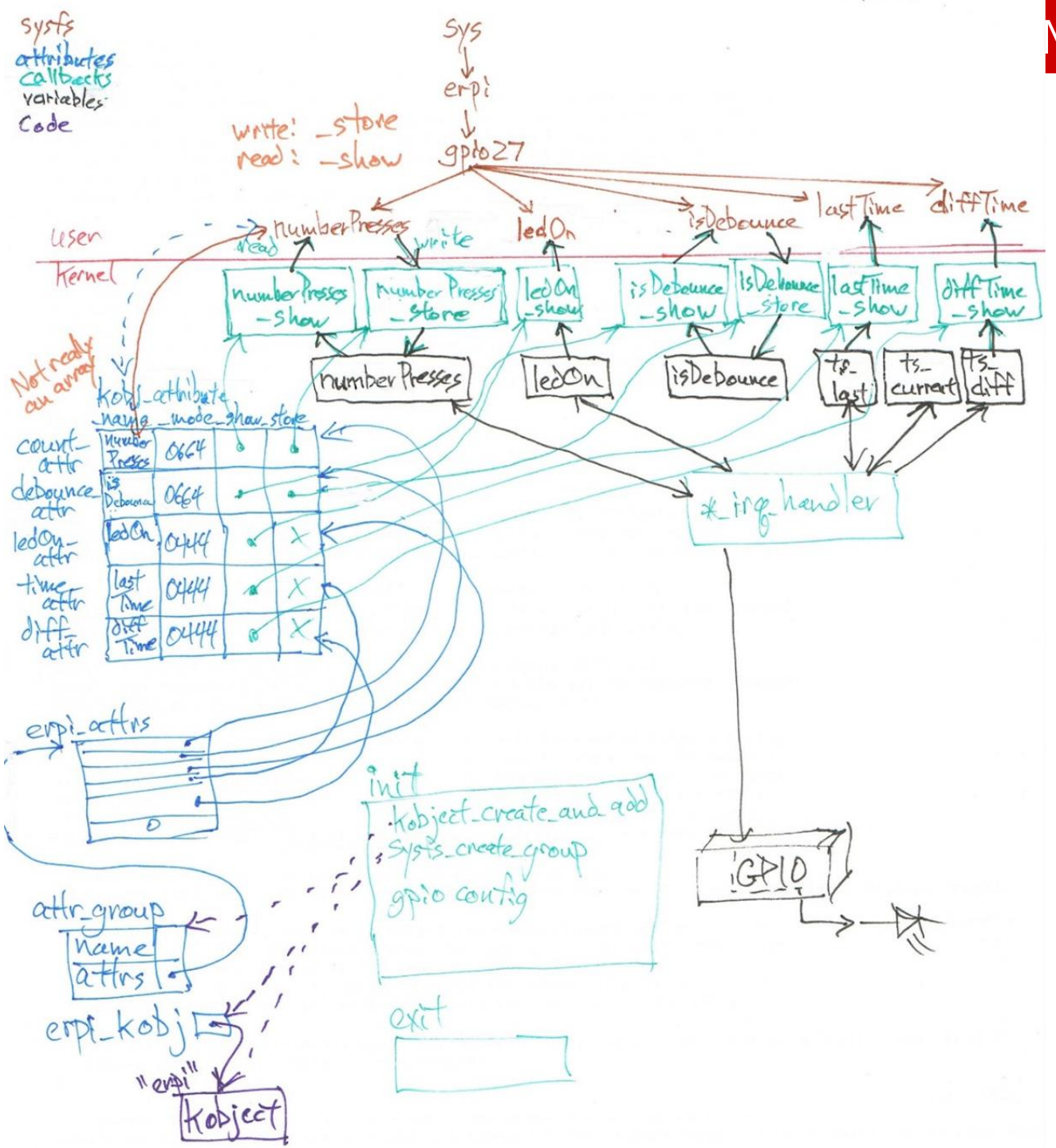



```
// add the attributes to /sys/erpi/ e.g., /sys/erpi/gpio27/numberPresses
result = sysfs_create_group(erpi_kobj, &attr_group);
```

# Example 3: Details

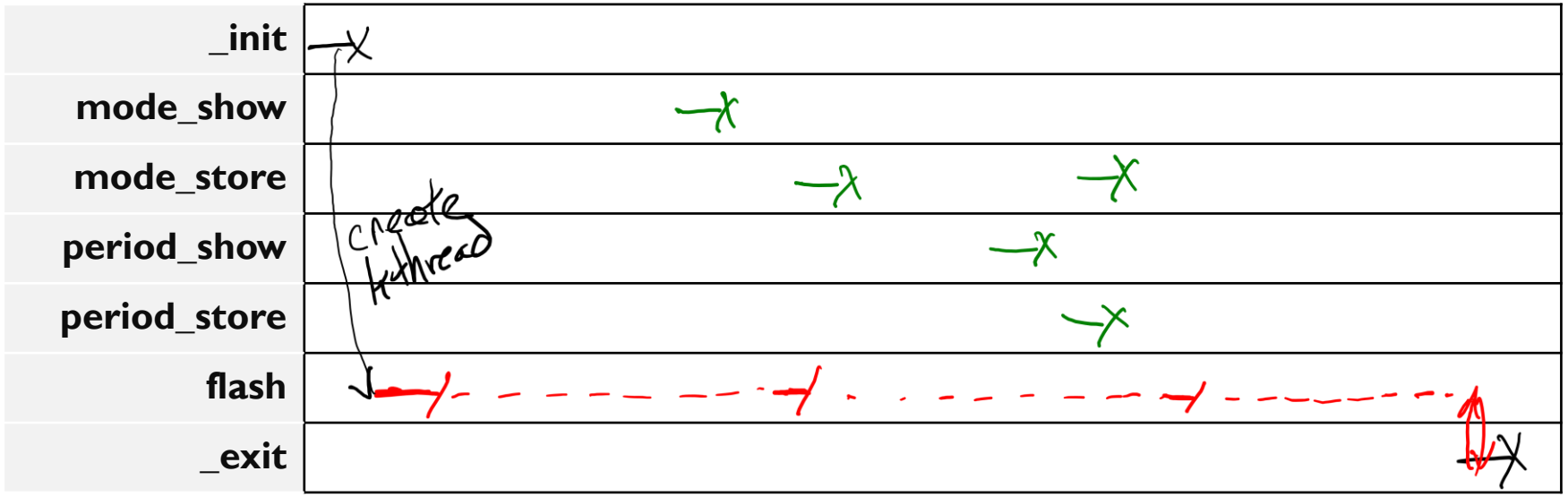


# Example 3: Old Details

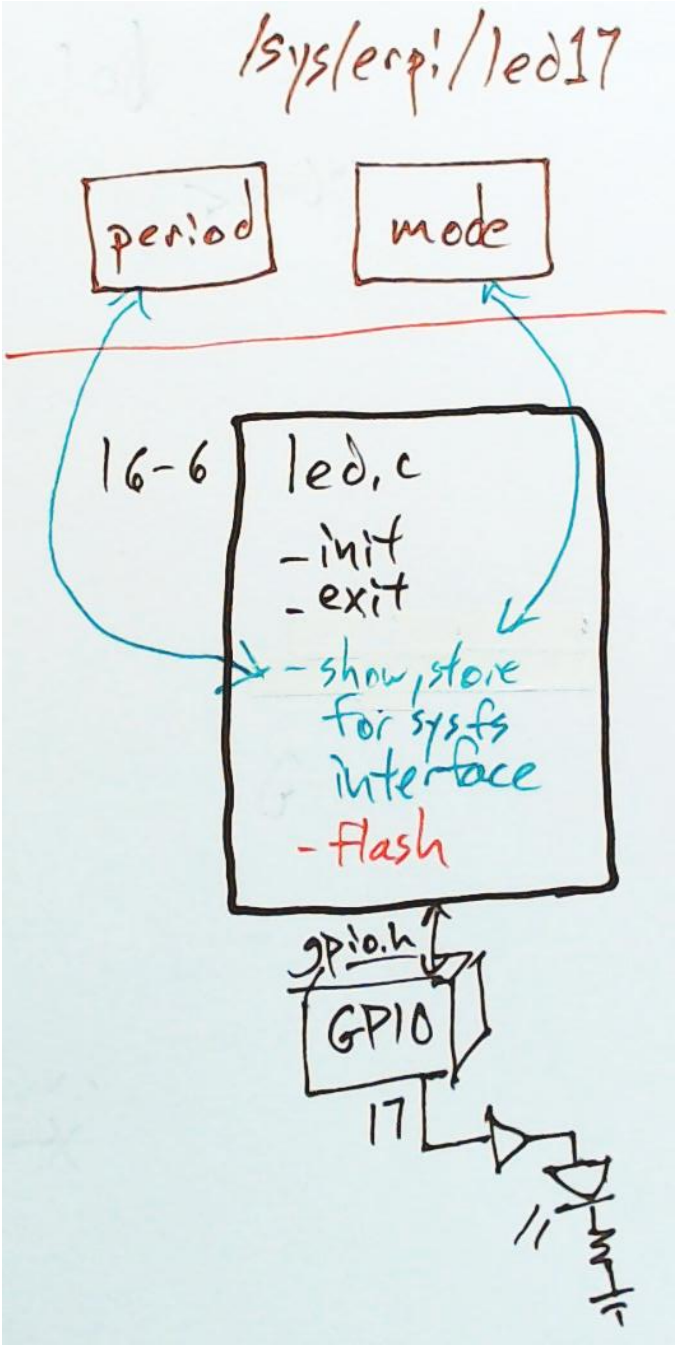


# Example 4: led

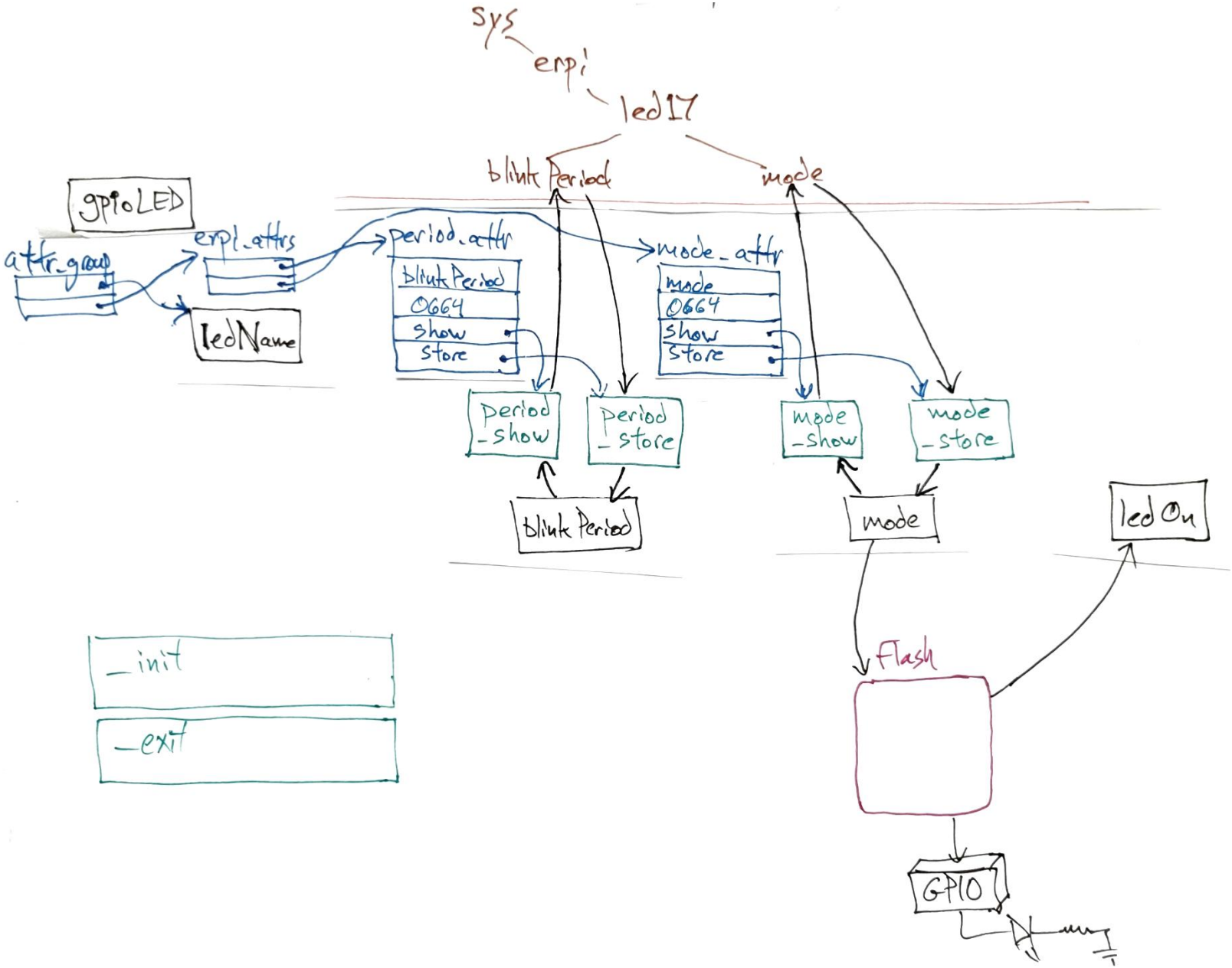
- Uses kernel thread to flash LED based on sysfs parameters
- Functions:
  - erpi_led_init
  - erpi_led_exit
  - mode_show, mode_store
  - period_show, period_store
  - flash (kthread)



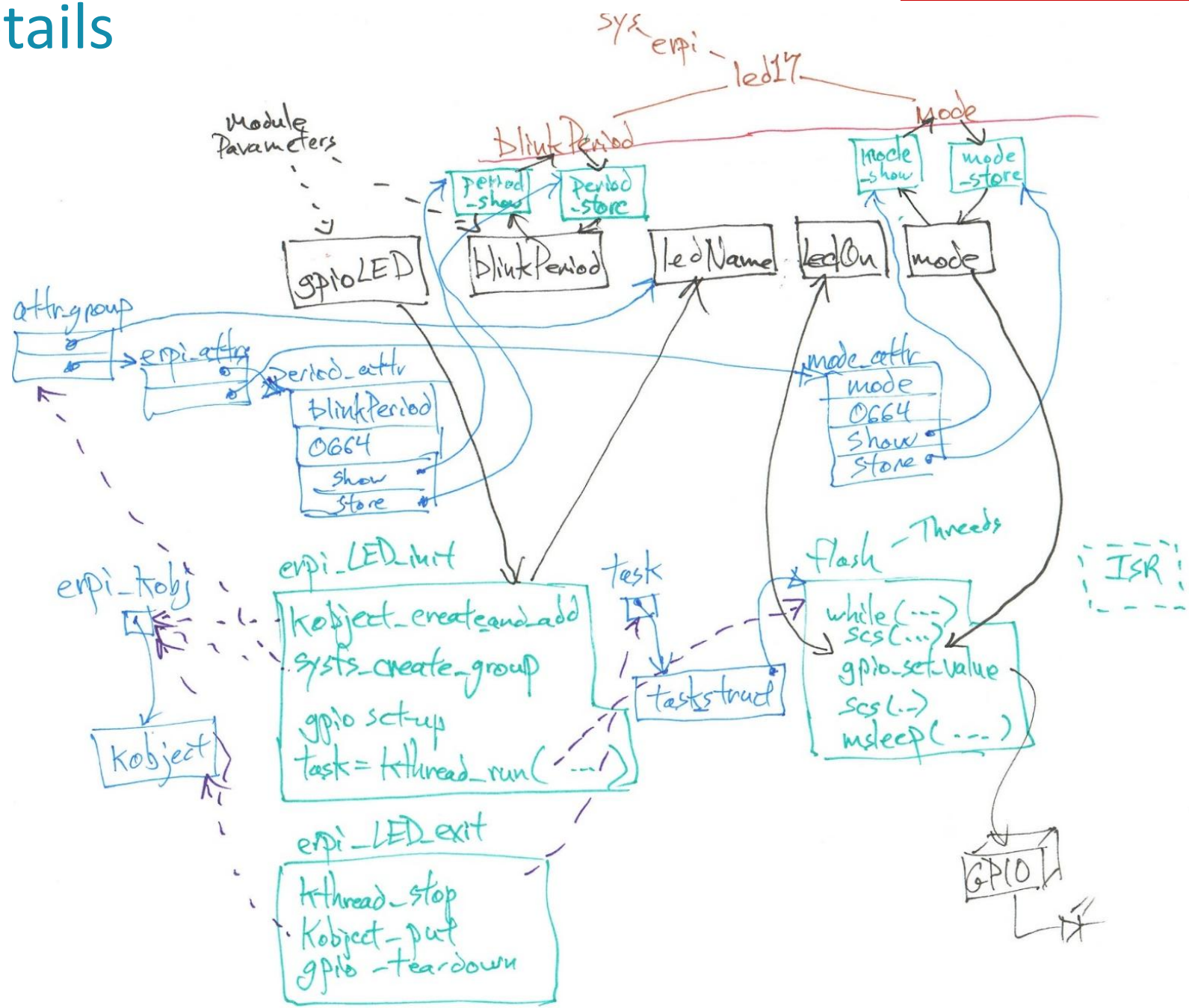

BES/GPIO-In/LKM_ISR/LED/led.c



# Example 4: Details



# Example 4: Old Details



# GPIO Access from the Kernel

# Descriptor-Based GPIO Interface

- Old interface used gpio_* functions, new uses gpiod_* functions
  - *The use of the legacy functions is strongly discouraged, new code should use* ***<linux/gpio/consumer.h>*** *and descriptors exclusively.*
  - *All the functions that work with the descriptor-based GPIO interface are prefixed with* ***gpiod_****. The* ***gpio_*** *prefix is used for the legacy interface. No other function in the kernel should use these prefixes.*
- Kernel modules use gpio consumer interface
  - See <linux/gpio/consumer.h>
- Documentation
  - https://docs.kernel.org/driver-api/gpio/consumer.html
  - Guidelines for GPIOs consumers
  - Obtaining and Disposing GPIOs
    - Via Device Tree
    - Via legacy gpio numbers
  - Using GPIOs
    - Setting Direction
    - Spinlock-Safe GPIO Access
    - Access Multiple GPIOs with a Single Function Call
    - GPIOs mapped to IRQs
  - Interacting with the Legacy GPIO Subsystem