

# Kernel Course: Lecture 17

## Communicating with Hardware (part 2)

---

Sam Protsenko  
<joe.skb7@gmail.com>

May 30, 2020

# Agenda

1. Kernel Driver: Proper Way
2. User Space
3. Assignments
4. Appendixes

## Kernel Driver: Proper Way

---

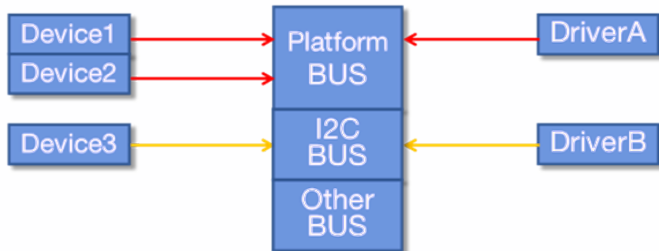
# API Overview

## Device/driver matching

- Driver must be platform-independend
- Device-specific data is obtained from Device Tree
- In real drivers we rarely use just `module_init()`
- Driver **binding** is the process of associating a device with a driver
- Bus drivers (*driver core*) handle the binding  
(as bus driver knows about all devices and drivers on this bus)

## Device/driver matching (cont'd)

- When driver or device is registered, driver core will check their **compatible** strings
- If those strings **match**, driver core will invoke **probe( )** function
- Platform devices should be registered very early during system boot
- Drivers usually register later during booting, or by module loading



## API: miscdevice

- **miscdevice** = Miscellaneous Character Device
- Major number is the same for all misc devices (see `/proc/devices`)
- Easier to implement than regular char dev

---

```
#include <linux/miscdevice.h>

struct miscdevice {
    int minor;
    const char *name;
    const struct file_operations *fops;
    ...
};

int misc_register(struct miscdevice *misc);
void misc_deregister(struct miscdevice *misc);
```

---

## API: New GPIO Kernel API

- Similar set of functions as in “Legacy GPIO API”
- Operates on **struct gpio\_desc** (instead of GPIO number)
- GPIO descriptor is usually obtained from device tree

---

```
#include <linux/gpio/consumer.h>

/* Get descriptor structure from dts */
struct gpio_desc *__must_check devm_gpiod_get(struct device *dev,
                                              const char *con_id,
                                              enum gpiod_flags flags);

int gpiod_to_irq(const struct gpio_desc *desc);
void gpiod_set_value(struct gpio_desc *desc, int value);
int gpiod_get_value(const struct gpio_desc *desc);
int gpiod_set_debounce(struct gpio_desc *desc, unsigned debounce); /* usec */
```

---

For details see [Documentation/driver-api/gpio/consumer.rst](#).



# API: Device Tree

---

```
#include <linux/of.h>

/*
 * - can be obtained from struct device (.of_node field), in probe function
 * - contains matched device properties (can be read using functions below)
 */
struct device_node;

/*
 * - to store "compatible" strings table
 * - provide it to struct device (.of_match_table field)
 */
struct of_device_id;

/* Device Tree access functions */
int of_property_read_u32(const struct device_node *np, const char *propname,
                        u32 *out_value);
bool of_property_read_bool(const struct device_node *np, const char *propname);
```

---

For details see [Documentation/devicetree/usage-model.txt](#).

## API: Managed Device Resources

- **devres** is linked list of memory areas associated with a **struct device**
- Each devres entry is associated with a release function
- All devres entries are released on driver detach
- On release, the associated release function is invoked and then the devres entry is freed

---

```
void *devm_kzalloc(struct device *dev, size_t size, gfp_t gfp);
struct gpio_desc *devm_gpiod_get(struct device *dev, const char *con_id,
                                enum gpiod_flags flags);
int devm_request_irq(struct device *dev, unsigned int irq,
                    irq_handler_t handler, unsigned long irqflags,
                    const char *devname, void *dev_id);
```

---

For details see [Documentation/driver-model/devres.txt](#).

# Two Flavours of Power Management

## Runtime power management (Runtime PM)

Turn off (stop clock or remove power) hardware components that aren't going to be used in the near future, **transparently** from the user space's viewpoint.

## System sleep

Knowing in advance that **the whole system** is not going to be used in the near future, turn off **everything** (possibly **by force**) except for the RAM chips.

- We are only going to use *System sleep* PM API
- For details see “System Sleep vs Runtime Power Management” slides (by Rafael J. Wysocki)

# API: Power Management

---

```
#include <linux/pm.h>

/* Populate it and set to struct driver (.pm field) */
struct dev_pm_ops {
    int (*suspend)(struct device *dev);
    int (*resume)(struct device *dev);
    int (*freeze)(struct device *dev);
    int (*thaw)(struct device *dev);
    int (*poweroff)(struct device *dev);
    int (*restore)(struct device *dev);
};

SIMPLE_DEV_PM_OPS(name, suspend_fn, resume_fn); /* returns struct dev_pm_ops */

int device_init_wakeup(struct device *dev, bool enable);
bool device_may_wakeup(struct device *dev);

int enable_irq_wake(unsigned int irq);
int disable_irq_wake(unsigned int irq);
```

---

## Attempt #3

Platform driver + char dev + new GPIO API + device tree

**PRODUCTION READY!**

# Decisions Overview

- Previous attempts are fine for experiments and low-level platform drivers (e.g. GPIO controller driver)
- For our device we have to use either:
  - Special frameworks for device classes (like IIO, RTC, etc)
  - or generic kernel devices (char devs, block devs, etc)
- We don't have any special frameworks in kernel for our device class
- So we will use:
  - Character device (misc): for user space interface
  - Platform device: for integration in device tree and PM ops

## Driver Features

- User space interface (file operations):
  - read: button state (0 or 1); blocking/non-blocking read
  - write: LED state (0 or 1)
  - poll: wait for button to be pressed
  - ioctl: **SETLED, GETLED, KERN\_CONTROL**
- Proper integration with Device Tree
- Power Management: button can wake up BBB from suspend
- Button is handled via interrupt
- Kernel can implement device's logic, or delegate it to user space
- Handle multi-threading scenarios
- Expose ioctl constants to user space

## Third Attempt: Device Tree

Listing 1: am335x-boneblack.dts

```
1 &am33xx_pinmux {
2     hw3_pins: hw3_pins {
3         pinctrl-single,pins = <
4             AM33XX_IOPAD(0x82c, PIN_INPUT | MUX_MODE7)      /* gpmc_ad11.gpio0_27 */
5             AM33XX_IOPAD(0x83c, PIN_OUTPUT | MUX_MODE7)     /* gpmc_ad15.gpio1_15 */
6         >;
7     };
8 };
9
10 / {
11     hw3 {
12         compatible = "globallogic,hw3";
13         button-gpios = <&gpio0 27 GPIO_ACTIVE_LOW>;
14         led-gpios = <&gpio1 15 GPIO_ACTIVE_HIGH>;
15         debounce-delay-ms = <5>;
16         wakeup-source;
17         pinctrl-names = "default";
18         pinctrl-0 = <&hw3_pins>;
19     };
20 };
```



### Listing 2: hw3.c

---

```
1 // SPDX-License-Identifier: GPL-2.0
2 /*
3  * Driver for handling externally connected button and LED.
4  */
5
6 #include <linux/module.h>
7 #include <linux/platform_device.h>
8 #include <linux/gpio/consumer.h>
9 #include <linux/interrupt.h>
10 #include <linux/of.h>
11 #include <linux/miscdevice.h>
12 #include <linux/fs.h>
13 #include <linux/spinlock.h>
14 #include <linux/poll.h>
15 #include <linux/sched/signal.h>
16 #include <linux/wait.h>
17 #include <linux/pm.h>
18
19 #include "hw3.h"
20
```

## Third Attempt: Code (2/14)

```
21 #define DRIVER_NAME      "hw3"
22 #define WRITE_BUF_LEN    10
23 #define READ_BUF_LEN     2      /* one character and \0 */
24
25 struct hw3 {
26     struct miscdevice mdev;
27     struct gpio_desc *btn_gpio;
28     struct gpio_desc *led_gpio;
29     int btn_irq;
30     int led_on;
31     int btn_on;
32     int control;              /* kernel controls LED switching? */
33     spinlock_t lock;
34     wait_queue_head_t wait;
35     bool data_ready;         /* new data ready to read */
36 };
37
38 static inline struct hw3 *to_hw3_struct(struct file *file)
39 {
40     struct miscdevice *miscdev = file->private_data;
41
42     return container_of(miscdev, struct hw3, mdev);
43 }
```

## Third Attempt: Code (3/14)

```
44
45 static ssize_t hw3_read(struct file *file, char __user *buf, size_t count,
46                         loff_t *ppos)
47 {
48     struct hw3 *hw3 = to_hw3_struct(file);
49     unsigned long flags;
50     unsigned long copy_len;
51     ssize_t ret;
52     const char *val;
53
54     spin_lock_irqsave(&hw3->lock, flags);
55     while (!hw3->data_ready) {
56         spin_unlock_irqrestore(&hw3->lock, flags);
57         if (file->f_flags & O_NONBLOCK)
58             return -EAGAIN;
59         if (wait_event_interruptible(hw3->wait, hw3->data_ready))
60             return -ERESTARTSYS;
61         spin_lock_irqsave(&hw3->lock, flags);
62     }
63
64     val = hw3->btn_on ? "1" : "0";
65     hw3->data_ready = false;
66     spin_unlock_irqrestore(&hw3->lock, flags);
```

## Third Attempt: Code (4/14)

```
67
68     /* Handle case when user requested 1 byte read */
69     copy_len = min(count, (size_t)READ_BUF_LEN);
70
71     /* Do not advance ppos, do not use simple_read_from_buffer() */
72     if (copy_to_user(buf, val, copy_len))
73         ret = -EFAULT;
74     else
75         ret = copy_len;
76
77     return ret;
78 }
79
80 static ssize_t hw3_write(struct file *file, const char __user *buf,
81                         size_t count, loff_t *ppos)
82 {
83     struct hw3 *hw3 = to_hw3_struct(file);
84     unsigned long flags;
85     char kbuf[WRITE_BUF_LEN];
86     long val;
87     int res;
88
89     /* Do not advance ppos, do not use simple_write_to_buffer() */
```

## Third Attempt: Code (5/14)

```
90     if (copy_from_user(kbuf, buf, WRITE_BUF_LEN))
91         return -EFAULT;
92
93     kbuf[1] = '\\0'; /* get rid of possible \n from "echo" command */
94     res = kstrtoul(kbuf, 0, &val);
95     if (res)
96         return -EINVAL;
97     val = !!val;
98
99     spin_lock_irqsave(&hw3->lock, flags);
100    if (hw3->led_on != val) {
101        hw3->led_on = val;
102        gpio_set_value(hw3->led_gpio, hw3->led_on);
103    }
104    spin_unlock_irqrestore(&hw3->lock, flags);
105
106    return count;
107 }
108
109 static __poll_t hw3_poll(struct file *file, poll_table *wait)
110 {
111     struct hw3 *hw3 = to_hw3_struct(file);
112     unsigned long flags;
```

## Third Attempt: Code (6/14)

```
113     __poll_t mask = 0;
114
115     poll_wait(file, &hw3->wait, wait);
116
117     spin_lock_irqsave(&hw3->lock, flags);
118     if (hw3->data_ready)
119         mask = EPOLLIN | EPOLLRDNORM;
120     spin_unlock_irqrestore(&hw3->lock, flags);
121
122     return mask;
123 }
124
125 static long hw3_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
126 {
127     struct hw3 *hw3 = to_hw3_struct(file);
128     unsigned long flags;
129     int val;
130
131     switch (cmd) {
132     case HW3IOC_SETLED:
133         if (get_user(val, (int __user *)arg))
134             return -EFAULT;
135     }
```

## Third Attempt: Code (7/14)

```
136         spin_lock_irqsave(&hw3->lock, flags);
137         hw3->led_on = !!val;
138         gpiod_set_value(hw3->led_gpio, hw3->led_on);
139         spin_unlock_irqrestore(&hw3->lock, flags);
140
141         /* Fall through */
142     case HW3IOC_GETLED:
143         spin_lock_irqsave(&hw3->lock, flags);
144         val = hw3->led_on;
145         spin_unlock_irqrestore(&hw3->lock, flags);
146
147         return put_user(val, (int __user *)arg);
148     case HW3IOC_KERN_CONTROL:
149         if (get_user(val, (int __user *)arg))
150             return -EFAULT;
151
152         spin_lock_irqsave(&hw3->lock, flags);
153         hw3->control = !!val;
154         spin_unlock_irqrestore(&hw3->lock, flags);
155
156         break;
157     default:
158         return -ENOTTY;
```

## Third Attempt: Code (8/14)

```
159     }
160
161     return 0;
162 }
163
164 static const struct file_operations hw3_fops = {
165     .owner          = THIS_MODULE,
166     .read           = hw3_read,
167     .write          = hw3_write,
168     .poll           = hw3_poll,
169     .unlocked_ioctl = hw3_ioctl,
170     .llseek         = no_llseek,
171 };
172
173 static irqreturn_t hw3_btn_isr(int irq, void *data)
174 {
175     struct hw3 *hw3 = data;
176     unsigned long flags;
177
178     spin_lock_irqsave(&hw3->lock, flags);
179     hw3->data_ready = true;
180     hw3->btn_on = gpiod_get_value(hw3->btn_gpio);
181     if (hw3->btn_on && hw3->control) {
```



## Third Attempt: Code (9/14)

```
182             hw3->led_on ^= 0x1;
183             gpiod_set_value(hw3->led_gpio, hw3->led_on);
184     }
185     spin_unlock_irqrestore(&hw3->lock, flags);
186
187     wake_up_interruptible(&hw3->wait);
188
189     return IRQ_HANDLED;
190 }
191
192 static int hw3_probe(struct platform_device *pdev)
193 {
194     struct device *dev = &pdev->dev;
195     struct device_node *node = pdev->dev.of_node;
196     struct hw3 *hw3;
197     u32 debounce;
198     bool wakeup_source;
199     int ret;
200
201     hw3 = devm_kzalloc(&pdev->dev, sizeof(*hw3), GFP_KERNEL);
202     if (!hw3)
203         return -ENOMEM;
204 }
```

## Third Attempt: Code (10/14)

```
205     hw3->control = 1;
206
207     /* "button-gpios" in dts */
208     hw3->btn_gpio = devm_gpiod_get(dev, "button", GPIOD_IN);
209     if (IS_ERR(hw3->btn_gpio))
210         return PTR_ERR(hw3->btn_gpio);
211
212     /* "led-gpios" in dts */
213     hw3->led_gpio = devm_gpiod_get(dev, "led", GPIOD_OUT_LOW);
214     if (IS_ERR(hw3->led_gpio))
215         return PTR_ERR(hw3->led_gpio);
216
217     hw3->btn_irq = gpiod_to_irq(hw3->btn_gpio);
218     if (hw3->btn_irq < 0)
219         return hw3->btn_irq;
220
221     ret = of_property_read_u32(node, "debounce-delay-ms", &debounce);
222     if (ret == 0) {
223         ret = gpiod_set_debounce(hw3->btn_gpio, debounce * 1000);
224         if (ret < 0)
225             dev_warn(dev, "No HW support for debouncing\n");
226     }
227
```

## Third Attempt: Code (11/14)

```
228     wakeup_source = of_property_read_bool(node, "wakeup-source");
229
230     ret = devm_request_irq(dev, hw3->btn_irq, hw3_btn_isr,
231                          IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING,
232                          dev_name(dev), hw3);
233     if (ret < 0)
234         return ret;
235
236     device_init_wakeup(dev, wakeup_source);
237     platform_set_drvdata(pdev, hw3);
238     spin_lock_init(&hw3->lock);
239     init_waitqueue_head(&hw3->wait);
240
241     hw3->mdev.minor      = MISC_DYNAMIC_MINOR;
242     hw3->mdev.name       = DRIVER_NAME;
243     hw3->mdev.fops       = &hw3_fops;
244     hw3->mdev.parent     = dev;
245     ret = misc_register(&hw3->mdev);
246     if (ret)
247         return ret;
248
249     gpiod_set_value(hw3->led_gpio, 0);
250
```

## Third Attempt: Code (12/14)

```
251         return 0;
252     }
253
254     static int hw3_remove(struct platform_device *pdev)
255     {
256         struct hw3 *hw3 = platform_get_drvdata(pdev);
257
258         misc_deregister(&hw3->mdev);
259         return 0;
260     }
261
262     #ifdef CONFIG_PM_SLEEP
263     static int hw3_suspend(struct device *dev)
264     {
265         struct hw3 *hw3 = dev_get_drvdata(dev);
266
267         if (device_may_wakeup(dev))
268             enable_irq_wake(hw3->btn_irq);
269
270         return 0;
271     }
272
273     static int hw3_resume(struct device *dev)
```

## Third Attempt: Code (13/14)

```
274 {
275     struct hw3 *hw3 = dev_get_drvdata(dev);
276
277     if (device_may_wakeup(dev))
278         disable_irq_wake(hw3->btn_irq);
279
280     return 0;
281 }
282 #endif /* CONFIG_PM_SLEEP */
283
284 static SIMPLE_DEV_PM_OPS(hw3_pm, hw3_suspend, hw3_resume);
285
286 static const struct of_device_id hw3_of_match[] = {
287     { .compatible = "globallogic,hw3" },
288     { .compatible = "globallogic,hw4" },
289     { .compatible = "globallogic,hw5" },
290     {}, /* sentinel */
291 };
292 MODULE_DEVICE_TABLE(of, hw3_of_match);
293
294 static struct platform_driver hw3_driver = {
295     .probe = hw3_probe,
296     .remove = hw3_remove,
```

## Third Attempt: Code (14/14)

```
297     .driver = {
298         .name = DRIVER_NAME,
299         .pm = &hw3_pm,
300         .of_match_table = hw3_of_match,
301     },
302 };
303
304 module_platform_driver(hw3_driver);
305
306 MODULE_ALIAS("platform:hw3");
307 MODULE_AUTHOR("Sam Protsenko <joe.sk7@gmail.com>");
308 MODULE_DESCRIPTION("Test module 3");
309 MODULE_LICENSE("GPL");
```

---

## Third Attempt: Header

### Listing 3: hw3.h

---

```
1  /* SPDX-License-Identifier: GPL-2.0 */
2  #ifndef _UAPI_LINUX_HW3_H
3  #define _UAPI_LINUX_HW3_H
4
5  #include <linux/ioctl.h>
6  #include <linux/types.h>
7
8  /* Chosen to be unique w.r.t. Documentation/ioctl/ioctl-number */
9  #define HW3_IOCTL_MAGIC      0x91
10
11 #define HW3IOC_SETLED        _IOWR(HW3_IOCTL_MAGIC, 0, int)
12 #define HW3IOC_GETLED        _IOR(HW3_IOCTL_MAGIC, 1, int)
13 #define HW3IOC_KERN_CONTROL  _IOW(HW3_IOCTL_MAGIC, 2, int)
14
15 #endif /* _UAPI_LINUX_HW3_H */
```

---

Take five



## User Space

---

## Listing 4: hw3-app.c

---

```
1 #include <sys/types.h>
2 #include <sys/ioctl.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #include <poll.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <string.h>
9
10 #include "hw3.h"
11
12 #define DEV_FILE      "/dev/hw3"
13 #define TIMEOUT      10000      /* msec */
14 #define BUF_SIZE     4096
15 #define COUNT        20
16
17 int main(int argc, char *argv[])
18 {
19     int fd;
20     int ret = EXIT_FAILURE;
```

## User Space Application (2/4)

```
21     int i;
22     struct pollfd pfd;
23     int ready;
24     int led_on;
25     char buf[BUF_SIZE];
26
27     fd = open(DEV_FILE, O_RDWR | O_NONBLOCK);
28     if (fd == -1) {
29         perror("Failed to open dev file");
30         return EXIT_FAILURE;
31     }
32
33     i = 0;
34     if (ioctl(fd, HW3IOC_KERN_CONTROL, &i) == -1) {
35         perror("Error occurred on ioctl");
36         goto end;
37     }
38
39     ioctl(fd, HW3IOC_GETLED, &led_on);
40
41     pfd.fd = fd;
42     pfd.events = POLLIN;
43     for (i = 0; i < COUNT; ++i) {
```

## User Space Application (3/4)

```
44     printf("Waiting for button interrupt [%d/%d]...\n", i+1, COUNT);
45     ready = poll(&pfd, 1, TIMEOUT);
46     if (ready < 0) {
47         perror("poll() error");
48         goto end;
49     } else if (ready == 0) {
50         fprintf(stderr, "poll() timeout\n");
51         goto end;
52     }
53     if (!(pfd.revents & POLLIN)) {
54         fprintf(stderr, "poll() returned with no POLLIN\n");
55         goto end;
56     }
57
58     ret = read(fd, buf, BUF_SIZE);
59     if (ret == -1)
60         perror("read() error");
61     else
62         printf("read: %s\n", buf);
63
64     /* Handle "release button" event */
65     if (buf[0] == '0') {
66         led_on ^= 0x1;
```

## User Space Application (4/4)

```
67             //ioctl(fd, HW3IOC_SETLED, &led_on);
68             sprintf(buf, "%d", led_on);
69             write(fd, buf, 2);
70         }
71     }
72
73     i = 1;
74     ioctl(fd, HW3IOC_KERN_CONTROL, &i);
75     ret = EXIT_SUCCESS;
76
77 end:
78     close(fd);
79     return ret;
80 }
```

---

## Listing 5: Makefile

---

```
1  ifneq ($(KERNELRELEASE),) # kbuild part of makefile
2
3  CFLAGS_hw3.o := -DDEBUG
4  obj-m := hw3.o
5
6  else # normal makefile
7
8  KDIR ?= /lib/modules/$(shell uname -r)/build
9
10 default: module app
11 module:
12     $(MAKE) -C $(KDIR) M=$(PWD) C=1 modules
13 clean:
14     $(MAKE) -C $(KDIR) M=$(PWD) C=1 clean
15     rm -f hw3-app
16 app: CC = $(CROSS_COMPILE)gcc
17 app: CFLAGS = -O2 -Wall
18 app:
19     $(CC) $(CFLAGS) hw3-app.c -o hw3-app
20
21 .PHONY: module clean app
22
23 endif
```

# Building Everything (On Host)

Setup environment:

---

```
$ export PATH=/opt/gcc-arm-8.3-2019.03-x86_64-arm-linux-gnueabihf/bin:$PATH
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ export ARCH=arm
$ export KDIR=~/repos/linux-stable
```

---

Build new dtb:

---

```
$ cd $KDIR
$ make am335x-boneblack.dtb
$ cp arch/arm/boot/dts/am335x-boneblack.dtb $TFTP_DIR
$ cd -
```

---

Build module and app:

---

```
$ cd $MODULE_DIR
$ make
$ cp hw3.ko hw3-app $NFS_DIR/root
```

---

## Testing Driver From Bash (On Target)

---

```
# insmod hw3.ko           # Load module
# echo 0 > /dev/hw3       # set LED off
# echo 1 > /dev/hw3       # set LED on
# cat /dev/hw3            # poll button

010010...                # press Ctrl-C to interrupt

# rmmod hw3.ko           # unload module
```

---

Notice that when driver is in control, LED will toggle on button **press**.



## Testing Driver Using App (On Target)

---

```
# insmod hw3.ko
# ./hw3-app

Waiting for button interrupt [1/20]...
read: 1
Waiting for button interrupt [2/20]...
read: 0
...

# rmmod hw3.ko
```

---

Notice that when app is in control, LED will toggle on button **release**.

# Testing Power Management (On Target)

---

```
# insmod hw3.ko
# echo mem > /sys/power/state      # issue suspend

PM: suspend entry (deep)
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.001 seconds) done.
OOM killer disabled.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
Suspending console(s) (use no_console_suspend to debug)

=== PRESS OUR BUTTON FOR RESUME ===

Disabling non-boot CPUs ...
pm33xx pm33xx: PM: Successfully put all powerdomains to target state
OOM killer enabled.
Restarting tasks ... done.
PM: suspend exit

# rmmod hw3.ko
```

---

# Kernel Introspection (On Target)

---

```
# cat /proc/interrupts | grep hw3
63: 18 44e07000.gpio 27 Edge hw3

# cat /sys/kernel/debug/gpio
gpiochip0: GPIOs 0-31, parent: platform/44e07000.gpio, gpio-0-31:
  gpio-27 ( |button ) in hi IRQ
gpiochip1: GPIOs 32-63, parent: platform/4804c000.gpio, gpio-32-63:
  gpio-47 ( |led ) out hi

# find /sys/kernel/debug/pinctrl/ -exec grep -Hn hw3 {} \;
...

# ls -l /sys/firmware/devicetree/base/hw3/
button-gpios
compatible
debounce-delay-ms
led-gpios
wakeup-source
...

# cat /sys/firmware/devicetree/base/hw3/debounce-delay-ms | hexdump
```

---

# Assignments

---

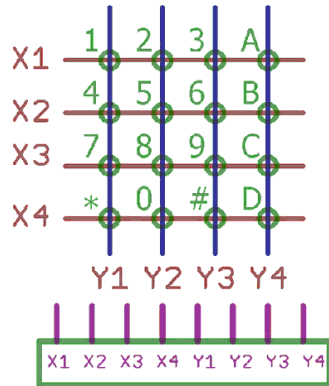
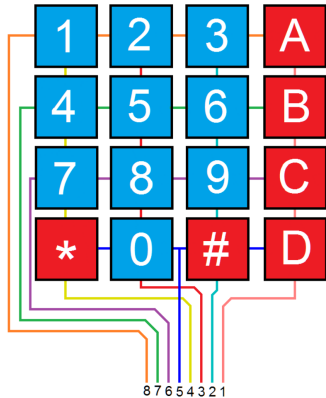
# Assignment 1

- Assemble LED + button device on the breadboard
- Build **hw3.ko**, **hw3-app** and modified **am335x-boneblack.dtb**
- Boot kernel with new **dtb** and load **hw3.ko**
- Test it with Bash commands
- Test it with **hw3-app**
- Inspect **hw3.ko** driver using kernel introspection capabilities
- Make sure you are using NFS for development
- Make sure you are able to jump between both kernel functions and your driver code functions

## Assignment 2

- Connect matrix keypad to BBB (directly or using breadboard)
- Implement driver for detecting pressed buttons
- Report detected buttons to kernel log (**dmesg**)
- Use work queue for scanning columns
- Obtain scan interval from device tree definition
- Perform reading rows on interrupt
- Configure debouncing for rows lines
- Use platform driver and device tree

## Assignment 2 (cont'd)



## Assignment 2 (cont'd)

GPIO line	Pin name	BBB pin
gpio0_26	gpmc_ad10	P8.14
gpio0_27	gpmc_ad11	P8.17
gpio1_12	gpmc_ad12	P8.12
gpio1_13	gpmc_ad13	P8.11
gpio1_14	gpmc_ad14	P8.16
gpio1_15	gpmc_ad15	P8.15
gpio1_17	gpmc_a1	P9.23
gpio1_29	gpmc_csn0	P8.26



## Assignment 2 (cont'd)

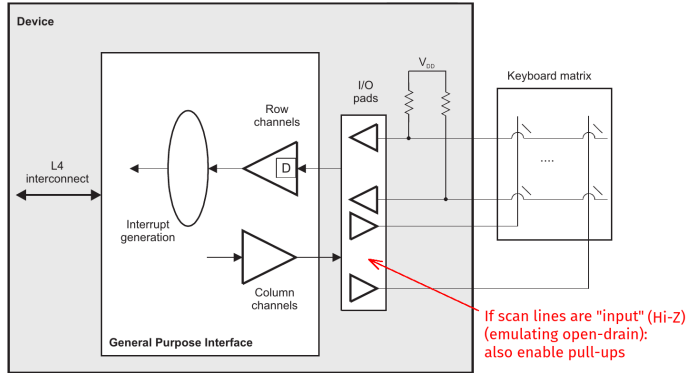


Figure 1: Using internal pull-ups

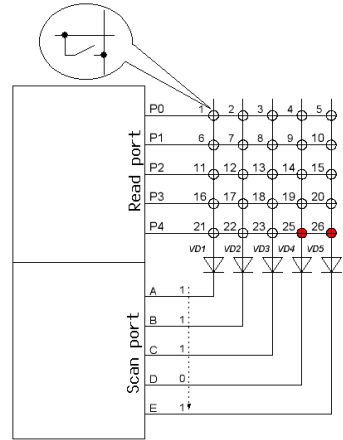


Figure 2: Short circuit protection (external)

## Assignment 2 (cont'd)

### Initialization:

- Pull-up all lines (“scan” and “read”)
- Set all scan and read lines into input
- Configure debouncing on read lines

### Polling:

- Set all scan lines into input (Hi-Z)
- Set one scan line into “0”
- Repeat for next scan line

### Scanning (on interrupt):

- Read the state of all read lines
- Detect which button was pressed

## Assignment 3 (advanced)

- Find existing driver for matrix keypad in kernel
- Find device tree bindings documentation for it
- Use this driver instead of your own, make it work
- Read and understand that driver's code (especially `input_dev` API)

Thank you!

## Appendixes

---

## Appendix A: Power Management on BBB

## Appendix A: Power Management on BBB

- Cortex-M3 core is used as a power supervisor
- Make sure your kernel has **CONFIG\_AMX3\_PM** option
- Obtain firmware for Cortex-M3 and copy it to kernel dir:

---

```
$ git clone git://git.ti.com/processor-firmware/ti-amx3-cm3-pm-firmware.git
$ cd ti-amx3-cm3-pm-firmware
$ git checkout ti2018.05
$ cp bin/am335x-pm-firmware.elf ~/repos/linux-stable/firmware
```

---

## Appendix A: Power Management on BBB (cont'd)

- Add next options to `bbb.cfg` fragment and build the kernel:

---

```
# --- PM (see sprac74a.pdf) ---  
# Embed firmware in kernel  
CONFIG_EXTRA_FIRMWARE="am335x-pm-firmware.elf"  
CONFIG_EXTRA_FIRMWARE_DIR="firmware"  
# AMx3 Power Config Options  
CONFIG_MAILBOX=y  
CONFIG_OMAP2PLUS_MBOX=y  
CONFIG_REMOTEPROC=y  
CONFIG_WKUP_M3_RPROC=y  
CONFIG_WKUP_M3_IPC=y  
CONFIG_SOC_TI=y  
CONFIG_TI_EMIF_SRAM=y  
CONFIG_AMX3_PM=y  
# RTC  
CONFIG_RTC_DRV_OMAP=y
```

---



## Appendix A: Power Management on BBB (cont'd)

- Kernel will upload the firmware to Cortex-M3
- Now it's possible to issue PM operations (suspend/resume)
- Don't try to load the firmware from user space (using `CONFIG_FW_LOADER_USER_HELPER*`), mdev doesn't support it
- For details see:

<http://www.ti.com/lit/an/sprac74a/sprac74a.pdf>

## Appendix B: Device Tree Overlays

## Appendix B: Device Tree Overlays

- Instead of messing with `am335x-boneblack.dtb`, it would be nice to load Device Tree definitions for external devices incrementally
- Device Tree Overlays (`.dtbo`) to the rescue!
- **Bad news:** `CONFIG_OF_CONFIGFS` is still not merged in kernel, so we can't merge overlays in kernel (via ConfigFS)
- **Good news:** we still can merge overlays into `dtb` in U-Boot
- See `fdt apply` command in U-Boot shell

## Appendix B: Device Tree Overlays (cont'd) (1/2)

Listing 6: hw3.dtso

---

```
1 /dts-v1/;
2 /plugin/;
3
4 #include <dt-bindings/gpio/gpio.h>
5 #include <dt-bindings/pinctrl/am33xx.h>
6
7 / {
8     compatible = "ti,beaglebone", "ti,beaglebone-black";
9     part_number = "GLOBALLOGIC-HW3";
10    version = "A1";
11
12    fragment@0 {
13        target = <&am33xx_pinmux>;
14        __overlay__ {
15            hw3_pins: hw3_pins {
16                pinctrl-single,pins = <
17                    /* gpmc_ad11.gpio0_27 */
18                    AM33XX_IOPAD(0x82c, PIN_INPUT | MUX_MODE7)
19                    /* gpmc_ad15.gpio1_15 */
20                    AM33XX_IOPAD(0x83c, PIN_OUTPUT | MUX_MODE7)
```

## Appendix B: Device Tree Overlays (cont'd) (2/2)

```
21         >;
22     };
23 };
24 };
25
26 fragment@1 {
27     target-path = "/";
28     __overlay__ {
29         hw3 {
30             compatible = "globallogic,hw3";
31             button-gpios = <&gpio0 27 GPIO_ACTIVE_LOW>;
32             led-gpios = <&gpio1 15 GPIO_ACTIVE_HIGH>;
33             debounce-delay-ms = <5>;
34             wakeup-source;
35             pinctrl-names = "default";
36             pinctrl-0 = <&hw3_pins>;
37         };
38     };
39 };
40 };
```

---

## Appendix B: Device Tree Overlays (cont'd)

### Commands to build .dtso → .dtbo

---

```
in=hw3.dtso
gen=hw3_gen.dtso
out=hw3.dtbo
kernel_dir=~/repos/linux-stable
gcc_flags="-E -P -x assembler-with-cpp -I$kernel_dir/include"
dtc_flags="-W no-unit_address_vs_reg -I dts -O dtb -b 0 -@"

rm -f $out
gcc $gcc_flags -o $gen $in
dtc $dtc_flags -o $out $gen
rm -f $gen
```

---