Kernel Course: Lecture 16

Communicating with Hardware (part 1)

Sam Protsenko

November 7, 2019

GlobalLogic

Agenda

1. Architecture Design
2. Hardware Overview
3. Kernel Driver: Naïve Approach
4. Assignments

Architecture Design

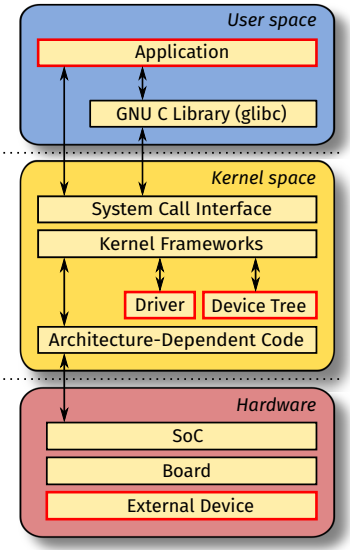
Objectives

- Learn how to communicate with hardware
  (Hardware = SoC + Board + External)
- Grasp the whole development chain: **Hardware → Kernel → User space**
- Learn how to develop **upstreamable** and **production ready** driver
- Structure previously obtained knowledge:
  - Get deeper understanding
  - Learn missing parts
- Prepare base for further development

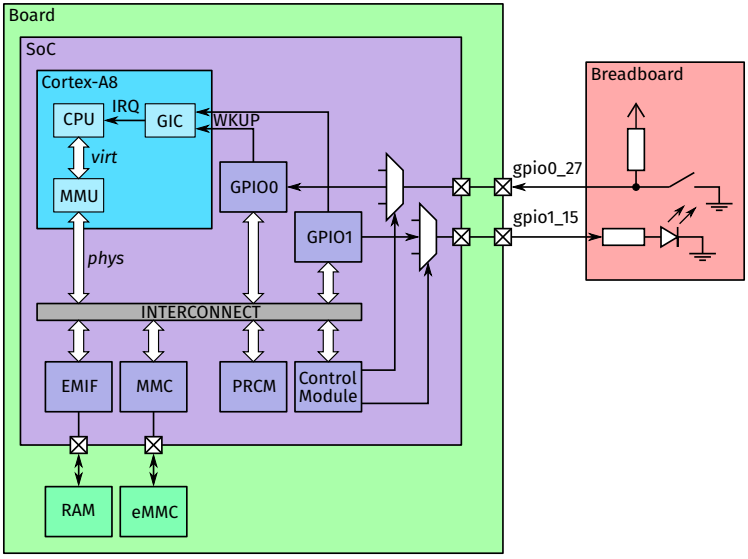
Design Requirements

- Hardware:
  - Prototyping on breadboard
  - External LED and button
  - Make it accessible via GPIOs
  - Prepare HW info for development
- Kernel:
  - Device Tree (definition in dts; obtain in module)
  - Control the LED and button (GPIO, interrupt)
  - User-space interface (char dev, file ops)
  - Power management
- User space:
  - Test driver via `echo` and `cat`
  - Develop application for testing driver

Software Archictecture



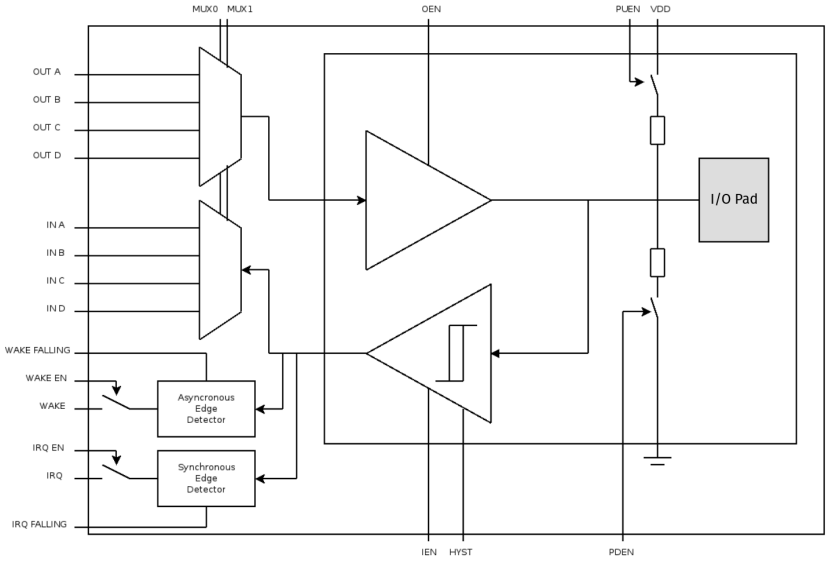
Device Functional Block Diagram



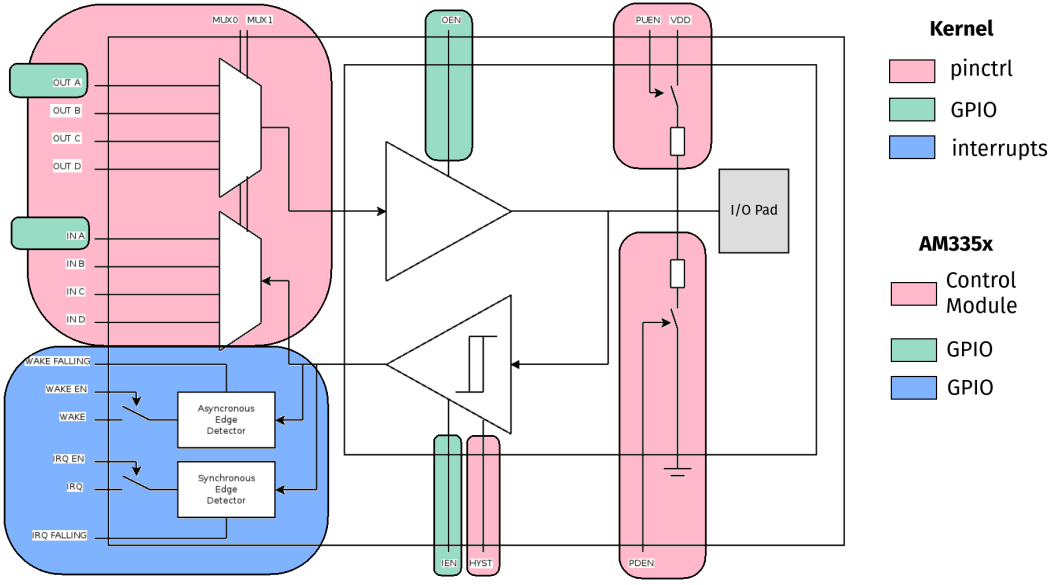
Hardware Overview

GPIO Module

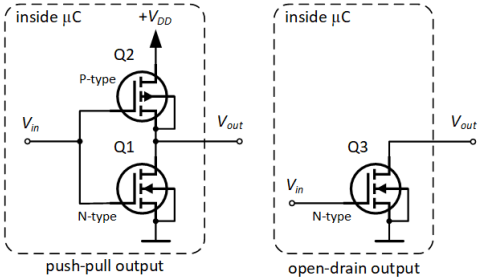
Generic GPIO Block

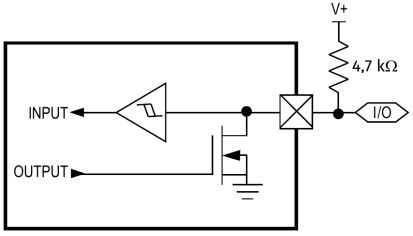


Generic GPIO Block



Output Buffers





- Push-pull states: **0** and **1**
- Open-drain states: **0** and **Hi-Z**
- Open-drain can be bidirectional
- External **pull-up** is usually used
- Allows several devices (line sharing, **wired-AND**)
- Often used for 1-wire or I2C (bit-banged)
- Different voltage can be used

Open-Drain Outputs in AM335x

- In AM335x we don't have **open-drain outputs**
- All GPIO output buffers are push-pull
- But we can emulate open-drain output behavior by switching input/output modes

Push-Pull Output:

- `GPIO_OE[n] = 0` (fixed, output)
- `GPIO_DATAOUT[n]:`
  - 0: low state
  - 1: high state

Open-Drain Output:

- `GPIO_DATAOUT[n] = 0` (fixed)
- `GPIO_OE[n]`:
  - 0 (output): low state
  - 1 (input): Hi-Z state

PRCM: Clock for GPIO

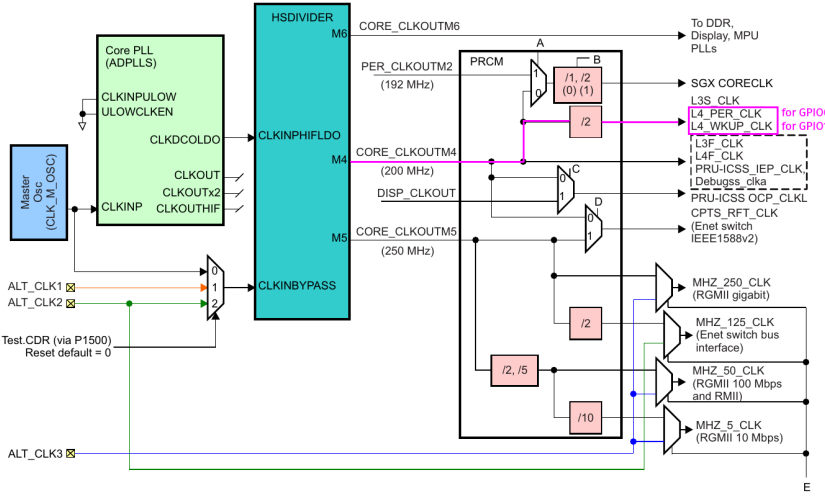


Figure 1: GPIO interface clock derivation

Clock Handling inside GPIO

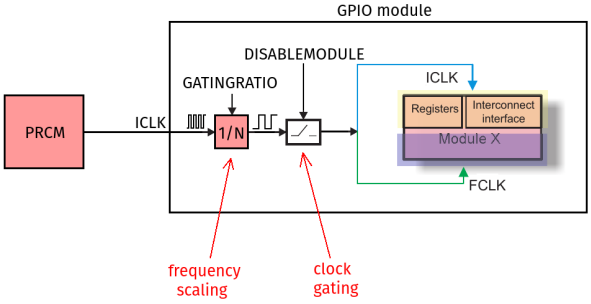


Figure 2: Clock path inside of GPIO module

$$P \approx P_{dyn} = C \cdot f \cdot V^2$$

Power Management Consideration

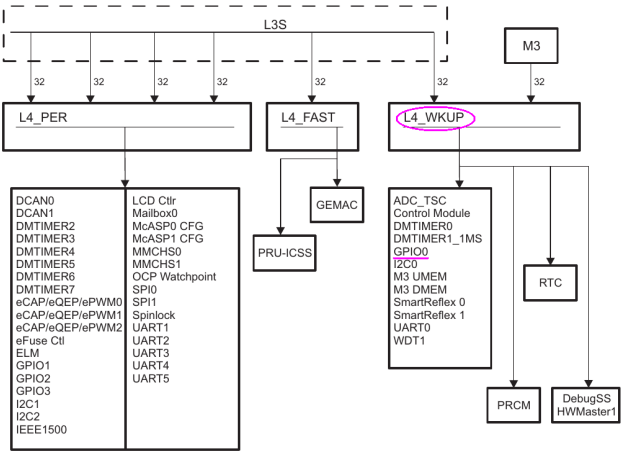


Figure 3: L4 Topology (Figure 10-2 in TRM)

Power Management Consideration

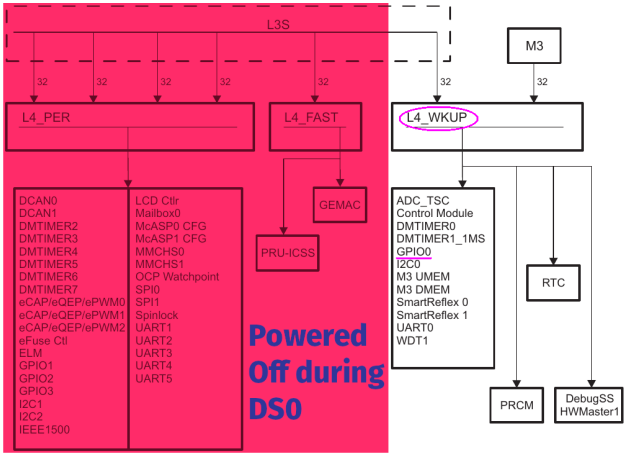


Figure 3: L4 Topology (Figure 10-2 in TRM)

Debouncing

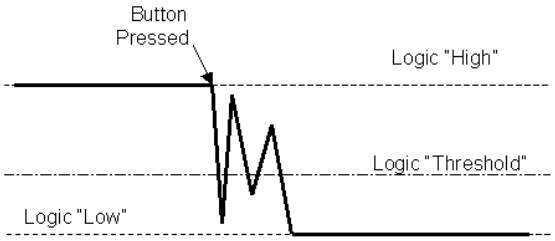


Figure 4: Button Debouncing

Debouncing

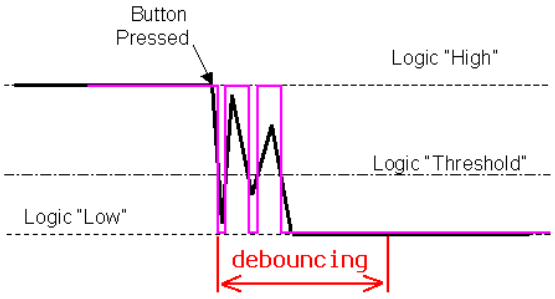


Figure 4: Button Debouncing

Debouncing (cont'd)

- Debouncing: software or hardware
- AM335x has GPIO debouncing capability:
  - `GPIO_DEBOUNCEENABLE[31:0]`
  - `GPIO_DEBOUNCINGTIME[7:0]`
- Kernel API:
  - Old: `gpio_set_debounce()`
  - New: `gpiod_set_debounce()`

Choosing Pins

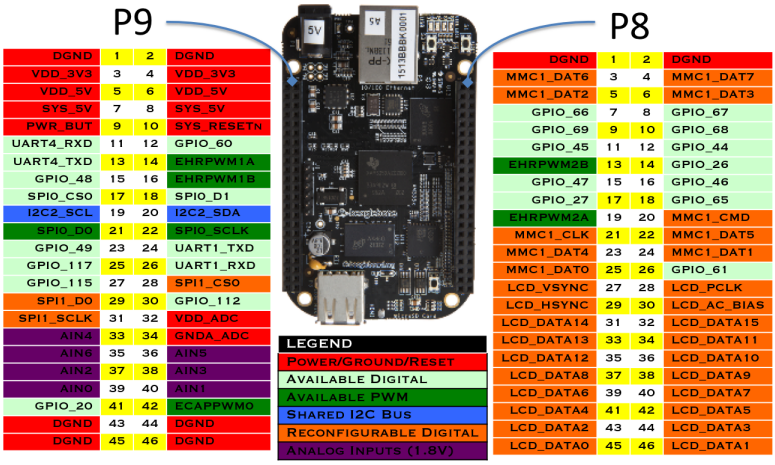


Figure 5: BBB Cape Expansion Headers

Choosing GPIO Pins

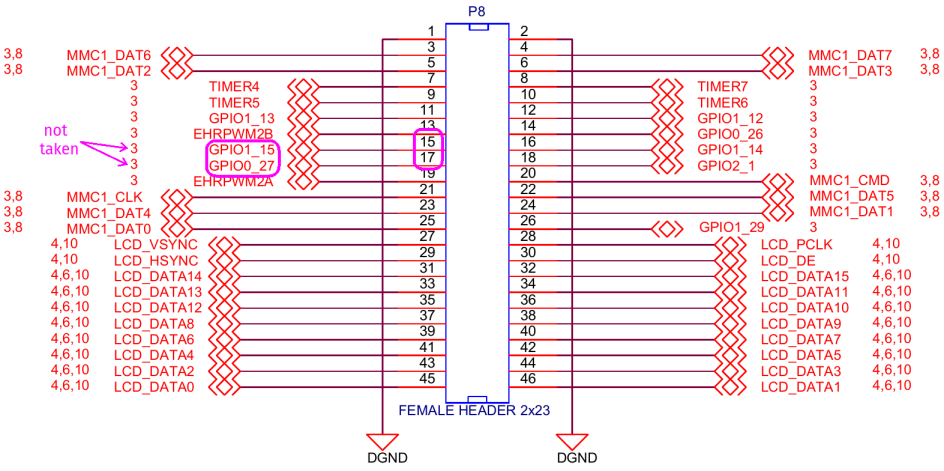


Figure 6: P8 Expansion Header

Choosing GPIO Pins (cont'd)

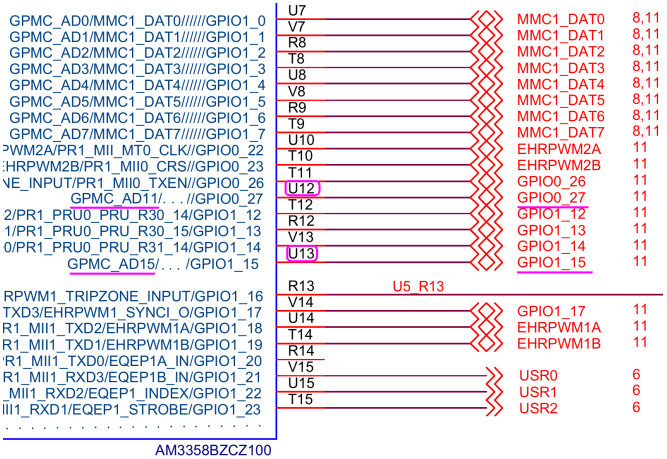


Figure 7: AM3358 pins

Registers and Values

Pin Multiplexing

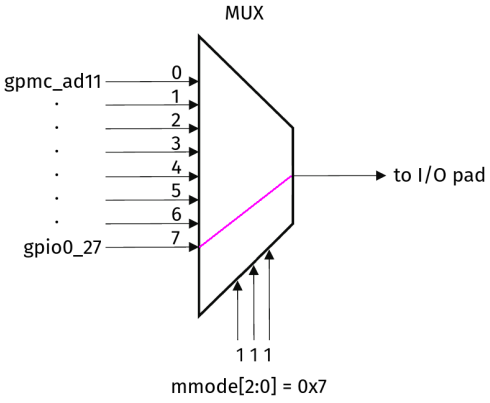


Figure 8: Pin mux example

Pin Modes

ZCZ BALL NUMBER [1]	PIN NAME [2]	SIGNAL NAME [3]	MODE [4]	TYPE [5]	BALL RESET STATE [6][25]	BALL RESET REL. STATE [7]	RESET REL. MODE [8]
U12	GPMC_AD11	gpmc_ad11	0	I/O	L	L	7
		lcd_data20	1	O			
		mmc1_dat3	2	I/O			
		mmc2_dat7	3	I/O			
		ehrpwm0_synco	4	O			
		pr1_mii0_txd3	5	O			
		gpio0_27	7	I/O			
U13	GPMC_AD15	gpmc_ad15	0	I/O	L	L	7
		lcd_data16	1	O			
		mmc1_dat7	2	I/O			
		mmc2_dat3	3	I/O			
		eQEP2_strobe	4	I/O			
		pr1_ecap0_ecap_capin_apwm_o	5	I/O			
		pr1_pru0_pru_r31_15	6	I			
		gpio1_15	7	I/O			

Figure 9: Pin attributes (Table 4-2 in datasheet)

Modules Addresses

Table 2-2. L4_WKUP Peripheral Memory Map

Region Name	Start Address (hex)	End Address (hex)	Size	Description
GPIO0	0x44E0_7000	0x44E0_7FFF	4KB	GPIO Registers
	0x44E0_8000	0x44E0_8FFF	4KB	Reserved
Control Module	0x44E1_0000	0x44E1_1FFF	128KB	Control Module Registers

Table 2-3. L4_PER Peripheral Memory Map

Device Name	Start_address (hex)	End_address (hex)	Size	Description
GPIO1	0x4804_C000	0x4804_CFFF	4KB	GPIO1 Registers
	0x4804_D000	0x4804_DFFF	4KB	Reserved

Figure 10: Base addresses of registers (“Memory Map” section)

9.3.1.50 conf_<module>_<pin> Register (offset = 800h–A34h)

See the device datasheet for information on default pin mux configurations.

See Table 9-10, *Control Module Registers Table*, for the full list of offsets for each module/pin configuration.

Figure 9-51. conf_<module>_<pin> Register

31	30	29	28	27	26	25	24
Reserved							
R-0h							

23	22	21	20	19	18	17	16
Reserved				Reserved			
R-0h				R-0h			

15	14	13	12	11	10	9	8
Reserved							
R-0h							

7	6	5	4	3	2	1	0
Reserved	conf_<module>_<pin>_slewctrl	conf_<module>_<pin>_rxactive	conf_<module>_<pin>_putypesel	conf_<module>_<pin>_puden	conf_<module>_<pin>_mmode		
R-0h	R/W-0h	R/W-1h	R/W-0h	R/W-0h	R/W-0h		

Figure 11: Pin mux registers description (“Control Module” section)

Pin Control Registers (cont'd)

Offset	Acronym
82Ch	conf_gpmc_ad11
83Ch	conf_gpmc_ad15

Figure 12: Pin mux registers offsets ("Control Module" section)

GPIO Registers Offset

Offset	Acronym	Section
130h	GPIO_CTRL	Section 25.4.1.15
134h	GPIO_OE	Section 25.4.1.16
138h	GPIO_DATAIN	Section 25.4.1.17
13Ch	GPIO_DATAOUT	Section 25.4.1.18

Figure 13: Address offset for GPIO registers

GPIO_CTRL Register

31	30	29	28	27	26	25	24
RESERVED							
R-0h							
23	22	21	20	19	18	17	16
RESERVED							
R-0h							
15	14	13	12	11	10	9	8
RESERVED							
R-0h							
7	6	5	4	3	2	1	0
RESERVED					GATINGRATIO		DISABLEMODULE
R-0h					R/W-0h		R/W-0h

Figure 14: GPIO_CTRL register

Rest of GPIO Registers

Figure 25-23. GPIO_OE Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUTPUTEN[n]																															
R/W-FFFFFFFFh																															

Figure 25-24. GPIO_DATAIN Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATAIN																															
R-0h																															

Figure 25-25. GPIO_DATAOUT Register

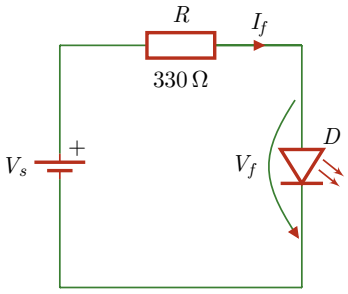
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATAOUT																															
R/W-0h																															

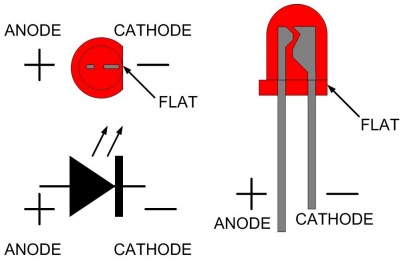
Figure 15: GPIO common registers

Take five

Building the Device

Interfacing LED





$V_s = 3.3\,\mathrm{V}$ (BBB GPIO voltage)

$V_f = 2.0\,\mathrm{V}$ (for red LED)

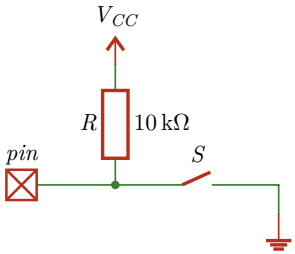
$I_{LED} = 20\,\mathrm{mA}$ (nominal LED current)

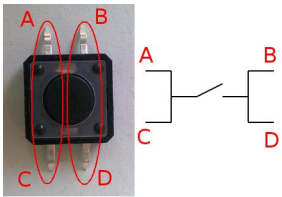
$I_{GPIO} = 4\,\mathrm{mA}$ (max BBB GPIO)

$$R = \frac{V_s - V_f}{I_f}$$

$$R = \frac{3.3 - 2.0}{0.004} \approx 330\,\Omega$$

Interfacing Button





Logic:

- Button pressed: $V_{pin} = 0\,\mathrm{V}$
- Button not pressed: $V_{pin} = V_{CC} = 3.3\,\mathrm{V}$

Pull-up resistor:

- Current must be strong enough to eliminate noise
- But we don't want too much power to be drawn either
- Pull-up resistor is usually $10\,\mathrm{k}\Omega$

Breadboard

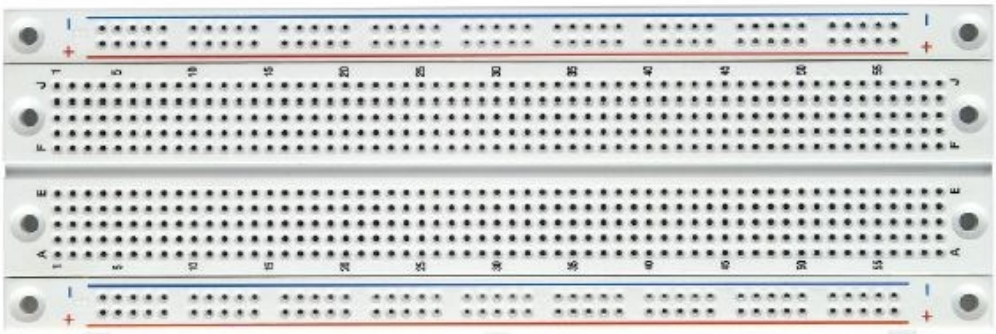


Figure 16: EIC-20020 Breadboard

Breadboard

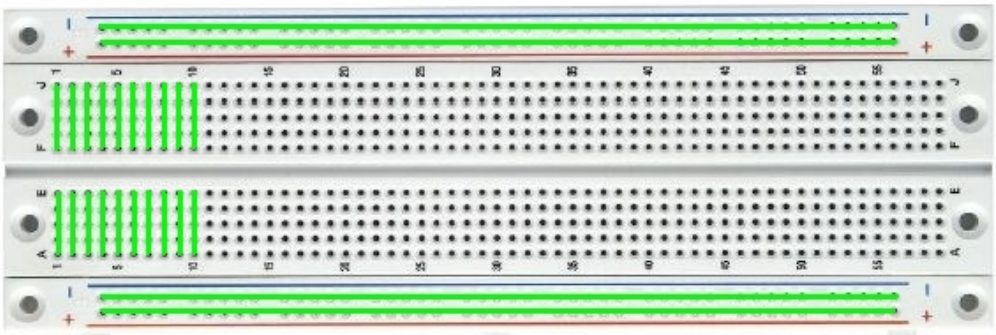


Figure 16: EIC-20020 Breadboard

Device: Schematics

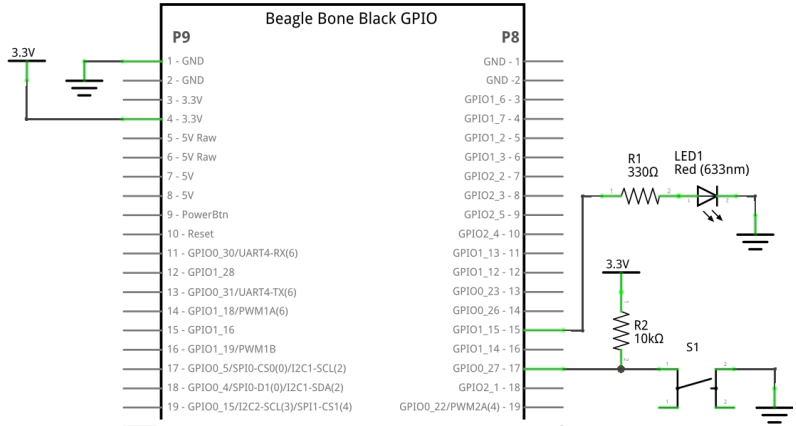


Figure 17: Schematics of device

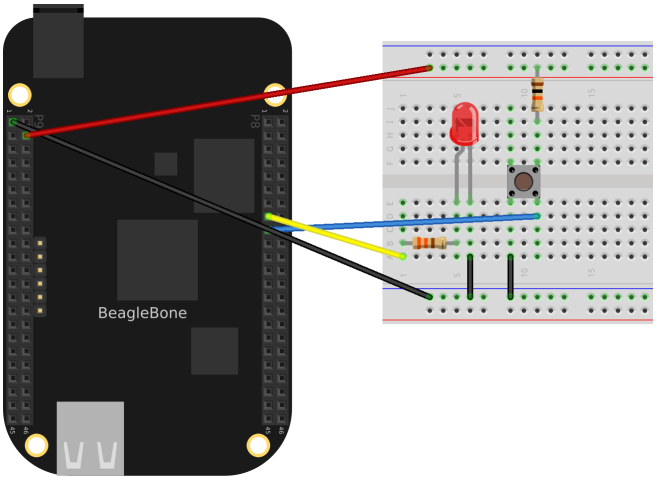


Figure 18: Breadboard connections

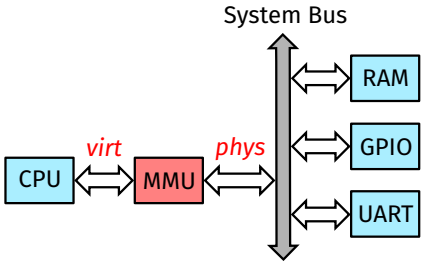
Kernel Driver: Naïve Approach

Attempt #1

Direct access to registers

DO NOT USE IN REAL LIFE!

API: ioremap




```
virt_addr = ioremap(phys_addr, size);
```

API: ioremap (cont'd)

```c
#include <asm/io.h>

/* Create/remove the mapping to virtual address space for I/O region */
void __iomem *ioremap(resource_size_t res_cookie, size_t size);
void iounmap(volatile void __iomem *iomem_cookie);

/* Read/write primitives: handle memory barriers, endianness, volatile */
void iowrite32(u32 value, volatile void __iomem *addr);
u32 ioread32(const volatile void __iomem *addr);
```

Read LDD3 chapter 9 for details.

First Attempt: Code (1/7)

Listing 1: hw1.c

```
1  #include <linux/module.h>
2  #include <linux/kthread.h>
3  #include <linux/delay.h>
4  #include <asm/io.h>
5
6  /* Modules base addresses and registers offsets */
7  #define CTRL_MODULE_BASE        0x44e10000
8  #define GPIO0_BASE              0x44e07000
9  #define GPIO1_BASE              0x4804c000
10 #define CONF_GPMC_AD11_OFFSET   0x82c
11 #define CONF_GPMC_AD15_OFFSET   0x83c
12 #define GPIO_CTRL_OFFSET        0x130
13 #define GPIO_OE_OFFSET          0x134
14 #define GPIO_DATAIN_OFFSET      0x138
15 #define GPIO_DATAOUT_OFFSET     0x13c
16
17 /* Registers addresses */
18 #define CONF_GPMC_AD11          (CTRL_MODULE_BASE + CONF_GPMC_AD11_OFFSET)
19 #define CONF_GPMC_AD15          (CTRL_MODULE_BASE + CONF_GPMC_AD15_OFFSET)
20 #define GPIO0_CTRL              (GPIO0_BASE + GPIO_CTRL_OFFSET)
```

First Attempt: Code (2/7)

```
21 #define GPIO1_CTRL              (GPIO1_BASE + GPIO_CTRL_OFFSET)
22 #define GPIO0_OE                (GPIO0_BASE + GPIO_OE_OFFSET)
23 #define GPIO1_OE                (GPIO1_BASE + GPIO_OE_OFFSET)
24 #define GPIO0_DATAIN            (GPIO0_BASE + GPIO_DATAIN_OFFSET)
25 #define GPIO1_DATAOUT           (GPIO1_BASE + GPIO_DATAOUT_OFFSET)
26
27 /* conf_<module>_<pin> registers flags */
28 #define MUX_MODE7               0x7
29 #define PULL_DISABLE            BIT(3)
30 #define INPUT_EN                BIT(5)
31 #define BTN_CONF                (MUX_MODE7 | PULL_DISABLE | INPUT_EN)
32 #define LED_CONF                (MUX_MODE7 | PULL_DISABLE)
33
34 /* GPIO line numbers for button and LED */
35 #define BTN_LINE                BIT(27)         /* gpmc_ad11.gpio0_27 */
36 #define LED_LINE                BIT(15)         /* gpmc_ad15.gpio1_15 */
37
38 static struct task_struct *thread;
39 static void __iomem *btn_gpio, *led_gpio;
40
41 static void hw1_mux_pins(void)
42 {
43         void __iomem *reg;
```

First Attempt: Code (3/7)

```
44
45          /* BTN: mux the pin */
46          reg = ioremap(CONF_GPMC_AD11, 4);
47          iowrite32(BTN_CONF, reg);
48          iounmap(reg);
49
50          /* LED: mux the pin */
51          reg = ioremap(CONF_GPMC_AD15, 4);
52          iowrite32(LED_CONF, reg);
53          iounmap(reg);
54  }
55
56  static void hw1_setup_gpio(void)
57  {
58          void __iomem *reg;
59          u32 val;
60
61          /* BTN: enable modules, clock = functional */
62          reg = ioremap(GPIO0_CTRL, 4);
63          iowrite32(0x0, reg);
64          iounmap(reg);
65
66          /* LED: enable modules, clock = functional */
```

First Attempt: Code (4/7)

```
67          reg = ioremap(GPIO1_CTRL, 4);
68          iowrite32(0x0, reg);
69          iounmap(reg);
70
71          /* BTN: configure input/output mode */
72          reg = ioremap(GPIO0_OE, 4);
73          val = ioread32(reg);
74          val |= BTN_LINE;        /* 1 = input */
75          iowrite32(val, reg);
76          iounmap(reg);
77
78          /* LED: configure input/output mode */
79          reg = ioremap(GPIO1_OE, 4);
80          val = ioread32(reg);
81          val &= ~LED_LINE;       /* 0 = output */
82          iowrite32(val, reg);
83          iounmap(reg);
84  }
85
86  static int hw1_thread_func(void *data)
87  {
88          /* Poll the button */
89          while (!kthread_should_stop()) {
```

First Attempt: Code (5/7)

```
90                  u32 btn_val, led_val;
91
92                  btn_val = ioread32(btn_gpio);
93                  led_val = ioread32(led_gpio);
94                  if (btn_val & BTN_LINE) {
95                          /* not pressed (pull-up): LED off */
96                          led_val &= ~LED_LINE;
97                  } else {
98                          /* pressed (GND): LED on */
99                          led_val |= LED_LINE;
100                 }
101
102                 iowrite32(led_val, led_gpio);
103
104                 msleep(100);
105         }
106
107         return 0;
108 }
109
110 static int __init hw1_init(void)
111 {
112         int ret;
```

First Attempt: Code (6/7)

```
113
114         hw1_mux_pins();
115         hw1_setup_gpio();
116
117         btn_gpio = ioremap(GPIO0_DATAIN, 4);
118         led_gpio = ioremap(GPIO1_DATAOUT, 4);
119
120         thread = kthread_run(hw1_thread_func, NULL, "hw1_thread");
121         if (IS_ERR(thread)) {
122                 pr_err("kthread_run() failed\n");
123                 ret = PTR_ERR(thread);
124                 goto err1;
125         }
126
127         return 0;
128
129 err1:
130         iounmap(led_gpio);
131         iounmap(btn_gpio);
132         return ret;
133 }
134
135 static void __exit hw1_exit(void)
```

First Attempt: Code (7/7)

```
136 {
137         if (thread)
138                 kthread_stop(thread);
139         iounmap(led_gpio);
140         iounmap(btn_gpio);
141 }
142
143 module_init(hw1_init);
144 module_exit(hw1_exit);
145
146 MODULE_AUTHOR("Sam Protsenko <semen.protsenko@globallogic.com>");
147 MODULE_DESCRIPTION("Test module 1");
148 MODULE_LICENSE("GPL");
```

Shortcomings

This approach is just plain wrong:

- Will work only on BBB
- Possible conflicts for GPIO (lines) and pinctrl (muxes)
- Possible race-condition due to `GPIO_DATAOUT` usage (can be solved by using `GPIO_SETDATAOUT` instead)
- Too much code
- Polling vs interrupts
- No user space interface

Use Device Tree, Luke!

IOREMAP

DEVICE TREE

Attempt #2

Legacy GPIO API + pinctrl in device tree

DO NOT USE IN REAL LIFE!

GPIO Kernel Frameworks

- Your kernel already has GPIO driver for SoC's GPIO module
  - It's platform-dependent: aware of platform-specific data (registers, interrupts, etc)
  - You can't use the driver directly (it's only setting callbacks)
  - Device Tree definition: `arch/arm/boot/dts/am33xx.dtsi`
  - Driver's code: `drivers/gpio/gpio-omap.c`

GPIO Kernel Frameworks

- Your kernel already has GPIO driver for SoC's GPIO module
  - It's platform-dependent: aware of platform-specific data (registers, interrupts, etc)
  - You can't use the driver directly (it's only setting callbacks)
  - Device Tree definition: `arch/arm/boot/dts/am33xx.dtsi`
  - Driver's code: `drivers/gpio/gpio-omap.c`
- Kernel has GPIO frameworks to access GPIO driver functions
  - Platform-independent API: the same usage for all possible boards
  - All API calls lead to GPIO driver calls
  - Legacy GPIO API: `include/linux/gpio.h`
  - New GPIO API: `linux/gpio/consumer.h`

API: Legacy GPIO Kernel API

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

int gpio_request(unsigned gpio, const char *label);
void gpio_free(unsigned gpio);
int gpio_to_irq(unsigned gpio);
int gpio_direction_output(unsigned gpio, int value);
int gpio_direction_input(unsigned gpio);
void gpio_set_value(unsigned gpio, int value);
int gpio_get_value(unsigned gpio);
```

Read `Documentation/driver-api/gpio/legacy.rst` for details.

Second Attempt: Device Tree

Listing 2: am335x-boneblack.dts

```
&am33xx_pinmux {
        hw_pins: hw_pins {
                pinctrl-single,pins = <
                        AM33XX_IOPAD(0x82c, PIN_INPUT | MUX_MODE7)              /* gpmc_ad11.gpio0_27 */
                        AM33XX_IOPAD(0x83c, PIN_OUTPUT | MUX_MODE7)             /* gpmc_ad15.gpio1_15 */
                >;
        };
};

/ {
        soc {
                pinctrl-names = "default";
                pinctrl-0 = <&hw_pins>;
        };
};
```

Second Attempt: Code (1/3)

Listing 3: hw2.c

```
1  #include <linux/module.h>
2  #include <linux/gpio.h>
3  #include <linux/interrupt.h>
4
5  #define AM335_GPIO(bank,line)   (32 * bank + line)
6  #define LED_GPIO                AM335_GPIO(1, 15)
7  #define BTN_GPIO                AM335_GPIO(0, 27)
8
9  static int irq;
10 static int led_on;
11
12 static irqreturn_t hw2_btn_isr(int num, void *priv)
13 {
14         led_on ^= 0x1;
15         gpio_set_value(LED_GPIO, led_on);
16         pr_info("interrupt!\n");
17         return IRQ_HANDLED;
18 }
19
20 static int __init hw2_init(void)
```

Second Attempt: Code (2/3)

```
21 {
22         int err;
23
24         err = gpio_request(BTN_GPIO, "my_button");
25         if (err) {
26                 pr_err("Unable to request button GPIO\n");
27                 return -EINVAL;
28         }
29         gpio_direction_input(BTN_GPIO);
30         irq = gpio_to_irq(BTN_GPIO);
31         err = request_threaded_irq(irq, NULL, hw2_btn_isr,
32                                    IRQF_TRIGGER_FALLING | IRQF_ONESHOT,
33                                    "my button key", NULL);
34         if (err) {
35                 pr_err("Unable to request interrupt for button\n");
36                 return -EINVAL;
37         }
38
39         err = gpio_request(LED_GPIO, "my_led");
40         if (err) {
41                 pr_err("Unable to request LED GPIO\n");
42                 return -EINVAL;
43         }
```

Second Attempt: Code (3/3)

```
44          gpio_direction_output(LED_GPIO, led_on);
45
46          return 0;
47  }
48
49  static void __exit hw2_exit(void)
50  {
51          free_irq(irq, NULL);
52          gpio_free(BTN_GPIO);
53          gpio_free(LED_GPIO);
54  }
55
56  module_init(hw2_init);
57  module_exit(hw2_exit);
58
59  MODULE_AUTHOR("Sam Protsenko <semen.protsenko@globallogic.com>");
60  MODULE_DESCRIPTION("Test module 2");
61  MODULE_LICENSE("GPL");
```

Shortcomings

Somewhat better, but still has its pitfalls:

- GPIO lines are still hard-coded in driver
- Obsolete legacy GPIO API
- Referencing pinctrl from `soc` node is hacky
- Still no user space interface

Assignments

Assignment 1 (mandatory)

Try out everything we discussed today:

- Assemble “LED + button” device on the breadboard
- Build and run provided module #1 (make sure the device works)
- Build and run provided module #2 (make sure the device works)
  - Apply provided patch for `.dts` file (in kernel)
  - Rebuild `.dtb` file for BBB
  - Boot kernel with new `dtb` and load `hw2.ko`
- Think about how mentioned problems can be solved

Assignment 2

Find out and use existing drivers in the kernel:

- Look for `gpio-leds` and `gpio-keys` drivers in `Documentation/devicetree/bindings/`
- Try to use those drivers instead of `hw1.ko` / `hw2.ko` from this lecture
- Manage to handle button and LED in user space
- How to use PWM module to control LED brightness?
  - Which pin to connect LED to?
  - How to mux that pin for PWM mode?
  - How to enable PWM module in device tree?
  - Now how to control the LED brightness from user space? (sysfs)
  - Which existing driver can be used to control the LED brigtness via PWM from device tree? Try to use it.

Assignment 3

Figure out how `ioremap()` actually works:

- Examine `Documentation/arm/memory.rst`:
  - At which addresses `ioremap()` mappings are stored?
  - Which function is used to prepare *static mappings*?
- Figure out how static mappings are being set up (hint: look where the function mentioned in Documentation is being called from (from `arch/` code)? what's passed to that function from there?)
- Which physical addresses are being statically mapped?
- What are the corresponding virtual addresses (calculate manually)?
- Look at AM335x TRM, section 2 "Memory Map": which hardware modules can be accessed via those statically mapped addresses in kernel?

Assignment 3 (cont'd)

- Where is that function (from `arch/`) used? It's set to some callback field in some struct. Provide the file path (it's actually a *board file*).
- What does "static mapping" mean? Use `git blame` on the function mentioned in Documentation and read commit messages for found commits to get more info. Also Google for it.
- Does `ioremap()` actually creates new MMU mapping in page table when you call it, passing some GPIO register address?
- Can we avoid using `ioremap()` in `hw1.ko`? Why is that possible?
- Why it's incorrect to use `ioremap()` for system memory (regular RAM)?
- Trace `ioremap()` function call as deep as possible, using your IDE capabilities (for BBB architecture)

Assignment 4

Implement user space GPIO driver for our device

- Create minimal user space driver (in C) using `/dev/gpiochip*` char devs
  - Toggle the LED on button press while your app is running
- Look here for insights:
  - Syscall interface: `Documentation/ABI/testing/gpio-cdev`
  - Examples: `tools/gpio/*`
- It's even better to use `libgpiod` (you'll have to cross-compile it for ARM)
- Don't use GPIO sysfs interface, it's obsolete
- Send me all source files
- **NOTE**: User-space GPIO interface is useful for makers, but usually it's a poor substitute for real kernel drivers

Thank you!