KAIST
Electrical Engineering School

Embedded Software - EE516

Project 5

"Device Driver for Embedded H/W"
Interrupt & GPIO Control in Beagle Board

Guarav Kalra

Mario Loaiciga

Fall Semester - 2016

Task 1

The solution of this first task required of two fundamental structures: `struct mem_block`, `struct linked_list` (see List. 1). Said structures are combined to allocate memory blocks linked by pointers to other blocks. On module initialization, the function `dummy_create()` allocates the required memory for the linked list. For this implementation, we assumed the linked list has a fixed size define as `LIST_SIZE` in the `linked_list.h` header file.

```c
typedef struct mem_block
{
  struct mem_block *next;//@ Pointer to next block
  int value;             //@ Pointer to the value stored in the block

}block;
```

```c
typedef struct linked_list
{
  int cnt;        //@ Number of blocks in the list
  block *head;    //@ Address of first block's structure

}linked_list;
```

Listing 1: Linked list structures (included in `linked_list.h`)

An extract of the `dummy_read()` function is shown in List. 2. This function has the particularity of changing the usage of the argument `size_t length`, that is, since the device operates only on fixed size integer values, the read and write functions always return the same amount if bytes. Thus, the `size_t length` argument can now be used by the reader application to communicate to the device driver the value it wants to receive. This behavior can be seen in line 95 of List. 2, where the request is stored in `rand_num`. The `dummy_read()` function then looks for the request in the list and uses `copy_to_user` to return a value to the user application.

The execution of the user space application operating in the linked list memory device is shown in Fig. 1. Observe that upon module initialization, the linked list is readied. The application in user space simply obtains the file descriptor of the device `DUMMY_DEVICE` and then pass it down to all is threads, which in turn read and write as the semaphores `sem mutex`, `sem full` and `sem empty` allow them to.

In the example shown in Fig. 1, a total of nine writers start by placing random values the list. Next, three readers request values from the list. The reader with ID 1 requests a value of 1, but since this value was now introduced in the previous write operations, it reads a default value -1. The following two readers obtain their requested values, which were introduced by writer 8 and writer 9. Fig. 2 shows the log of the clean up function which removes the module and also traverses the linked list freeing the allocated memory(`dummy_destroy() defined in linked_list.h`). Notice how in lines 126 and 127 semaphores protect all access to the linked list related structures.

```
ssize_t dummy_read(struct file *file, char *buffer, size_t length, loff_t *offset)
{
```

```
  /*@ Load read value requested from parameter lenght @*/
  rand_num = length;//@ length is now used to request values

  down(&full);//@ Sleep if there is zero full blocks
  down(&mutex);//@ Enter criticla region

  /*@ Loop until the hold list has been searched @*/
  while(i < list.cnt){

    if(read_block->value == rand_num){

      *dummy_buf = read_block->value;//@ Copy the value to the dummy
      read_block->value = INIT_VAL;//@ Delete the value from the block
      printk("@ Read HIT\n");
      goto HIT;//@  The value was found, copy to user the value
    }
    read_block = read_block->next;
    i++;//@ Save the iteartion number
  }

  /*@ Else the value was not located, copy -1 to user @*/
  *dummy_buf = INIT_VAL;
  printk(" @ Read MISS\n");
  HIT:
  if(copy_to_user(buffer, dummy_buf, sizeof(int))){
    kfree(dummy_buf);
    return -EFAULT;
  }

  i = (i>=LIST_SIZE)?-1:i;//@ Block number = -1 if value not found
  printk("Dummy Driver : Read Call (block: %d, value: %d)\n",i,*dummy_buf);

  up(&mutex);//@ Exit critical region
  up(&empty);//@ Signal another block has been emptied
```

Listing 2: Device read function

The operation of the `dummy_write()` function has the particularity of using the function `get_random_bytes()` (see List. 3, line 151). This function allows to generate signed random numbers in kernel space. Once the random number is generated, the linked list is traversed again to locate the block to be written.

```
136 ssize_t dummy_write(struct file *file,const char *buffer,size_t length,...
137 {
```

```
142   block *write_block = list.head;//@ Initialize the block pointer
143 
144 
145   if(copy_from_user(dummy_buf, buffer, sizeof(int))){//@ Copy user's data to dummy
146     kfree(dummy_buf);
147     return -EFAULT;
148   }
149 
```

```c
/*@ Create random number ( 0 ~ LIST_SIZE-1 )@*/
get_random_bytes(&rand_num,sizeof(int));
rand_num = abs(rand_num) % LIST_SIZE;

down(&empty);//@ Sleep if there is zero empty blocks
down(&mutex);//@ Enter critical region

  while( i < rand_num){
    write_block = write_block->next;//@ Trasverse the list until
    i++;                            //@ the random block is pointed at
  }

  write_block->value=*dummy_buf;

  printk("Dummy Driver : Write Call (block: %d, value: %d)\n",i,*dummy_buf);

up(&mutex);//@ Exit critical region
up(&full);//@ Signal another block has been filled
```

Listing 3: Device write function

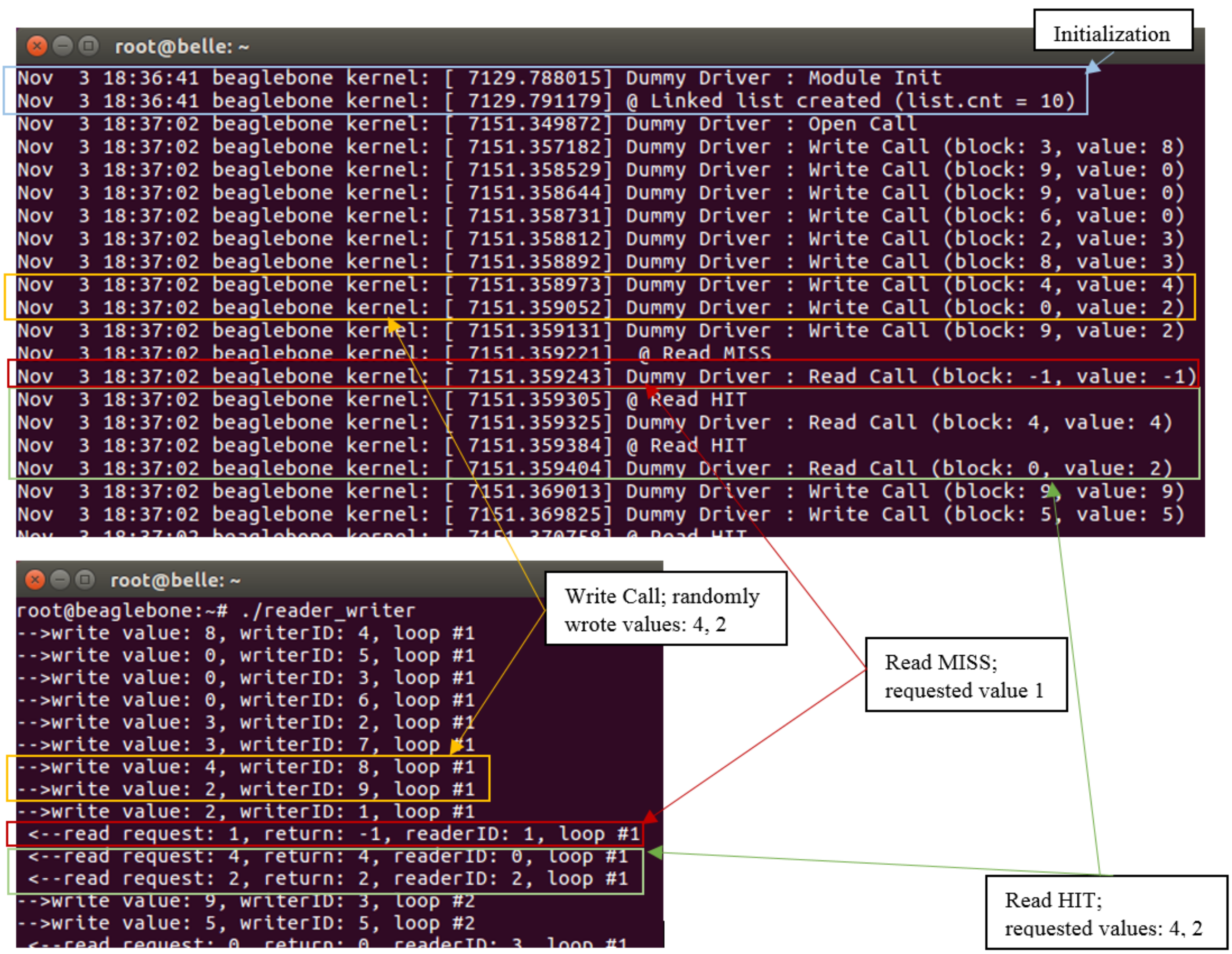

Figure 1: Linked list module operation

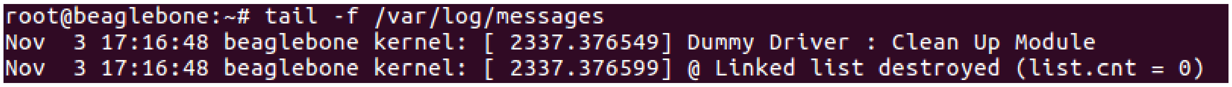

Figure 2: Module unmount and memory clean up

Task 2

The implementation of this task can be completed progressively building upon the a group of basic functions and structures that allows to handled the LEDs and create interruptions from both the kernel timers or physical inputs. All the four sub-tasks depend of `struct gpio_data`, a structure that holds the reference information of the LEDs (see List. 4). This structured is used to request, by means of a standard procedure, the control over this set of GPIO elements.

```
static struct {
    unsigned int gpio; /* GPIO of LED */
    const char *label; /* GPIO label */
    bool valid; /* If TRUE, GPIO is requested & allocated */
  } gpio_data[NUM_LED] = {
      [LED0] = {LED0_GPIO, "LED0_GPIO", FALSE},
      [LED1] = {LED1_GPIO, "LED1_GPIO", FALSE},
      [LED2] = {LED2_GPIO, "LED2_GPIO", FALSE},
      [LED3] = {LED3_GPIO, "LED3_GPIO", FALSE},
};
```

Listing 4: GPIO LED structure for signal mapping

Task 2.1

The completion of this task depends on the following GPIO functions:

```
gpio_is_valid()

gpio_request()

gpio_direction_output()

gpio_set_value()

gpio_free()
```

This fairly simple sub-task's operation flow is shown in List. 6. In order to request the GPIO of the LEDs, the costume function `_bb_module_startup()` is used. After the LED request is successful, the program turn on the LEDs, one after the other, using the `gpio_set_value()` function which is it at the core of the function `_turn_on_led()`. This is function is called in each one of the four LEDs.

```
static int __init
bb_module_init(void)
{
```

```
  dbg("");

  /* startup */
  _bb_module_startup();

  /* turn on LED 0 */
  _turn_on_led(LED0_GPIO);
  msleep(1000); //sleep
```

```
  /* turn on LED 3 */
  _turn_on_led(LED3_GPIO);
  msleep(1000); //sleep

  return 0;
}
```

Listing 5: Turn on LEDs in sequence

Task 2.2

The completion of this task depends on the creation of kernel timer interrupts. This is achieved by means of the following functions:

`init_timer()`

`add_timer()`

`del_timer()`

`get_jiffies_64()`

The code is also dependent of the `struct timer_list`, defined in `linux/timer.h`.

```
static struct timer_list kern_timer;
```

Said structure contains the value `expires` which is used to save the life time of the timer, as well as a pointer for a function handler that allows us specify an action upon time lapse completion. This structure also contain a value `data` that can be passed as an argument for later usage.

As was the case for the module initialization of the last sub-task, the module first registers GPIO elements and then turns on the LEDs, but now the latter accion is carried out by the `_bb_module_register_timer` (see List. 6)

```
static int __init
bb_module_init(void)
{
  dbg("");

  /* startup */
  _bb_module_startup();

  /* register timer */
```

```c
  _bb_module_register_timer();

  return 0;
}
```

Listing 6: Turn LEDs on kernel interrupt

List. 7 show the sequence of steps used to register the kernel timer, setting up its expiration time and the eventual callback to the `kern_timer_handler`, a function toggles the state of the LEDs (LED blink).

```c
static void
_bb_module_register_timer(void)
{
  // initialize timer
  init_timer(&kern_timer);

  //expire at current + TIME_STEP
  kern_timer.expires = get_jiffies_64() + TIME_STEP;

  // handler
  kern_timer.function = kern_timer_handler;
  kern_timer.data = 0;

  // add to kernel
  add_timer(&kern_timer);
}
```

Listing 7: Handling of the timer for kernel interrupts

Task 2.3

The completion of this task depends on the utilization of the following functions interruption request & handling functions and flags:

`gpio_to_irq()`	`IRQF_TRIGGER_RISING`
`request_irq()`	`IRQF_TRIGGER_FALLING`
`free_irq()`	`IRQF_TRIGGER_RISING`

In addition, we now introduce the value `irq_number` which will serve us to map the GPIO button signal to a IRQ signal in the kernel:

```c
static unsigned int irq_number;
```

Once more, we can see the module initialization (see List. 8) requests the GPIO of the LEDs though `_bb_module_startup()`. However, now the `_bb_module_register_button()` function is introduce to map the GPIO button to a signal interrupt and include an interruption handler that we have called `button_irq_handler()`.

```
static int __init
bb_module_init(void)
{
  int ret;
  dbg("");

  /* register button */
  ret = _bb_module_register_button();
  if (ret < 0) {
    err("Failed to setup button IRQ");
    return -1;
  }

  /* startup */
  _bb_module_startup();

  return 0;
}
```

Listing 8: Introduction of GPIO interrupt

In List. 9 we have shown the code of the `button_irq_handler()`. Naturally, this call-back function is executed on each rising edge of the GPIO button (`IRQF_TRIGGER_RISING`). The handler carries tow critical tasks:

1. Check the current state of the GPIO
2. Register the kernel timer which will later make the LEDs blink

```
static irq_handler_t button_irq_handler
(unsigned int irq, void *dev_id, struct pt_regs *regs)
{
  static bool blinking = FALSE;
  int iter;

  // if not blinking, start blinking
  if (blinking == FALSE) {
    /* turn on all LEDs (for quick response) */
    for (iter = LED0; iter < NUM_LED; iter++) {
      // turn on LED
      _turn_on_led(gpio_data[iter].gpio);
    }

    /* start pattern */
    _bb_module_register_timer();
    blinking = TRUE;
  }
  // blinking, stop blinking
  else {
    _bb_module_unregister_timer();

    /* turn off all LEDs (if still on) */
    for (iter = LED0; iter < NUM_LED; iter++) {
      // if on
      if (!!gpio_get_value(gpio_data[iter].gpio)) {
        // turn off LED
```

```
        _turn_off_led(gpio_data[iter].gpio);
      }
    }

    blinking = FALSE;
  }

  return (irq_handler_t)IRQ_HANDLED;
}
```

Listing 9: GPIO Interruption handling function

Task 2.4

```
static unsigned int counter;

static spinlock_t gpio_press_time_lock;
static ktime_t gpio_press_time;
```

```
static irq_handler_t button_irq_handler
(unsigned int irq, void *dev_id, struct pt_regs *regs)
{
  uint8_t value;
  unsigned int duration;

  spin_lock(&gpio_press_time_lock);
  value = gpio_get_value(BUTTON_GPIO);

  if (value == 1) {
    // If no time has elapsed, we probably already cleared on a FALLING
    // interrupt. So finish the handler.
    if (ktime_to_ms(gpio_press_time) == 0) {
      goto finished;
    }

    duration = ktime_to_ms(ktime_sub(ktime_get(), gpio_press_time));

    if (duration == 0) {
      goto finished;
    }

    gpio_press_time = ktime_set(0, 0);
    info("Detected button release, duration of %u", duration);

    if (duration >= 1000) { //1sec => reset
      _handle_counter_pattern(TRUE);
    } else {
      _handle_counter_pattern(FALSE);
    }
  } else {
    // If a time is already set, we already received a RISING interrupt.
    // So we can finish the handler.
    if (ktime_to_ms(gpio_press_time) > 0) {
      goto finished;
    }
```

```

    info("Detected button press");
    gpio_press_time = ktime_get();
  }

  finished:
  spin_unlock(&gpio_press_time_lock);
  return (irq_handler_t)IRQ_HANDLED;
}
```

Listing 10: GIPO interruption: Detects duration between rising edges

```
static void
_handle_counter_pattern(bool reset)
{
  unsigned int val;
  static bool timer_registered = FALSE;

  val = reset ? _reset_counter_value() : _increment_counter_value();

  /* for quick response */
  _turn_on_led_pattern(val);

  /* start blinking pattern */
  if (timer_registered == FALSE && val != 0) {
    _bb_module_register_timer();
    timer_registered = TRUE;
  }
  /* value has been reset (remove timer since all LEDs are off anyways) */
  else if (val == 0 && timer_registered == TRUE) {
    _bb_module_unregister_timer();
    timer_registered = FALSE;
  }
  /* renew timer */
  else if (timer_registered == TRUE) {
    _bb_module_unregister_timer();
    _bb_module_register_timer();
  }
}
```

Listing 11: LED and kernel timer handling