Interrupt Requests (IRQ)

Accessing interrupts in Linux using the beaglebone black.

Interrupts and sysfs

The basic template covered in the previous lecture can be expanded to create a more sophisticated device driver.

We can explore this by connecting the Beaglebone to some external hardware and interacting with the hardware via the LKM.

Interfacing and directly communicating with hardware is an important part of developing an embedded system

GPIO via kernel space

We can interact with the Beaglebone GPIO's via kernel space.

- LKM can use GPIO based interrupts to increase response time over GPIO access via user space.

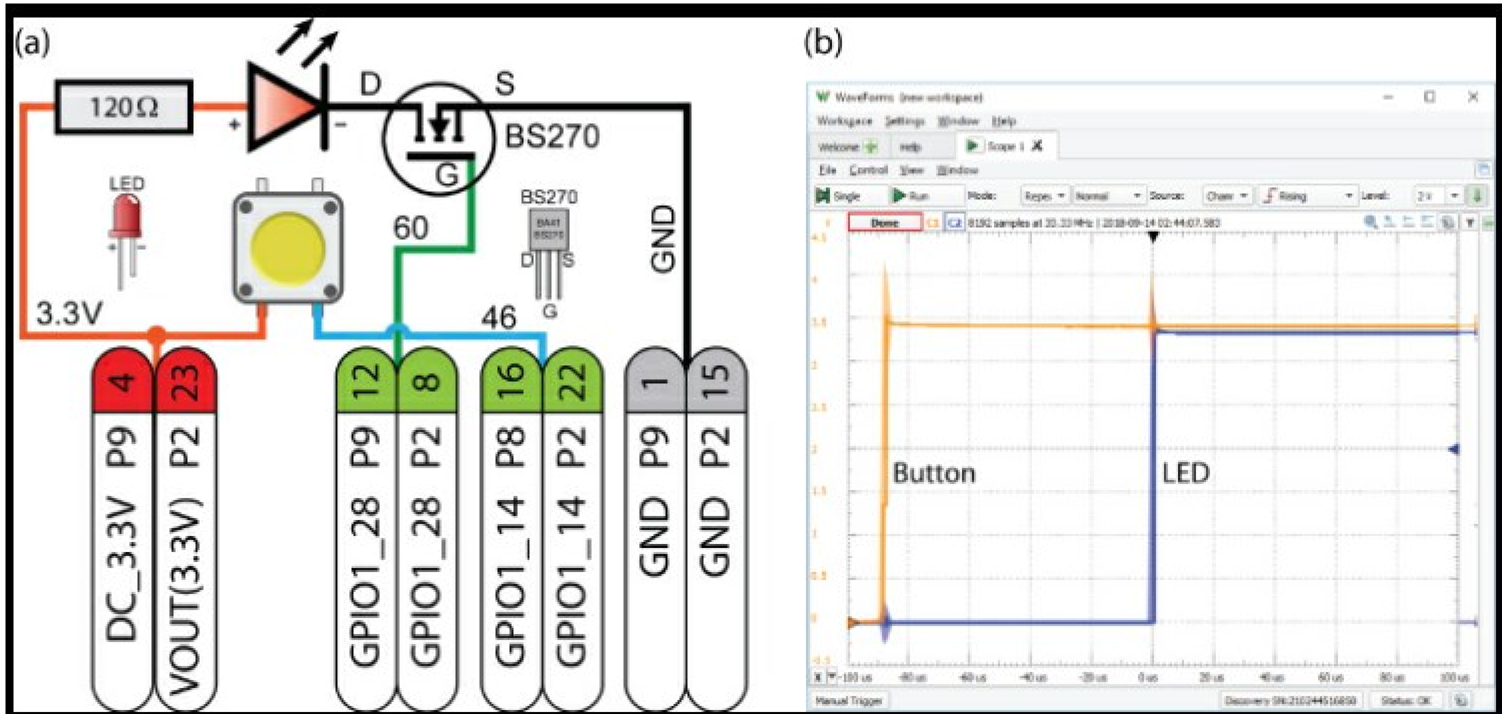

Figure 16-2: (a) An LED and pushbutton circuit for testing the GPIO LKM using the BBB and PocketBeagle, (b) LKM performance results (with software debouncing disabled)

Interrupt refresher

An interrupt does exactly what it states on the label.
“Interrupt what you are doing and do that shiny thing over there”

An interrupt

- stops the current process
- saves the current system state
- branches to an interrupt service routine (ISR)
- executes the ISR code
- reloads the previous system state
- continues the process execution

Enabling an IRQ

To enable an IRQ on a GPIO pin, we need to do the following:

- Configure the GPIO hardware as an interrupt, returns an IRQ number.
- Request an interrupt from Linux and associate the IRQ number with an ISR handler function.
- Write the ISR handler function code.

GPIO via kernel space

gpio.h contains some functions that are useful for developing our LKM.

```
static inline bool gpio_is_valid(int number) // Check if a GPIO number is valid
static inline int  gpio_request(unsigned gpio, const char *label) // Request access to the GPIO
static inline int  gpio_direction_input(unsigned gpio) // Set the GPIO to be an input
static inline int  gpio_get_value(unsigned gpio) // Obtain the valid of the GPIO
static inline int  gpio_direction_output(unsigned gpio, int value) // Set the GPIO to be an output
static inline int  gpio_set_debounce(unsigned gpio, unsigned debounce) // Enable software debounce in milliseconds to the GPIO
static inline void gpio_free(unsigned gpio) // release the IRQ associated with the GPIO
static inline int  gpio_to_irq(unsigned gpio) // obtain an IRQ for the GPIO (returns IRQ number)
```

Requesting an IRQ

```
result = request_irq(irqNumber,        // the interrupt number
  (irq_handler_t) ebb_gpio_irq_handler,// pointer to the handler
  IRQF_TRIGGER_RISING,                 // interrupt on rising edge
  "ebb_gpio_handler",                  // used to identify the owner
  NULL);                 // *dev_id for shared interrupt lines, NULL
```

- irqNumber => return value from gpio_to_irq()
- ebb_gpio_irq_handler => ISR function to call on interrupt
- IRQF_TRIGGER_RISING => The type of trigger event (interrupt.h)
- “ebb_gpio_handler” => string identifier in /proc/interrupts
- NULL => the device id that identifies the interrupt source.

Interrupt Service Routine format

```
static irq_handler_t ebb_gpio_irq_handler(unsigned int irq,
                                          void *dev_id, struct pt_regs *regs){
  // perform shiny interrupty things
}
```

- irq is the IRQ associated with the GPIO
- dev_id is the device identifier that caused the interrupt
- regs are hardware specific register values used for debugging.

- Return IRQ_HANDLED if successful – return IRQ_NONE otherwise.

Sysfs interaction

If we want a set and forget system, then the interrupt lines above are enough to interact with an example LKM with interrupts enabled.

What if we want to change the interrupt or LKM parameters while it is running?

We can use the kObject interface to interact with the kernel module.

This allows us to configure the kernel module from our applications!

kobjects – ktypes - ksets

A kobject gives us an entry in /sys/ which allows us to define attributes and pass information between /sys/ and the kernel.

A kobject is not useful on its own, it requires being embedded within other data structures to control access.

A ktype is the type of the object the kobject is embedded within. It controls what happens when the kobject is created and destroyed.

A kset is a group of kobjects that can be of different ktypes. A set of kobjects can be thought of as a sysfs directory with kobject subdirectories.

kobjects – ktypes - ksets

- Create a kobject located in /sys/ebb/
- Create an attribute group /sys/ebb/gpio
- Create attributes in the group
  /sys/ebb/gpio/debounce
  /sys/ebb/gpio/count
  /sys/ebb/gpio/ledon
  /sys/ebb/gpio/time
- Then, with the LKM loaded, can test in the terminal
  cat /sys/ebb/gpio/count

kobjects – ktypes - ksets

- Lets create a kobject

```
ebb_kobj = kobject_create_and_add("ebb", kernel_kobj->parent); // create object
```

- Give the kobject some attributes

```
result = sysfs_create_group(ebb_kobj, &attr_group); // associate attributes with that object
```

kobjects – ktypes - ksets

- Define the attribute group

```
static struct attribute_group attr_group = {
      .name  = gpioName,      // the name generated in ebb_button_init()
      .attrs = ebb_attrs,      // the attributes array defined just below
};
```

- Define an array of attributes

```
static struct attribute *ebb_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,
};
```

kobjects – ktypes - ksets

- Associate handlers for sysfs interaction

```
static struct kobj_attribute count_attr = __ATTR(numberPresses, 0664, numberPresses_show,
numberPresses_store);
```

- Create the show and store handlers

```
static ssize_t numberPresses_show(struct kobject *kobj, struct kobj_attribute *attr, char *buf) {
  return sprintf(buf, "%d\n", numberPresses);
}
static ssize_t numberPresses_store(struct kobject *kobj, struct kobj_attribute *attr, const char *buf,
size_t count) {
  sscanf(buf, "%du", &numberPresses);
  return
```

Examples

- http://derekmolloy.ie/kernel-gpio-programming-buttons-and-leds/
- https://github.com/derekmolloy/exploringBB/tree/version2/chp16

Summary

We’ve had a brief introduction into using interrupts with a LKM

- We’ve outlined how you would use interrupts in a LKM on the BBB
- We’ve outlined how you would interact between a LKM and sysfs. This allows a user mode application to interact with the LKM.