```
**********************************************************************
/

 #include <linux/cdev.h>
 #include <linux/device.h>
 #include <linux/fs.h>
 #include <linux/gpio.h>
 #include <linux/module.h>
 #include <linux/sched.h>
 #include <linux/version.h>
 #include <linux/uaccess.h>

 static int gpio_in = 11; // default GPIO
 module_param(gpio_in, int, 0644);

 static dev_t          example_dev;
 static struct cdev    example_cdev;
 static struct class * example_class = NULL;

 /* FIX: correct prototype */
 static ssize_t example_read(struct file * filp, char __user * buffer, size_t length, loff_t *
offset);

 static const struct file_operations fops_example = {
    .owner   = THIS_MODULE,
    .read    = example_read,
 };

 static int __init example_init(void)
 {
```

```
int err;

if ((err = gpio_request(gpio_in, THIS_MODULE->name)) != 0) {
    return err;
}

if ((err = gpio_direction_input(gpio_in)) != 0) {
    gpio_free(gpio_in);
    return err;
}

if ((err = alloc_chrdev_region(&example_dev, 0, 1, THIS_MODULE->name)) < 0) {
    gpio_free(gpio_in);
    return err;
}

example_class = class_create(THIS_MODULE, "class_example");
if (IS_ERR(example_class)) {
    unregister_chrdev_region(example_dev, 1);
    gpio_free(gpio_in);
    return PTR_ERR(example_class);
}

device_create(example_class, NULL, example_dev, NULL, THIS_MODULE->name);

cdev_init(&example_cdev, &fops_example);

if ((err = cdev_add(&example_cdev, example_dev, 1)) != 0) {
    device_destroy(example_class, example_dev);
```

```
        class_destroy(example_class);
        unregister_chrdev_region(example_dev, 1);
        gpio_free(gpio_in);
        return err;
    }

    pr_info("GPIO input module loaded, gpio=%d\n", gpio_in);
    return 0;
}

static void __exit example_exit(void)
{
    cdev_del(&example_cdev);
   device_destroy(example_class, example_dev);
    class_destroy(example_class);
    unregister_chrdev_region(example_dev, 1);
    gpio_free(gpio_in);
    pr_info("GPIO input module unloaded\n");
}

/* FIX: correct return type + __user qualifier */
static ssize_t example_read(struct file * filp, char __user * buffer, size_t length, loff_t *
offset)
{
    char chaine[32];
    int lg;

    snprintf(chaine, sizeof(chaine), "%d\n", gpio_get_value(gpio_in));
    lg = strlen(chaine);
```

```
    if (*offset > 0)  // EOF handling
        return 0;

    if (lg > length)
        return -EINVAL;

    if (copy_to_user(buffer, chaine, lg))
        return -EFAULT;

    *offset += lg;
    return lg;
}

module_init(example_init);
module_exit(example_exit);

MODULE_LICENSE("GPL");
```

1. Header Files

```
15 #include <linux/cdev.h>
16 #include <linux/device.h>
17 #include <linux/fs.h>
18 #include <linux/gpio.h>
19 #include <linux/module.h>
20 #include <linux/sched.h>
21 #include <linux/version.h>
22 #include <linux/uaccess.h>
```

Explanation:

- cdev.h → Provides support for **character devices** (struct cdev).
- device.h → Kernel **device model** functions (e.g., class_create, device_create).
- fs.h → File operations (struct file_operations).
- gpio.h → GPIO access functions (gpio_request, gpio_direction_input, gpio_get_value).
- module.h → Required for every kernel module (module_init, module_exit).
- sched.h → Process scheduling info (not heavily used here, sometimes for TASK_* macros).
- version.h → To check kernel version for conditional compilation.
- uaccess.h → Functions to safely copy data between **user space and kernel space** (copy_to_user).

These headers form the backbone of any **kernel driver** that exposes GPIOs to user space.

---

2. Module Parameters & GPIO Default

33 static int gpio_in = 11; // default GPIO

34 module_param(gpio_in, int, 0644);

Explanation:

- Sets **GPIO pin 11** as default input.
- module_param allows you to override gpio_in from **insmod**, e.g.,
- sudo insmod gpio-input.ko gpio_in=49
- 0644 → permission for /sys/module/<modulename>/parameters/gpio_in.

Relation to BBB:

- On BBB, GPIOs are mapped using GPIOx_y = x*32 + y. For example, GPIO1_17 = 1*32 + 17 = 49.

---

3. Device Variables

36 static dev_t example_dev;

37 static struct cdev example_cdev;

38 static struct class * example_class = NULL;

Explanation:

- dev_t → Represents **device number** (major + minor).
- struct cdev → Character device structure.
- struct class * → Device class for **sysfs entries** (/dev/class_example).

This allows user-space apps to open /dev/<device> and read GPIO values.

---

4. File Operations Prototype

41 static ssize_t example_read(struct file * filp, char __user * buffer, size_t length, loff_t * offset);

- Prototype for the read function exposed to user-space apps (read() syscall).

43 static const struct file_operations fops_example = {

44 .owner = THIS_MODULE,

45 .read = example_read,

46 };

- fops_example links **user-space operations** to **kernel functions**.
- Only .read is implemented; write/open/ioctl are omitted.

In **U-Boot**, this is analogous to providing **command handlers**, but U-Boot does not have struct file_operations. Kernel drivers need this for user-space interaction.

---

5. Module Initialization

48 static int __init example_init(void)

49 {

50 int err;

- __init → Function discarded from memory after initialization to save RAM.
- err → variable to hold error codes.

5.1 GPIO Request & Set Direction

52 if ((err = gpio_request(gpio_in, THIS_MODULE->name)) != 0) {

53 return err;

54 }

- Requests exclusive access to GPIO pin.

- Fails if another driver already owns it.

```
56 if ((err = gpio_direction_input(gpio_in)) != 0) {
57     gpio_free(gpio_in);
58     return err;
59 }
```

- Sets GPIO as **input**.
- If fails, **frees GPIO** to avoid resource leak.

BBB Relation: Before reading GPIO pins connected to buttons or sensors, you must **request them in kernel** (not needed in U-Boot; U-Boot directly writes registers).

5.2 Allocate Device Number

```
61 if ((err = alloc_chrdev_region(&example_dev, 0, 1, THIS_MODULE->name)) < 0) {
62     gpio_free(gpio_in);
63     return err;
64 }
```

- Allocates **dynamic major number** for character device.
- 0 → minor number start, 1 → count of devices.

5.3 Create Device Class & Device Node

```
66 example_class = class_create(THIS_MODULE, "class_example");
67 if (IS_ERR(example_class)) {
68     unregister_chrdev_region(example_dev, 1);
69     gpio_free(gpio_in);
70     return PTR_ERR(example_class);
71 }


73 device_create(example_class, NULL, example_dev, NULL, THIS_MODULE->name);
```

- Creates /sys/class/class_example/ in **sysfs**.
- device_create makes /dev/gpio-input node automatically.

BBB Relation: This allows user-space apps (Python/C) to **open /dev/gpio-input and read values**, unlike U-Boot where you poll registers.

5.4 Initialize and Add cdev

```
75 cdev_init(&example_cdev, &fops_example);


77 if ((err = cdev_add(&example_cdev, example_dev, 1)) != 0) {
78     device_destroy(example_class, example_dev);
79     class_destroy(example_class);
80     unregister_chrdev_region(example_dev, 1);
81     gpio_free(gpio_in);
82     return err;
83 }
```

- Registers **character device** with the kernel.
- Links the device number to fops_example.

5.5 Module Load Log

```
85 pr_info("GPIO input module loaded, gpio=%d\n", gpio_in);
86 return 0;
```

- Prints message to **dmesg** when loaded.

---

6. Module Exit

```
 static void __exit example_exit(void)
 {
    cdev_del(&example_cdev);
    device_destroy(example_class, example_dev);
    class_destroy(example_class);
    unregister_chrdev_region(example_dev, 1);
    gpio_free(gpio_in);
```

```
    pr_info("GPIO input module unloaded\n");
}
```

- Cleans up everything:
    - Deletes cdev
    - Removes device node
    - Destroys class
    - Frees device number
    - Frees GPIO
- Important to prevent **resource leaks** when module is removed.

Analogous in **U-Boot**: You would disable GPIO clocks or cleanup memory, though U-Boot rarely has dynamic cleanup.

---

7. Read Function

```
static ssize_t example_read(struct file * filp, char __user * buffer, size_t length, loff_t * offset)
{
    char chaine[32];
    int lg;
    snprintf(chaine, sizeof(chaine), "%d\n", gpio_get_value(gpio_in));
    lg = strlen(chaine);
    if (*offset > 0)  // EOF handling
        return 0;
    if (lg > length)
        return -EINVAL;
    if (copy_to_user(buffer, chaine, lg))
        return -EFAULT;
    *offset += lg;
    return lg;
}
```

Step-by-step:

1. gpio_get_value(gpio_in) → Reads **current value** of GPIO (0 or 1).
2. snprintf() → Converts to string for user-space read.
3. Checks for **EOF**: If offset > 0, returns 0.
4. Checks if **buffer is large enough**.
5. Copies value to user-space with copy_to_user.
6. Updates offset and returns number of bytes read.

BBB Relation: User-space apps can do:

fd = open("/dev/gpio-input", O_RDONLY);

read(fd, buf, sizeof(buf));

- This reads the **real-time GPIO pin** state.

---

8. Module Macros

121 module_init(example_init);

122 module_exit(example_exit);

124 MODULE_LICENSE("GPL");

- Registers **init** and **exit** functions.
- Declares **GPL license**, required for kernel symbols.

---

9. Full Functionality Layout (BBB Perspective)

Flow:

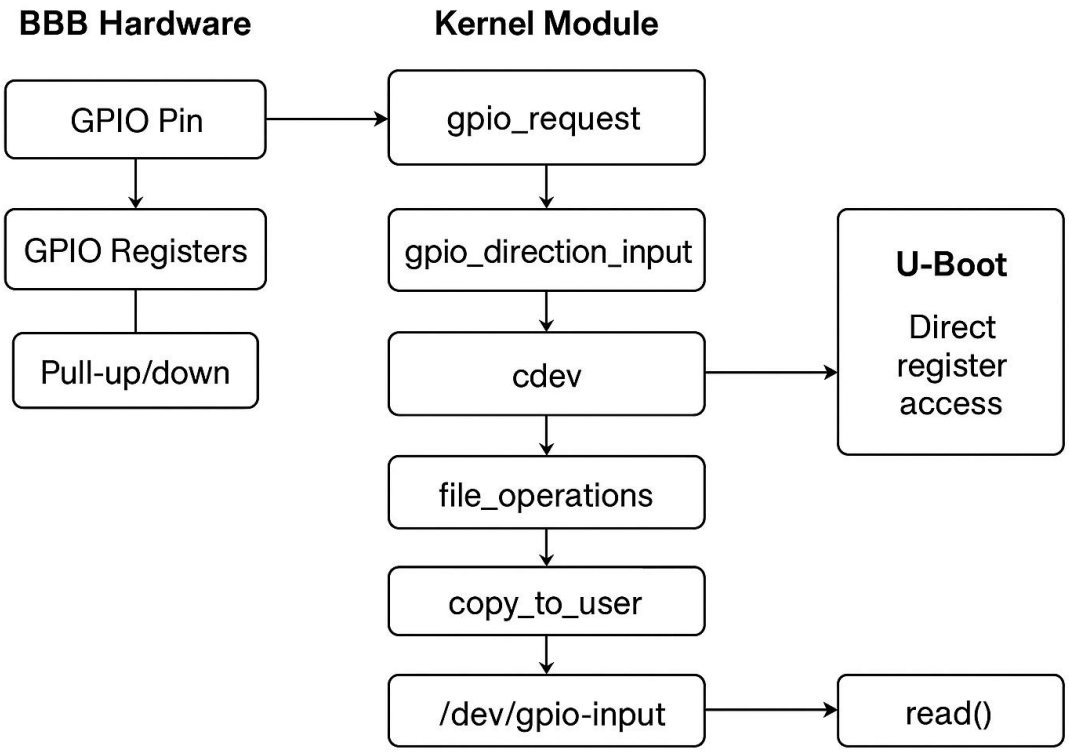
1. **Module load (insmod)**
    - Request GPIO pin.
    - Configure as input.
    - Allocate device number.
    - Create sysfs class + device.
    - Register cdev and link file_operations.
2. **User-space read**
    - Open /dev/<device> → triggers example_read.
    - example_read reads GPIO value.

    - Copies value to user buffer.
3. **Module unload (rmmod)**
    - Cleanup all resources: cdev_del, device_destroy, gpio_free.

BBB hardware relation:

- GPIO registers are mapped in memory (gpio_get_value() abstracts register access).
- User-space apps see /dev/gpio-input.
- In **U-Boot**, you directly manipulate GPIO registers (__raw_writel) without device model.

Kernel Concept	**Function in This Driver**	**U-Boot Equivalent**
gpio_request	Claim GPIO ownership	Direct register use (no ownership)
gpio_direction_input	Configure GPIO direction	Manual register config
cdev	Character device object	Not present
file_operations	Links user-space read/write	Command handlers (do_gpio)
class_create / device_create	Creates /dev node	Not present
copy_to_user	User-space safe transfer	Not needed, no userspace
module_init / module_exit	Load/unload kernel module	board_init or main() in U-Boot




```
#include <linux/cdev.h>
 #include <linux/device.h>
 #include <linux/fs.h>
 #include <linux/gpio.h>
 #include <linux/module.h>
 #include <linux/sched.h>
 #include <linux/version.h>
 #include <linux/timer.h>
 #include <linux/moduleparam.h>
 #include <asm/uaccess.h>

 static void timer_gpio(void);
 //static struct timer_list timer;
```

```
static int gpio_out=10;

module_param(gpio_out,int,0);

static int init_gpio (void)
{
  int err;

  if ((err = gpio_request(gpio_out, THIS_MODULE->name)) != 0) {
    return err;
  }
  if ((err = gpio_direction_output(gpio_out, 1)) != 0) {
    gpio_free(gpio_out);
    return err;
  }

  timer_gpio();
 // init_timer (& timer);
 // timer.function = timer_gpio;
  //timer.data = 0;
 // timer.expires = jiffies + 1*HZ;
 // add_timer(&timer);

  return 0;
}

static void exit_gpio (void)
{
 // del_timer(& timer);
```

```
  gpio_free(gpio_out);
}
static void timer_gpio()
{
  static int value = 0;
  int i=0;
  for(i=0;i<10;i++)
  {
  printk (KERN_INFO "my_sender_thread sent a message at jiffies=%ld\n", jiffies);
  set_current_state(TASK_INTERRUPTIBLE);
  schedule_timeout(1*HZ);
  gpio_set_value(gpio_out, value);
  value = 1 - value;
  //mod_timer(& timer, jiffies + 1*HZ);
  }
}
module_init(init_gpio);
module_exit(exit_gpio);
MODULE_LICENSE("GPL");
```

1️⃣ Header Files (Lines 16–25)

```
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/gpio.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/version.h>
#include <linux/timer.h>
```

#include <linux/moduleparam.h>

#include <asm/uaccess.h>

- linux/cdev.h → Char device support (not actually used here but often included in device drivers).
- linux/device.h → Device model interface (creates /dev nodes, classes).
- linux/fs.h → File operations, needed if you implement read/write.
- linux/gpio.h → GPIO kernel API (request, set direction, set/get value).
- linux/module.h → Required for **loadable kernel modules** (module_init/module_exit).
- linux/sched.h → Task state macros (TASK_INTERRUPTIBLE).
- linux/version.h → Kernel version checks.
- linux/timer.h → Kernel timers (struct timer_list).
- linux/moduleparam.h → Allow passing module parameters at insmod.
- asm/uaccess.h → User-space memory access (copy_to_user), not used in this module.

Relation to BBB/Kernels:

- This is a **Linux kernel module**, not U-Boot. U-Boot only initializes hardware and loads the kernel. GPIO access in U-Boot is limited; here, you are **manipulating GPIOs from the running Linux kernel**.

2️⃣ Global Variables and Module Parameter (Lines 27–32)

static void timer_gpio(void);

//static struct timer_list timer;

static int gpio_out=10;

module_param(gpio_out,int,0);

- gpio_out → GPIO pin number to control. Default = 10.
- module_param → Lets you override the pin number when loading the module:
- sudo insmod gpio-output.ko gpio_out=11
- timer_gpio → function that toggles GPIO value periodically.
- struct timer_list timer; → commented out, indicates intention to use kernel timers.

Relation to BBB:

- This pin number refers to the **BeagleBone Black's GPIO mapping** (e.g., GPIO0_10 = P8_32). Linux GPIO numbers differ from physical pins.

3️⃣ Module Initialization (Lines 34–54)

```
static int init_gpio(void)
{
   int err;
   if ((err = gpio_request(gpio_out, THIS_MODULE->name)) != 0) {
      return err;
   }
   if ((err = gpio_direction_output(gpio_out, 1)) != 0) {
      gpio_free(gpio_out);
      return err;
   }
   timer_gpio(); // calls function to toggle GPIO
   // init_timer (& timer);
   // timer.function = timer_gpio;
   // timer.data = 0;
   // timer.expires = jiffies + 1*HZ;
   // add_timer(&timer);

   return 0;
}
```

Step by step:

1. gpio_request(gpio_out, THIS_MODULE->name)
    - Reserves the GPIO pin for this module. Prevents conflicts with other drivers.
2. gpio_direction_output(gpio_out, 1)
    - Configures the GPIO as output and sets initial value = HIGH.
3. timer_gpio()

- Immediately calls the function to toggle GPIO (currently **blocking**).
- Commented out timer code indicates a **non-blocking timer-based approach** was planned (better practice).

BBB Relation:

- At this stage, Linux kernel has already been booted by U-Boot.
- This module runs in **kernel space**, controlling BBB hardware registers for GPIO0_10 via the GPIO driver in the kernel.

4️⃣ Module Exit (Lines 56–60)

```
static void exit_gpio(void)
{
   // del_timer(& timer);
   gpio_free(gpio_out);
}
```

- gpio_free(gpio_out) → Releases GPIO when module is removed.
- Commented timer deletion → would cancel periodic toggling if timer were used.

Functional meaning: ensures no resources are held after the module is removed.

5️⃣ Timer GPIO Function (Lines 62–76)

```
static void timer_gpio()
{
   static int value = 0;
   int i=0;
   for(i=0;i<10;i++)
   {
      printk(KERN_INFO "my_sender_thread sent a message at jiffies=%ld\n", jiffies);
      set_current_state(TASK_INTERRUPTIBLE);
      schedule_timeout(1*HZ);
```

```
        gpio_set_value(gpio_out, value);
        value = 1 - value;
        //mod_timer(& timer, jiffies + 1*HZ);
    }
}
```

- **Purpose:** Toggle GPIO 10 **10 times**, with 1 second delay between toggles.
- printk → Logs current jiffies to kernel log.
- set_current_state(TASK_INTERRUPTIBLE) + schedule_timeout(1*HZ) → sleeps current kernel thread **without busy waiting**.
- gpio_set_value(gpio_out, value) → Sets GPIO HIGH or LOW.
- value = 1 - value → toggles state.

Important Notes:

- Currently blocking in module init → not ideal. Should use kernel timers or a kernel thread.
- Commented mod_timer → intended to create a **repeating timer**, which is a better approach for production.

6️⃣ Module Registration (Lines 78–80)

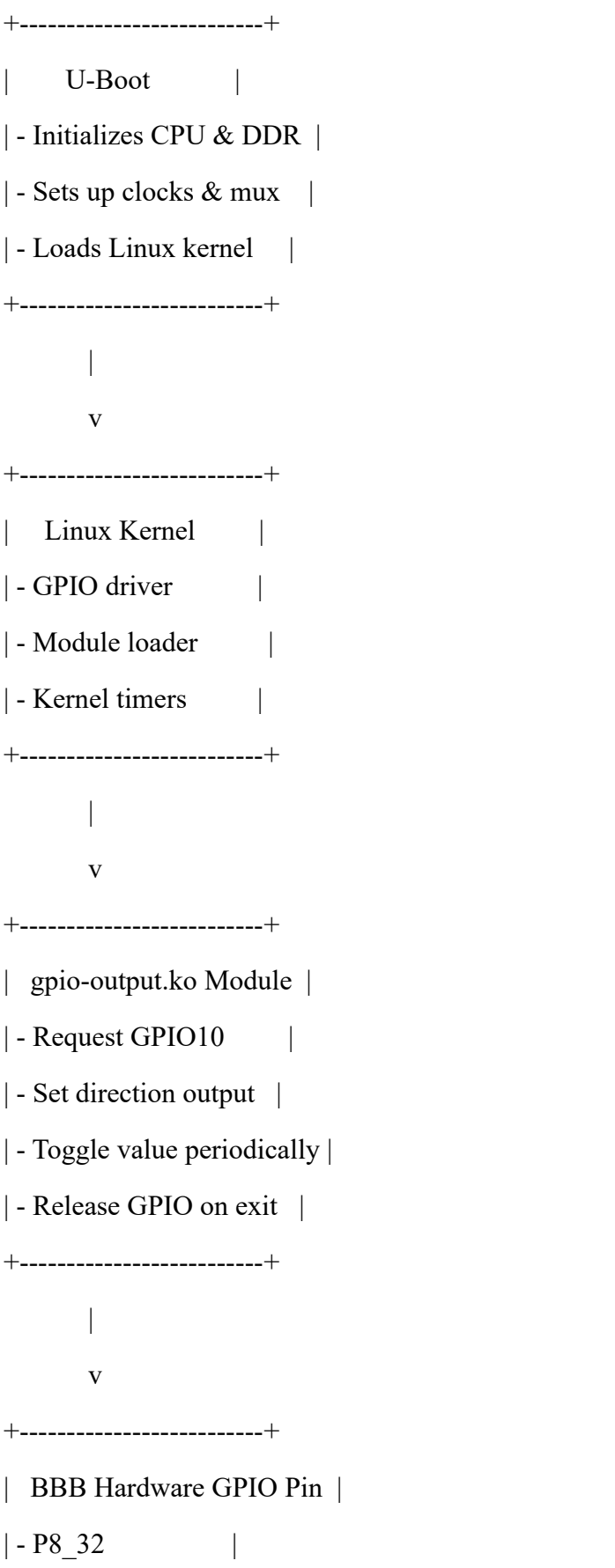
```
module_init(init_gpio);
module_exit(exit_gpio);
MODULE_LICENSE("GPL");
```

- module_init(init_gpio) → called when insmod loads module.
- module_exit(exit_gpio) → called when rmmod unloads module.
- MODULE_LICENSE("GPL") → tells kernel module is GPL licensed (needed for accessing symbols).

Relation to BBB:

- These hooks run **inside kernel space**, after U-Boot has initialized DDR, clocks, and memory mapping.
- This is how you safely control GPIO pins without touching hardware registers manually.

7️⃣ Functional Layout Diagram (BBB Perspective)



| - Connected to LED/Relay |

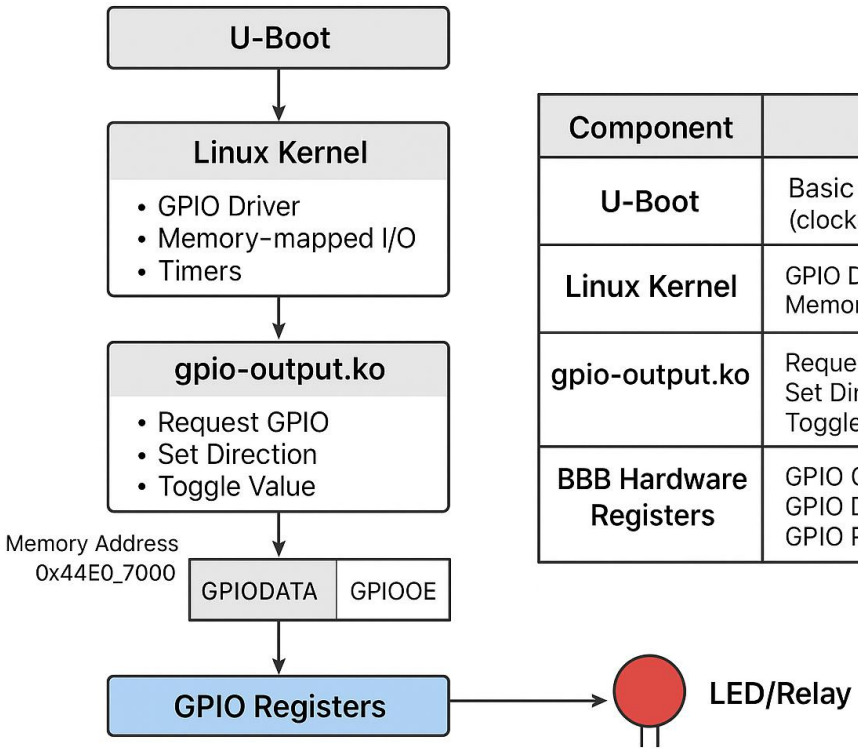
+-------------------------+

Flow Explanation:

1. **U-Boot** runs first → sets up basic hardware.
2. **Linux kernel** boots → initializes GPIO subsystem.
3. **gpio-output. ko** loaded → requests GPIO10, toggles output using timer or loop.
4. **BBB hardware** reacts → LED turns ON/OFF, relay toggles, etc.

Summary of Functionality

Feature	Implementation in Code	Kernel/BBB Relation
GPIO reservation	gpio_request ()	Ensures no conflict in Linux GPIO driver
GPIO direction setup	gpio_direction_output ()	Configures pin as output
GPIO toggling	gpio_set_value ()	Writes to hardware register
Delay between toggles	schedule_timeout(1*HZ)	Sleeps kernel thread without busy wait
Periodic callback (planned)	struct timer_list + mod_timer()	Repeats action asynchronously
Logging	printk()	Kernel log (viewable via dmesg)
Module load/unload hooks	module_init()/module_exit()	Entry/exit points in kernel



Component	Function
U-Boot	Basic initialization (clocks, DDR, pin muxing)
Linux Kernel	GPIO Driver Memory-mapped I/O
gpio-output.ko	Request GPIO Set Direction Toggle Value
BBB Hardware Registers	GPIO Control Registers GPIO Data Registers GPIO Pin Outputs

```
#include <linux/fs.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/module.h>
#include <linux/moduleparam.h>

int gpio_in=11;
int gpio_out=10;
module_param(gpio_in,int,0);
module_param(gpio_out,int,0);
irqreturn_t handler_irq_gpio(int irq, void * ident)
{
 static int value = 1;
 gpio_set_value(gpio_out, value);
 printk ("ISR\n");
```

```
  value = 1 - value;
  return IRQ_HANDLED;
}
static int gpio_irq = 0;
int example_init (void)
{
  int err;
  gpio_irq = gpio_to_irq(gpio_in);
   printk("interrupt request number:%d",gpio_irq);
  if ((err = gpio_request(gpio_out, THIS_MODULE->name)) != 0)
    return err;
  if ((err = gpio_direction_output(gpio_out, 0)) != 0) {
    gpio_free(gpio_out);
    return err;
  }
  if ((err = gpio_request(gpio_in, THIS_MODULE->name)) != 0) {
    gpio_free(gpio_out);
    return err;
  }
  if ((err = gpio_direction_input(gpio_in)) != 0) {
    gpio_free(gpio_in);
    gpio_free(gpio_out);
    return err;
  }
  if ((err = request_irq(gpio_irq, handler_irq_gpio, 0  , THIS_MODULE->name,
THIS_MODULE->name)) != 0) {
    gpio_free(gpio_in);
    gpio_free(gpio_out);
    return err;
```

```
  }
 irq_set_irq_type(gpio_irq, IRQF_TRIGGER_FALLING);
  return 0;
 }
 void example_exit (void)
 {
  free_irq(gpio_irq, THIS_MODULE->name);
  gpio_free(gpio_in);
  gpio_free(gpio_out);
 }
 module_init(example_init);
 module_exit(example_exit);
 MODULE_LICENSE("GPL");
```

Header Files

```
16 #include <linux/gpio.h>       // Provides GPIO API (request, free, set, get)
17 #include <linux/interrupt.h>  // Provides interrupt handling API
18 #include <linux/irq.h>        // Provides irq number related functions
19 #include <linux/module.h>     // Required for all kernel modules
20 #include <linux/moduleparam.h>// Allows passing parameters at module load time
```

- These headers are standard for Linux kernel GPIO and interrupt drivers.
- They give access to functions like gpio_request(), gpio_set_value(), request_irq(), etc.

GPIO Parameters

```
23 int gpio_in=11;
24 int gpio_out=10;
25 module_param(gpio_in,int,0);
26 module_param(gpio_out,int,0);
```

- gpio_in → Input GPIO number (default 11, can be changed at module load time)

- gpio_out → Output GPIO number (default 10, can be changed at module load time)
- module_param → Makes these configurable when loading the module using insmod.

Example:

insmod gpio_irq.ko gpio_in=49 gpio_out=60

Interrupt Service Routine

```
29 irqreturn_t handler_irq_gpio(int irq, void * ident)
30 {
31   static int value = 1;
32   gpio_set_value(gpio_out, value);
33   printk ("ISR\n");
34   value = 1 - value;
35   return IRQ_HANDLED;
36 }
```

- **Line 31:** static int value = 1; → Keeps track of output state across interrupts.
- **Line 32:** Sets the output GPIO to value (toggles LED/relay, etc.).
- **Line 33:** Prints a kernel log for ISR call.
- **Line 34:** Toggles value between 0 and 1.
- **Line 35:** Returns IRQ_HANDLED to indicate the interrupt was handled.

Global Variables

```
38 static int gpio_irq = 0;
```

- Stores the IRQ number mapped to gpio_in.

Module Initialization

```
40 int example_init (void)
41 {
42   int err;
```

```
44  gpio_irq = gpio_to_irq(gpio_in);
46  printk("interrupt request number:%d",gpio_irq);
```

- gpio_to_irq(gpio_in) → Converts a GPIO number to the IRQ number used by the kernel.
- printk logs the IRQ number to dmesg.

Request and Configure GPIOs

```
48  if ((err = gpio_request(gpio_out, THIS_MODULE->name)) != 0)
49    return err;
50  if ((err = gpio_direction_output(gpio_out, 0)) != 0) {
51    gpio_free(gpio_out);
52    return err;
53  }
```

- Requests control over gpio_out.
- Sets it as output, initially LOW (0).
- Frees GPIO if there's an error.

```
55  if ((err = gpio_request(gpio_in, THIS_MODULE->name)) != 0) {
56    gpio_free(gpio_out);
57    return err;
58  }
59  if ((err = gpio_direction_input(gpio_in)) != 0) {
60    gpio_free(gpio_in);
61    gpio_free(gpio_out);
62    return err;
63  }
```

- Requests control over gpio_in.
- Sets it as input (button/switch).
- Cleans up on failure.

Request Interrupt

```
65   if ((err = request_irq(gpio_irq, handler_irq_gpio, 0  , THIS_MODULE->name,
THIS_MODULE->name)) != 0) {
66     gpio_free(gpio_in);
67     gpio_free(gpio_out);
68     return err;
69   }
70 irq_set_irq_type(gpio_irq, IRQF_TRIGGER_FALLING);
```

- request_irq() → Binds gpio_irq to handler_irq_gpio().
- IRQF_TRIGGER_FALLING → Interrupt triggers on falling edge (high → low).
- Kernel now calls handler_irq_gpio() whenever input GPIO changes.

Module Exit

```
76 void example_exit (void)
77 {
78   free_irq(gpio_irq, THIS_MODULE->name);
79   gpio_free(gpio_in);
80   gpio_free(gpio_out);
81 }
```

- Frees IRQ and GPIOs on module removal (rmmod).

Module Macros

```
83 module_init(example_init);
84 module_exit(example_exit);
85 MODULE_LICENSE("GPL");
```

- Registers init and exit functions with kernel.
- Declares GPL license.

Functionality Summary

1. Module initializes:
    - Requests control of input (gpio_in) and output (gpio_out) pins.
    - Configures input as GPIO interrupt and output as GPIO output.
2. Converts gpio_in to IRQ number.
3. Registers ISR for falling edge interrupts.
4. When input pin falls (e.g., button pressed):
    - ISR toggles output pin state.
    - Logs "ISR" to kernel log.
5. On module removal, resources are cleaned.

Relation to BBB, Kernel, and U-Boot

Layer	Role in this Driver
BBB Hardware	GPIO pins physically connected to LED/button. Interrupt controller detects edge transitions.
Device Tree (DTS)	Maps physical pins to kernel GPIO numbers. Must match gpio_in/gpio_out.
Linux Kernel GPIO Subsystem	Provides APIs: gpio_request, gpio_set_value, gpio_to_irq. Manages pin muxing and interrupt routing.
ISR	Registered in kernel via request_irq. Runs in interrupt context, toggles output pin.
U-Boot	Bootloader. Does **not** run this driver. Only initializes SoC and loads kernel. GPIO states may be manipulated in U-Boot if commands like gpio set are used.
User-space	Can access GPIO via sysfs or devfs, but this module handles GPIO at kernel level using IRQs.

Driver Flow Diagram (Text Version)

BBB Hardware (GPIO pins)

↓

Interrupt Controller

↓

Linux Kernel GPIO/IRQ Subsystem

↓

gpio_to_irq() → maps GPIO to IRQ

↓

request_irq() → ISR registration

↓

handler_irq_gpio() executes on falling edge

↓

gpio_set_value(gpio_out, value) → toggles output

↓

User sees LED blink (hardware effect)

Key Points

- The module **does not interact with U-Boot**, only kernel handles it.
- ISR executes in **interrupt context**, so keep it fast.
- module_param allows changing GPIO numbers without recompiling.
- Proper cleanup prevents kernel panics due to resource leaks.