Embedded Linux Device Drivers

Agenda

- Direct Hardware access revision
- Interrupt handling - Top half
- Interrupt handling - Bottom half
- Soft IRQs
- Tasklets
- Work queues
- USB introduction

Interrupt Context vs Process Context

1. Interrupt Context

When it runs:

- Triggered by hardware interrupts (GPIO, timers, peripherals)
- Invoked asynchronously by CPU's interrupt controller

Key Characteristics:

- Per processor one page stack (4 KB)
- Shared stack for all interrupts (on that processor)

```
in_irq()      // Returns true in interrupt context
in_interrupt() // True for any interrupt/bottom-half context
```

Constraints:

- **No direct access to user space** (no user memory, no current->)
- **Cannot sleep or block**:
  - No `mutex_lock()` (use `spin_lock()` instead)
  - No `kmalloc(GFP_KERNEL)` (only `GFP_ATOMIC`)
  - No `msleep()` (use `udelay()` for short waits)
- **Stack limited** (~4KB on ARM, shared across interrupts)

Typical Uses:

- Acknowledge hardware
- Read critical status
- Schedule bottom half
- Simple register operations

2. Process Context

When it runs:

- Regular kernel code (system calls, kernel threads)
- Scheduled by kernel's process scheduler

Key Characteristics:

- Per process/thread one two-page stack (8 KB).

```
current // Valid pointer to task_struct
in_task() // Returns true in process context
```

Advantages:

- **Full kernel API available**:

  - Can sleep (`mutex_lock()`, `msleep()`)
  - Can access user space (`copy_to_user()`)
  - Memory allocation with `GFP_KERNEL`
- **Larger stack** (typically 8-16KB per thread)

Typical Uses:

- System call implementations
- Driver ioctl() handlers
- Complex processing deferred from interrupts

3. Critical Comparison Table

Feature	Interrupt Context	Process Context
Trigger	Hardware event	System call/scheduler
Preemption	Never	Possible
Stack	Small, shared	Per-process, larger
Sleep operations	Forbidden	Allowed
User memory access	Impossible	Possible
`current` pointer	Invalid	Valid
Latency requirements	Ultra-low (~μs)	Relaxed (~ms)

4. Practical Implications for BBB Drivers

GPIO Interrupt Example:

```
// BAD (in interrupt context):
static irqreturn_t bad_isr(int irq, void *dev)
{
```

```c
    mutex_lock(&lock); // WILL CAUSE KERNEL PANIC
    kmalloc(sizeof(buf), GFP_KERNEL); // WILL SLEEP
}

// GOOD:
static irqreturn_t good_isr(int irq, void *dev)
{
    spin_lock(&lock); // Atomic lock
    schedule_work(&deferred_work); // Push to process context
    return IRQ_HANDLED;
}
```

Interrupt Handling in Linux Kernel

1. Overview of Interrupt Handling

Key Concepts:

- **Interrupt Context**: Interrupt handler (ISR) always runs in interrupt context.
- **Top Half**: Time-critical part (minimal work)
- **Bottom Half**: Deferred processing (we'll cover later)
- **IRQ Numbers**: Hardware interrupt lines (view with `cat /proc/interrupts`)

2. Core APIs for Interrupt Handling

Essential Functions:

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

// Request interrupt
int request_irq(unsigned int irq, irq_handler_t handler,
                unsigned long flags, const char *name, void *dev);
```

```c
// Free interrupt
void free_irq(unsigned int irq, void *dev);

// Enable/disable IRQs locally
local_irq_disable();
local_irq_enable();

// Save/restore IRQ state
unsigned long flags;
local_irq_save(flags);
local_irq_restore(flags);
```

Flags for `request_irq()`:

Flag	Purpose
`IRQF_TRIGGER_RISING`	Trigger on rising edge
`IRQF_TRIGGER_FALLING`	Trigger on falling edge
`IRQF_SHARED`	Allow IRQ sharing
`IRQF_ONESHOT`	Keep IRQ disabled after handler

3. Basic LED Toggle Driver

Using GPIO_49 (LED) and GPIO_115 (Button):

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

#define LED_GPIO 49     // P9.23
```

```c
#define BTN_GPIO 115   // P9.27

static int irq_number;
static struct gpio_desc *led_gpio;

// Top Half - Minimal work
static irqreturn_t button_isr(int irq, void *dev_id)
{
    gpiod_set_value(led_gpio, !gpiod_get_value(led_gpio));
    pr_info("Interrupt! LED toggled\n");
    return IRQ_HANDLED;
}

static int __init gpio_isr_init(void)
{
    int ret;

    // LED setup
    led_gpio = gpio_to_desc(LED_GPIO);
    gpiod_direction_output(led_gpio, 0);

    // Button setup
    if (!gpio_is_valid(BTN_GPIO)) {
        pr_err("Invalid button GPIO\n");
        return -EINVAL;
    }

    ret = gpio_request(BTN_GPIO, "btn_gpio");
    if (ret) {
        pr_err("GPIO %d request failed\n", BTN_GPIO);
        return ret;
    }

    gpio_direction_input(BTN_GPIO);

    // Get Linux IRQ number
```

```
    irq_number = gpio_to_irq(BTN_GPIO);
    pr_info("Button GPIO %d maps to IRQ %d\n", BTN_GPIO, irq_number);

    // Request interrupt
    ret = request_irq(irq_number, button_isr,
                      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                      "bb-gpio-isr", NULL);
    if (ret) {
        pr_err("IRQ request failed\n");
        gpio_free(BTN_GPIO);
        return ret;
    }

    return 0;
}

static void __exit gpio_isr_exit(void)
{
    free_irq(irq_number, NULL);
    gpio_free(BTN_GPIO);
    gpiod_set_value(led_gpio, 0);
}

module_init(gpio_isr_init);
module_exit(gpio_isr_exit);
MODULE_LICENSE("GPL");
```

4. Testing the Driver

Load and Observe:

```
sudo insmod gpio_isr.ko
tail -f /var/log/kern.log  # Watch interrupts
```

```
# Press button - LED should toggle
ls /proc/irq/*/bb-gpio-isr  # Verify handler

sudo rmmod gpio_isr
```

Expected Output:

```
[ 1234.567890] Button GPIO 115 maps to IRQ 42
[ 1234.567901] Interrupt! LED toggled
[ 1234.570123] Interrupt! LED toggled
```

Bottom Halves in Linux Kernel

1. Overview of Bottom Halves

Why Bottom Halves?

- **Problem**: Interrupt handlers (top halves) must execute quickly
- **Solution**: Defer non-critical work to bottom halves

Key Characteristics:

Feature	Description
Execution Context	Process context (except softirq)
Scheduling	Non-preemptible (softirq/tasklet) or preemptible (workqueue)
Concurrency	Softirqs: Parallel on SMP Tasklets: Serialized per-CPU Workqueues: Fully preemptible

2. Softirqs

Core Concepts:

- **Fastest** bottom-half mechanism
- **Static allocation** (fixed at compile time)
- **Run in interrupt context** (with interrupts enabled)
- **Used by**: Network stack, block layer, timer subsystem

APIs:

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

// Kernel representation
struct softirq_action {
    void (*action)(struct softirq_action *);
};

// Statically declared (kernel pre-defined)
enum {
    HI_SOFTIRQ=0, TIMER_SOFTIRQ, NET_TX_SOFTIRQ, ...
};

// Registering softirq handler
open_softirq(NET_TX_SOFTIRQ, net_tx_action);

// Mark pending
raise_softirq(NET_RX_SOFTIRQ);

// Check execution context
in_softirq();
```

May execute in

- In the return from hardware interrupt code path
- In the ksoftirqd kernel thread
- In any code that explicitly checks for and executes pending softirqs, such as the networking subsystem

Constraints:

- No sleeping allowed
- Must be reentrant (can run on multiple CPUs simultaneously)
- Fixed list (cannot dynamically register new types)

3. Tasklets

Core Concepts:

- **Built on top of softirqs** (HI_SOFTIRQ/TASKLET_SOFTIRQ)
- **Dynamic registration** possible
- **Serialized execution** (same tasklet won't run concurrently)
- **Run in interrupt context or process context** based on softirq

APIs:

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

// Tasklet structure
struct tasklet_struct {
    unsigned long state; // zero, TASKLET_STATE_SCHED, or TASKLET_STATE_RUN
    void (*func)(unsigned long);
    unsigned long data;
    // ...
};

// Initialize (static)
DECLARE_TASKLET(my_tasklet, tasklet_fn, data);
// state = 0
```

```c
// Initialize (dynamic)
struct tasklet_struct my_tasklet;
tasklet_init(&my_tasklet, tasklet_fn, data);
// state = 0

// Schedule for execution
tasklet_schedule(&my_tasklet);
tasklet_hi_schedule(&my_tasklet);
// state = TASKLET_STATE_SCHED

// Disable/enable
tasklet_disable(&my_tasklet);
tasklet_enable(&my_tasklet);

// Kill permanently
tasklet_kill(&my_tasklet);
```

Example Usage:

```c
void my_tasklet_fn(unsigned long data) {
    // state = TASKLET_STATE_RUN
    printk("Running in tasklet context\n");
}

DECLARE_TASKLET(my_tasklet, my_tasklet_fn, 0);

// In interrupt handler:
tasklet_schedule(&my_tasklet);
```

4. Workqueues

Core Concepts:

- **Most flexible** bottom-half mechanism
- **Runs in process context** (can sleep!)
- **Dynamic creation** possible
- **Default shared queues** (kernel-managed) or **dedicated queues**

APIs:

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

// Work structure
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    void (*func)(struct work_struct *work);
    // ...
};

// Initialize
INIT_WORK(&my_work, work_fn);

// Schedule on shared queue
schedule_work(&my_work);

// Create dedicated queue
struct workqueue_struct *my_wq = alloc_workqueue("my_wq", flags, max_active);

// Schedule on custom queue
queue_work(my_wq, &my_work);
```

Flags for Workqueues:

Flag	Purpose
`WQ_UNBOUND`	No CPU affinity
`WQ_MEM_RECLAIM`	Needed for I/O during memory pressure
`WQ_HIGHPRI`	High-priority execution

Example Usage:

```
void work_fn(struct work_struct *work) {
    printk("Running in process context, can sleep!\n");
    msleep(10); // Valid!
}

DECLARE_WORK(my_work, work_fn);

// In interrupt handler:
schedule_work(&my_work);
```

5. Decision Guide

Mechanism	Context	Sleep?	SMP Safety	When to Use
Softirq	Either	No	Fully parallel	Core kernel needs
Tasklet	Either	No	Serialized	Driver deferred work
Workqueue	Process	Yes	Fully preemptible	Sleep-needed operations

Summary

1. **Softirqs**: For high-frequency, low-latency, non-sleeping tasks
2. **Tasklets**: Simpler alternative to softirqs for drivers

3. **Workqueues**: When you need to sleep or complex processing
4. **Rule of Thumb**:
   - Start with tasklets for simple drivers
   - Use workqueues if sleeping is needed
   - Avoid softirqs unless writing core kernel code

Here are complete driver examples for your BeagleBone Black (LED on GPIO49, switch on GPIO115) using both tasklet and workqueue approaches:

6. Tasklet-Based Implementation

(Fast, interrupt-context deferral)

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

#define LED_GPIO 49
#define BTN_GPIO 115

static struct tasklet_struct btn_tasklet;
static struct gpio_desc *led_gpio;
static int irq_number;

// Tasklet function (still atomic context)
static void toggle_led_tasklet(unsigned long data)
{
    gpiod_set_value(led_gpio, !gpiod_get_value(led_gpio));
    pr_info("LED toggled by tasklet\n");
}

// Top half ISR
static irqreturn_t button_isr(int irq, void *dev_id)
{
    tasklet_schedule(&btn_tasklet); // Defer to tasklet
```

```c
    return IRQ_HANDLED;
}

static int __init btnled_init(void)
{
    int ret;

    // LED setup
    led_gpio = gpio_to_desc(LED_GPIO);
    gpiod_direction_output(led_gpio, 0);

    // Button setup
    if (!gpio_is_valid(BTN_GPIO)) {
        pr_err("Invalid button GPIO\n");
        return -EINVAL;
    }

    ret = gpio_request(BTN_GPIO, "btn_gpio");
    if (ret) {
        pr_err("GPIO request failed\n");
        return ret;
    }

    gpio_direction_input(BTN_GPIO);
    irq_number = gpio_to_irq(BTN_GPIO);

    // Tasklet init
    tasklet_init(&btn_tasklet, toggle_led_tasklet, 0);

    // Request IRQ
    ret = request_irq(irq_number, button_isr,
                      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                      "bb-btn-irq", NULL);
    if (ret) {
        pr_err("IRQ request failed\n");
        tasklet_kill(&btn_tasklet);
```

```
        gpio_free(BTN_GPIO);
        return ret;
    }

    return 0;
}

static void __exit btnled_exit(void)
{
    free_irq(irq_number, NULL);
    tasklet_kill(&btn_tasklet);
    gpio_free(BTN_GPIO);
    gpiod_set_value(led_gpio, 0);
}

module_init(btnled_init);
module_exit(btnled_exit);
MODULE_LICENSE("GPL");
```

7. Workqueue-Based Implementation

(Flexible, process-context with sleep capability)

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/workqueue.h>

#define LED_GPIO 49
#define BTN_GPIO 115

static struct work_struct btn_work;
static struct gpio_desc *led_gpio;
static int irq_number;
```

```c
// Work function (can sleep)
static void toggle_led_work(struct work_struct *work)
{
    gpiod_set_value(led_gpio, !gpiod_get_value(led_gpio));
    pr_info("LED toggled by workqueue\n");

    // Example of sleep capability
    // msleep(10); // Valid in workqueue!
}

// Top half ISR
static irqreturn_t button_isr(int irq, void *dev_id)
{
    schedule_work(&btn_work); // Defer to workqueue
    return IRQ_HANDLED;
}

static int __init btnled_init(void)
{
    int ret;

    // LED setup
    led_gpio = gpio_to_desc(LED_GPIO);
    gpiod_direction_output(led_gpio, 0);

    // Button setup
    if (!gpio_is_valid(BTN_GPIO)) {
        pr_err("Invalid button GPIO\n");
        return -EINVAL;
    }

    ret = gpio_request(BTN_GPIO, "btn_gpio");
    if (ret) {
        pr_err("GPIO request failed\n");
        return ret;
```

```c
    }

    gpio_direction_input(BTN_GPIO);
    irq_number = gpio_to_irq(BTN_GPIO);

    // Workqueue init
    INIT_WORK(&btn_work, toggle_led_work);

    // Request IRQ
    ret = request_irq(irq_number, button_isr,
                     IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                     "bb-btn-irq", NULL);
    if (ret) {
        pr_err("IRQ request failed\n");
        gpio_free(BTN_GPIO);
        return ret;
    }

    return 0;
}

static void __exit btnled_exit(void)
{
    free_irq(irq_number, NULL);
    cancel_work_sync(&btn_work); // Ensure work completes
    gpio_free(BTN_GPIO);
    gpiod_set_value(led_gpio, 0);
}

module_init(btnled_init);
module_exit(btnled_exit);
MODULE_LICENSE("GPL");
```

8. Which to Choose?

1. **Use Tasklets When**:
   - Need minimal latency
   - Handling simple hardware events
   - No sleeping required
2. **Use Workqueues When**:
   - Need to perform I/O operations
   - Require mutex/sleep functionality
   - Doing substantial processing