Kernel Time Management

- Absolute & Relative time
  - Absolute time is also referred as "wall time". It is time of calendar day. In Hardware it is tracked by "RTC".
  - Typically RTC is backed by the seperate small battery, which ensures that wall time is tracked, even if main circuit/computer is switched off.
  - Relative time is used for duration between two events. Relative time is managed by the timer circuits. This timing is required for scheduling, resource management, etc.
- Hardware Clocks/Timers
  - RTC -> For Absolute time.
  - System Timer (PIT) -> periodic interrupts used by OS (ms).
  - HPET (High Precision Evolation Timer) -> Fine time calculation (us) a.k.a. HR (high resolution) timer.
  - TSC (Time Stamp Counter) -> Incremented on each CPU clock -- fastest counter in the system. But it is not generating interrupts.
- System Timer
  - Configured during booting of the system.
  - Generate Periodic Interrupts called as "Ticks"
  - Tick Rate (HZ)
    - HZ --> CONFIG_HZ configuration --> 100 (10ms) or 250 (4ms) or 1000 (1ms).
    - HZ ticks per second.
  - Advantages Large HZ i.e. Small Tick duration
    - Kernel timers execute with finer resolution.
    - System calls timeout with higher precision.
    - Resource usage fine calculation.
    - Accurate Process preemption.
  - Disavantages of Large HZ
    - More frequent timer interrupt and hence higher overhead.
    - More CPU time spent in execution of ISR.
- System uptime is maintained in "jiffies".
  - jiffies (32-bit) --> max value 2^32 ticks
  - jiffies_64 (64-bit)
  - Both these variables share the same memory. So increment operation is not repeated for both variables per tick.
- Conversion from jiffies to seconds and vice-versa:

    - sec = jiffies / HZ.
    - jiffies = sec * HZ
- Timer Interrupt ISR (will be called for each tick interrupt)
    - Update system uptime (increment jiffies)
    - Update wall time (time of day)
    - On SMP system, balance runqueues
    - Run expired dynamic timers
    - Update resource/processor time usage
    - Invoke scheduler
- Small delays in kernel space
    - `#include <linux/delay.h>`
    - `#include <asm/delay.h>`
    - void udelay(unsigned long usecs);
    - void ndelay(unsigned long nsecs);
    - void mdelay(unsigned long msecs);
- https://www.kernel.org/doc/Documentation/timers/timers-howto.txt

Dynamic Timer

- Applications
    - Delayed execution of a task.
    - Periodic execution of the task.
- Dynamic Timer struct: `<linux/timer.h>`

```
struct timer_list {
    struct hlist_node entry;                // add timer in timer linked list.
    unsigned long expires;                  // time-out/expire in jiffies (current jiffies + delay period)
    void (*function)(struct timer_list *);  // addr of timer func to be called when time-out/expire
    u32 flags;                              // timer state: added or expired or ...
    // ...
};
```

- terminal> sudo cat /proc/timer_list
- APIs:

```
struct timer_list timer;
```

```
// timer initialization
timer_setup(&timer, my_function, 0);    // interval in jiffies = seconds * HZ
timer.expires = jiffies + 1 * HZ;
add_timer(&timer);
```

```
// timer function
void my_function(struct timer_list *ptimer) {
    // ...
    mod_timer(&timer, jiffies + 1 * HZ);    // interval in jiffies = seconds * HZ
}
```

```
// timer destroy
del_timer_sync(&timer);
```

Kernel thread

- Like user space we can create threads in kernel space as well.
- The kernel threads should be used only for "dedicated" task/sub-system.
- Kernel threads use more resources and hence not advised to use often.

- It is created using kernel api: kthread_create() or kthread_run().

```
struct task_struct *task = kthread_run(thread_func, NULL, "lcd_scroll");
```

- To stop the kernel thread use kernel api: kthread_stop().
- If kernel thread is created with kthread_create(), it should be woke up explicitly using wake_up_process(). The macro kthread_run() internally calls both these functions.
- Each thread is associated with some thread function

```
int thread_func(void *param) {
    // code to be executed by thread
}
```

- The kernel threads are created and managed by "kthreadd" kernel daemon.

IO Ports/memory

- IO registers can be memory mapped or IO mapped based on arch.

IO mapped IO

- x86 -- IO mapped -- separate address space for IO registers.
    - Serial port 1: address = 0x3F8 and irq number = 4
    - Serial port 2: address = 0x2F8 and irq number = 3
    - Parallel port: address = 0x378
    - Keyboard controller: address = 0x60
- Example to disable keyboard.
    - http://embeddedguruji.blogspot.com/2019/01/linux-device-driver-to-disableenable.html

    - However, above example doesn't work in Linux kernel 5.0+ due to more restrictions on io memory.
- HAL macros to write/read from IO port registers --> inb(), outb(), inl(), outls(), ...
- Use request_region() before actual read/write operations on the port. Only one module can get ownership of the io port at a time.
    - The list of acquired io ports is visible under /proc/ioports
    - release_region() should be used at the end of module to release the io port.
- To access IO ports from the user-space ioperm() syscall can be used.
- For sake of demonstration (on x86 architecture), write and read can be performed on a unused/dummy port 0x80. Traditionally this port was used for debugging. Nowadays, writing/reading on it have no effect.

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/io.h>

/*
Traditionally, port 0x80 is used as a debug port on x86 systems. Writing to it doesn't affect critical
hardware and is generally safe for experiments.
*/
#define DUMMY_IO_PORT 0x80

static int __init io_port_demo_init(void) {
    u8 value;
    pr_info("%s: Module Loaded\n", THIS_MODULE->name);
    // Perform an outb() operation (writing data to the port)
    outb(0xAB, DUMMY_IO_PORT);
    pr_info("%s: Wrote 0xAB to port 0x80\n", THIS_MODULE->name);
    // Perform an inb() operation (reading data from the port)
    value = inb(DUMMY_IO_PORT);
    pr_info("%s: Read 0x%X from port 0x80\n", THIS_MODULE->name, value);
    return 0;
}
```

```
static void __exit io_port_demo_exit(void) {
    pr_info("%s: Module Unloaded\n", THIS_MODULE->name);
}

module_init(io_port_demo_init);
module_exit(io_port_demo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Nilesh Ghule");
MODULE_DESCRIPTION("Simple Kernel Module for I/O Port Programming");
```

Memory mapped IO

- ARM -- Memory mapped -- combined address space for IO registers and memory.
    - AM335x: For each GPIO there are 4KB address ranges (memory mapped). This 4KB have sevaral addresses at definite offset for controlling GPIO operations (for particular bits).
- HAL functions to write/read from IO memory --> ioread8(), iowrite8(), ioread32(), iowrite32(), ...
- Every module e.g. GPIO module has its own memory map i.e. physical address specified in the processor's technical reference manual.
- First you need to check if the memory region is being used or not using check_mem_region(). This step is deprecated in newer kernel.
- If it is free, request access to this memory region using request_mem_region(), then map the GPIO module using ioremap() or ioremap_nocache() (map bus memory into CPU space), which returns a void*.
- The returned address is not guaranteed to be usable directly as a virtual address; it is only usable by ioread*|iowrite*|read*|write*, etc. functions.
- Use ioread8|16|32/iowrite8|16|32 functions to read or write from/to i/o ports.
- Finally you need to iounmap() to unmap the memory and then you need to release memory region using release_mem_region().

- For of demonstration (on ARM architecture), write and read can be performed on a GPIO pin. Traditionally this port was used for debugging. Nowadays, writing/reading on it have no effect.

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/io.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

#define GPIO1_BASE_ADDR   0x4804C000 // Base address of GPIO1
#define GPIO_OE           0x134     // Offset for Output Enable register
#define GPIO_SETDATAOUT   0x194     // Offset for Set Data Output register
#define GPIO_CLEARDATAOUT 0x190     // Offset for Clear Data Output register
#define GPIO_PIN          (1 << 28) // GPIO1_28 (P9.12 pin)

void __iomem *gpio_base; // MMIO base address pointer

static int __init gpio_driver_init(void)
{
    // Map the physical address to a virtual address
    gpio_base = ioremap(GPIO1_BASE_ADDR, 0x1000);
    if (!gpio_base) {
        pr_err("Failed to ioremap GPIO1_BASE_ADDR\n");
        return -ENOMEM;
    }

    // Configure GPIO1_28 as an output
    u32 oe_reg = ioread32(gpio_base + GPIO_OE);
    oe_reg &= ~GPIO_PIN; // Clear the bit to set as output
    iowrite32(oe_reg, gpio_base + GPIO_OE);

    // Turn on the GPIO1_28 pin
    iowrite32(GPIO_PIN, gpio_base + GPIO_SETDATAOUT);
    pr_info("GPIO1_28 is set HIGH (LED ON)\n");
```

```
    return 0;
}

static void __exit gpio_driver_exit(void)
{
    // Turn off the GPIO1_28 pin
    iowrite32(GPIO_PIN, gpio_base + GPIO_CLEARDATAOUT);
    pr_info("GPIO1_28 is set LOW (LED OFF)\n");

    // Unmap the memory region
    iounmap(gpio_base);
}

module_init(gpio_driver_init);
module_exit(gpio_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Nilesh Ghule");
MODULE_DESCRIPTION("Simple GPIO driver using ioread32() and iowrite32()");
```

- Note: ioread8()... or iowrite8()... functions can also be used for memory mapped IO devices on x86 architecture e.g. some PCI devices.

```
#include <linux/module.h>
#include <linux/io.h>
#include <linux/pci.h>

#define MMIO_BAR 0 // PCI Base Address Register index

void __iomem *mmio_base;

static int __init mmio_example_init(void) {
    struct pci_dev *pdev = NULL;

    // Locate the PCI device (for example, a specific vendor/device ID)
```

```
    pdev = pci_get_device(0x1234, 0x5678, NULL);
    // Enable the PCI device
    pci_enable_device(pdev);
    // Get MMIO region (assumes MMIO is in BAR 0)
    mmio_base = pci_iomap(pdev, MMIO_BAR, pci_resource_len(pdev, MMIO_BAR));
    // Read a 32-bit register from the MMIO region
    u32 value = ioread32(mmio_base);
    pr_info("Read value: 0x%x\n", value);
    return 0;
}

static void __exit mmio_example_exit(void) {
    if (mmio_base)
        pci_iounmap(NULL, mmio_base);
    pr_info("Unloaded MMIO example module\n");
}

module_init(mmio_example_init);
module_exit(mmio_example_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Nilesh Ghule");
MODULE_DESCRIPTION("MMIO Example on x86 using ioread32()");
```

- Reference: LDD -- Communicating with Hardware

Writing Hardware Device Drivers

- Method 1:
  - Acquire IO port addresses using request_region() -- in module initialization
    - terminal> sudo cat /proc/ioports
  - Initialize the device using inb()/outb() -- in open()
  - Read/write data on IO device using inb()/outb() -- in read()/write()
  - De-initialize the device using inb()/outb() -- in release()

    - Release IO port addresses using release_region() -- in module exit
    - Note: If existing device driver is already occupying the port, then your driver request_region() will fail. In this case, existing driver should be blacklisted (/etc/modprobe.d/blacklist.conf) and then your driver can control the device.
- Method 2:
    - Implement your device driver depending on existing device driver/kernel sub-system i.e. your device operations invokes the functions exported by existing device driver/kernel sub-system.
    - Note: Here you don't have direct access to IO ports (no request_region()) and you should not blacklist existing device driver.

GPIO Device Driver Notes

1. Linux GPIO Subsystem

Core Concepts:

- **GPIO (General Purpose Input/Output)**: Programmable pins that can be configured as input or output
- **Two Perspectives**:
    - **Consumer**: Driver that *uses* GPIOs (our focus)
    - **Provider**: Driver that *implements* GPIO control (e.g., SoC-specific)

Key Features:

- Hardware-independent interface (`#include <linux/gpio.h>`)
- Automatic pin contention handling
- Support for input/output, pull-up/down, interrupts (covered later)

BBB GPIO Specifics:

- **AM335x SoC** has 4 GPIO banks (0-3), each with 32 pins
- Pin naming: `GPIO<bank>_<number>` (e.g., `GPIO0_26` for P8.14)
- Many pins are multiplexed (configure via Device Tree later)

2. BBB GPIO Consumer Driver (Legacy APIs)

Key Functions:

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

Example Driver Code:

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

#define MY_GPIO 48 // Example: GPIO1_16 (P9.15 on BBB)

static int __init gpio_demo_init(void)
{
    int ret;

    if (!gpio_is_valid(MY_GPIO)) {
        pr_err("Invalid GPIO %d\n", MY_GPIO);
        return -EINVAL;
    }

    // Request GPIO
    ret = gpio_request(MY_GPIO, "my_gpio");
    if (ret) {
        pr_err("GPIO %d request failed: %d\n", MY_GPIO, ret);
        return ret;
    }
```

```
    // Set as output with initial high
    ret = gpio_direction_output(MY_GPIO, 1);
    if (ret) {
        pr_err("GPIO %d direction set failed: %d\n", MY_GPIO, ret);
        gpio_free(MY_GPIO);
        return ret;
    }

    pr_info("GPIO %d initialized\n", MY_GPIO);
    return 0;
}

static void __exit gpio_demo_exit(void)
{
    gpio_set_value(MY_GPIO, 0); // Turn off before release
    gpio_free(MY_GPIO);
    pr_info("GPIO %d released\n", MY_GPIO);
}

module_init(gpio_demo_init);
module_exit(gpio_demo_exit);
MODULE_LICENSE("GPL");
```

BBB Pin Mapping Cheat Sheet:

Pin	GPIO Number	SoC Name
P8.14	26	GPIO0_26
P9.12	60	GPIO1_28
P9.15	48	GPIO1_16

3. Practical Lab Steps - Cross-Compilation

1. Compile the Driver:

```
# On host machine (cross-compilation)
make -C <kernel_src> M=$(pwd) ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

2. Test on BBB:

```
# Copy .ko file to BBB
scp gpio_demo.ko debian@beaglebone.local:~

# On BBB:
sudo insmod gpio_demo.ko
dmesg | tail # Check init message

# Manually verify GPIO (alternative to driver operations)
echo 48 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio48/direction
echo 1 > /sys/class/gpio/gpio48/value # Should match driver

sudo rmmod gpio_demo
```

4. Common Issues & Debugging

1. **GPIO Request Fails (-EBUSY)**:
    - Pin already in use (check `/sys/kernel/debug/gpio`)
    - Pin multiplexed for other function (check `config-pin` utility)
2. **Wrong GPIO Number**:
    - Verify BBB pin-to-GPIO mapping

- Use `gpioinfo` from `gpiod` package

3. **Permission Issues**:
   - Ensure running as root or user in `gpio` group
4. **Check GPIO States**:

```
sudo cat /sys/kernel/debug/gpio
```

5. **Verify Pin Muxing**:

```
config-pin -q P9.23
# Should show "gpio" mode
```

GPIO Driver using Modern gpiod API

1. Why gpiod?

Advantages over Legacy GPIO API:

1. **Hardware-neutral** - Doesn't assume numeric GPIOs
2. **Safer** - Better error handling
3. **Device Tree aware** (though we're not using DT yet)
4. **Multiple GPIOs** - Handles groups of GPIOs elegantly
5. **Standardized** - Recommended for new drivers

Key Headers:

```
```

```
#include <linux/gpio/consumer.h>  // Consumer interface
#include <linux/gpio.h>           // Optional helpers
```

2. Core gpiod Functions

Essential Functions:

Function	Purpose
`gpiod_get()`	Get GPIO descriptor
`gpiod_put()`	Release GPIO descriptor
`gpiod_direction_input()`	Set as input
`gpiod_direction_output()`	Set as output
`gpiod_get_value()`	Read input
`gpiod_set_value()`	Set output
`gpiod_to_irq()`	Get IRQ number (for future interrupts)

Flags for `gpiod_get()`:

```
GPIOD_IN       // Default input
GPIOD_OUT_LOW  // Output init low
GPIOD_OUT_HIGH // Output init high
```

3. Complete Driver Example

Using P9.23 (GPIO1_17) on BBB:

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

static struct gpio_desc *led_gpio;

static int gpiod_demo_probe(struct platform_device *pdev)
{
    led_gpio = gpiod_get(&pdev->dev, "led", GPIOD_OUT_HIGH);
    if (IS_ERR(led_gpio)) {
        dev_err(&pdev->dev, "Failed to get LED GPIO\n");
        return PTR_ERR(led_gpio);
    }

    dev_info(&pdev->dev, "GPIO controlled LED ready\n");
    return 0;
}

static int gpiod_demo_remove(struct platform_device *pdev)
{
    gpiod_set_value(led_gpio, 0); // Turn off LED
    gpiod_put(led_gpio);
    dev_info(&pdev->dev, "GPIO released\n");
    return 0;
}

static struct platform_driver gpiod_driver = {
    .driver = {
        .name = "bb-gpiod-demo",
        .owner = THIS_MODULE,
    },
    .probe = gpiod_demo_probe,
    .remove = gpiod_demo_remove,
};
module_platform_driver(gpiod_driver);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
```

4. Testing on BeagleBone Black (Native Compilation)

Compilation:

- Copy source code and Makefile on BBB.

```
make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
```

Manual Test without Device Tree:

```
# Load module with GPIO parameter (using P9.23 - GPIO1_17)
sudo insmod gpiod_demo.ko gpios="1,17"

# Verify:
ls /sys/class/leds/ # Should show gpiod-controlled LED
cat /sys/kernel/debug/gpio # Check GPIO states

# Unload:
sudo rmmod gpiod_demo
```

5. Key Differences from Legacy API

Aspect	Legacy API	gpiod API
GPIO Reference	Integer	`struct gpio_desc*`

Aspect	Legacy API	gpiod API
Error Handling	Return codes	`IS_ERR()`/`PTR_ERR()`
Multiple GPIOs	Individual calls	`gpiod_get_array()`
Active Low	Manual handling	Built-in support
Direction	Separate call	Combined with get