Linux High-Resolution Kernel Timers

The `struct hrtimer` Subsystem

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer

Overview

Roadmap:

- What is “high-resolution” timing?
- Issues with high-resolution timing
- The `struct hrtimer` API
- Examples

What is “High-Resolution” Timing?

The “hrtimer” API:

- Expresses time values in 64-bit, nanosecond units
- Facilitiates the precision offered by the platform

Compared to the “timer wheel”:

- More precision
- More range
- Much higher performance

What is “High-Resolution” Timing?

Why two implementations?

- Timer wheel is optimized for rare, low-precision timeouts
- Hrtimer is for high-precision interval timing

```
“The timer wheel code is fundamentally not suitable
for [high-resolution timing].  We initially didn’t
believe this...”
                              -- Thomas Gleixner and Ingo Molnar
```

What is “High-Resolution” Timing?

Implementation details:

- `ktime_t` data type
- `struct clocksource` and `struct clock_event_device`
- `struct hrtimer`

See:

- `Documentation/timers/highres.txt`
- `Documentation/timers/hrtimers.txt`

hrtimer.c

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

static long hrtimer_demo_secs = 0;
static unsigned long hrtimer_demo_nsecs = 500000000UL;

struct hrtimer_demo {
  int timeouts;
  struct hrtimer hrt;
};
struct hrtimer_demo hd;
```

hrtimer.c

```
static int __init hrtimer_demo_init(void)
{
  hrtimer_init(&hd.hrt, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
  hd.hrt.function = hrtimer_demo_timeout;
  hd.timeouts = 0;

  hrtimer_start(&hd.hrt, ktime_set(hrtimer_demo_secs,
                                   hrtimer_demo_nsecs),
                HRTIMER_MODE_REL);

  return 0;
}
```

hrtimer.c

```
static enum hrtimer_restart
hrtimer_demo_timeout (struct hrtimer *t)
{
  struct hrtimer_demo *hd =
    container_of(t, struct hrtimer_demo, hrt);

  printk(KERN_INFO "%s: hd->timeouts = %d\n",
         __FUNCTION__, ++hd->timeouts);
  hrtimer_start(&hd->hrt, ktime_set(hrtimer_demo_secs,
                                    hrtimer_demo_nsecs),
                HRTIMER_MODE_REL);
  return HRTIMER_NORESTART;
}
```

hrtimer.c

```
static void hrtimer_demo_exit(void)
{
  hrtimer_cancel(&hd.hrt);
}
module_exit(hrtimer_demo_exit);
```

Important!

Timer handler runs in interrupt context:

- No sleeping!
- No I2C, SPI, etc. communications

(We'll fix this later)

Drift

Timer “drift”:

- Slippage from deadline due to delayed start
- Unavoidable with previous example (why?)

Not the same as “jitter”:

- (Which is mostly due to interrupt latency)

Drift

```
static enum hrtimer_restart
hrtimer_demo_timeout (struct hrtimer *t)
{
  struct hrtimer_demo *hd =
    container_of(t, struct hrtimer_demo, hrt);

  printk(KERN_INFO "%s: hd->timeouts = %d\n",
         __FUNCTION__, ++hd->timeouts);
  hrtimer_add_expires(&hd->hrt, ktime_set(hrtimer_demo_secs,
                                          hrtimer_demo_nsecs))
  return HRTIMER_RESTART;
}
```

Recap

High-resolution timers:

- High precision, performance
- Optimized for interval timing
- Builds on several clock-related APIs

Linux High-Resolution Kernel Timers

The `struct hrtimer` Subsystem

Bill Gatliff
`bgat@billgatliff.com`

Freelance Embedded Systems Developer

Demonstration

Assignment

Repeat the demonstration:

- Build, run `hrtimer.c` on your platform

Develop your own PWM implementation:

- Generate signal via gpio kernel API
- Use high-resolution kernel timer to control duty cycle
- Provide a device interface to users

Assignment

GPIO API:

- Pick a GPIO output
- Use `gpio_direction_output()` to assert its value

Kernel timer for duty cycle:

- `struct hrtimer`
- Turn GPIO line on, off in the timer function
- (You can assume that the GPIO implementation doesn't sleep)

Assignment

Interface:

- `struct miscdevice`
- Allows users to change signal from applications
- Potentially tricky— concurrent access issues

Implement in small steps!

Assignment

First step:

- Turn GPIO line on and off in module init, exit
- Use the LED output as a test device

Next:

- Implement static duty cycle, period
- Specify values as module parameters

Assignment

Finally:

- Implement the interface
- Adjust values in `write()` and/or `ioctl()` (your preference)
- Query state during `read()`