

Performance Measurement of Kernel Routines

Pallav Joshi



Performance Measurement of Kernel Routines

Measuring performance of critical loops is one of the most important requirements of any embedded system. In this tip, we will look into how to measure the performance of your routines running in kernel space.

❖ **Compiling with Kernel:**

Include the performance measuring code into the Linux kernel by the following steps:

Step 1:

Copy the 'time_diff' folder in kernel source tree under 'drivers' folder.

(Please refer Appendix for the time_diff.h and time_diff.c files.)

Step 2:

Modify 'drivers/Makefile' by adding the following line at the end (without any conditions)

```
obj-y += time_diff/
```

Step 3:

Compile your kernel and make sure that 'drivers/time_diff' is compiled in your kernel image.

❖ **Calling from Kernel drivers to measure time in cycles:**

Step 1:

Include 'drivers/time_diff/time_diff.h' file in your source file (adjust the path according to location of your source file).

Step 2:

Call StartPerfCounter(counter_id) to start sampling the time stamp

Step 3:

Call StopPerfCounter(counter_id) to stop sampling the time stamp

The 'counter_id' at present is from 0-14, this range can be changed by modifying value of 'MAXCOUNTER', defined in 'time_diff.c'.

❖ **Obtain Results from user space:**

Now that you have kernel with timer support and a time measurement routines compiled driver, the step is to check results generated by the utilities provided.

Step 1:

Load your kernel image on target board.

Step 2:

Load drivers compiled with measurement functions into kernel.


```

static int displayHeader(char *buffer)
{
    int count = 0;

    count = sprintf(buffer, "%s%s",
                    "Idx   Start   Stop   Min   Max   Total   Count\n",
                    "-----\n");

    return count;
}

static int displayRow(int counter_id, char *buffer, int max)
{
    timeCounter *pRow;
    int count;

    if (counter_id > MAXCOUNTER)
    {
        printk("%s: Invalid counter number\n", __FUNCTION__);
        return 0;
    }
    pRow = &counterList[counter_id];
    count = snprintf(buffer, max, "%3d %12lu%12lu%12lu%12lu%12lu\n",
                    counter_id, pRow->start, pRow->stop, pRow->min, pRow->max, pRow->acc, pRow->no_acc);
    return count;
}

void StartPerfCounter(int counter_id)
{
    timeCounter *pRow;
    if (!timerEnable)
        return;
    if (counter_id > MAXCOUNTER)
    {
        printk("%s: Invalid counter number\n", __FUNCTION__);
        return;
    }
    pRow = &counterList[counter_id];
    if (pRow->busy)
    {
        printk("%s: Error Counter busy %d\n", __FUNCTION__, counter_id);
        return;
    }
    pRow->busy = 1;
    pRow->start = get_cycles();
}

void StopPerfCounter(int counter_id)
{
    timeCounter *pRow;

```

```

cycles_t diff;
if (!timerEnable)
    return;
if (counter_id > MAXCOUNTER)
{
    printk("%s: Invalid counter number \n", __FUNCTION__);
    return;
}
pRow = &counterList[counter_id];
if (!(pRow->busy))
{
    printk("%s: Error Counter Free %d\n", __FUNCTION__, counter_id);
    return;
}
pRow->stop = get_cycles();
diff = pRow->stop - pRow->start;
if (pRow->min > diff)
    pRow->min = diff;
if (pRow->max < diff)
    pRow->max = diff;
pRow->acc += diff;
++pRow->no_acc;
pRow->busy = 0;
}

static int timer_read_proc(char *page, char **start,
                           off_t off, int count, int *eof, void *data)
{
    char *buffer = page;
    int loopCount;
    int copyCount;

    copyCount = snprintf(buffer, count, "PerfCounter: %s\n", (timerEnable) ? "ON" : "OFF");
    if (copyCount > 0)
    {
        buffer += copyCount;
        count -= copyCount;
    }

    if (count > 0) {
        copyCount = displayHeader(buffer);
        if (copyCount > 0) {
            buffer += copyCount;
            count -= copyCount;
        }
    }

    for(loopCount = 0; loopCount < MAXCOUNTER && count > 0; loopCount++)
    {
        copyCount = displayRow(loopCount, buffer, count);
        if (copyCount > 0)

```

```

        {
            buffer += copyCount;
            count -= copyCount;
        }
    }
    *eof = 1;
    return (buffer - page) - off;
}

static int timer_write_proc(struct file *file, const char __user *buffer,
                           unsigned long count, void *data)
{
    char cmd;
    copy_from_user(&cmd, buffer, 1);

    switch (cmd) {
    case 'E': /* Enable the performance counters */
    case 'e':
        timerEnable = 1;
        break;
    case 'D': /* Disable the performance counters */
    case 'd':
        timerEnable = 0;
        break;
    case 'C': /* Disable & clear the performance counters */
    case 'c':
        timerEnable = 0;
        InitCounterList();
        break;
    }
    return 1;
}

int __init time_diff_init_module(void)
{
    proc_entry = create_proc_entry ("myPerfTimers", 0600, NULL);
    if (proc_entry != NULL)
    {
        proc_entry->read_proc = timer_read_proc ;
        proc_entry->write_proc = timer_write_proc;
    }
    else
    {
        printk ("Error in creation of Proc entry\n");
        return -1;
    }
    InitCounterList();

    return 0;
}

```

