CONFIG_HZ 和 USER_HZ

2013-08-31 22:57:01

分类： LINUX

内核时钟的频率是由CONFIG_HZ决定的，以前默认是100HZ，现在内核默认是250HZ。而1个jiffy是1个时钟滴答，时间间隔是有CONFIG_HZ决定的，频率是250HZ，也就是周期为4ms。每4ms，增加一个时钟滴答，也即jiffies++。

原理比较简单，如何查看自己的Linux的CONFIG_HZ的值呢?

```
1. root@manu:~/code/c/self/ticks# grep ^CONFIG_HZ /boot/config-3.2.0-29-
generic-pae
2. CONFIG_HZ_250=y
3. CONFIG_HZ=250
```

有意思的是，systemtap tutorial有个比较好玩的实验，也可以确定CONFIG_HZ的大小。

```
1.
2. global count_jiffies, count_ms
3. probe timer.jiffies(100) { count_jiffies ++ }
4. probe timer.ms(100) { count_ms ++ }
5. probe timer.ms(543210)
6. {
7.     hz=(1000*count_jiffies) / count_ms
8.     printf ("jiffies:ms ratio %d:%d => CONFIG_HZ=%d\n",
9.     count_jiffies, count_ms, hz)
10.     exit ()
11. }
```

输出如下:

```
1. jiffies:ms ratio 1358:5420 => CONFIG_HZ=250
```

实验等待的时间有点久，需要等待543210ms，9分钟左右，时间越久，误差越小，如果等待的时间段一些，会出现误差。感兴趣的筒子自行实验。

除此外，还有一个值是USER_HZ,不了解这个的，可能会了解times系统调用，这个系统调用是统计进程时间消耗的，

```
1.     #include <sys/times.h>
2.
3.        clock_t times(struct tms *buf);
```

times系统调用的时间单位是由USER_HZ决定的，所以，times系统调用统计的时间是以10ms为单位的。说100HZ空口无凭，如何获取USER_HZ。

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/times.h>
#include<time.h>

int main()
{
    int user_ticks_per_second ;
     user_ticks_per_second = (int)sysconf(_SC_CLK_TCK);
     if(user_ticks_per_second == -1)
     {
         fprintf(stderr,"failed to get ticks per second by sysconf\n");
         return -1;
     }

     printf("The Number of USER ticks per second is
%d\n",user_ticks_per_second);

     return 0;
}
```

输出如下：

```
The Number of USER ticks per second is 100
```

如果你嫌用C代码调用SYSCONF太麻烦了，可以通过shell命令getconf获得USER_HZ的值：

```
root@manu:~/code/c/self/ticks# getconf CLK_TCK
100
```

times系统调用来统计进程信息我不建议使用了，精度太低了。提出这个USER_HZ，只是希望不要困惑，为什么CONFIG_HZ是250,而sysconf(_SC_CLK_TCK)却是100.

参考文献：

1 man 7 time

2 systemtap tutorial

内核中的HZ 及延迟等

时钟中断由系统定时硬件以周期性的间隔产生，这个间隔由内核根据 HZ 值来设定，HZ 是一个体系依赖的值，在 <linux/param.h>中定义或该文件包含的某个子平台相关文件中。作为通用的规则，即便如果知道 HZ 的值，在编程时应当不依赖这个特定值，而始终使用HZ。对于当前版本，我们应完全信任内核开发者，他们已经选择了最适合的HZ值，最好保持 HZ 的默认值。

对用户空间，内核HZ几乎完全隐藏，用户 HZ 始终扩展为 100。当用户空间程序包含 param.h，且每个报告给用户空间的计数器都做了相应转换。对用户来说确切的 HZ 值只能通过 /proc/interrupts 获得：/proc/interrupts 的计数值除以 /proc/uptime 中报告的系统运行时间。

对于ARM体系结构：在<linux/param.h>文件中的定义如下：

```
#ifdef __KERNEL__
# define HZ                 CONFIG_HZ       /* Internal kernel timer frequency */
# define USER_HZ          100               /* 用户空间使用的HZ，User interfaces are in "ticks" */
# define CLOCKS_PER_SEC (USER_HZ) /* like times() */
#else
# define HZ                 100
#endif
```

也就是说：HZ 由__KERNEL__和CONFIG_HZ决定。若未定义__KERNEL__，HZ为100；否则为CONFIG_HZ。而CONFIG_HZ是在内核的根目录的.config文件中定义，并没有在make menuconfig的配置选项中出现。Linux的\arch\arm\configs\s3c2410_defconfig文件中的定义为：

```
#
# Kernel Features
#
# CONFIG_PREEMPT is not set
# CONFIG_NO_IDLE_HZ is not set
CONFIG_HZ=200
# CONFIG_AEABI is not set
# CONFIG_ARCH_DISCONTIGMEM_ENABLE is not set
```

所以正常情况下s3c24x0的HZ为200。这一数值在后面的实验中可以证实。

每次发生一个时钟中断，内核内部计数器的值就加一。这个计数器在系统启动时初始化为 0， 因此它代表本次系统启动以来的时钟嘀哒数。这个计数器是一个 64-位 变量(即便在 32-位的体系上)并且称为 “jiffies_64”。但是驱动通常访问 jiffies 变量（unsigned long） （根据体系结构的不同：可能是 jiffies_64 ，可能是jiffies_64 的低32位）。使用 jiffies 是首选，因为它访问更快，且无需在所有的体系上实现原子地访问 64-位的 jiffies_64 值。

使用 jiffies 计数器

这个计数器和用来读取它的工具函数包含在 <linux/jiffies.h>， 通常只需包含 <linux/sched.h>，它会自动放入 jiffies.h 。 jiffies 和 jiffies_64 必须被当作只读变量。当需要记录当前 jiffies 值（被声明为 volatile 避免编译器优化内存读）时，可以简单地访问这个 unsigned long 变量，如：

```
#include <linux/jiffies.h>
unsigned long j, stamp_1, stamp_half, stamp_n;
j = jiffies; /* read the current value */
```

```
stamp_1 = j + HZ; /* 1 second in the future */
stamp_half = j + HZ/2; /* half a second */
stamp_n = j + n * HZ / 1000; /* n milliseconds */
```

以下是一些简单的工具宏及其定义：

```
#define time_after(a,b)                 \
(typecheck(unsigned long, a) && \
         typecheck(unsigned long, b) && \
((long)(b) - (long)(a) < 0))
#define time_before(a,b)          time_after(b,a)
#define time_after_eq(a,b)          \
(typecheck(unsigned long, a) && \
         typecheck(unsigned long, b) && \
((long)(a) - (long)(b) >= 0))
#define time_before_eq(a,b)          time_after_eq(b,a)
```

用户空间的时间表述法（struct timeval 和 struct timespec ）与内核表述法的转换函数：

```
#include <linux/time.h> /* #i nclude <linux/jiffies.h> --> \kernel\time.c*/
struct timespec {
time_t          tv_sec; /* seconds */
long          tv_nsec; /* nanoseconds */
};
#endif
struct timeval {
time_t                  tv_sec; /* seconds */
         suseconds_t          tv_usec; /* microseconds */
};
unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
```

访问jiffies_64 对于 32-位 处理器不是原子的，这意味着如果这个变量在你正在读取它们时被更新你可能读到错误的值。若需要访问jiffies_64，内核有一个特别的辅助函数，为你完成适当的锁定：

```
#include <linux/jiffies.h>
u64 get_jiffies_64(void);
```

处理器特定的寄存器

若需测量非常短时间间隔或需非常高的精度，可以借助平台依赖的资源。许多现代处理器包含一个随时钟周期不断递增的计数寄存器，他是进行高精度的时间管理任务唯一可靠的方法。最有名的计数器寄存器是 TSC (timestamp counter), 在 x86 的 Pentium 处理器开始引入并在之后所有的 CPU 中出现（包括 x86_64 平台）。它是一个 64-位 寄存器，计数 CPU 的时钟周期，可从内核和用户空间读取。在包含了 <asm/msr.h> (一个 x86-特定的头文件, 它的名子代表"machine-specific registers")的代码中可使用这些宏：

```
rdtsc(low32,high32);/*原子地读取 64-位TSC 值到 2 个 32-位 变量*/
rdtscl(low32);/*读取TSC的低32位到一个 32-位 变量*/
rdtscll(var64);/*读 64-位TSC 值到一个 long long 变量*/
```

```
/*下面的代码行测量了指令自身的执行时间:*/
unsigned long ini, end;
rdtscl(ini); rdtscl(end);
printk("time lapse: %li\n", end - ini);
```

一些其他的平台提供相似的功能，并且内核头文件提供一个体系无关的功能用来代替 rdtsc，称 get_cycles（定义在 <asm/timex.h>(由 <linux/timex.h> 包含)），原型如下:

```
#include <linux/timex.h>
cycles_t get_cycles(void);
/*这个函数在每个平台都有定义，但在没有时钟周期计数器的平台上返回 0 */
/*由于s3c2410系列处理器上没有时钟周期计数器所以get_cycles定义如下：*/
typedef unsigned long cycles_t;
static inline cycles_t get_cycles (void)
{
return 0;
}
```

获取当前时间

驱动一般无需知道时钟时间（用年月日、小时、分钟、秒来表达的时间），只对用户程序才需要，如 cron 和 syslogd。 内核提供了一个将时钟时间转变为秒数值的函数:

```
unsigned long
mktime(const unsigned int year0, const unsigned int mon0,
const unsigned int day, const unsigned int hour,
const unsigned int min, const unsigned int sec)
{
unsigned int mon = mon0, year = year0;
/* 1..12 -> 11,12,1..10 */
if (0 >= (int) (mon -= 2)) {
                mon += 12; /* Puts Feb last since it has leap day */
                year -= 1;
}
return ((((unsigned long)
(year/4 - year/100 + year/400 + 367*mon/12 + day) +
                year*365 - 719499
)*24 + hour /* now have hours */
)*60 + min /* now have minutes */
)*60 + sec; /* finally seconds */
}
/*这个函数将时间转换成从1970年1月1日0小时0分0秒到你输入的时间所经过的秒数，溢出时间为2106-02-07 06:28:16。本人认为这个函数的使用应这样：若你要计算2000-02-07 06:28:16 到2000-02-09 06:28:16 所经过的秒数：unsigned long time1 = mktime(2000,2,7,6,28,16)-mktime(2000,2,9,6,28,16)； 若还要转成jiffies，就再加上:unsigned long time2 = time1*HZ.注意溢出的情况！*/
```

为了处理绝对时间，<linux/time.h> 导出了 do_gettimeofday 函数，它填充一个指向 struct timeval 的指针变量。绝对时间也可来自 xtime 变量，一个 struct timespec 值，为了原子地访问它，内核提供了函数 current_kernel_time。它们的精确度由硬件决定，原型是:

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
struct timespec current_kernel_time(void);
/*得到的数据都表示当前时间距UNIX时间基准1970－01－01 00：00：00的相对时间*/
```

以上两个函数在ARM平台都是通过 xtime 变量得到数据的。

全局变量xtime：它是一个timeval结构类型的变量，用来表示当前时间距UNIX时间基准1970 - 01 - 01 00：00：00的相对秒数值。

结构timeval是Linux内核表示时间的一种格式（Linux内核对时间的表示有多种格式，每种格式都有不同的时间精度），其时间精度是微秒。该结构是内核表示时间时最常用的一种格式，它定义在头文件include/linux/time.h中，如下所示：

```
struct timeval {
time_t tv_sec; /* seconds */
suseconds_t tv_usec; /* microseconds */
};
```

其中，成员tv_sec表示当前时间距UNIX时间基准的秒数值，而成员tv_usec则表示一秒之内的微秒值，且1000000>tv_usec>＝0。

Linux内核通过timeval结构类型的全局变量xtime来维持当前时间，该变量定义在kernel/timer.c文件中，如下所示：

```
/* The current time */
volatile struct timeval xtime __attribute__ ((aligned (16)));
```

但是，全局变量xtime所维持的当前时间通常是供用户来检索和设置的，而其他内核模块通常很少使用它（其他内核模块用得最多的是jiffies），因此对xtime的更新并不是一项紧迫的任务，所以这一工作通常被延迟到时钟中断的底半部（bottom half）中来进行。由于bottom half的执行时间带有不确定性，因此为了记住内核上一次更新xtime是什么时候，Linux内核定义了一个类似于jiffies的全局变量wall_jiffies，来保存内核上一次更新xtime时的jiffies值。时钟中断的底半部分每一次更新xtime的时侯都会将wall_jiffies更新为当时的jiffies值。全局变量wall_jiffies定义在kernel/timer.c文件中：

```
/* jiffies at the most recent update of wall time */
unsigned long wall_jiffies;
```

```
#include <linux/time.h>
 struct timeval t1;
 struct timeval t2;
 struct timeval t3;

 gettimeofday (&t1, NULL);*/

 usleep(2000);
 gettimeofday (&t2, NULL);
 printf("t1->tv_sec = %ld,t1->tv_usec = %ld\n",t1.tv_sec,t1.tv_usec);
 printf("t2->tv_sec = %ld,t2->tv_usec = %ld\n",t2.tv_sec,t2.tv_usec);
```

```
        printf("t2-t1 = %ld\n",1000000*(t2.tv_sec-t1.tv_sec)+(t2.tv_usec-
t1.tv_usec));
        }
        gettimeofday (&t2, NULL);
        gettimeofday (&t3, NULL);
        printf("t1->tv_sec = %ld,t1->tv_usec = %ld\n",t1.tv_sec,t1.tv_usec);
        printf("t2->tv_sec = %ld,t2->tv_usec = %ld\n",t2.tv_sec,t2.tv_usec);
        printf("t3->tv_sec = %ld,t3->tv_usec = %ld\n",t2.tv_sec,t2.tv_usec);
        printf("t2-t1 = %ld\n",1000000*(t2.tv_sec-t1.tv_sec)+(t2.tv_usec-
t1.tv_usec));*/
```