# 一、概要

时间管理在内核中占有非常重要的地位，内核中有大量的函数都是基于时间驱动的。有些函数需要周期执行，而有些需要等待一个相对时间后才运行，此外内核还必须管理系统的运行时间以及当前日期和时间。内核必须在硬件的帮助下才能计算和管理时间。

硬件为内核提供了一个系统定时器用以计算流逝的时间，该时钟在内核中可看成是一个电子时间资源。系统定时器以某种频率自行触发或射中时钟中断，当时钟中断发生时内核就通过一种特殊的中断处理程序对其进行处理。该频率可以通过编程预定(即节拍率)，连续两次时钟中断的间隔时间称为节拍（tick）。利用时间中断周期执行的主要工作：

A．更新系统运行时间

B．更新墙上时间

C．在smp系统上，均衡调度程序中各处理器上的运行队里

D．检查当前进程是否用尽了自己的时间片

E．运行超时的动态定时器

F．更新资源消耗和处理器时间的统计值

## 1、节拍率：HZ

系统定时器频率是通过静态预处理定义的，就是HZ，在系统启动时按照HZ值对硬件进行设置。不同体系结构，HZ不同，但大部分都是100即一个tick是10ms。

一个合适的HZ是相当重要的，下面分析了高HZ的优势和劣势：

优势：

A．内核定时器能够以更高的频度和更高的准确度运行

B．依赖定时值的系统调用（poll和select）能够以更高的精度运行

C．对诸如资源消耗和系统运行时间等的测量会有更精细的解析度

D．提高进程抢占的准确度

劣势：

A．中断频率越高，切换频繁，系统负担越重

B．中断处理程序占用的CPU时间越多

C．更频繁的打乱处理器高速缓存并增加耗电

## 2、jiffies

全局变量jiffies用来记录自系统启动以来产生的节拍的总数。内核给jiffies赋了一个特殊的初值，引起尽早溢出捕捉bug，每次时钟中断处理程序就会增加该变量的值。

在include/linux/jiffies.h文件中定义了该变量：

extern unsigned long volatile \_\_jiffy\_data jiffies;

jiffies为无符号长整形，在32位体系结构上是32位，在64位体系结构上是64位。

当jiffies变量的值超过它的最大存放范围后就会发生溢出，回绕到0。因此，内核提供了四个宏来帮助比较节拍计数，能正确地处理节拍计数回绕情况：

time\_after(a, b) time\_before(a, b) time\_after\_eq(a, b) time\_before\_eq(a, b)

## 3、硬件时钟和定时器

体系结构提供了两种设备进行计时：实时时钟和系统定时器。

实时时钟：

用来持久存放系统时间的设备，即便系统关闭后也可以靠主板上的微型电池提供的电力保持系统的计时。当系统启动时，内核通过读取RTC来初始化墙上时间，该时间存放在xtime变量中，然后定期同步两者时间保持一致，重启时可以得到一个相对准确的时间。

系统定时器：

内核定时机制中最为重要的角色，提供一种周期性触发中断机制。相应的中断处理程序主要完成以下工作：

A．更新jiffies

B．更新墙上时间

C．更新资源消耗的统计值，比如当前进程所消耗的系统时间和用户时间

D．执行到期的动态定时器

E．执行scheduler\_tick()函数

F．计算平均负载值

## 4、实际时间

当前实际时间定义在文件kernel/time/timekeeping.c中：

static struct timekeeper timekeeper

struct timekeeper {

…

u64 xtime\_sec;

u64 xtime\_nsec;

…

}

xtime\_sec以秒为单位，存放着自1970年1月1日（UTC）以来经过的时间，该时间点被称为纪元，多数Unix系统的墙上时间都是基于该纪元而言的。xtime\_nsec上一秒开始经过的ns数。

通过gettimeofday和settimeofday来获取和设置当前时间，设置时间需要具有CAP\_SYS\_TIME权能。

## 5、动态定时器

它是管理内核流逝的时间的基础，使用简单。只需要执行一些初始化工作，设置一个超时时间，指定超时发生后执行的函数，然后激活定时器就可以了。

定时器由结构timer\_list表示，定义在文件< include/linux/timer.h >

struct timer\_list {

/\*

\* All fields that change during normal runtime grouped to the

\* same cacheline

\*/

struct list\_head entry; //定时器链表入口

unsigned long expires; //以jiffies为单位的定时值

struct tvec\_base \*base; //定时器内部值，用户不使用

void (\*function)(unsigned long); //定时器处理函数

unsigned long data; //传递给处理函数的长整型参数

int slack;

…

}

使用方法如下：

创建定时器：struct timer\_list my\_timer

初始化定时器：init\_timer(&my\_timer)

定时值初始化：

my\_timer.expires = jiffies + delay

my\_timer.data = 0

my\_timer.function = my\_function( void my\_function(unsigned long data))

激活定时器：add\_timer(&my\_timer)

修改已经激活的定时器超时时间：mod\_timer(&my\_timer, jiffies+new\_delay)

在定时器超市前停止定时器：del\_timer(&my\_timer) 或 del\_timer\_sync(&my\_timer)

定时器作为软中断在下半部上下文中执行，时钟中断处理程序会通过raise\_softirq(TIMER\_SOFTIRQ) 唤醒定时器软中断，从而在当前处理器上运行所有的超时定时器。虽然所有定时器都以链表形式存放在一起，但是为了提高搜索效率，内核将定时器按照他们的超时时间划分成五组，当定时器超时时间接近时，定时器将随组一起下移。

## 6、延迟执行

内核代码（尤其是驱动程序）除了使用定时器或下半部机制以外，还需要其他方法来推迟执行任务。内核提供了许多延迟方法处理各种延迟要求：

A．忙等待

该方法想要延迟的时间是节拍的整数倍，或者精确率要求不高时使用

unsinged long timeout = jiffies + 10

while (time\_before(jiffies, timeout))

;

或者

while(time\_before(jiffies, delay))

cond\_resched();

B．短延迟：

void udelay(unsigned long usecs)

void ndelay(unsigned long nsecs)

void mdelay(unsigned long msecs)

/proc/cpuinfo的BogoMIPS主要被udelay()和mdelay()函数使用，记录处理器在给定时间内忙循环执行的次数。

C．schedule\_timeout()

该方法会让需要延迟执行的任务睡眠到指定的延迟时间耗尽后再重新运行。唯一的参数是延迟的相对时间，单位是jiffies。需要注意一点:

该函数需要调用调度程序，所以调用它的代码必须保证能够睡眠。换句话说，调用代码必须处于进程上下文中，并且不能持有锁。

# 二、linux时间框架

## 1、框架

随着技术发展，出现了下面两种新的需求：

（1）嵌入式设备需要较好的电源管理策略。传统的linux会有一个周期性的时钟，即便是系统无事可做的时候也要醒来，这样导致系统不断的从低功耗（idle）状态进入高功耗的状态。这样的设计不符合电源管理的需求。

（2）多媒体的应用程序需要非常精确的timer，例如为了避免视频的跳帧、音频回放中的跳动，这些需要系统提供足够精度的timer。

因此，内核时间子系统也进行了框架调整，引进了高精度timer。和低精度timer不同，高精度timer使用了人类的最直观的时间单位ns（低精度timer使用的tick是和内核配置相关，不够直接）。本质上linux kernel提供了高精度timer之后，其实不必提供低精度timer了，不过由于低精度timer存在了很长的历史，并且渗入到内核各个部分，如果去掉容易引起linux kernel稳定性和健壮性的问题，因此kernel保持两种并存。其示意图如下：



在smp情况下，driver硬件定时器划分成了两部分：一个提供给clocksource模块使用，一个提供给clockevent事件使用。而clockevent又简单化为两个小部分：一个是每个CPU自己的定时器，一个是全局定时器；每个CPU定时器管理本CPU的任务运行情况、资源统计等，第一个启动CPU（一般是CPU0）还会更新tick和墙上时钟，而global timer则主要用于低功耗模式下CPU进入睡眠时唤醒所有CPU。

Kernel抽象了底层驱动，划分为clockevent和clocksource连个模块，驱动加载时会调用两个模块接口进行注册。clocksource的精度就是定时器时钟频率的精度（ns级别），可以认为是一个timeline，而clockevent则是这个timeline上的特定时间点，产生中断后调用相应的callback函数进行事件处理。

tick device layer 基于clock event设备进行工作：每个CPU都有自己唯一的tick device，管理自己的调度，进程统计等。Tick device可以工作在两种模式：periodic 和 one shot mode。有多少CPU就有多少个tick device称之为local tick device，在所有的local tick device中会有一个被选为global tick device（一般是CPU0的tick device），该device负责维护整个系统的jiffies，更新wall clock，计算全局负荷什么的。

高精度hrtimer需要高精度的clockevent，工作在one shot mode的tick device提供高精度的clockevent。虽然有了高精度hrtimer的出现，内核并没有抛弃老的低精度timer机制。当系统处于高精度timer的时候，系统会setup一个特别的高精度hrtimer（sched\_timer），该高精度timer会周期性的触发，从而模拟传统的periodic tick，推动传统低精度timer的运转（代码没有弄明白sched\_timer？）。

## 2、内核配置

（1）GENERIC\_CLOCKEVENTS和GENERIC\_CLOCKEVENTS\_BUILD：代表使用新的时间子系统架构，默认就是配置好的

（2）新时间子系统框架下，Timers subsystem配置选项主要和tick 已经是否支持高精度hrtimer有关。Tick有2种配置（二选一）：

CONFIG\_HZ\_PERIODIC：无论何时都启用周期性的tick，即便是在系统idle的时候。

CONFIG\_NO\_HZ\_IDLE：该选项默认会打开 CONFIG\_TICK\_ONESHOT 和CONFIG\_NO\_HZ\_COMMON，在系统idle的时候，停掉周期性tick。

CONFIG\_HIGH\_RES\_TIMERS：支持高精度hrtimer

配置了高精度hrtimer或NO\_HZ\_COMMOM就一定配置CONFIG\_TICK\_ONESHOT，表示系统支持one-shot类型的tick\_device。

因此，在新时间子系统下有4种配置：

A．低精度timer和周期tick

B．低精度timer和dynamic tick（tickless idle）（当前系统使用情况）

C．高精度hrtimer和周期tick

D．高精度hrtimer和dynamic tick（tickless idle）

## 3、四种配置（转载自附录网站）

### （1）低精度timer + 周期tick

我们首先看周期性tick的实现。起始点一定是底层的clock source chip driver ，该driver会调用接口clockevents\_register\_device向clock event注册。一旦增加了一个clock event device，需要通知上层的tick device layer，有可能新注册的这个device更好、更适合某个tick device。要是这个clock event device被某个tick device收留了（要么该tick device之前没有匹配的clock event device，要么新的clock event device更适合该tick device），那么就启动对该tick device的配置（参考tick\_setup\_device）。根据当前系统的配置情况（周期性tick），会调用tick\_setup\_periodic函数，这时候，该tick device对应的clock event device的clock event handler被设置为tick\_handle\_periodic。底层硬件会周期性的产生中断，从而会周期性的调用tick\_handle\_periodic从而驱动整个系统的运转。需要注意的是：即便是配置了CONFIG\_NO\_HZ和CONFIG\_TICK\_ONESHOT，系统中没有提供one shot的clock event device，这种情况下，整个系统仍然是运行在周期tick的模式下。

下面来到低精度timer模块了，其实即便没有使能高精度timer，内核也会把高精度timer模块的代码编译进kernel的image中，这一点可以从Makefile文件中看出。在这种构架下，各个内核模块也可以调用linux kernel中的高精度timer模块的接口函数来实现高精度timer，但是，这时候高精度timer模块是运行在低精度的模式，也就是说这些hrtimer虽然是按照高精度timer的红黑树进行组织，但是系统只是在每一周期性tick到来的时候调用hrtimer\_run\_queues函数，来检查是否有expire的hrtimer。毫无疑问，这里的高精度timer也就是没有意义了。

由于存在周期性tick，低精度timer的运作毫无压力，和过去一样。

### （2）低精度timer + Dynamic Tick

系统开始的时候并不是直接进入Dynamic tick mode的，而是经历一个切换过程。开始的时候，系统运行在周期tick的模式下，各个cpu对应的tick device的（clock event device的）event handler是tick\_handle\_periodic。在timer的软中断上下文中，会调用tick\_check\_oneshot\_change进行是否切换到one shot模式的检查，如果系统中有支持one-shot的clock event device，并且没有配置高精度timer的话，那么就会发生tick mode的切换（调用tick\_nohz\_switch\_to\_nohz），这时候，tick device会切换到one shot模式，而event handler被设置为tick\_nohz\_handler。由于这时候的clock event device工作在one shot模式，因此当系统正常运行的时候，在event handler中每次都要reprogram clock event，以便正常产生tick。当cpu运行idle进程的时候，clock event device不再reprogram产生下次的tick信号，这样，整个系统的周期性的tick就停下来。

### （3）高精度timer + Dynamic Tick

同样的，系统开始的时候并不是直接进入Dynamic tick mode的，而是经历一个切换过程。系统开始的时候是运行在周期tick的模式下，event handler是tick\_handle\_periodic。在周期tick的软中断上下文中（参考run\_timer\_softirq），如果满足条件，会调用hrtimer\_switch\_to\_hres将hrtimer从低精度模式切换到高精度模式上。这时候，系统会有下面的动作：

A．Tick device的clock event设备切换到oneshot mode（参考tick\_init\_highres函数）

B．Tick device的clock event设备的event handler会更新为hrtimer\_interrupt（参考tick\_init\_highres函数）

C．设定sched timer（也就是模拟周期tick那个高精度timer，参考tick\_setup\_sched\_timer函数）

这样，当下一次tick到来的时候，系统会调用hrtimer\_interrupt来处理这个tick（该tick是通过sched timer产生的）。

在Dynamic tick的模式下，各个cpu的tick device工作在one shot模式，该tick device对应的clock event设备也工作在one shot的模式，这时候，硬件Timer的中断不会周期性的产生，但是linux kernel中很多的模块是依赖于周期性的tick的，因此，在这种情况下，系统使用hrtime模拟了一个周期性的tick。在切换到dynamic tick模式的时候会初始化这个高精度timer，该高精度timer的回调函数是tick\_sched\_timer。这个函数执行的函数类似周期性tick中event handler执行的内容。不过在最后会reprogram该高精度timer，以便可以周期性的产生clock event。当系统进入idle的时候，就会stop这个高精度timer，这样，当没有用户事件的时候，CPU可以持续在idle状态，从而减少功耗。

### （4）高精度timer + 周期性Tick

这种配置不多见，多半是由于硬件无法支持one shot的clock event device，这种情况下，整个系统仍然是运行在周期tick的模式下。

# 三、用户接口

## 1、系统时间相关服务

### （1）秒级函数：time和stime

#include <time.h>

time\_t time(time\_t \*t); //获取时间秒

int stime(time\_t \*t); //设置时间秒

对应的系统调用为：sys\_time和sys\_stime。time函数返回当前点到linux epoch的秒数，stime设定当前时间点到linux epoch的秒数。

与上面函数配套的还有一系列时间点与linux epoch转换函数：mktime, localtime\_r。

### （2）微秒级函数：gettimeofday和settimeofday

#include <sys/time.h>

int gettimeofday(struct timeval \*tv, struct timezone \*tz);

int settimeofday(const struct timeval \*tv, const struct timezone \*tz);

struct timeval {

time\_t tv\_sec; /\* seconds \*/

suseconds\_t tv\_usec; /\* microseconds \*/

};

struct timezone {

int tz\_minuteswest; /\* minutes west of Greenwich \*/

int tz\_dsttime; /\* type of DST correction \*/

};

对应的系统调用:sys\_gettimeofday和sys\_settimeday。Gettimeofday获取linux epoch到当前时间点的秒数以及微秒数；settimeofday则设定从linux epoch到当前时间点的秒数以及微秒数。

值得一提的是：这些系统调用在新的POSIX标准中接口被clock\_gettime和clock\_settime取代。

### （3）纳秒级别的时间函数：clock\_gettime和clock\_settime

#include <time.h>

int clock\_getres(clockid\_t clk\_id, struct timespec \*res); //获取clock\_id的系统时钟精度

int clock\_gettime(clockid\_t clk\_id, struct timespec \*tp);

int clock\_settime(clockid\_t clk\_id, const struct timespec \*tp);

struct timespec {

time\_t tv\_sec; /\* seconds \*/

long tv\_nsec; /\* nanoseconds \*/

};

clk\_id识别system clock的ID，定义如下：

CLOCK\_REALTIME：真实的墙上时钟，前面函数就是从获取该ID的值

CLOCK\_MONOTONIC：该时钟是单调递增的，也是真实的墙上时钟只是起始点不一定是linux epoch，一般会把系统启动的时间点设定为基准点。除了NTP和adjtime对该时钟进行调整外，其他任何接口不允许设定该时钟，保证该时钟的单调性。可以了解系统启动时间。

CLOCK\_MONOTONIC\_RAW：具备CLOCK\_MONOTONIC的特性，但它不受NTP和adjtime影响，是完全基于本地晶振的时钟。一般程序员不大用。

CLOCK\_BOOTTIME：类似CLOCK\_MONOTONIC，但是系统suspend时依然增加。

CLOCK\_PROCESS\_CPUTIME\_ID：每个CPU的高精度进程定时器，clock\_getcpuclockid获取进程的clock\_id

CLOCK\_THREAD\_CPUTIME\_ID：线程的CPU时间，pthread\_getcpuclockid获取线程的clock\_id。

### （4）系统时钟调整

上面设定系统时间是一个比较粗暴的做法，一旦修改了系统时间，系统中很多以来绝对时间的进程会有各种奇怪的行为。所以系统提供了时间同步的接口函数，可以让外部的精准计时服务器不断的修正系统时钟。

A．adjtime

int adjtime(const struct timeval \*delta, struct timeval \*olddelta);

struct timeval {

time\_t tv\_sec; /\* seconds \*/

suseconds\_t tv\_usec; /\* microseconds \*/

};

该函数可以根据delta参数缓慢的修正系统时钟（CLOCK\_REALTIME那个）。olddelta返回上一次调整中尚未完成的delta。

B．adjtimex

#include <sys/timex.h>

int adjtimex(struct timex \*buf);

struct timex {

int modes; /\* mode selector \*/

long offset; /\* time offset (usec) \*/

long freq; /\* frequency offset (scaled ppm) \*/

long maxerror; /\* maximum error (usec) \*/

long esterror; /\* estimated error (usec) \*/

int status; /\* clock command/status \*/

long constant; /\* pll time constant \*/

long precision; /\* clock precision (usec) (read-only) \*/

long tolerance; /\* clock frequency tolerance (ppm)

(read-only) \*/

struct timeval time; /\* current time (read-only) \*/

long tick; /\* usecs between clock ticks \*/

};

该函数用来显示或这修改linux内核的时间变量的工具，提供了对与内核时间变量直接访问功能，可以实现对于系统时间的飘逸进行修正。任何用户都可以使用它查看，但是只有root用户才可以更改这些参数。

## 2、进程睡眠

### （1）秒级函数：sleep

#include <unistd.h>

unsigned int sleep(unsigned int seconds);

该函数会导致当前进程sleep seconds之后（基于CLOCK\_REALTIME）返回继续执行程序。返回值说明了进程没有进入睡眠的时间。

### （2）微秒级别函数：usleep

#include <unistd.h>

int usleep(useconds\_t usec)；

该函数功能和上面一样，不过返回值定义不同。0：表示执行成功，-1：执行失败，错误码在errno中。

### （3）纳秒级别函数：nanosleep

#include <time.h>

int nanosleep(const struct timespec \*req, struct timespec \*rem);

struct timespec {

time\_t tv\_sec; /\* seconds \*/

long tv\_nsec; /\* nanoseconds \*/

};

该函数取代了usleep函数，req中设定你要sleep的秒以及纳秒值，rem表示还有多少时间没睡完。返回0表示成功，返回-1说明失败。

sleep/usleep/nanosleep的系统都是通过kernel的sys\_nanosleep系统调用实现（底层基于hrtimer）。

### （4）更高级的sleep函数：clock\_nanosleep

#include <time.h>

int clock\_nanosleep(clockid\_t clock\_id, int flags,

const struct timespec \*request,

struct timespec \*remain);

clock\_id说明该函数不仅能基于real\_time clock睡眠，还可以基于其他的系统时钟睡眠。flag等0或1，分别指明request参数设定的时间值是相对时间还是绝对时间。

## 3、timer相关的服务

### （1）alarm函数

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

该函数在指定秒数（基于CLOCK\_REALTIME）的时间过去后，向该进程发送SIGALRM信号。调用该接口的程序需要设定signal handler。

# 四、代码解析

第二章节讲述内核配置有两种主要模式：periodic 和 one-shot，下面代码分析主要根据实际使用的低精度tick和dynamic tick (tickles tick)即使用one-shot配置进行讲解(Linux-3.10.y)。

## 1、数据结构

### （1）clocksource

内核使用struct clocksource数据结构记录时钟源所有信息，主要作为系统时间的基准，当有多个时钟源时选择最优那个，没有时钟源时默认使用基于jiffies的时钟clocksource\_jiffies。内核通过一个链表clocksource\_list管理所有注册的时钟源，每个时钟源定义了一个单调增加的计数器并以ns为单位。

struct clocksource结构体详细如下（include/linux/clocksource.h）:

struct clocksource {

cycle\_t (\*read)(struct clocksource \*cs); //读取指定CS的cycle值（定时器当前计数值）

cycle\_t cycle\_last; //保存最近一次read的cycle值（其中一个重要作用翻转）

cycle\_t mask; //counter是32位还是64位

//公式：ns = (cycles/F) \* NSEC\_PER\_SEC = (cycle \* mult) >> shift

u32 mult; //cycle转化为ns的乘数

u32 shift; //cycle转化为ns的除数，采用移位的方式

u64 max\_idle\_ns; //该时钟允许的最大空闲时间（没搞明白如何用）

u32 maxadj; //最大调整值与mult相关

#ifdef CONFIG\_ARCH\_CLOCKSOURCE\_DATA //未用

struct arch\_clocksource\_data archdata;

#endif

const char \*name; //时钟源名字

struct list\_head list; //注册时钟源链表头

int rating; //时钟源精度值，

1--99： 不适合于用作实际的时钟源，只用于启动过程或用于测试；

100--199：基本可用，可用作真实的时钟源，但不推荐；

200--299：精度较好，可用作真实的时钟源；

300--399：很好，精确的时钟源；

400--499：理想的时钟源，如有可能就必须选择它作为时钟源；

int (\*enable)(struct clocksource \*cs); //使能时钟源

void (\*disable)(struct clocksource \*cs); //禁止时钟源

unsigned long flags; //时钟源属性，CLOCK\_SOURCE\_IS\_CONTINUOUS连续时钟

void (\*suspend)(struct clocksource \*cs); //挂起时钟源

void (\*resume)(struct clocksource \*cs); //恢复时钟源

#ifdef CONFIG\_CLOCKSOURCE\_WATCHDOG //未用

/\* Watchdog related data, used by the framework \*/

struct list\_head wd\_list;

cycle\_t cs\_last;

cycle\_t wd\_last;

#endif

}\_\_\_\_cacheline\_aligned;

### （2）clockevent

内核使用struct clock\_event\_device数据结构记录时钟的事件信息，包括硬件时钟中断发生时要执行的那些操作。提供了对周期性事件和单触发事件的支持。还提供了高精度定时器和动态定时器的支持。内核通过一个clockevent\_devices管理所有注册的clock event设备。

该结构体在头文件include/linux/clockchips.h中定义，详细定义如下：

struct clock\_event\_device {

void (\*event\_handler)(struct clock\_event\_device \*); //事件处理函数，主要有三种：peridioc\one-shot\broadcast

int (\*set\_next\_event)(unsigned long evt, struct clock\_event\_device \*); //设置下次触发事件基于clocksource 的cycles

int (\*set\_next\_ktime)(ktime\_t expires, struct clock\_event\_device \*); //设置下次触发事件基于ktime（用得比较少）

ktime\_t next\_event;

u64 max\_delta\_ns; //可设置的最大时间差

u64 min\_delta\_ns; //可设置的最小时间差

u32 mult; //和clocksource一样

u32 shift; //和clocksource一样

enum clock\_event\_mode mode; //clockevent工作模式，见下面

unsigned int features; //clockevent设备特征，见下面

unsigned long retries;

void (\*broadcast)(const struct cpumask \*mask); //广播所有CPU函数

void (\*set\_mode)(enum clock\_event\_mode mode, struct clock\_event\_device \*); //设置模式

void (\*suspend)(struct clock\_event\_device \*);

void (\*resume)(struct clock\_event\_device \*);

unsigned long min\_delta\_ticks;

unsigned long max\_delta\_ticks;

const char \*name;

int rating;

int irq;

const struct cpumask \*cpumask; //CPU掩码，判断是否属于某一个CPU，或广播支持的CPU

struct list\_head list;

} \_\_\_\_cacheline\_aligned;

clockevent设备工作模式：

enum clock\_event\_mode {

CLOCK\_EVT\_MODE\_UNUSED = 0,

CLOCK\_EVT\_MODE\_SHUTDOWN, //关闭模式

CLOCK\_EVT\_MODE\_PERIODIC, //周期性模式

CLOCK\_EVT\_MODE\_ONESHOT, //单次模式

CLOCK\_EVT\_MODE\_RESUME, //恢复模式

};

clockevent设备特征：

#define CLOCK\_EVT\_FEAT\_PERIODIC 0x000001 //可以产生周期触发事件特征

#define CLOCK\_EVT\_FEAT\_ONESHOT 0x000002 //可以产生单触发事件特征

#define CLOCK\_EVT\_FEAT\_KTIME 0x000004 //产生事件的事件基准ktime

//X86下使用，进入省电情况

#define CLOCK\_EVT\_FEAT\_C3STOP 0x000008 //clocksource停止，需要广播事件支持，本ARM平台也使用了该选项

#define CLOCK\_EVT\_FEAT\_DUMMY 0x000010 // Local APIC timer使用该选项

### （3）tick\_device

struct tick\_device只是对struct clock\_event\_device的一个封装，加入了运行模式变量，支持PERIODIC和ONESHOT两种模式。

struct tick\_device {

struct clock\_event\_device \*evtdev;

enum tick\_device\_mode mode;

};

enum tick\_device\_mode {

TICKDEV\_MODE\_PERIODIC,

TICKDEV\_MODE\_ONESHOT,

};

## 2、内核初始化

在内核启动函数start\_kernel里对时间系统进行了初始化

### （1）tick\_init

该函数初始化tick控制。向clockevents\_chain通知链中添加一个tick通知分发器tick\_notifier（分发回调函数：tick\_notify）。在底层驱动注册设备时，CLOCK\_EVT\_NOTIFY\_ADD消息就是添加了一个新的clockevent设备。

初始化tick broadcast掩码，如果配置了CONFIG\_TICK\_ONESHOT相关掩码也要初始化。

### （2）init\_timers

初始化本CPU上的低精度定时器相关的数据结构，将通知分发器timers\_nb添加到cpu\_chain通知链；初始化定时器软中断open\_softirq(TIMER\_SOFTIRQ, run\_timer\_softirq);

### （3）hrtimers\_init

初始化本CPU上的高精度精度定时器相关的数据结构，将通知分发器hrtimers\_nb添加到cpu\_chain通知链；如果开启高精度定时器宏，则初始化高精度定时器软中断open\_softirq(HRTIMER\_SOFTIRQ, run\_hrtimer\_softirq)（本平台为使用）。

### （4）timekeeping\_init

初始化时钟源clocksource及timekeeping模块时间初始值，如果平台没有更好的时钟源，系统使用jiffies作为时钟源。

clock = clocksource\_default\_clock();

struct clocksource \* \_\_init \_\_weak clocksource\_default\_clock(void)

{

return &clocksource\_jiffies;

}

struct clocksource clocksource\_jiffies = {

.name = "jiffies",

.rating = 1, /\* lowest valid rating\*/

.read = jiffies\_read,

.mask = 0xffffffff, /\*32bits\*/

.mult = NSEC\_PER\_JIFFY << JIFFIES\_SHIFT, /\* details above \*/

.shift = JIFFIES\_SHIFT,

};

### （5）time\_init

前面函数时内核通用架构，该函数为硬件时钟初始化平台相关，一般由各个平台自己实现，细节见下节。

void \_\_init time\_init(void)

{

if (machine\_desc->init\_time)

machine\_desc->init\_time(); （=hi3536\_timer\_init）

else

clocksource\_of\_init();

sched\_clock\_postinit();

}

## 3、硬件时钟初始化

### （1）平台注册

MACHINE\_START(HI3536, "hi3536")

.atag\_offset = 0x100,

.map\_io = hi3536\_map\_io,

.init\_early = hi3536\_init\_early,

.init\_irq = hi3536\_gic\_init\_irq,

#ifdef CONFIG\_HI3536\_SYSCNT

.init\_time = arch\_timer\_init,

#else

.init\_time = hi3536\_timer\_init,

#endif

.init\_machine = hi3536\_init,

.smp = smp\_ops(hi3536\_smp\_ops),

.reserve = hi3536\_reserve,

.restart = hi3536\_restart,

MACHINE\_END

上一节machine\_desc的具体实现即为该宏定义，init\_time就是hi3536\_timer\_init，其具体内容见下面。

### （2）平台定时器初始化

void \_\_init hi3536\_timer\_init(void)

{

/\* 设置所有定时器的工作时钟（未初始化，默认3MHZ）

Hi3536有time0~9 10个时钟

根据配置，所有定时器都配置成了总线时钟125Mhz(8 ns)

\*/

writel(readl((const volatile void \*)IO\_ADDRESS(REG\_BASE\_SCTL)) |

(1 << 16) | (1 << 18),

(volatile void \*)IO\_ADDRESS(REG\_BASE\_SCTL));

#ifdef CONFIG\_SP804\_LOCAL\_TIMER

hi3536\_local\_timer\_init(); //每个CPU使用一个定时器作为local timer，该平台有4核CPU0~CPU3分别对应：timer4~timer7，注册为clockevent设备

#endif

hi3536\_clocksource\_init((void \*)TIMER(0)->addr,

TIMER(0)->name); //timer0作为clocksource设备

sp804\_clockevents\_init((void \*)TIMER(1)->addr,

TIMER(1)->irq.irq, TIMER(1)->name); //timer1作为clockevent设备的global timer

}

从上面看，hi3536总共支持10个timer，实际使用了6个，timer0/timer1，timer4~timer7，其他保留。其注册的顺序如下：timer0(clocksource) —> timer1(clockevent global timer) —> timer4(clockevent cpu0 local timer) —> time5~7(clockevent cpu1~3 local timer)。每类定时器的初始化，见下面细节分析。

### （3）clocksource初始化

初始化函数源码如下：

static void \_\_init hi3536\_clocksource\_init(void \_\_iomem \*base, const char \*name)

{

long rate = sp804\_get\_clock\_rate(name); //获取定时器时钟62.5MHz

struct clocksource \*clksrc = &hi3536\_clocksource.clksrc;

if (rate < 0)

return;

clksrc->name = name; //name=timer0

clksrc->rating = 200; //时钟源精度值

clksrc->read = hi3536\_clocksource\_read; //获取计数值，系统主要调用该接口转化为系统时间

clksrc->mask = CLOCKSOURCE\_MASK(32), //计数值32位

clksrc->flags = CLOCK\_SOURCE\_IS\_CONTINUOUS, //持续的时钟源

clksrc->resume = hi3536\_clocksource\_resume,

hi3536\_clocksource.base = base;

hi3536\_clocksource\_start(base); //初始化寄存器

clocksource\_register\_hz(clksrc, rate); //计算出mult和shift，为系统选择更好的时钟源

setup\_sched\_clock(hi3536\_sched\_clock\_read, 32, rate); //通用sched\_clock模块，这个模块主要是提供一个sched\_clock的接口函数，获取当前时间点和系统启动之间的纳秒值。

}

### （4）per CPU定时器初始化

每CPU定时器初始化函数：

static void \_\_init hi3536\_local\_timer\_init(void)

{

unsigned int cpu = 0;

unsigned int ncores = num\_possible\_cpus(); //获取CPU个数

local\_timer\_rate = sp804\_get\_clock\_rate("sp804"); //获取定时器时钟

for (cpu = 0; cpu < ncores; cpu++) { //为每个CPU分配各自的定时器

struct hi\_timer\_t \*cpu\_timer = GET\_SMP\_TIMER(cpu);

cpu\_timer->irq.handler = sp804\_timer\_isr; //中断处理函数

cpu\_timer->irq.dev\_id = (void \*)cpu\_timer; //定时器分别是timer0~3

setup\_irq(cpu\_timer->irq.irq, &cpu\_timer->irq); //注册中断号

disable\_irq(cpu\_timer->irq.irq); //关闭中断

}

local\_timer\_register(&hi3536\_timer\_tick\_ops); //注册定时器操作函数

/\* 以上只是为每个CPU分配了定时器，并没有注册每个cpu的clockevent，启动CPU（CPU0）在稍后注册，而其他次CPU则直到kernel\_init启动它们后才会注册 \*/

}

//定时器注册clockevent操作函数

static struct local\_timer\_ops hi3536\_timer\_tick\_ops \_\_cpuinitdata = {

.setup = hi3536\_local\_timer\_setup,

.stop = hi3536\_local\_timer\_stop,

};

static int \_\_cpuinit hi3536\_local\_timer\_setup(struct clock\_event\_device \*evt)

{

unsigned int cpu = smp\_processor\_id();

struct hi\_timer\_t \*timer = GET\_SMP\_TIMER(cpu);

struct irqaction \*irq = &timer->irq;

evt->name = timer->name;

evt->irq = irq->irq;

evt->features = CLOCK\_EVT\_FEAT\_PERIODIC | CLOCK\_EVT\_FEAT\_ONESHOT

| CLOCK\_EVT\_FEAT\_C3STOP;

evt->set\_mode = sp804\_set\_mode;

evt->set\_next\_event = sp804\_set\_next\_event;

evt->rating = 350;

timer->priv = (void \*)evt;

clockevents\_config\_and\_register(evt, local\_timer\_rate, 0xf, 0xffffffff); //注册clockevent

irq\_set\_affinity(evt->irq, evt->cpumask);

enable\_irq(evt->irq);

return 0;

}

//中断处理函数

static irqreturn\_t sp804\_timer\_isr(int irq, void \*dev\_id)

{

struct hi\_timer\_t \*timer = (struct hi\_timer\_t \*)dev\_id;

unsigned int clkevt\_base = timer->addr;

struct clock\_event\_device \*evt

= (struct clock\_event\_device \*)timer->priv;

/\* clear the interrupt \*/

writel(1, IOMEM(clkevt\_base + TIMER\_INTCLR));

evt->event\_handler(evt); //periodic 和 one-shot模式处理函数不一样，见4小节细节。

return IRQ\_HANDLED;

}

### （5）per CPU注册clockevent

上面只是初始化为每个CPU分配一个定时器，并没有将定时器注册到clockevent。而注册则是在各个CPU启动后，主CPU和次CPU注册是分开的，具体如下：

A．主CPU（cpu0）

注册过程：kernel\_init —> kernel\_init\_freeable —> smp\_prepare\_cpus —> percpu\_timer\_setup :

static void \_\_cpuinit percpu\_timer\_setup(void)

{

unsigned int cpu = smp\_processor\_id();

struct clock\_event\_device \*evt = &per\_cpu(percpu\_clockevent, cpu);

evt->cpumask = cpumask\_of(cpu);

if (!lt\_ops || lt\_ops->setup(evt)) //这里的setup就是前面初始化的：hi3536\_local\_timer\_setup，将CPU0的timer注册到clockevent设备。

broadcast\_timer\_setup(evt);

}

B．次CPU（cpu1~cpu3）

注册过程：kernel\_init —> kernel\_init\_freeable —> smp\_init：

void \_\_init smp\_init(void)

{

…

/\* FIXME: This should be done in userspace --RR \*/

for\_each\_present\_cpu(cpu) {

if (num\_online\_cpus() >= setup\_max\_cpus)

break;

if (!cpu\_online(cpu)) //启动所有未启动的CPU

cpu\_up(cpu);

}

…

}

cpu\_up —> \_cpu\_up —> \_\_cpu\_up —> boot\_secondary —> smp\_ops.smp\_boot\_secondary —> hi3536\_boot\_secondary —> hi3536\_secondary\_startup(汇编) —> secondary\_startup(汇编) —> \_\_secondary\_switched(汇编) —> secondary\_start\_kernel —> percpu\_timer\_setup该函数就是上面主CPU注册本地定时器的函数。

### （6）clockevent初始化

这里主要是注册clockevent的global timer，在periodic模式下不起作用，在one-shot模式下才会作用。注册过程过程如下：

sp804\_clockevents\_init —> \_\_sp804\_clockevents\_init —> clockevents\_config\_and\_register —> clockevents\_register\_device —> clockevents\_do\_notify —> tick\_notify（tick\_init初始化的通知分发器） —> tick\_check\_new\_device

## 4、periodic和one-shot模式

针对clockevent有periodic和one-shot两种模式，主要的不同点：event\_handler事件处理函数不同，细节见下列分析。

### （1）periodic模式

根据上一节的注册初始化，其先后顺序如下：timer1(clockevent global timer) —> timer4(clockevent cpu0 local timer) —> time5~7(clockevent cpu1~3 local timer)。

注册timer1时运行在CPU0上，次CPU1~3还没有启动，这时tick\_check\_new\_device接口会把timer1注册为CPU0的local timer，此刻timer1的event\_handler= tick\_handle\_periodic。当CPU0调用smp\_prepare\_cpus注册自己的local timer时会用timer4替换timer1，此刻timer4的event\_handler= tick\_handle\_periodic，而timer1的event\_handler在tick\_setup\_device中被修改为clockevents\_handle\_noop（空实现），并在接口tick\_check\_broadcast\_device中被注册为global timer设备（在周期模式下不起任何作用）。

CPU1~3启动时调用secondary\_start\_kernel注册local timer时只有一个定时器分别是timer5~7，它们的event\_handler = tick\_handle\_periodic。

clockevents\_handle\_noop：空实现

tick\_handle\_periodic主要完成如下工作：

A．如果是CPU0，则调用do\_timer完成tick更新和墙上时钟更新

B．update\_process\_times完成：启动本地软中断；更新本CPU的运行队列；调度任务；SMP下触发运行队列均衡。

### （2）one-shot模式

设备启动时一开始是periodic模式，过程和上面一模一样。各个CPU触发本地软中断后发生切换，run\_timer\_softirq —> hrtimer\_run\_pending —> tick\_check\_oneshot\_change —> tick\_nohz\_switch\_to\_nohz —> tick\_switch\_to\_oneshot：

将CPU0~3的event\_handler切换为：tick\_nohz\_handler

将timer1的event\_handler切换为：tick\_handle\_oneshot\_broadcast

切换只会发生一次，切换好后hrtimer\_run\_pending接口就会直接返回。

tick\_nohz\_handler：

和tick\_handle\_periodic做的事情大致一样。

tick\_handle\_oneshot\_broadcast主要完成：唤醒各个CPU，补偿tick的偏差。

one-shot模式下event\_handler处理函数每次都要重新设置next\_event。该模式的好处就是可以节省CPU功耗。

## 5、系统调用例子讲解

### （1）gettimeofday

该函数主要用于获取微妙级别的时间。其对应的系统调用为sys\_gettimeofday，实际起作用的是do\_gettimeofday：

void do\_gettimeofday(struct timeval \*tv)

{

struct timespec now;

getnstimeofday(&now);

tv->tv\_sec = now.tv\_sec;

tv->tv\_usec = now.tv\_nsec/1000;

}

int \_\_getnstimeofday(struct timespec \*ts)

{

struct timekeeper \*tk = &timekeeper;

unsigned long seq;

s64 nsecs = 0;

do {

seq = read\_seqcount\_begin(&timekeeper\_seq); //使用顺序锁进行数据同步访问

ts->tv\_sec = tk->xtime\_sec; //获取秒数，xtime\_sec更新由update\_all\_time进行

nsecs = timekeeping\_get\_ns(tk); //调用clocksource的read函数（即上面的hi3536\_clocksource\_read），将计数转化为相应的ns数

} while (read\_seqcount\_retry(&timekeeper\_seq, seq));

ts->tv\_nsec = 0;

timespec\_add\_ns(ts, nsecs); //调整ns数，传递给用户

…..

}

### （2）nanosleep

这是一个ns级别的睡眠函数，对应的系统调用sys\_nanosleep（kernel/hrtimer.c），实际起作用的是hrtimer\_nanosleep—> do\_nanosleep：

static int \_\_sched do\_nanosleep(struct hrtimer\_sleeper \*t, enum hrtimer\_mode mode)

{

//初始化一个新的hrtimer

hrtimer\_init\_sleeper(t, current);

do {

set\_current\_state(TASK\_INTERRUPTIBLE); //设置当前任务睡眠

hrtimer\_start\_expires(&t->timer, mode); //将新的hrtimer加入到timer\_list

if (!hrtimer\_active(&t->timer)) //如果没有激活hrtimer则直接退出

t->task = NULL;

if (likely(t->task)) //如果激活了，就开始调度

freezable\_schedule();

hrtimer\_cancel(&t->timer); //运行到这里，说明定时器到期，那么取消定时器。

mode = HRTIMER\_MODE\_ABS;

} while (t->task && !signal\_pending(current));

\_\_set\_current\_state(TASK\_RUNNING); //设置进程状态可执行

return t->task == NULL;

}

不管内核有没有配置HIGH\_RES\_TIMERS，内核都编译httimer.c接口。如果没有配置hrtimer则使用低精度的tick方案，这时定时器是相当不准确；如果进行了配置，则使用高精度方案。两种方案下，定时器中断处理方法不同：

低精度下的hrtimer：

update\_process\_times —> run\_local\_timers —> hrtimer\_run\_queues

高精度下的hrtimer：

run\_hrtimer\_softirq—> hrtimer\_peek\_ahead\_timers —> \_\_hrtimer\_peek\_ahead\_timers —> hrtimer\_interrupt

# 附录A

参考资料

<http://www.wowotech.net/timer_subsystem/time-subsyste-architecture.html>

<http://www.wowotech.net/timer_subsystem/timer_subsystem_userspace.html>