Linux时间子系统

何良斌

# 1、相关数据结构

Linux为了管理硬件时钟，主要分为两类clocksource和clockevents。

（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

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）clockevents

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。

本文主要讲系统时间，该结构体在头文件include/linux/clockchips.h中定义，不展开讲解。

# 2、内核相关初始化函数

在内核启动函数start\_kernel里对时间系统进行了初始化。

（1）tick\_init

tick为时钟事件设备，该函数初始化tick控制，基于clockevent框架注册一个tick分发器tick\_notifier（分发回调函数：tick\_notify），并加入队列clockevents\_chain。

（2）init\_timers

通用定时器初始化，注册timers\_nb，加入到cpu\_chain队列；开启定时器软中断open\_softirq(TIMER\_SOFTIRQ, run\_timer\_softirq);

（3）hrtimers\_init

高精度定时器初始化，注册hrtimers\_nb，加入到cpu\_chain队列；如果开启高精度定时器宏，则开启高精度定时器软中断open\_softirq(HRTIMER\_SOFTIRQ, run\_hrtimer\_softirq);

（4）timekeeping\_init

初始化时钟源及时间初始值，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)

{

system\_timer = machine\_desc->timer;

*system*\_timer->init();

}

# 3、硬件时钟（GM为例）

（1）平台注册

MACHINE\_START(GM, BOARD\_NAME)

.atag\_offset = BOOT\_PARAMETER\_PA\_OFFSET, //boot command line, after kernel 3.2 change as relative address

.map\_io = board\_map\_io,

.init\_irq = board\_init\_irq,

.timer = &board\_sys\_timer,

.fixup = board\_fixup\_memory,

.init\_machine = board\_init\_machine,

.restart = arch\_reset,

MACHINE\_END

上一节中的machine\_desc就是该宏定义的具体值。

struct sys\_timer board\_sys\_timer = {

.init = board\_sys\_timer\_init, //硬件定时器初始化函数

};

（2）定时器初始化

static struct fttmr010\_clockevent fttmr010\_0\_clockevent = {

.clockevent = {

.name = "fttmr010:0",

.irq = TIMER\_FTTMR010\_IRQ0,

},

.base = (void \_\_iomem \*)TIMER\_FTTMR010\_VA\_BASE,

.id = 0,

};

static struct fttmr010\_clocksource fttmr010\_1\_clocksource = {

.clocksource = {

.name = "fttmr010:1",

},

.base = (void \_\_iomem \*)TIMER\_FTTMR010\_VA\_BASE,

.id = 1,

};

static void \_\_init board\_sys\_timer\_init(void)

{

unsigned int pclk = pmu\_get\_apb0\_clk();

fttmr010\_0\_clockevent.freq = pclk;

fttmr010\_clockevent\_init(&fttmr010\_0\_clockevent);

fttmr010\_1\_clocksource.freq = pclk;

fttmr010\_clocksource\_init(&fttmr010\_1\_clocksource);

}

该函数主要完成clockevent和clocksource的定时器初始化，从上面可以看出timer0用于clockevent且要注册中断，timer1用于clocksource，使用APB0 clk为定时器时钟源。

（3）clockevent定时器初始化

fttmr010\_clockevent\_init该函数主要实现：timer0寄存器配置，中断注册，clockevent设备数据结构初始化，注册clockevent设备。

（4）clocksource定时器初始化

初始化函数源码如下：

void \_\_init fttmr010\_clocksource\_init(struct fttmr010\_clocksource \*fttmr010)

{

struct clocksource \*cs = &fttmr010->clocksource;

…

cs->rating = 300; //时钟源精度值

cs->read = fttmr010\_clocksource\_read; //获取计数值，系统主要调用该接口转化为系统时间

cs->mask = CLOCKSOURCE\_MASK(32); //计数值32位

cs->shift = 20; //除法移位，此处写死，实际可以根据freq自动计算出来

cs->flags = CLOCK\_SOURCE\_IS\_CONTINUOUS;

cs->mult = clocksource\_hz2mult(fttmr010->freq, cs->shift); //根据shift计算出对应的mult

/\* 初始化定时器1，该定时器单调递增，不产生中断 \*/

fttmr010\_disable(fttmr010->base, fttmr010->id);

fttmr010\_set\_match1(fttmr010->base, fttmr010->id, 0);

fttmr010\_set\_match2(fttmr010->base, fttmr010->id, 0);

fttmr010\_set\_reload(fttmr010->base, fttmr010->id, 0xffffffff);

fttmr010\_set\_counter(fttmr010->base, fttmr010->id, 0xffffffff);

fttmr010\_enable\_noirq(fttmr010->base, fttmr010->id);

clocksource\_register(cs); //注册clocksource设备

}

以下函数主要返回定时器的计数值

static cycle\_t fttmr010\_clocksource\_read(struct clocksource \*cs)

{

struct fttmr010\_clocksource \*fttmr010;

cycle\_t counter;

fttmr010 = container\_of(cs, struct fttmr010\_clocksource, clocksource);

counter = fttmr010\_get\_counter(fttmr010->base, fttmr010->id);

return ~counter;

}

# 4、系统时间

1、系统时间转化流程

结合上面的硬件定时器，其基本流程图如下：

2、整个调用流程



（1）timekeeping\_get\_ns具体实现

static inline s64 timekeeping\_get\_ns(void)

{

cycle\_t cycle\_now, cycle\_delta;

struct clocksource \*clock;

/\* read clocksource: \*/

clock = timekeeper.clock;

cycle\_now = clock->read(clock); //此处的read就为上面函数fttmr010\_clocksource\_read

/\* calculate the delta since the last update\_wall\_time: \*/

cycle\_delta = (cycle\_now - clock->cycle\_last) & clock->mask;

/\* return delta convert to nanoseconds using ntp adjusted mult. 通过这条注释也说明通过ntp可以调整mult \*/

return clocksource\_cyc2ns(cycle\_delta, timekeeper.mult, timekeeper.shift); //将定时器timer1的cycle转化为对应的ns数

}

static inline s64 clocksource\_cyc2ns(cycle\_t cycles, u32 mult, u32 shift)

{

return ((u64) cycles \* mult) >> shift;

}

这里的mult和shift就是上面根据timer1的freq计算出来的。

T=cycles/F:F为timer1的时钟频率，但内核一般不允许浮点运算而且影响精度，因此，内核使用了另外一个变通的办法，根据时钟的频率和期望的精度事先计算出两个辅助常数mult和shift，然后使用以下公式进行cycle和t的转换：

t = (cycle \* mult) >> shift;

只要我们保证以下计算无限接近F就可以保证精度

F = (1 << shift) / mult;

# 5、时间修正

公式：t = (cycle \* mult) >> shift，误差来源：timer定时器本身精确度cycle及mult和shift的精确度。下面以GM8286为例。

freq=41500000时，得到mult:404270265/sift:24。也就是说1S=1000000000ns需要41500000个时钟周期，利用上面公式理论计算出：

t = (cycle \* mult) >> shift =（ 41500000 \* 404270265 ） >> 24 = 999999999.85 ns。因此，每S偏差为0.15ns，一天才偏差12960ns。实际情况：GM8286每天都偏快4~5S，mult和shift是死的，只能是cycle本身不够准确导致系统偏差较大，当然也包括其他地方引入误差。

修正方法：上面说过通过shift值得到mult值，当cycle不准时通过修正mult值来达到目的：频率：41500000，每天平均快：4.67S 即每S偏差54051ns

mult =( t<<24) / cycle，t=1000000000 – 54051, cycle= cycle，最终得到如下：

mult = ( 999945949 << 24 ( = 16776342192259072) ) / 41500000 = 404248414

后续实现ntp功能后就可以根据偏差自动计算出该值，一般ntp一次就可以，一个设备的偏差基本固定不变，另外需要注意的就是t<<24可能超出32位范围。