调用ktime API需要include如下header file

> `#include/linux/ktime.h`
>
> `#include/linux/timekeeping.h`

```
/*
 * ktime_t:
 *
 * A single 64-bit variable is used to store the hrtimers
 * internal representation of time values in scalar nanoseconds. The
 * design plays out best on 64-bit CPUs, where most conversions are
 * NOPs and most arithmetic ktime_t operations are plain arithmetic
 * operations.
 *
 */
union ktime {
    s64 tv64;
};

typedef union ktime ktime_t;        /* Kill this */
```

ktime_t就是64-bit的有符号数！

```
ktime_t start, now;
start = ktime_get();
... // do something

now = ktime_get();
ktime_us_delta(now, start) < 5000) {
    ...
}
```

ktime_us_delta(const ktime_t later, const ktime_t earlier)
把前后时间之间的差转换成us(microsecond)

ktime_get()返回的其实就是纳秒。
有下面code为证

```
/* Convert ktime_t to nanoseconds - NOP in the scalar storage format: */
#define ktime_to_ns(kt)         ((kt).tv64)
```

```
static inline s64 ktime_to_us(const ktime_t kt)
{
    return ktime_divns(kt, NSEC_PER_USEC);
}

static inline s64 ktime_to_ms(const ktime_t kt)
{
    return ktime_divns(kt, NSEC_PER_MSEC);
}
```

kt / NSEC_PER_USEC ==> 纳秒 / 1000 (即每微秒1000纳秒)，转换成us(微秒)

kt / NSEC_PER_MSEC ==> 纳秒 / 1000000 (即每毫秒1000000纳秒)，转换成ms(毫秒)

```
static inline ktime_t ktime_add_us(const ktime_t kt, const u64 usec)
{
    return ktime_add_ns(kt, usec * NSEC_PER_USEC);
}

static inline ktime_t ktime_add_ms(const ktime_t kt, const u64 msec)
{
    return ktime_add_ns(kt, msec * NSEC_PER_MSEC);
}
```

把kt与微秒数 / 毫秒数相加，得到kt(纳秒数)

ktime_t的比较最好不要直接相比，而要调用如下API

```
static inline bool ktime_after(const ktime_t cmp1, const ktime_t cmp2)
{
    return ktime_compare(cmp1, cmp2) > 0;
}

static inline bool ktime_before(const ktime_t cmp1, const ktime_t cmp2)
{
    return ktime_compare(cmp1, cmp2) < 0;
}

static inline int ktime_equal(const ktime_t cmp1, const ktime_t cmp2)
{
    return cmp1.tv64 == cmp2.tv64;
}
```

ktime_t (纳秒)与struct timespec / struct timeval之间的互相转换

```c
struct timespec {
    __kernel_time_t tv_sec;          /* seconds */
    long        tv_nsec;        /* nanoseconds */
};

struct timeval {
    __kernel_time_t     tv_sec;     /* seconds */
    __kernel_suseconds_t    tv_usec;    /* microseconds */
};
```

__kernel_time_t ==> long

即把纳秒数转换成秒+纳秒 / 秒 + 毫秒

To ktime_t

```c
/* convert a timespec to ktime_t format: */
static inline ktime_t timespec_to_ktime(struct timespec ts)
{
    return ktime_set(ts.tv_sec, ts.tv_nsec);
}

/* convert a timespec64 to ktime_t format: */
static inline ktime_t timespec64_to_ktime(struct timespec64 ts)
{
    return ktime_set(ts.tv_sec, ts.tv_nsec);
}

/* convert a timeval to ktime_t format: */
static inline ktime_t timeval_to_ktime(struct timeval tv)
{
    return ktime_set(tv.tv_sec, tv.tv_usec * NSEC_PER_USEC);
}
```

From ktime_t

```c
/* Map the ktime_t to timespec conversion to ns_to_timespec function */
#define ktime_to_timespec(kt)       ns_to_timespec((kt).tv64)

/* Map the ktime_t to timespec conversion to ns_to_timespec function */
#define ktime_to_timespec64(kt)     ns_to_timespec64((kt).tv64)

/* Map the ktime_t to timeval conversion to ns_to_timeval function */
#define ktime_to_timeval(kt)        ns_to_timeval((kt).tv64)
```

---

in kernel/time/timekeeping.c

getboottime()返回自boot以后到现在的时间。

```c
/**
 * getboottime - Return the real time of system boot.
 * @ts:      pointer to the timespec to be set
 *
 * Returns the wall-time of boot in a timespec.
 *
 * This is based on the wall_to_monotonic offset and the total suspend
 * time. Calls to settimeofday will affect the value returned (which
 * basically means that however wrong your real time clock is at boot time,
 * you get the right time here).
 */
void getboottime(struct timespec *ts)
{
    struct timekeeper *tk = &tk_core.timekeeper;
    ktime_t t = ktime_sub(tk->offs_real, tk->offs_boot);

    *ts = ktime_to_timespec(t);
}
EXPORT_SYMBOL_GPL(getboottime);
```