NAME

eventtimers - kernel event timers subsystem

SYNOPSIS

#include <sys/timeet.h>

```
struct eventtimer;

typedef int et_start_t(struct eventtimer *et,
    sbintime_t first, sbintime_t period);
typedef int et_stop_t(struct eventtimer *et);
typedef void et_event_cb_t(struct eventtimer *et, void *arg);
typedef int et_deregister_cb_t(struct eventtimer *et, void *arg);

struct eventtimer {
        SLIST_ENTRY(eventtimer)  et_all;
        char                     *et_name;
        int                      et_flags;
#define ET_FLAGS_PERIODIC        1
#define ET_FLAGS_ONESHOT         2
#define ET_FLAGS_PERCPU                  4
#define ET_FLAGS_C3STOP                  8
#define ET_FLAGS_POW2DIV         16
        int                      et_quality;
        int                      et_active;
        uint64_t            et_frequency;
        sbintime_t               et_min_period;
        sbintime_t               et_max_period;
        et_start_t          *et_start;
        et_stop_t           *et_stop;
        et_event_cb_t            *et_event_cb;
        et_deregister_cb_t  *et_deregister_cb;
        void                     *et_arg;
        void                     *et_priv;
        struct sysctl_oid   *et_sysctl;
};
```

int

et_register(*struct eventtimer *et*);

int

et_deregister(*struct eventtimer *et*);

void
et_change_frequency(*struct eventtimer *et, uint64_t newfreq*);

ET_LOCK();

ET_UNLOCK();

*struct eventtimer **
et_find(*const char *name, int check, int want*);

int
et_init(*struct eventtimer *et, et_event_cb_t *event, et_deregister_cb_t *deregister, void *arg*);

int
et_start(*struct eventtimer *et, sbintime_t first, sbintime_t period*);

int
et_stop(*struct eventtimer *et*);

int
et_ban(*struct eventtimer *et*);

int
et_free(*struct eventtimer *et*);

DESCRIPTION

Event timers are responsible for generating interrupts at specified time or periodically, to run different time-based events. Subsystem consists of three main parts:

Drivers Manage hardware to generate requested time events.

Consumers *sys/kern/kern_clocksource.c* uses event timers to supply kernel with **hardclock**(), **statclock**() and **profclock**() time events.

Glue code *sys/sys/timeet.h*, *sys/kern/kern_et.c* provide APIs for event timer drivers and consumers.

DRIVER API

Driver API is built around eventtimer structure. To register its functionality driver allocates that

structure and calls **et_register**(). Driver should fill following fields there:

et_name Unique name of the event timer for management purposes.

et_flags Set of flags, describing timer capabilities:

ET_FLAGS_PERIODIC	Periodic mode supported.
ET_FLAGS_ONESHOT	One-shot mode supported.
ET_FLAGS_PERCPU	Timer is per-CPU.
ET_FLAGS_C3STOP	Timer may stop in CPU sleep state.
ET_FLAGS_POW2DIV	Timer supports only 2^n divisors.

et_quality Abstract value to certify whether this timecounter is better than the others. Higher value means better.

et_frequency Timer oscillator's base frequency, if applicable and known. Used by consumers to predict set of possible frequencies that could be obtained by dividing it. Should be zero if not applicable or unknown.

et_min_period, *et_max_period*
Minimal and maximal reliably programmable time periods.

et_start Driver's timer start function pointer.

et_stop Driver's timer stop function pointer.

et_priv Driver's private data storage.

After the event timer functionality is registered, it is controlled via *et_start* and *et_stop* methods. *et_start* method is called to start the specified event timer. The last two arguments are used to specify time when events should be generated. *first* argument specifies time period before the first event generated. In periodic mode NULL value specifies that first period is equal to the *period* argument value. *period* argument specifies the time period between following events for the periodic mode. The NULL value there specifies the one-shot mode. At least one of these two arguments should be not NULL. When event time arrive, driver should call *et_event_cb* callback function, passing *et_arg* as the second argument. *et_stop* method is called to stop the specified event timer. For the per-CPU event timers *et_start* and *et_stop* methods control timers associated with the current CPU.

Driver may deregister its functionality by calling **et_deregister**().

If the frequency of the clock hardware can change while it is running (for example, during power-saving modes), the driver must call **et_change_frequency**() on each change. If the given event timer is the active timer, **et_change_frequency**() stops the timer on all CPUs, updates *et->frequency*, then restarts the timer on all CPUs so that all current events are rescheduled using the new frequency. If the given timer is not currently active, **et_change_frequency**() simply updates *et->frequency*.

CONSUMER API

et_find() allows consumer to find available event timer, optionally matching specific name and/or capability flags. Consumer may read returned eventtimer structure, but should not modify it. When wanted event timer is found, **et_init**() should be called for it, submitting *event* and optionally *deregister* callbacks functions, and the opaque argument *arg*. That argument will be passed as argument to the callbacks. Event callback function will be called on scheduled time events. It is called from the hardware interrupt context, so no sleep is permitted there. Deregister callback function may be called to report consumer that the event timer functionality is no longer available. On this call, consumer should stop using event timer before the return.

After the timer is found and initialized, it can be controlled via **et_start**() and **et_stop**(). The arguments are the same as described in driver API. Per-CPU event timers can be controlled only from specific CPUs.

et_ban() allows consumer to mark event timer as broken via clearing both one-shot and periodic capability flags, if it was somehow detected. **et_free**() is the opposite to **et_init**(). It releases the event timer for other consumers use.

ET_LOCK() and **ET_UNLOCK**() macros should be used to manage mutex(9) lock around **et_find**(), **et_init**() and **et_free**() calls to serialize access to the list of the registered event timers and the pointers returned by **et_find**(). **et_start**() and **et_stop**() calls should be serialized in consumer's internal way to avoid concurrent timer hardware access.

SEE ALSO

eventtimers(4)

AUTHORS

Alexander Motin <*mav@FreeBSD.org*>