(Hardware) und kopiert sie in den von der Applikation übergebenen Speicherbereich (in den Userspace). Danach übergibt sie die Kontrolle wieder an die Funktion `I/O read` (9). Der Systemcall ist nun abgearbeitet. Sollte die Applikation höchste Priorität haben, wird der Scheduler die CPU wieder an diesen Rechenprozess übergeben. Die über den Systemcall `read` gelesenen Daten werden noch in der Bibliotheksfunktion `fread` verarbeitet und können schließlich von der Applikation selbst ausgewertet werden.

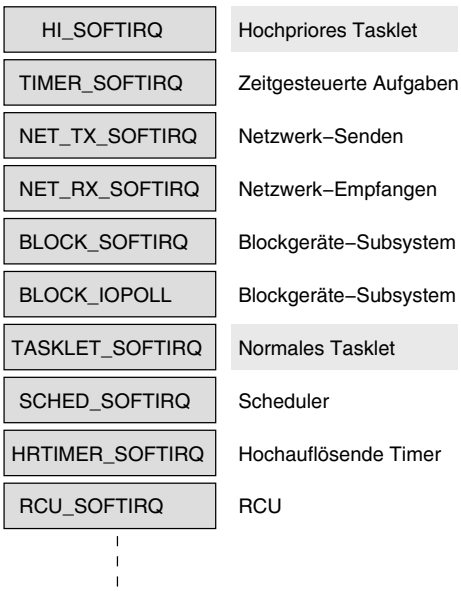
6.3 Softirqs

Softirqs ermöglichen die hochpriore Verarbeitung von Funktionen.

Direkt nach Abarbeitung einer Hardware-ISR beziehungsweise nach Freigabe der Interrupts überprüft der Kernel, ob weitere wichtige Funktionen/Routinen abzuarbeiten sind. Diese Funktionen werden Softirqs genannt.

In den Kernel-Quellen ist hierzu ein Feld von 32 Softirq-Routinen (siehe Datei `<kernel/softirq.c>` im Kernel-Quellcode) angelegt, wovon zehn bereits vordefiniert sind. Ein zugehöriges Bit im Variablenfeld `irq_stat` gibt an, ob eine Routine aufgerufen werden muss (Bit=1) oder nicht (Bit=0).

Abb. 6-3
Vordefinierte Softirqs



Das Feld der Softirq-Routinen wird über die Funktion `open_softirq` belegt. Die vordefinierten Softirqs sind in Abbildung 6-3 dargestellt.

Für Softirqs gilt generell:

- Die Abarbeitung erfolgt im Interruptkontext. Interrupts selbst sind dabei aber zugelassen und unterbrechen – falls sie auftreten – die Abarbeitung des Softirqs.
- Die vordefinierten Softirqs werden entsprechend der oben dargestellten Reihenfolge priorisiert. Ein Timer-Softirq wird daher vor den Net-Softirqs abgearbeitet.
- Ein Softirq kann bei Mehrprozessorsystemen mehrfach parallel ablaufen.

Für den Treiberentwickler sind insbesondere die Ausprägungen Tasklet sowie Timer interessant. Sie werden im Folgenden genauer vorgestellt.

6.3.1 Tasklets

Tasklets stellen eine spezifische Ausprägung der Softirqs dar.

Der Kernel-Programmierer setzt Tasklets ein, um längere Berechnungen, die im Kontext eines Interrupts notwendig werden, abarbeiten zu lassen. Fänden diese Berechnungen innerhalb der Hardware-ISR statt, hätte das längere Interrupt-Latenzzeiten zur Folge, was in jedem Fall zu vermeiden ist. Daher teilt man die Verarbeitung einer ISR in zwei Schritte auf: Während des ersten Schrittes sind Interrupts gesperrt, nur die (zeit-)kritischen Aktionen werden durchgeführt. Die übrigen Berechnungen werden in einem zweiten Schritt – bei freigegebenen Interrupts – behandelt. Dieser früher Bottom-Half genannte Teil wird typischerweise in Form eines Tasklets realisiert.

Tasklets sind vom Entwickler kodierte Funktionen, die vom Kernel zusammen mit einem Parameter aufgerufen werden. Die Adresse der Tasklet-Funktion und das ihr als Parameter zu übergebende Datum werden in einer Instanz der Datenstruktur `struct tasklet_struct` (Prototyp in `<linux/interrupt.h>`) beschrieben. Die Initialisierung der Datenstruktur `struct tasklet_struct` kann statisch (also durch den Compiler) durch das Makro `DECLARE_TASKLET` oder dynamisch (also innerhalb einer Funktion zur Laufzeit) mit Hilfe der Funktion `tasklet_init` erfolgen (siehe Beispiele 6-2 und 6-3).

***Beispiel 6-2** Statische Initialisierung einer Tasklet-Struktur*

```
static void tasklet_function( unsigned long data );                    ❶
DECLARE_TASKLET( tl_descr, tasklet_function, 0L ); // Statische        ❷
Definition
```

❶ Deklaration der Funktion, die abgearbeitet werden soll, wenn das Tasklet aufgerufen wird

❷ Mit `DECLARE_TASKLET` wird ein Element vom Typ `struct tasklet_struct` mit Namen `tl_descr` angelegt. Dieses Element wird mit der Adresse der aufzurufenden Funktion (`tasklet_function`) und einem zu übergebenden Datum (hier »0L«) initialisiert.

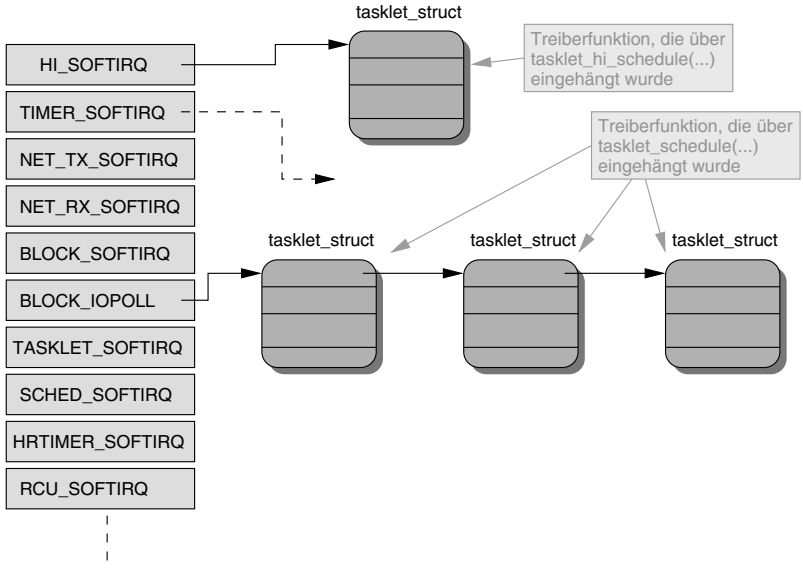
Beispiel 6-3 *Dynamische Initialisierung einer Tasklet-Struktur*

```
struct tasklet_struct tl_descr;
...
static int __init mod_init(void)
{
    tasklet_init( &tl_descr, tasklet_function, 0L );
    ...
}
```

Zur Abarbeitung freigeben

Das so definierte Tasklet muss nur noch zum richtigen Zeitpunkt zur Abarbeitung freigegeben werden. Typischerweise findet dieser Vorgang innerhalb einer Hardware-ISR durch Aufruf der Funktion `tasklet_schedule` oder der Funktion `tasklet_hi_schedule` statt. Welche der beiden Funktionen verwendet wird, hängt von der Wichtigkeit der innerhalb des Tasklets zu erledigenden Aufgaben ab. Für die Bearbeitung stehen nämlich zwei Prioritätsstufen zur Verfügung. Ein mit `tasklet_hi_schedule` eingehängtes Tasklet wird auf der Prioritätsstufe `HI_SOFTIRQ` abgearbeitet. Das bedeutet, dass die Tasklet-Funktion wirklich direkt nach dem Ende einer Hardware-ISR die CPU zugeteilt bekommt (siehe Abbildung 6-4). Ein mit der Funktion `tasklet_schedule` eingehängtes Tasklet dagegen wird erst dann abgearbeitet, wenn kein anderer Softirq mehr anliegt; dieses Tasklet hat also innerhalb der Gruppe der Softirqs die niedrigste Priorität (`TASKLET_SOFTIRQ`).

Abb. 6-4 *Tasklets als Teil der Softirq-Verarbeitung*



Um ein übersichtliches und abgeschlossenes Beispiel zur Hand zu haben, wird das Tasklet im Beispiel 6-4 nicht während einer ISR, sondern bereits während der Modulinitialisierung »geschedult«. Wird der Code kompi-

liert und das so generierte Modul geladen, erscheint in den Syslogs die Meldung »Tasklet called...«.

Beispiel 6-4 *Einfaches Tasklet*

```
#include <linux/module.h>
#include <linux/interrupt.h>

static void tasklet_func( unsigned long data )
{
    printk(KERN_INFO "Tasklet called...\n");
}

DECLARE_TASKLET( tl_descr, tasklet_func, 0L );

static int __init mod_init(void)
{
    printk(KERN_INFO "mod_init called\n");
    tasklet_schedule( &tl_descr ); /* Tasklet sobald als möglich ausführen */
    return 0;
}

static void __exit mod_exit(void)
{
    printk(KERN_INFO "mod_exit called\n");
    tasklet_kill( &tl_descr );
}

module_init( mod_init );
module_exit( mod_exit );

MODULE_LICENSE("GPL");
```

Einhängen ohne direkten Aufruf

Zum Schedulen eines Tasklets werden innerhalb der Funktionen `tasklet_schedule` beziehungsweise `tasklet_hi_schedule` Interrupts gesperrt. Zwar ist dies in den meisten Fällen unproblematisch, doch gibt es Situationen, in denen Interrupts nicht gesperrt werden sollen oder können. Für solche Fälle bietet Kernel 2.6 die Möglichkeit, das Tasklet vorzeitig einzuhängen, die Abarbeitung aber durch Setzen eines Flags zu verhindern. Zum Setzen dieses Flags dient die Funktion `tasklet_disable`. Die Ausführung der Tasklet-Funktion kann zum gewünschten Zeitpunkt – ohne Interrupts sperren zu müssen – eingefordert werden. Der Aufruf von `tasklet_enable` reicht hierzu aus. Wird anstelle des Makros `DECLARE_TASKLET` das Makro `DECLARE_TASKLET_DISABLED` verwendet, wird das Disable-Flag bereits direkt bei einer statischen Initialisierung gesetzt.

Mit einem Tasklet lässt sich zwar auf der einen Seite die Interrupt-Latenzzeit des Kernels verbessern, es beeinflusst aber auf der anderen Seite auch die Task-Latenzzeit. Aus diesem Grund sollte ein Tasklet so kurz wie möglich gehalten werden. Hinzu kommt, dass Tasklets im Interruptkon-

text abgearbeitet werden, also sich nicht schlafen legen dürfen! Spätestens wenn diese Funktionalität benötigt wird, muss der Treiberentwickler auf die Verwendung von Kernel-Threads ausweichen.

Ein Tasklet wird zu einem Zeitpunkt maximal einmal aufgerufen – so die Spezifikation. Das gilt auch für Mehrprozessorsysteme. Unterschiedliche Tasklets zur gleichen Zeit einzusetzen, ist allerdings möglich.

Achtung, Race Condition!

Abschließend noch eine Warnung: Beim Gebrauch von Tasklets kann der Entwickler leicht in eine Race Condition verstrickt werden. Wird nämlich das Modul entladen und ruft der Kernel im Anschluss ein von diesem Modul geschedultes Tasklet auf, soll der Prozessor Code abarbeiten, der längst nicht mehr vorhanden ist. Das aber ist unmöglich. Es kommt zu einer Oops-Meldung. Zu den Aufgaben des Treiberentwicklers gehört von daher, sicherzustellen, dass jegliches Tasklet vor Entladen des Moduls entweder abgearbeitet oder entfernt wurde. Die entsprechende Funktion heißt `tasklet_kill`. Sie kann auch dann gefahrlos aufgerufen werden, wenn zum Zeitpunkt des Aufrufes das Tasklet nicht geschedult ist.

6.3.2 Timer-Funktionen

Neben Tasklets ist für Treiberentwickler noch eine weitere Variante der Softirqs überaus interessant, die sogenannten Timer. Mit ihrer Hilfe kann der Entwickler den Kernel beauftragen, Funktionen zu einem definierten, späteren Zeitpunkt auszuführen.

Abarbeitung zu definierten Zeitpunkten

Dazu muss man zunächst wissen, dass Zeiten innerhalb des Kernels historisch nicht absolut, sondern relativ zum Einschaltzeitpunkt verarbeitet werden. Als Basis gilt die Anzahl der seit dem Einschalten ausgelösten, periodischen Timerinterrupts. Sie wird in einem Zähler mit dem Namen `jiffies` gezählt (siehe Kapitel 6.6.1).

Um einen Kernel-Timer zu verwenden, wird im Treiber eine Datenstruktur vom Typ `struct timer_list` alloziert. Die für den Kernel spezifischen Felder dieser Datenstruktur werden über die Funktion `init_timer` initialisiert (hier ist nur eine dynamische Initialisierung, also innerhalb einer Funktion, möglich). Anschließend sind die übrigen Felder mit der Adresse der aufzurufenden Funktion (Feld `function`), mögliche Daten für die Funktion (Feld `data`) und dem Zeitpunkt, zu dem die Funktion aufgerufen werden soll (Feld `expires`), zu spezifizieren (Beispiel 6-5). Der Abarbeitungszeitpunkt wird absolut in `jiffies` angegeben. Um eine Funktion relativ zum momentanen Zeitpunkt aufzurufen, wird zum aktuellen Zeitpunkt der Relativwert aufaddiert. Liegt der Zeitpunkt, zu dem die Timer-Funktion aufgerufen werden soll, bereits in der Vergangenheit, wird die Routine sofort ausgeführt.

Beispiel 6-5 *Initialisierung der timer_list*

```
static struct timer_list ptimer;
static void timer_funktion(unsigned long);
...
static int __init mod_init(void)
{
    init_timer( &ptimer );
    ptimer.function = timer_funktion;
    ptimer.data = 0;
    ptimer.expires = jiffies + (2*HZ); // alle 2 Sekunden
    ...
```

Periodische Timer

Sobald der im Expires-Feld angegebene Zeitpunkt erreicht beziehungsweise überschritten ist, wird die dort spezifizierte Funktion genau einmal aufgerufen. Soll eine Funktion periodisch abgearbeitet werden, muss sie die zugehörige `timer_list` mit dem nächsten Aufrufzeitpunkt initialisieren und dann dem Kernel erneut übergeben (so zu sehen in Beispiel 6-7; hier wird die Timer-Funktion periodisch alle zwei Sekunden aufgerufen).

Die Funktion `add_timer` schließlich übergibt den Treiber zur Ausführung an den Kernel. Damit ist der Timer »aktiviert«. Der Kernel sorgt dafür, dass die in der Struktur angegebene Funktion mit den Daten als Parameter zum spezifizierten Zeitpunkt aufgerufen wird.

Es ist nicht erlaubt, einen bereits aktivierten Timer ein zweites Mal zu aktivieren!

Wie andere Softirqs auch, wird die Timer-Funktion im Interruptkontext aufgerufen. Das bedeutet: Die Timer-Funktion ist möglichst kurz zu halten. Sie kann sich nicht schlafen legen! Überdies ist ein Zugriff auf User-Prozesse (Applikationen) innerhalb der Timer-Funktion nicht möglich.

Deaktivieren.

Ist der Timer Teil eines Moduls und soll das Modul wieder entladen werden, ist jeder noch nicht abgelaufene Timer aus dem System zu entfernen (Deaktivieren des Timers). Dazu existiert die Funktion `del_timer_sync`. Auf SMP-Maschinen deaktiviert diese Funktion nicht nur einen Timer, sondern wartet darüber hinaus noch so lange, bis die möglicherweise auf einer anderen CPU aktive Timer-Funktion beendet ist. Auf einer Einprozessormaschine wird `del_timer_sync` auf die Funktion `del_timer` abgebildet, die den Timer nur deaktiviert.

```
static void __exit mod_exit(void)
{
    del_timer_sync( &ptimer );
    ...
```

Bevor ein Timer deaktiviert wird, ist zu verhindern, dass der Timer erneut eingehängt (`add_timer`) wird. Anders als bei den Tasklets muss dazu der Treiber seinen eigenen Mechanismus implementieren (beispielsweise über ein Flag, wie in Beispiel 6-6).

Beispiel 6-6
Stoppen periodischer Funktionen

```
static atomic_t nicht_mehr_einhaengen = 0;                          ❶

void timer_funktion( unsigned long parameter )
{
    if( atomic_read( &nicht_mehr_einhaengen ) ) {                   ❷
        complete( &on_exit );
    } else {
        add_timer( &ptimer );
    }
    ...
}

static void __exit mod_exit(void)                                   ❸
{
    atomic_set( &nicht_mehr_einhaengen, 1 );
    wait_for_completion( &on_exit );
    del_timer_sync( &ptimer ); // nicht mehr notwendig              ❹
    ...
```

❶ Mit diesem Flag wird synchronisiert, ob der Timer noch eingehängt werden darf oder nicht. Der Datentyp »atomic_t« wird in Tabelle 6-1 vorgestellt.
❷ Wenn das Flag nicht gesetzt ist, darf der Timer weiterhin verwendet werden.
❸ Das Modul wird entladen, der Timer ist aus dem System zu entfernen. Durch das Setzen des Flags wird sichergestellt, dass nicht zwischenzeitlich der Timer erneut aktiviert wird.
❹ Mit dieser Funktion wird ein Timer aus dem Kernel entfernt. Es sollte grundsätzlich die »sync«-Variante (`del_timer_sync`) verwendet werden.

Um festzustellen, ob ein Timer aktiviert ist, gibt es die Funktion `timer_pending`. Diese gibt `1` zurück, falls der Timer aktiviert ist, andernfalls `0`.

```
if( timer_pending( &ptimer ) ) {
    printk( "Timer ist aktiviert.\n" );
} else {
    printk( "Timer ist nicht aktiviert.\n" );
}
```

Weitere Timer-Funktionen

Noch zwei weitere Funktionen erleichtern den Umgang mit Timern. Die Funktion `mod_timer` ermöglicht es, bei einem aktivierten Timer den Zeitpunkt zu modifizieren, zu dem die Timer-Funktion aufgerufen werden soll.

Mit der Funktion `add_timer_on` ist der Entwickler in der Lage, für ein SMP-System festzulegen, auf welchem Prozessor eine Timer-Funktion ablaufen soll.

Beispiel 6-7 *Verwendung eines Timers*

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/timer.h>
#include <linux/sched.h>
#include <linux/init.h>

static struct timer_list mytimer;

static void inc_count(unsigned long arg)
{
    printk("inc_count called (%ld)...\n", mytimer.expires );
    mytimer.expires = jiffies + (2*HZ); // 2 second
    add_timer( &mytimer );
}

static int __init ktimer_init(void)
{
    init_timer( &mytimer );
    mytimer.function = inc_count;
    mytimer.data = 0;
    mytimer.expires = jiffies + (2*HZ); // 2 second
    add_timer( &mytimer );
    return 0;
}

static void __exit ktimer_exit(void)
{
    if( timer_pending( &mytimer ) )
        printk("Timer ist aktiviert ...\n");
    if( del_timer_sync( &mytimer ) )
        printk("Aktiver Timer deaktiviert\n");
    else
        printk("Kein Timer aktiv\n");
}

module_init( ktimer_init );
module_exit( ktimer_exit );

MODULE_LICENSE("GPL");
```

Ein Timer darf nur einmal aktiviert sein.

Etwas komplizierter wird der Umgang mit Timern, wenn ein und derselbe Treiber an mehreren Stellen die Funktion `add_timer` mit dem gleichen Timer-Objekt aufruft. Da es zu Abstürzen kommen kann, falls der Kernel damit beauftragt wird, ein einzelnes Timer-Objekt zu einem Zeitpunkt gleich mehrfach abzuarbeiten, muss der Entwickler vor dem Aktivieren des Timers überprüfen, ob der Timer nicht bereits aktiviert wurde. Der Vorgang des Überprüfens und das Einhängen stellen einen kritischen Abschnitt dar, der zu schützen ist. Dazu bietet sich ein Spinlock an (siehe Beispiel 6-8).

Beispiel 6-8 *Sicheres Einhängen eines Timer-Objekts*

```
static struct spinlock_t timer_lock;
...

void secure_add_timer( struct timer_list *ptimer )
{
    unsigned long flags;

    spin_lock_irqsave( &timer_lock, flags );
    if( !timer_pending( ptimer ) ) {
        add_timer( ptimer );
    }
    spin_unlock_irqrestore( &timer_lock, flags );
}
```

6.3.3 High Resolution Timer

Grundlagen

Ab Kernel 2.6.16 haben die Entwickler sukzessive die auf periodische Interrupts beruhende Zeitverwaltung gegen ein auf dynamischen Ticks basierendes Zeitmanagement (dyntick) ausgetauscht. Der Kernel berechnet intern den nächsten Zeitpunkt, zu dem er aktiv werden muss, und programmiert die Timerbausteine so, dass genau zu diesem Zeitpunkt ein Interrupt auftritt, der die notwendige Aktion anstößt. Der neue Code bildet nicht nur die Basis für wirklich hoch zeitauflösende Timer, sondern auch für ein beeindruckend hochgenaues Zeitverhalten (high precision timer). Der Programmierer hat Zugriff auf dieses Zeitmanagement über die sogenannten High Resolution Timer (hrtimer).

Zeitquellen

Das Zeitmanagement stellt zwei Zeitquellen zur Verfügung. Die Zeitquelle `CLOCK_MONOTONIC` repräsentiert die Zeit, so wie sie im Rechner mitgeführt wird. Sprünge kommen dabei nicht vor, auch wenn die Uhr mal vor oder mal zurückgesetzt werden sollte. Sie zählt immer weiter. `CLOCK_REALTIME` repräsentiert die Zeit außerhalb des Rechners. Dreht der Admin auf seinem Rechner an der Zeitschraube, hat das direkten Einfluss auf diese Quelle.

Außerdem führt die auf hrtimer beruhende Zeitverwaltung den neuen Zeitdatentyp `ktime_t` ein, der auf Nanosekunden beruht. Dieser Datentyp und die Operatoren darauf werden in Abschnitt Zeitverwaltung auf Basis von dynamischen Timerticks (Seite 224) genauer vorgestellt.

Funktionen

Um hrtimer programmtechnisch zu nutzen, definieren Sie ein Objekt vom Typ `struct hrtimer`. Zur Initialisierung des Objekts mit Hilfe der Funktion `hrtimer_init` definieren Sie die Zeitquelle (`CLOCK_MONOTONIC` oder `CLOCK_REALTIME`) und ob der Zeitpunkt, zu dem der Timer aktiv werden soll, relativ oder absolut angegeben wird (`HRTIMER_MODE_REL` oder `HRTIMER_MODE_ABS`). Schließlich weisen Sie noch die Callback-Funktion zu, die im Fall des Auslösens aufgerufen wird. Zum eigentlichen Starten definieren Sie den Auslösezeitpunkt mit Hilfe der Funktionen, die Sie in Tabelle 6-2 fin-

den, und übergeben den Zeitpunkt durch Aufruf von `hrtimer_start_range_ns` oder `hrtimer_start`. Allerdings sollten Sie – falls möglich – die Funktion `hrtimer_start_range_ns` einsetzen. Diese gibt dem Kernel die Chance, Aufgaben zu bündeln. Dadurch wird die Chance erhöht, häufiger nicht genutzte Systemteile abzuschalten und Energie zu sparen (Leistungsbündelung).

Callback

Die Callback-Funktion selbst bekommt, wenn sie aufgerufen wird, die Adresse des zugehörigen Timerobjekts übergeben. Da sie im Interruptkontext abläuft, gelten die üblichen Einschränkungen beispielsweise beim Reservieren von Speicher. Rückgabewert der Funktion ist entweder `HRTIMER_RESTART` oder `HRTIMER_NORESTART`. Soll der Callback gleich wieder gestartet werden (`HRTIMER_RESTART`), dürfen Sie nicht vergessen, durch Aufruf beispielsweise der Funktion `hrtimer_add_expires_ns` einen neuen, in der Zukunft liegenden Zeitpunkt zu setzen.

Beispiel 6-9
High Resolution Timer

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/hrtimer.h>
#include <linux/ktime.h>

struct hrtimer hrt_monotonic, hrt_realtime;

static enum hrtimer_restart timer_function( struct hrtimer *hrt )
{
    printk("%ld: timer_function( %p )\n", jiffies, hrt );
    return HRTIMER_RESTART;
}

static int __init mod_init(void)
{
    struct timespec tp;
    ktime_t tim, absolut;

    printk("mod_init called\n");
    hrtimer_get_res( CLOCK_MONOTONIC, &tp );
    printk("monotonic(%p): %ld, %ld\n",
        &hrt_monotonic, tp.tv_sec, tp.tv_nsec );
    hrtimer_get_res( CLOCK_REALTIME, &tp );
    printk("realtime(%p): %ld, %ld\n",
        &hrt_realtime, tp.tv_sec, tp.tv_nsec );

    hrtimer_init( &hrt_monotonic, CLOCK_MONOTONIC, HRTIMER_MODE_REL );
    hrtimer_init( &hrt_realtime, CLOCK_REALTIME, HRTIMER_MODE_ABS );
    hrt_monotonic.function = timer_function;
    hrt_realtime.function = timer_function;
    tim = ktime_set( 15, 0 );
    hrtimer_start( &hrt_monotonic, tim, HRTIMER_MODE_REL );
    ktime_get_real_ts( &tp );
    absolut = timespec_to_ktime(tp);
```

```
    tim = ktime_add( tim, absolut );
    hrtimer_start( &hrt_realtime, tim, HRTIMER_MODE_ABS );
    printk("Timer activated at %ld jiffies\n", jiffies );
    return 0;
}

static void __exit mod_exit(void)
{
    hrtimer_cancel( &hrt_monotonic );
    hrtimer_cancel( &hrt_realtime );
    printk("mod_exit called\n");
}

module_init( mod_init );
module_exit( mod_exit );
MODULE_LICENSE("GPL");
```

Beispiel 6-9 zeigt Ihnen sowohl die Verwendung der beiden Zeitquellen, als auch den Umgang mit dem neuen Zeitdatentyp `ktime_t`. Die beiden Timer werden so aktiviert, dass sie nach 15 Sekunden die Callback-Funktion aufrufen, die eine Ausgabe per `printk` macht. Anhand der Ausgaben können Sie den Unterschied zwischen den beiden Zeitquellen sehen, falls Sie während der Wartezeit an der Zeitschraube drehen. Dieses Experiment sollten Sie natürlich keinesfalls auf einem Produktivsystem durchführen! Übersetzen Sie den Quellcode und laden Sie den Treiber. Beobachten Sie dabei die Ausgabe (Syslog), die erwartungsgemäß nach exakt 15 Sekunden zweimal erfolgt. Entfernen Sie den Treiber wieder und laden Sie ihn erneut. Danach stellen Sie die Systemzeit um fünf Sekunden vor. Der Timer `hrt_realtime` bekommt diese Modifikation mit und verkürzt die Wartezeit um fünf Sekunden, bevor er die Funktion `print_jiffies` aufruft. Der Timer `hrt_monotonic` ignoriert den Zeithüpfer, wartet genau 15 Sekunden und ruft dann die Funktion auf. Absolut betrachtet kommt dieser Aufruf fünf Sekunden zu spät.

6.3.4 Tasklet auf Basis des High Resolution Timers

Neben den klassischen Timern bieten Kernel ab Version 2.6.16 die Möglichkeit, Tasklets auf Basis der hrtimer zu realisieren. Wie in Abschnitt 6.6 ausgeführt, stellen hrtimer hochauflösende und hochpräzise Timer zur Verfügung, die auf einem neuen Zeitverwaltungssystem beruhen.

Initialisierung

Diese Timer-Tasklets sind leicht einzusetzen. Sie benötigen ein Tasklet-Objekt, welches mit Hilfe der Funktion `tasklet_hrtimer_init` initialisiert wird. Dabei geben Sie die Zeitquelle (`CLOCK_REALTIME` oder `CLOCK_MONOTONIC`) an. Weiterhin benötigen Sie eine Callback-Funktion, die aufgerufen wird, wenn der Timer abläuft. Diese Funktion gibt am Ende

HRTIMER_NORESTART oder HRTIMER_RESTART zurück, je nachdem, ob das Tasklet nur einmal aufgerufen werden soll oder – ohne erneute äußere Aktivierung – mehrfach. Falls die Callback-Funktion HRTIMER_RESTART zurückgibt, dürfen Sie jedoch nicht vergessen, vorher den Auslösezeitpunkt beispielsweise über die Funktion hrtimer_add_expires_ns neu zu setzen.

Aktivierung

Den Timer selbst aktivieren Sie, indem Sie die Funktion tasklet_hrtimer_start aufrufen. Die Zeit, zu der die Callback-Funktion aufgerufen wird, geben Sie dabei in der Form von ktime_t an. Sie haben die Möglichkeit, diese Zeit absolut (HRTIMER_MODE_ABS) oder relativ (HRTIMER_MODE_REL) anzugeben.

Aufräumen

Beim Entladen des Moduls stellen Sie durch Aufruf von tasklet_hrtimer_cancel noch sicher, dass keine durch das Modul aktivierte Callback-Funktion im System verbleibt.

Beispiel 6-10 zeigt Ihnen die Verwendung der Funktionen.

Beispiel 6-10
High Resolution Timer per Tasklet aufgerufen

```
#include <linux/module.h>
#include <linux/interrupt.h>
#include <linux/hrtimer.h>

static struct tasklet_hrtimer fireup;

static enum hrtimer_restart hrtimer_func( struct hrtimer *hrt )
{
    printk(KERN_INFO "hrtimer_func called at %ld...\n", jiffies);
    return HRTIMER_NORESTART;
}

static int __init mod_init(void)
{
    static ktime_t start_in;

    printk(KERN_INFO "mod_init called at %ld\n", jiffies);
    tasklet_hrtimer_init( &fireup, hrtimer_func, CLOCK_REALTIME,
        HRTIMER_MODE_REL );
    start_in = ktime_set( 10, 0 );
    tasklet_hrtimer_start( &fireup, start_in, HRTIMER_MODE_REL );
    return 0;
}

static void __exit mod_exit(void)
{
    printk(KERN_INFO "mod_exit called\n");
    tasklet_hrtimer_cancel( &fireup );
}

module_init( mod_init );
module_exit( mod_exit );
MODULE_LICENSE("GPL");
```