## High Resolution Timers (HRT/hrtimer)

**Core Concept:**

High-resolution timers in the Linux kernel offer timer functionality with nanosecond-level precision, a capability that distinguishes them from traditional kernel timers, which are based on jiffies (system ticks).

**Rationale for Dual Timer Mechanisms:**

* Initially, the Linux kernel incorporated a single timer mechanism predicated on the Cascading Timer Wheel (CTW).
* Subsequent attempts to integrate this mechanism with high-resolution timing requirements proved unsuccessful, primarily because the CTW's fundamental structure is not optimally suited for maintaining time-ordered events with efficiency.

**Key Differences:**

| **Feature** | **Kernel Timer (Traditional)** | **High Resolution Timer (HRT)** |
| --- | --- | --- |
| **Resolution** | Jiffies (system ticks) | 64-bit nanoseconds |
| **Data Structure** | Cascading Timer Wheel (CTW) | Red-Black Tree |
| **Suitability** | General kernel tasks | Performance-critical, precise timing |

**Terminology:**

* **Jiffies:** A global counter within the Linux kernel that represents the number of system ticks that have transpired since system initialization. The duration of a jiffy is system-dependent.
* **Cascading Timer Wheel (CTW):** A data structure employed in the traditional kernel timer implementation. It comprises an array of lists, wherein timers are positioned based on their respective expiry times. This structure employs a cascading approach to manage timers with varying timeouts.
* **Red-Black Tree:** A self-balancing binary search tree that facilitates efficient insertion, deletion, and searching of elements in logarithmic time. HRT utilizes this structure to maintain timers in a time-ordered manner.
* **Nanosecond:** A unit of time equal to one billionth of a second (10−9 seconds).
* **Time-ordered Data Structure:** A data structure in which elements are organized according to their time of occurrence or expiry. Within the context of HRT, timers are ordered by their expiration time.

**Availability:**

* Introduced in Linux kernel version 2.6.21 and subsequent releases.
* Requires that the kernel be compiled with the **CONFIG\_HIGH\_RES\_TIMERS=y** configuration parameter enabled.

**Verification of HRT Availability:**

1. **Kernel Configuration File:** Examine /boot/config-\* for the line CONFIG\_HIGH\_RES\_TIMERS=y.
2. **/proc/timer\_list:** Inspect for .resolution: 1 ns and event\_handler: hrtimer\_interrupt.
3. **clock\_getres() System Call:** This system call can be invoked to ascertain the resolution of a specific clock.

**Users of High Resolution Timers:**

* **User-space applications:** Employing nanosleep, POSIX timers (timer\_create, etc.), and Interval Timers (itimer).
* **Kernel:** Drivers and subsystems that necessitate precisely timed events, including, for example, multimedia synchronization and high-speed data acquisition.

**High Resolution Timer API (<linux/hrtimer.h>):**

**1. struct hrtimer:** Represents an HRT timer.

struct hrtimer {  
 struct rb\_node node; // Red-black tree node for time-ordered insertion  
 ktime\_t expires; // Absolute expiry time  
 int (\* function) (struct hrtimer \*); // Timer expiry callback function  
 struct hrtimer\_base \* base; // Pointer to the timer base (per CPU and per clock)  
};

**2. ktime\_t:** A data type designated for storing time values in nanoseconds.

* **64-bit systems:** Typically a 64-bit integer representing nanoseconds.
* **32-bit systems:** A structure comprising separate fields for seconds and nanoseconds.

ktime\_t ktime\_set(long secs, long nanosecs);

* **Purpose:** To generate a ktime\_t value from seconds and nanoseconds.

**3. hrtimer\_init():** Initializes an hrtimer structure.

void hrtimer\_init( struct hrtimer \*timer, clockid\_t clock\_id, enum hrtimer\_mode mode );

* timer: A pointer to the hrtimer structure to be initialized.
* clock\_id: Specifies the clock source to be utilized (defined in <linux/time.h>).
  + CLOCK\_MONOTONIC: Time since system boot, guaranteed to move forward.
  + CLOCK\_REALTIME: System's wall-clock time.
* mode: Timer mode:
  + HRTIMER\_MODE\_ABS: Absolute expiry time (relative to the chosen clock's epoch).
  + HRTIMER\_MODE\_REL: Relative expiry time (offset from the current time).

**4. hrtimer\_start():** Initiates (or restarts) an initialized HRT on the current CPU.

int hrtimer\_start(struct hrtimer \*timer, ktime\_t time, const enum hrtimer\_mode mode);

* timer: A pointer to the hrtimer to be started.
* time: The expiry time (absolute or relative, depending on mode).
* mode: Expiry mode (HRTIMER\_MODE\_ABS or HRTIMER\_MODE\_REL).
* Returns: 0 upon success, 1 if the timer was already active.

**5. Halting High Resolution Timers:**

* **hrtimer\_cancel():** Cancels a timer and awaits the completion of the timer's callback function execution, if it is currently running.

int hrtimer\_cancel(struct hrtimer \* timer);

\* Returns: 0 if the timer was not active, 1 if it was active.

* **hrtimer\_try\_to\_cancel():** Attempts to deactivate a timer but does not wait for the callback function.

int hrtimer\_try\_to\_cancel(struct hrtimer \* timer);

\* Returns: 0 if not active, 1 if active, -1 if the callback is currently executing.

**6. Adjusting the Timeout (for Periodic Timers):**

* The callback function of a periodic timer is required to define a new expiration time before returning HRTIMER\_RESTART.
* **hrtimer\_forward():** Advances the timer's expiry time by a specified interval, relative to a designated time.

u64 hrtimer\_forward(struct hrtimer \* timer, ktime\_t now, ktime\_t interval);

\* `timer`: HRT to advance.  
  
\* `now`: The time to advance past.  
  
\* `interval`: The interval by which to advance.  
  
\* Returns: The number of overruns.

* **hrtimer\_forward\_now():** Advances the timer's expiry time by a given interval, relative to the current time.

u64 hrtimer\_forward\_now(struct hrtimer \*timer, ktime\_t interval);

\* `timer`: HRT to advance.  
  
\* `interval`: The interval by which to advance.  
  
\* Returns: The number of overruns.

**7. Verifying Timer Status:**

* **hrtimer\_get\_remaining():** Returns the remaining time until the timer expires.

ktime\_t hrtimer\_get\_remaining(const struct hrtimer \* timer);

* **hrtimer\_callback\_running():** Checks if the timer's callback function is currently executing.

int hrtimer\_callback\_running(struct hrtimer \*timer);

\* Returns: 0 if not running, 1 if running.

* **hrtimer\_cb\_get\_time():** Retrieves the current time of the given timer.  
  ktime\_t hrtimer\_cb\_get\_time(struct hrtimer \*timer);

**Utilization of HRT in a Linux Device Driver (Example Analysis):**

The provided driver.c illustrates the fundamental usage of HRT in a kernel module:

1. **Initialization (etx\_driver\_init)**:
   * Allocates a character device.
   * Initializes the HRT by invoking hrtimer\_init(&etx\_hr\_timer, CLOCK\_MONOTONIC, HRTIMER\_MODE\_REL). This employs the monotonic clock for relative timing.
   * Assigns the timer's callback function to timer\_callback.
   * Establishes the initial timeout as 5 seconds (4 seconds + 1 billion nanoseconds) using ktime\_set(TIMEOUT\_SEC, TIMEOUT\_NSEC).
   * Initiates the timer through a call to hrtimer\_start(&etx\_hr\_timer, ktime, HRTIMER\_MODE\_REL).
2. **Timer Callback (timer\_callback)**:
   * Prints a message indicating the execution of the callback.
   * Advances the timer's expiry time by an additional 5 seconds by calling hrtimer\_forward\_now(&etx\_hr\_timer, ktime\_set(TIMEOUT\_SEC, TIMEOUT\_NSEC)). This renders the timer periodic.
   * Returns HRTIMER\_RESTART to signal that the timer should be reactivated upon reaching the new expiry time.
3. **Cleanup (etx\_driver\_exit)**:
   * Halts the timer by invoking hrtimer\_cancel(&etx\_hr\_timer).
   * Unregisters the character device.

**Important Considerationsntext**.

* Therefore, you **cannot** perform the following actions within the callback:
  + Go to sleep or relinquish the processor (e.g., calling msleep(), acquiring a blocking semaphore).
  + Acquire a mutex (blocking operations are not allowed).
  + Perform time-consuming tasks (should be kept short and efficient).
  + Access user-space virtual memory.

**Potential Questions and Answers:**

1. Q: Why was the High Resolution Timer introduced in addition to the existing kernel timer?  
   A: The traditional kernel timer, based on jiffies and the Cascading Timer Wheel, lacks the nanosecond-level precision required by certain applications. Attempts to merge it with high-resolution needs failed because the CTW's structure isn't optimized for maintaining a time-ordered list of events with fine-grained accuracy. HRT, using a red-black tree, efficiently manages timers with high resolution.
2. Q: What is the resolution of the High Resolution Timer? How does it compare to the kernel timer?  
   A: The High Resolution Timer offers a resolution of 64-bit nanoseconds. In contrast, the kernel timer's resolution is determined by the jiffy, which is system-dependent but typically in the millisecond range (e.g., 1ms, 4ms, or 10ms). HRT provides significantly finer-grained timing.
3. Q: What data structure does the High Resolution Timer use to manage timers, and why is it suitable?  
   A: HRT uses a red-black tree to store and manage timers. This self-balancing binary search tree is ideal for performance-focused applications because it allows efficient insertion, deletion, and searching of timers based on their expiry times in logarithmic time complexity (O(log n)). This ensures that timer management remains efficient even with a large number of active timers.
4. Q: How can you check if High Resolution Timers are enabled and available on a Linux system?  
   A: You can check in a few ways:
   * Examine the kernel configuration file (e.g., /boot/config-\*) for CONFIG\_HIGH\_RES\_TIMERS=y.
   * Inspect the /proc/timer\_list file for an entry with .resolution: 1 ns and event\_handler: hrtimer\_interrupt.
   * Use the clock\_getres() system call to get the resolution of system clocks.
5. Q: What are some use cases for High Resolution Timers in both user-space and the kernel?  
   A:
   * **User-space:** Applications requiring precise delays (e.g., multimedia playback synchronization using nanosleep), POSIX timers for accurate event scheduling, and interval timers for profiling or periodic tasks.
   * **Kernel:** Device drivers needing precise timing for hardware interactions (e.g., controlling sensors, communication protocols), multimedia subsystems for synchronization, and other kernel modules requiring accurate event scheduling.
6. Q: Explain the purpose of ktime\_t and the ktime\_set() function.  
   A: ktime\_t is a kernel data type designed to store time values with nanosecond precision. It's typically a 64-bit integer on 64-bit systems and a structure on 32-bit systems. The ktime\_set(long secs, long nanosecs) function is used to create a ktime\_t value by combining a given number of seconds and nanoseconds.
7. Q: What is the difference between HRTIMER\_MODE\_ABS and HRTIMER\_MODE\_REL when initializing or starting an HRT?  
   A:
   * HRTIMER\_MODE\_ABS (Absolute): The expiry time is specified as an absolute point in time, relative to the epoch of the chosen clock (CLOCK\_MONOTONIC or CLOCK\_REALTIME).
   * HRTIMER\_MODE\_REL (Relative): The expiry time is specified as an offset from the current time when hrtimer\_start() is called.
8. Q: What is the role of the callback function in an HRT, and what value should it return for a one-shot timer versus a periodic timer?  
   A: The callback function is executed when the HRT expires.
   * For a **one-shot timer** (that should not run again), the callback should return HRTIMER\_NORESTART.
   * For a **periodic timer** (that should run repeatedly), the callback must set a new expiration time (usually by using hrtimer\_forward() or hrtimer\_forward\_now()) and return HRTIMER\_RESTART.
9. Q: What are the key differences between hrtimer\_cancel() and hrtimer\_try\_to\_cancel()?  
   A:
   * hrtimer\_cancel() attempts to cancel the timer and waits for the timer's callback function to finish executing if it's currently running. It guarantees that the timer is not active after the call returns.
   * hrtimer\_try\_to\_cancel() attempts to deactivate the timer but returns immediately without waiting for the callback to finish. It can return -1 if the callback is currently executing and cannot be stopped at that moment.
10. Q: Why are there restrictions on the actions that can be performed within an HRT callback function?  
    A: The HRT callback function runs in interrupt context, which has strict limitations. Interrupt context code must be non-blocking and cannot access user-space memory or perform actions that might cause the system to sleep or wait. This is because interrupts need to be handled quickly and efficiently to maintain system responsiveness.

**Real-time Examples:**

1. **Multimedia Synchronization:** In a video playback application, HRTs can be used to precisely schedule the display of video frames and the playback of audio samples. This ensures smooth and synchronized multimedia output, preventing stuttering or audio-video desynchronization. For instance, if a video frame needs to be displayed every 33.33 milliseconds (for 30 frames per second), an HRT can be set with this relative timeout.
2. **High-Speed Data Acquisition:** Scientific instruments or industrial control systems might require sampling data at very precise intervals (e.g., every microsecond). HRTs can be used to trigger the data acquisition process at these precise times, ensuring accurate and consistent data collection. A driver for a high-speed ADC (Analog-to-Digital Converter) could use HRTs to schedule the sampling.
3. **Network Time Protocol (NTP):** While NTP primarily operates at a higher level, the underlying operating system needs accurate timers to measure and adjust system time. HRTs can provide the necessary high-resolution timing for implementing precise timekeeping and synchronization mechanisms within the kernel.
4. **Motor Control:** In robotics or automation, precise control of motor movements often requires generating Pulse Width Modulation (PWM) signals with very accurate timing. HRTs can be used to generate these PWM signals with the required frequency and duty cycle precision, leading to smoother and more accurate motor control.
5. **Real-time Operating Systems (RTOS) Emulation:** On general-purpose Linux systems, HRTs can be used to emulate the behavior of an RTOS scheduler for specific tasks. This allows for more precise control over task execution timing than standard Linux scheduling policies might offer, which is useful for certain embedded development or testing scenarios.