Chapter 11: Timer and Timer Services

11.1 Introduction

In embedded systems, system tasks and user tasks often schedule and perform activities after some time has elapsed. For example, a RTOS scheduler must perform a context switch of a preset time interval periodically-among tasks of equal priorities-to ensure execution fairness when conducting a round-robin scheduling algorithm. A software-based memory refresh mechanism must refresh the dynamic memory every so often or data loss will occur. In embedded networking devices, various communication protocols schedule activities for data retransmission and protocol recovery. The target monitor software sends system information to the host-based analysis tool periodically to provide system-timing diagrams for visualization and debugging.

In any case, embedded applications need to schedule future events. Scheduling future activities is accomplished through timers using timer services.

Timers are an integral part of many real-time embedded systems. A *timer* is the scheduling of an event according to a predefined time value in the future, similar to setting an alarm clock.

A complex embedded system is comprised of many different software modules and components, each requiring timers of varying timeout values. Most embedded systems use two different forms of timers to drive time-sensitive activities: hard timers and soft timers. Hard timers are derived from physical timer chips that directly interrupt the processor when they expire. Operations with demanding requirements for precision or latency need the predictable performance of a hard timer. Soft timers are software events that are scheduled through a software facility.

A soft-timer facility allows for efficiently scheduling of non-high-precision software events. A practical design for the soft-timer handling facility should have the following properties:

- efficient timer maintenance, i.e., counting down a timer,
- efficient timer installation, i.e., starting a timer, and
- efficient timer removal, i.e., stopping a timer.

While an application might require several high-precision timers with resolutions on the order of microseconds or even nanoseconds, not all of the time requirements have to be high precision. Even demanding applications also have some timing functions for which resolutions on the order of milliseconds, or even of hundreds of milliseconds, are sufficient. Aspects of applications requiring timeouts with course granularity (for example, with tolerance for bounded inaccuracy) should use soft timers. Examples include the Transmission Control Protocol module, the Real-time Transport Protocol module, and the Address Resolution Protocol module.

Another reason for using soft timers is to reduce system-interrupt overhead. The physical timer chip rate is usually set so that the interval between consecutive timer interrupts is within tens of milliseconds or even within tens of microseconds. The interrupt latency and overhead can be substantial and can grow with the increasing number of outstanding timers. This issue particularly occurs when each timer is implemented by being directly interfaced with the physical timer hardware.

This chapter focuses on:

- real-time clocks versus system clocks,
- programmable interval timers,
- timer interrupt service routines,
- timer-related operations,
- soft timers, and
- implementing soft-timer handling facilities.

11.2 Real-Time Clocks and System Clocks

In some references, the term *real-time clock* is interchangeable with the term *system clock* . Within the context of this book, however, these terminologies are separate, as they are different on various architectures.

Real-time clocks exist in many embedded systems and track time, date, month, and year. Commonly, they are integrated with battery-powered DRAM as shown in Figure 11.1. This integrated real-time clock becomes independent of the CPU and the programmable interval timer, making the maintenance of real time between system power cycles possible.

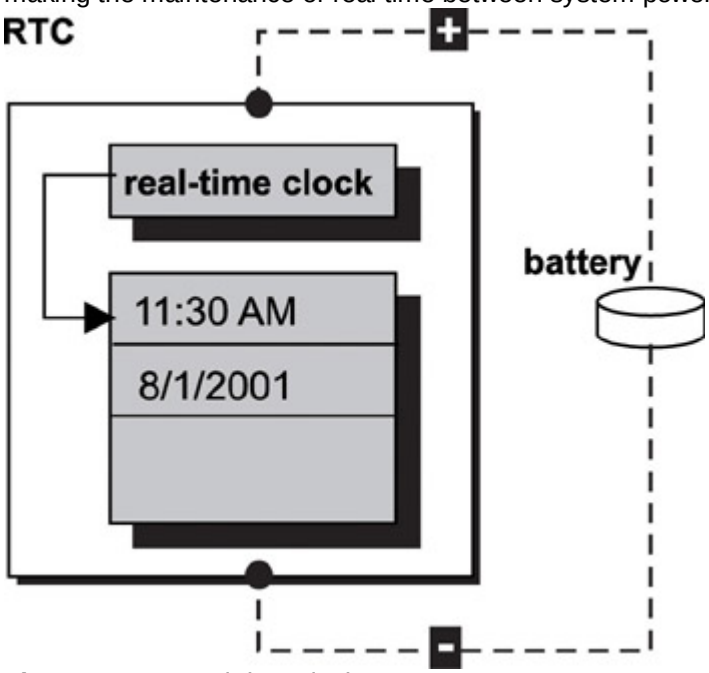

Figure 11.1: A real-time clock.

The job of the system clock is identical to that of the real-time clock: to track either real-time or elapsed time following system power up (depending on implementation). The initial value of the system clock is typically retrieved from the real-time clock at power up or is set by the user. The programmable interval timer drives the system clock, i.e. the system clock increments in value per timer interrupt. Therefore, an important function performed at the timer interrupt is maintaining the system clock, as shown in Figure 11.2.

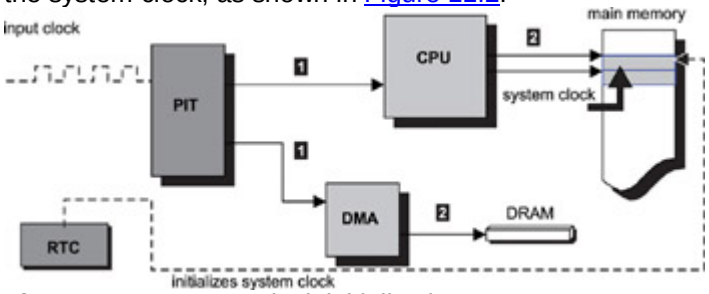

Figure 11.2: System clock initialization.

11.3 Programmable Interval Timers

The *programmable interval timer* (PIT), also known as the *timer chip* , is a device designed mainly to function as an event counter, elapsed time indicator, rate-controllable periodic event generator, as well as other applications for solving system-timing control problems.

The functionality of the PIT is commonly incorporated into the embedded processor, where it is called an *on-chip timer*. Dedicated stand-alone timer chips are available to reduce processor overhead. As different as the various timer chips can be, some general characteristics exist among them. For example, timer chips feature an input clock source with a fixed frequency, as well as a set of programmable timer control registers. The *timer interrupt rate* is the number of timer interrupts generated per second. The timer interrupt rate is calculated as a function of the input clock frequency and is set into a timer control register.

A related value is the *timer countdown value*, which determines when the next timer interrupt occurs. It is loaded in one of the timer control registers and decremented by one every input clock cycle. The remaining timer control registers determine the other modes of timer operation, such as whether periodic timer interrupts are generated and whether the countdown value should be automatically reloaded for the next timer interrupt.

Customized embedded systems come with schematics detailing the interconnection of the system components. From these schematics, a developer can determine which external components are dependent on the timer chip as the input clock source. For example, if a timer chip output pin interconnects with the control input pin of the DMA chip, the timer chip controls the DRAM refresh rate.

Timer-chip initialization is performed as part of the system startup. Generally, initialization of the timer chip involves the following steps:

- Resetting and bringing the timer chip into a known hardware state.
- Calculating the proper value to obtain the desired timer interrupt frequency and programming this value into the appropriate timer control register.
- Programming other timer control registers that are related to the earlier interrupt frequency with correct values. This step is dependent on the timer chip and is specified in detail by the timer chip hardware reference manual.
- Programming the timer chip with the proper mode of operation.
- Installing the timer interrupt service routine into the system.
- Enabling the timer interrupt.

The behavior of the timer chip output is programmable through the control registers, the most important of which is the *timer interrupt-rate register* (TINTR), which is as follows:

TINTR = F(x)

where *x* = frequency of the input crystal

Manufacturers of timer chips provide this function and the information is readily available in the programmer's reference manual.

The timer interrupt rate equals the number of timer interrupt occurrences per second. Each interrupt is called a *tick,* which represents a unit of time. For example, if the timer rate is 100 ticks, each tick represents an elapsed time of 10 milliseconds.

The periodic event generation capability of the PIT is important to many real-time kernels. At the heart of many real-time kernels is the announcement of the timer interrupt occurrence, or the *tick announcement*, from the ISR to the kernel, as well as to the kernel scheduler, if one exists. Many of these kernel schedulers run through their algorithms and conduct task scheduling at each tick.

11.4 Timer Interrupt Service Routines

Part of the timer chip initialization involves installing an interrupt service routine (ISR) that is called when the timer interrupt occurs. Typically, the ISR performs these duties:

- **Updating the system clock**-Both the absolute time and elapsed time is updated. *Absolute time* is time kept in calendar date, hours, minutes, and seconds. *Elapsed time* is usually kept in ticks and indicates how long the system has been running since power up.
- **Calling a registered kernel function to notify the passage of a preprogrammed period**-For the following discussion, the registered kernel function is called `announce_time_tick`.
- **Acknowledging the interrupt, reinitializing the necessary timer control register(s), and returning from interrupt.**

The `announce_time_tick` function is invoked in the context of the ISR; therefore, all of the restrictions placed on an ISR are applicable to `announce_time_tick`. In reality, `announce_time_tick` is part of the timer ISR. The `announce_time_tick` function is called to notify the kernel scheduler about the occurrence of a timer tick. Equally important is the announcement of the timer tick to the soft-timer handling facility. These concepts are illustrated in Figure 11.3.

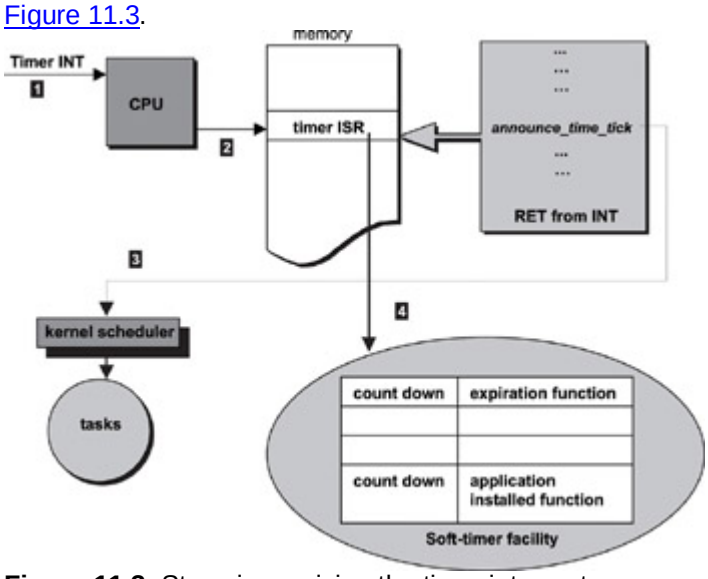

Figure 11.3: Steps in servicing the timer interrupt.

The soft-timer handling facility is responsible for maintaining the soft timers at each timer tick.

11.5 A Model for Implementing the Soft-Timer Handling Facility

The functions performed by the soft-timer facility, called the *timer facility* from now on, include:

- allowing applications to start a timer,
- allowing applications to stop or cancel a previously installed timer, and
- internally maintaining the application timers.

The soft-timer facility is comprised of two components: one component lives within the timer tick ISR and the other component lives in the context of a task.

This approach is used for several reasons. If all of the soft-timer processing is done with the ISR and if the work spans multiple ticks (i.e., if the timer tick handler does not complete work before the next clock tick arrives), the system clock might appear to drift as seen by the software that

tracks time. Worse, the timer tick events might be lost. Therefore, the timer tick handler must be short and must be conducting the least amount of work possible. Processing of expired soft timers is delayed into a dedicated processing task because applications using soft timers can tolerate a bounded timer inaccuracy. The *bounded timer inaccuracy* refers to the imprecision the timer may take on any value. This value is guaranteed to be within a specific range.

Therefore, a workable model for implementing a soft-timer handling facility is to create a dedicated processing task and call it a worker task, in conjunction with its counter part that is part of the system timer ISR. The ISR counterpart is given a fictitious name of `ISR_timeout_fn` for this discussion.

The system timer chip is programmed with a particular interrupt rate, which must accommodate various aspects of the system operation. The associated timer tick granularity is typically much smaller than the granularity required by the application-level soft timers. The `ISR_timeout_fn` function must work with this value and notify the worker task appropriately.

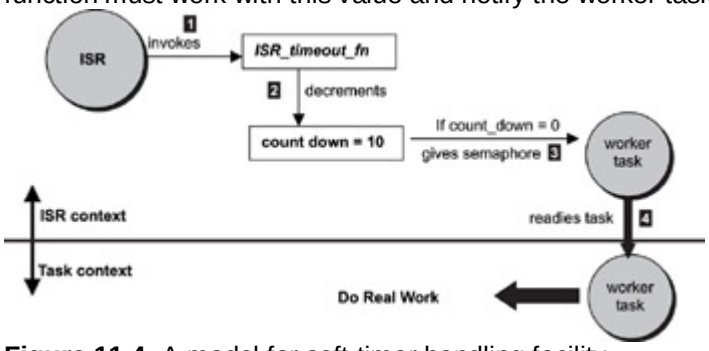

Figure 11.4: A model for soft-timer handling facility.

In the following example, assume that an application requires three soft timers. The timeout values equal 200ms, 300ms, and 500ms. The least common denominator is 100ms. If each hardware timer tick represents 10ms, then 100ms translates into a countdown value of 10. The `ISR_timeout_fn` keeps track of this countdown value and decrements it by one during each invocation. The `ISR_timeout_fn` notifies the worker task by a "give" operation on the worker task's semaphore after the countdown value reaches zero, effectively allowing the task to be scheduled for execution. The `ISR_timeout_fn` then reinitializes the countdown value back to 10. This concept is illustrated in Figure 11.4.

In the ISR-to-processing-task model, the worker task must maintain an application-level, timer-countdown table based on 100ms granularity. In this example, the timer table has three countdown values: 2, 3, and 5 representing the 200ms, 300ms, and the 500ms application-requested timers. An application-installed, timer-expiration function is associated with each timer. This concept is illustrated in Figure 11.5.

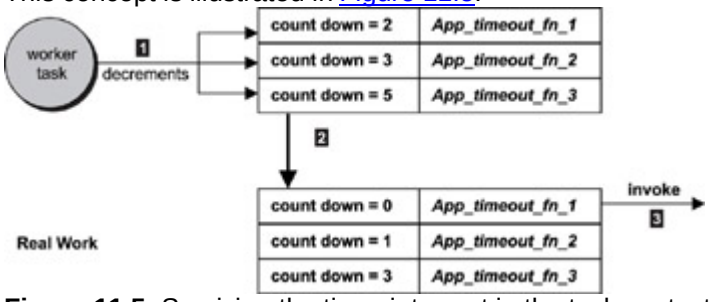

Figure 11.5: Servicing the timer interrupt in the task context.

The three soft timers, which are simply called timers unless specified otherwise, are decremented by the worker task each time it runs. When the counter reaches zero, the application timer has expired. In this example, the 200ms timer and the associated function `App_timeout_fn_1`,

which the application installs, is invoked. As shown in Figures 11.4 and 11.5, a single ISR-level timer drives three application timers at the task-level, providing a good reason why these timers are called soft timers. The decrease in the number of ISR timers installed improves the overall system performance.

These application-installed timers are called *soft timers* because processing is not synchronized with the hardware timer tick. It is a good idea to explore this concept further by examining possible delays that can occur along the delivery path of the timer tick.

11.5.1 Possible Processing Delays

The first delay is the event-driven, task-scheduling delay. As shown in the previous example, the maintenance of soft timers is part of `ISR_timeout_fn` and involves decrementing the expiration time value by one. When the expiration time reaches zero, the timer expires and the associated function is invoked. Because `ISR_timeout_fn` is part of the ISR, it must perform the smallest amount of work possible and postpone major work to a later stage outside the context of the ISR. Typical implementations perform real work either inside a worker task that is a dedicated daemon task or within the application that originally installed the timer. The minimum amount of work completed within the ISR by the installed function involves triggering an asynchronous event to the worker task, which typically translates into the kernel call `event_send`, should one exist. Alternatively, the triggering can also translate into the release of a semaphore on which the worker task is currently blocked. The notification delay caused by event generation from the ISR to the daemon task is the first level of delay, as shown in Figure 11.6. Note that the hypothetical kernel function `event_send` and the semaphore release function must be callable from within an ISR.

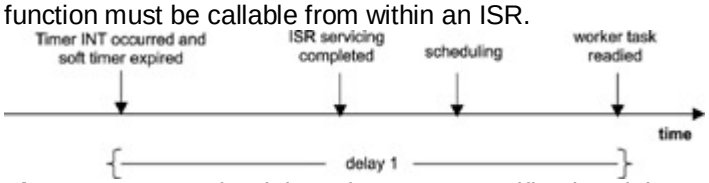

Figure 11.6: Level 1 delays-timer event notification delay.

The second delay is the priority-based, task-scheduling delay. In a typical RTOS, tasks can execute at different levels of execution priorities. For example, a worker task that performs timer expiration-related functions might not have the highest execution priority. In a priority-based, kernel-scheduling scheme, a worker task must wait until all other higher priority tasks complete execution before being allowed to continue. With a round-robin scheduler, the worker task must wait for its scheduling cycle in order to execute. This process represents the second level of delay as shown in Figure 11.7.

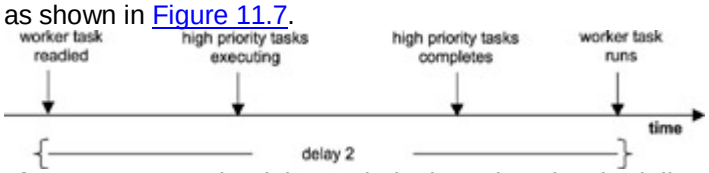

Figure 11.7: Level 2 delays-priority-based, task-scheduling delays.

Another delay is introduced when an application installs many soft timers. This issue is explored further in the next section when discussing the concept of timing wheels.

11.5.2 Implementation Considerations

A soft-timer facility should allow for efficient timer insertion, timer deletion and cancellation, and timer update. These requirements, however, can conflict with each other in practice. For example, imagine the linked list-timer implementation shown in Figure 11.8. The fastest way to start a timer is to insert it either at the head of the timer list or at the tail of the timer list if the timer entry data structure contains a double-linked list.

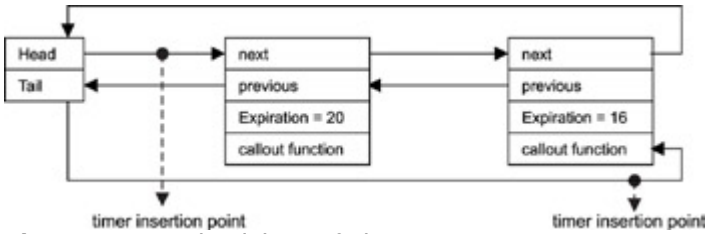

Figure 11.8: Maintaining soft timers.

Because the timer list is not sorted in any particular order, maintaining timer ticks can prove costly. Updating the timer list at each tick requires the worker task to traverse the entire linked list and update each timer entry. When the counter reaches zero, the callout function is invoked. A timer handle is returned to the application in a successful timer installation. The cancellation of a timer also requires the worker task to traverse the entire list. Each timer entry is compared to the timer handle, and, when a match is found, that particular timer entry is removed from the timer list.

As shown in Figure 11.9, while timer installation can be performed in constant time, timer cancellation and timer update might require O(*N*) in the worst case.

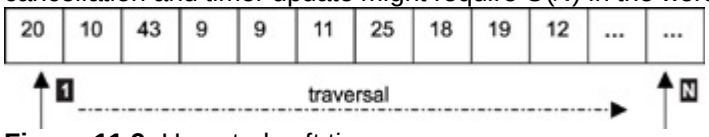

Figure 11.9: Unsorted soft timers.

Sorting expiration times in ascending order results in efficient timer bookkeeping. In the example, only the first timer-entry update is necessary, because all the other timers are decremented implicitly. In other words, when inserting new timers, the timeout value is modified according to the first entry before inserting the timer into the list.

As shown in Figure 11.10, while timer bookkeeping is performed in constant time, timer installation requires search and insertion. The cost is `O(log(N))`, where *N* is the number of entries in the timer list. The cost of timer cancellation is also `O(log(N))`.

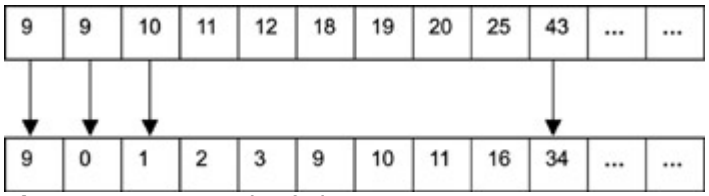

Figure 11.10: Sorted soft timers.

11.6 Timing Wheels

As shown in Figure 11.11, the *timing wheel* is a construct with a fixed-size array in which each slot represents a unit of time with respect to the precision of the soft-timer facility. The timing wheel approach has the advantage of the sorted timer list for updating the timers efficiently, and it also provides efficient operations for timer installation and cancellation.

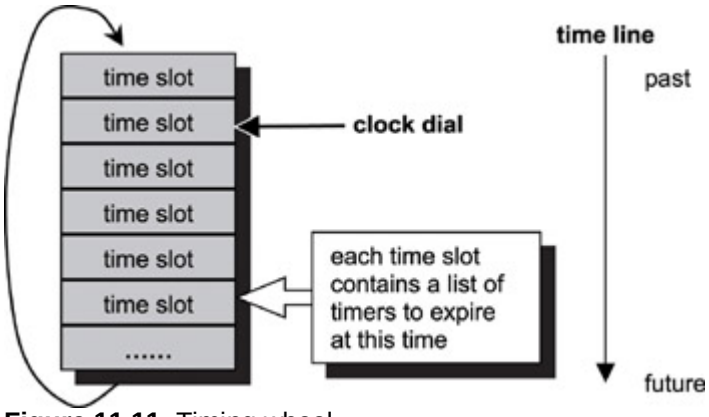

Figure 11.11: Timing wheel.

The soft-timer facility installs a periodic timeout (a clock tick) using the underlying timer hardware. This hardware-based periodic timer, drives all of the soft timers installed within the facility. The frequency of the timeout determines the precision of the soft-timer facility. For example, if the precision defines a tick occurrence every 50ms, each slot represents the passing of 50ms, which is the smallest timeout that can be installed into the timer facility. In addition, a doubly linked list of timeout event handlers (also named *callback functions* or *callbacks* for short) is stored within each slot, which is invoked upon timer expiration. This list of timers represents events with the same expiration time.

Each timer slot is represented in Figure 11.12.

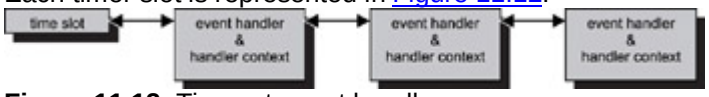

Figure 11.12: Timeout event handlers.

The clock dial increments to the next time slot on each tick and wraps to the beginning of the time-slot array when it increments past the final array entry. The idea of the timing wheel is derived from this property. Therefore, when installing a new timer event, the current location of the clock dial is used as the reference point to determine the time slot in which the new event handler will be stored. Consider the following example as depicted in Figure 11.13. Assume each time slot represents the passing of 50ms, which means that 50ms has elapsed between ticks.

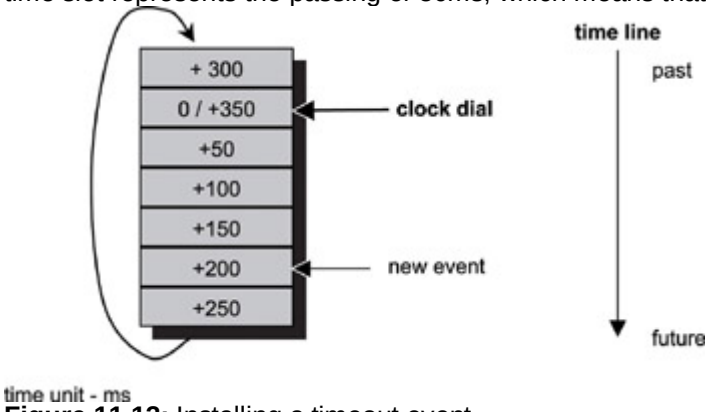

Figure 11.13: Installing a timeout event.

The time slot marked *+200* is the slot in which to store an event handler if the developer wants to schedule a 200ms timeout in the future. The location of the clock dial is the 'beginning of time' on the time line, in other words, the reference point. At a minimum, the timer handle returned to the calling application is the array index.

11.6.1 Issues

A number of issues are associated with the timing wheel approach. The number of slots in the timing wheel has a limit, whatever that might be for the system. The example in Figure 11.13 makes this problem obvious. The maximum schedulable event is 350ms. How can a 400ms timer be scheduled? This issue causes an overflow condition in the timing wheel. One approach is to deny installation of timers outside the fixed range. A better solution is to accumulate the events causing the overflow condition in a temporary event buffer until the clock dial has turned enough so that these events become schedulable. This solution is illustrated in Figure 11.14.

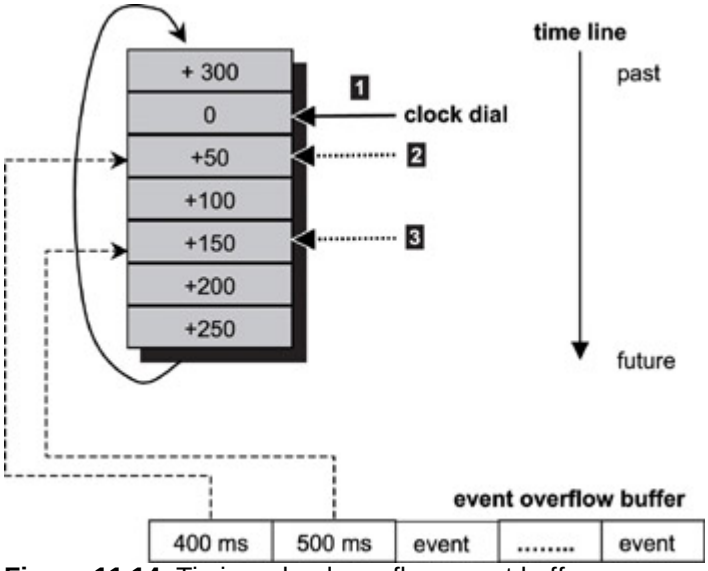

Figure 11.14: Timing wheel overflow event buffer.

For example, in order to schedule a 400ms timeout when the clock dial is at location 1, this event must be saved in the event overflow buffer until the clock dial reaches location 2. To schedule a 500ms timer when clock dial is at location 1, this event must be saved in the event overflow buffer until the clock dial reaches location 3. The expired events at location 2 and location 3 must be serviced first, and then the new events installed. The event overflow buffer must be examined to see if new events need to be scheduled when the clock dial moves at each clock tick to the next slot. This process implies that the events in the overflow buffer must be sorted in increasing order. New events are inserted in order and can be expensive if the overflow buffer contains a large number of entries.

Another issue associated with the timing wheel approach is the precision of the installed timeouts. Consider the situation in which a 150ms timer event is being scheduled while the clock is ticking but before the tick announcement reaches the timing wheel. Should the timer event be added to the +150ms slot or placed in the +200ms slot? On average, the error is approximately half the size of the tick. In this example, the error is about 25ms.

One other important issue relates to the invocation time of the callbacks installed at each time slot. In theory, the callbacks should all be invoked at the same time at expiration, but in reality, this is impossible. The work performed by each callback is unknown; therefore, the execution length of each callback is unknown. Consequently, no guarantee or predictable measures exist concerning when a callback in a later position of the list can be called, even in a worst-case scenario. This issue introduces non-determinism into the system and is undesirable. Figure 11.15 illustrates the problem.

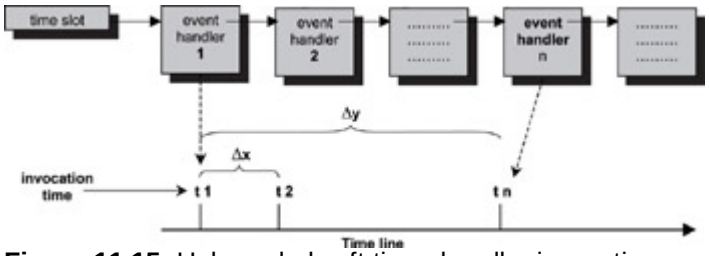

Figure 11.15: Unbounded soft-timer handler invocation.

Event handler 1 is invoked at *t1* when the timeout has just expired. Similarly, event handler *n* is invoked at *tn* when the previous (*n* -1) event handlers have finished execution. The interval *x* and *y* is non-deterministic because the length of execution of each handler is unknown. These intervals are also unbounded.

Ideally, the timer facility could guarantee an upper bound; for example, regardless of the number of timers already installed in the system, event handler *n* is invoked no later than 200ms from the actual expiration time.

This problem is difficult, and the solution is application specific.

11.6.2 Hierarchical Timing Wheels

The timer overflow problem presented in the last section can be solved using the *hierarchical timing wheel* approach.

The soft-timer facility needs to accommodate timer events spanning a range of values. This range can be very large. For example accommodating timers ranging from 100ms to 5 minutes requires a timing wheel with 3,000 (5 × 60 × 10) entries. Because the timer facility needs to have a granularity of at least 100ms and there is a single array representing the timing wheel,

10 × 100ms = 1 sec
10 entries/sec

60 sec = 1 minute
60 × 10 entries / min

therefore:

5 × 60 × 10 = total number of entries needed for the timing wheel with a granularity of 100ms.

A hierarchical timing wheel is similar to a digital clock. Instead of having a single timing wheel, multiple timing wheels are organized in a hierarchical order. Each timing wheel in the hierarchy set has a different granularity. A clock dial is associated with each timing wheel. The clock dial turns by one unit when the clock dial at the lower level of the hierarchy wraps around. Using a hierarchical timing wheel requires only 75 (10 + 60 + 5) entries to allow for timeouts with 100ms resolution and duration of up to 5 minutes.

With a hierarchical timing wheels, there are multiple arrays, therefore

10 × 100ms = 1 sec
10 entries/sec
the 1st array (leftmost array as shown in Figure 11.16)

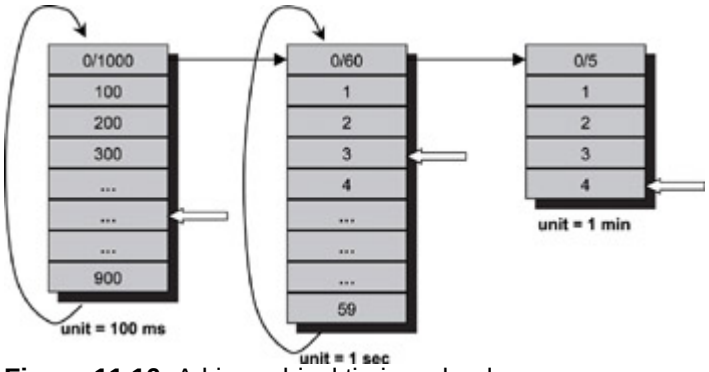

Figure 11.16: A hierarchical timing wheel.

60 sec = 1 minute
60 entries / min
the 2nd array (middle array shown in Figure 11.16)

5 entries for 5 minutes
3rd array

therefore:

5 + 60 + 10 = total number of entries needed for the hierarchal timing wheels.

The reduction in space allows for the construction of higher precision timer facilities with a large range of timeout values. Figure 11.16 depicts this concept.

For example, it is possible to install timeouts of 2 minutes, 4 seconds, and 300 milliseconds. The timeout handler is installed at the 2-minute slot first. The timeout handler determines that there are still 4.3 seconds to go when the 2 minutes is up. The handler installs itself at the 4-second timeout slot. Again, when 4 seconds have elapsed, the same handler determines that 300 milliseconds are left before expiring the timer. Finally, the handler is reinstalled at the 300-millisecond timeout slot. The real required work is performed by the handler when the last 300ms expire.

11.7 Soft Timers and Timer Related Operations

Many RTOSs provide a set of timer-related operations for external software components and applications through API sets. These common operations can be cataloged into these groups:

- **group 1**-provides low-level hardware related operations,
- **group 2**-provides soft-timer-related services, and
- **group 3**-provides access either to the storage of the real-time clock or to the system clock.

Not all of the operations in each of these three groups, however, are offered by all RTOSs, and some RTOSs provides additional operations not mentioned here.

The first group of operations is developed and provided by the BSP developers. The group is considered low-level system operations. Each operation in the group is given a fictitious function name for this discussion. Actual function names are implementation dependent.

Table 11.1: Group 1 Operations.

Typical Operations	Description
`sys_timer_enable`	Enables the system timer chip interrupts. As soon as this

Table 11.1: Group 1 Operations.

Typical Operations	Description
	operation is invoked, the timer interrupts occur at the preprogrammed frequency, assuming that the timer chip has been properly initialized with the desired values. Only after this operation is complete can kernel task scheduling take place.
`sys_timer_disable`	Disables the system timer chip interrupts. After this operation is complete, the kernel scheduler is no longer in effect. Other system-offered services based on time ticks are disabled by this operation as well.
`sys_timer_connect`	Installs the system timer interrupt service routine into the system exception vector table. The new timer ISR is invoked automatically on the next timer interrupt. The installed function is either part of the BSP or the kernel code and represents the 'timer ISR' depicted in Figure 11.3, page 172.
	Input Parameters: 1. New timer interrupt service routine
`sys_timer_getrate`	Returns the system clock rate as the number of ticks per second that the timer chip is programmed to generate.
	Output Parameter: 1. Ticks per second
`sys_timer_setrate`	Sets the system clock rate as the number of ticks per second the timer chip generates. Internally, this operation reprograms the PIT to obtain the desired frequency.
	Input Parameters: 1. Ticks per second
`sys_timer_getticks`	Returns the elapsed timer ticks since system power up. This figure is the total number of elapsed timer ticks since the system was first powered on.
	Output Parameters: 1. Total number of elapsed timer ticks

The second group of timer-related operations includes the core timer operations that are heavily used by both the system modules and applications. Either an independent timer-handling facility or a built-in one that is part of the kernel offers these operations. Each operation in the group is given a fictitious function name for this discussion. Actual function names are implementation dependent.

The `timer_create` and `timer_start` operations allow the caller to start a timer that expires some time in the future. The caller-supplied function is invoked at the time of expiration, which is specified as a time relative with respect to when the `timer_start` operation is invoked. Through these timer operations, applications can install soft timers for various purposes. For example, the TCP protocol layer can install retransmission timers, the IP protocol layer can install

packet-reassembly discard timers, and a device driver can poll an I/O device for input at predefined intervals.

Table 11.2: Group 2 Operations.

Typical Operations	Description
`timer_create`	Creates a timer. This operation allocates a soft-timer structure. Any software module intending to install a soft timer must first create a timer structure. The timer structure contains control information that allows the timer-handling facility to update and expire soft timers. A timer created by this operation refers to an entry in the soft-timers array depicted in Figure 11.3.
	Input Parameter: Expiration time User function to be called at the timer expiration
	Output Parameter: An ID identifying the newly created timer structure
	Note: This timer structure is implementation dependent. The returned timer ID is also implementation dependent.
`timer_delete`	Deletes a timer. This operation deletes a previously created soft timer, freeing the memory occupied by the timer structure.
	Input Parameter: 1. An ID identifying a previously created timer structure
	Note: This timer ID is implementation dependent.
`timer_start`	Starts a timer. This operation installs a previously created soft timer into the timer-handling facility. The timer begins running at the completion of this operation.
	Input Parameter: 1. An ID identifying a previously created timer structure
`timer_cancel`	Cancels a currently running timer. This operation cancels a timer by removing the currently running timer from the timer-handling facility.
	Input Parameter: 1. An ID identifying a previously created timer structure

The third group is mainly used by user-level applications. The operations in this group interact either with the system clock or with the real-time clock. A system utility library offers these operations. Each operation in the group is given a fictitious function name for this discussion. Actual function names are implementation dependent.

Table 11.3: Group 3 Operations.

Typical Operations	Description
`clock_get_time`	Gets the current clock time, which is the current running value either from the system clock or from the real-time clock.

Table 11.3: Group 3 Operations.

Typical Operations	Description
	Output Parameter: A time structure containing seconds, minutes, or hours[1]
`clock_set_time`	Sets the clock to a specified time. The new time is set either into the system clock or into the real-time clock.
	Input Parameter: A time structure containing seconds, minutes, or hours[1]

1. The time structure is implementation dependent.

11.8 Points to Remember

Some points to remember include the following:

- Hardware timers (hard timers) are handled within the context of the ISR. The timer handler must conform to general restrictions placed on the ISR.
- The kernel scheduler depends on the announcement of time passing per tick.
- Soft timers are built on hard timers and are less accurate because of various delays.
- A soft-timer handling facility should allow for efficient timer installation, cancellation, and timer bookkeeping.
- A soft-timer facility built using the timing-wheel approach provides efficient operations for installation, cancellation, and timer bookkeeping.