As a member of software engineering team, you are tasked to design a timer management system for the Real Time Operating System that is being developed in house for use in an embedded communication device. The following describes the high level design document.

[T](https://doc.micrium.com/display/osiiidoc/Timer+States+-+Timer+Management+Internals#ww1048457)he figure below shows the state diagram of a timer.

Tasks can call RTOSTmrStateGet() to find out the state of a timer. Also, at any time during the countdown process, the application code can call RTOSTmrRemainGet() to find out how much time remains before the timer reaches zero (0). The value returned is expressed in “timer ticks.” If timers are decremented at a rate of 10 Hz then a count of 50 corresponds to 5 seconds. If the timer is in the stop state, the time remaining will correspond to either the initial delay (one shot or periodic with initial delay), or the period if the timer is configured for periodic without initial delay.

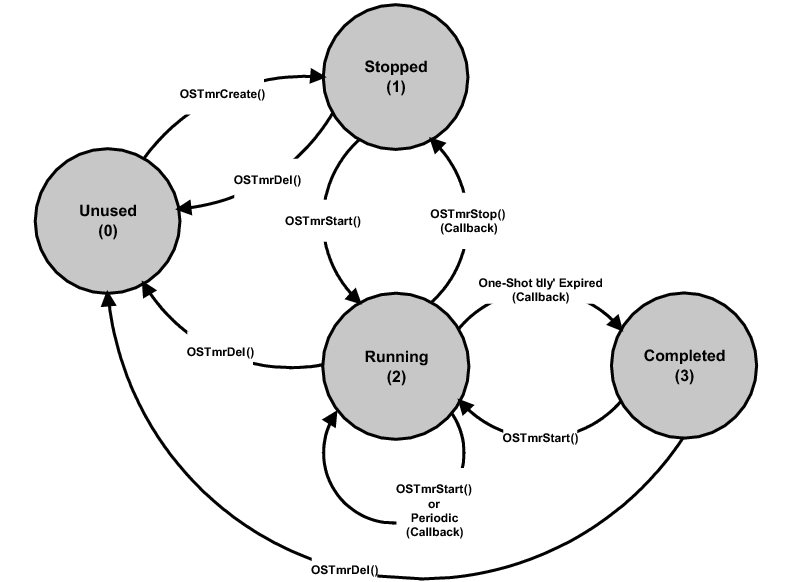


Figure - Timer State Diagram

**(1)**

 The “Unused” state is a timer that has not been created or has been “deleted.”

**(2)**

 When creating a timer or calling RTOSTmrStop(), the timer is placed in the “stopped” state.

**(3)**

 A timer is placed in running state when calling RTOSTmrStart(). The timer stays in that state unless it’s stopped, deleted, or completes its one shot.

**(4)**

 The “Completed” state is the state a one-shot timer is in when its delay expires.

A timer is a kernel object as defined by the RTOS\_TMR data type as shown in the listing below:

|  |
| --- |
| typedef  struct  os\_tmr { |
| INT8U             RTOSTmrType;   /\* Should always be set to RTOS\_TMR\_TYPE for timers \*/ |
| RTOS\_TMR\_CALLBACK RTOSTmrCallback;   /\* Function to call when timer expires \*/ |
| void             \*RTOSTmrCallbackArg; /\* Argument to callback function \*/ |
| void             \*RTOSTmrNext;       /\* Double link list pointers   \*/ |
| void             \*RTOSTmrPrev; |
| INT32U           RTOSTmrMatch;       /\* Timer expires when RTOSTmrTickCtr = RTOSTmrMatch\*/ |
| INT32U           RTOSTmrDly;         /\* Delay time before periodic update starts \*/ |
| INT32U           RTOSTmrPeriod;      /\* Period to repeat timer  \*/ |
| INT8U           \*RTOSTmrName;        /\* Name to give the timer   \*/ |
| INT8U             RTOSTmrOpt;         /\* Options (see RTOS\_TMR\_OPT\_xxx)  \*/ |
| INT8U             RTOSTmrState;       /\* Indicates the state of the timer:\*/ |
| /\*     RTOS\_TMR\_STATE\_UNUSED   \*/ |
| /\*     RTOS\_TMR\_STATE\_RUNNING  \*/ |
| /\*     RTOS\_TMR\_STATE\_STOPPED  \*/ |
| } RTOS\_TMR; |
|  |

The structure starts with a “RTOSTmrType” field, which allows it to be recognized by OS as a timer. Other kernel objects will also have a “Type” as the first member of the structure. If a function is passed a kernel object, OS is able to confirm that it is passed the proper data type. For example, if passing a message queue to a timer service (for example RTOSTmrStart()) then OS will be able to recognize that an invalid object was passed, and return an error code accordingly.

Each kernel object can be given a name (RTOSTmrName) for easier recognition by debuggers. This member is simply a pointer to an ASCII string which is assumed to be NUL terminated.

 The . RTOSTmrCallback member is a pointer to a function that is called when the timer expires. If a timer is created and passed a NULL pointer, a callback would not be called when the timer expires.

 If there is a non-NULL . RTOSTmrCallback then the application code could have also specified that the callback be called with an argument when the timer expires. This is the argument that would be passed in this call.

RTOSTmrNext and RTOSTmrPrev are pointers used to link a timer in a doubly linked list. These are described later.

The RTOSTmrDly field contains the one-shot time when the timer is configured (i.e., created) as a one-shot timer and the initial delay when the timer is created as a periodic timer. The value is expressed in multiples of 1/RTOS\_CFG\_TMR\_TASK\_RATE\_HZ of a second.

The RTOSTmrPeriod field is the timer period when the timer is created to operate in periodic mode. The value is expressed in multiples of 1/RTOS\_CFG\_TMR\_TASK\_RATE\_HZ of a second.

The RTOSTmrOpt field contains options that are passed to RTOSTmrCreate().

The RTOSTmrState field represents the current state of the timer (see the figure in [Timers States](https://doc.micrium.com/display/osiiidoc/Timer+States+-+Timer+Management+Internals)).

Even if the internals of the RTOS\_TMR data type are understood, the application code should never access any of the fields in this data structure directly. Instead, you should always use the Application Programming Interfaces (APIs) provided.

RTOS\_TmrTask() is a task created by OS (i.e. you) and its priority is configurable by the user. RTOS\_TmrTask() is typically set to a medium priority.

RTOS\_TmrTask() is a periodic task and uses the same interrupt source used to generate clock ticks. However, timers are generally updated at a slower rate (i.e., typically 10 Hz or so) and thus, the timer tick rate is divided down in software. If the tick rate is 1000 Hz and the desired timer rate is 10 Hz then the timer task will be signaled every 100th tick interrupt as shown in the figure below.

**1)**  The tick ISR occurs and assumes interrupts are enabled and executes.

**(2)** The tick ISR signals the tick task that it is time for it to update timers.

**(3)** The tick ISR terminates, however there might be higher priority tasks that need to execute (assuming the timer task has a lower priority). Therefore, OS runs the higher priority task(s).

**(4)** When all higher priority tasks have executed, OS switches to the timer task and determines that there are three timers that expired.

**(5)** The callback for the first timer is executed.

**(6)** The callback for the second expired timer is executed.

**(7)** The callback for the third expired timer is executed.

There are a few interesting things to notice:

* Execution of the callback functions is performed within the context of the timer task. This means that the application code will need to make sure there is sufficient stack space for the timer task to handle these callbacks.
* The callback functions are executed one after the other based on the order they are found in the timer list.
* The execution time of the timer task greatly depends on how many timers expire and how long each of the callback functions takes to execute. Since the callbacks are provided by the application code they have a large influence on the execution time of the timer task.
* The timer callback functions must never wait on events because this would delay the timer task for excessive amounts of time, if not forever.
* Callbacks should execute as quickly as possible.

[Timer List - Timer Management Internals](https://doc.micrium.com/display/osiiidoc/Timer+List+-+Timer+Management+Internals) Design Options

µC/OS-III applications may require many timers. The timer manager implements a simple linear list where each timer is linked in a doubly linked list as shown in the figure below.

1. RTOSTmrListEntries
2. RTOSTmrListPtr

0

1. RTOSTmrTickCtr

Figure - Empty Timer List

**(1)**

 RTOSTmrListEntries contains the current number of entries in the list. This variable is updated whenever timers are added or removed from the list.

**(2)**

 RTOSTmrListPtr contains a pointer to a doubly linked list of timers that the timer manager will need to update.

**(3)**

 RTOSTmrTickCtr is incremented by RTOS\_TmrTask() every time the tick ISR signals the task. This counter basically keeps track of the number of times the timer task has been signaled.

Timers are inserted in the timer list by calling RTOSTmrStart() and, a timer must be created before it can be used. Newly created timers are always inserted at the beginning of the list as shown in the figure following the code listing below and the code listing itself.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | RTOS\_TMR  MyTmr1;  RTOS\_TMR  MyTmr2;      void MyTmrCallbackFnct1 (void \*p\_arg)  {      /\* Do something when timer #1 expires \*/  }      void MyTmrCallbackFnct2 (void \*p\_arg)  {      /\* Do something when timer #2 expires \*/  }      void MyTask (void \*p\_arg)  {      RTOS\_ERR  err;          while (DEF\_ON) {          :          RTOSTmrCreate((RTOS\_TMR            \*)&MyTmr1,                     (RTOS\_CHAR           \*)“My Timer #1”,                     (RTOS\_TICK            )1,                     (RTOS\_TICK            )0,                     (RTOS\_OPT             )RTOS\_OPT\_TMR\_ONE\_SHOT,                     (RTOS\_TMR\_CALLBACK\_PTR)MyTmrCallbackFnct1,                     (void              \*)0,                     (RTOS\_ERR            \*)&err);          /\* Check ’err” \*/          RTOSTmrStart ((RTOS\_TMR \*)&MyTmr1,                     (RTOS\_ERR \*)&err);          /\* Check “err” \*/          // Continues in the next code listing! |

Listing - Creating and Starting a timer

Since this is the first timer inserted in the timer list, the .NextPtr and .PrevPtr both point to NULL.

RTOSTmrListEntries

RTOSTmrListPtr

RTOSTmrTickCtr

RTOSTmrNextPtr

RTOSTmrPrevPtr

RTOSTmrTickCtr

0

0

OS\_TMR

0

Figure - Inserting a timer in the timer list

The code below shows creating and starting another timer. This is performed “before” the timer task is signaled.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | // Continuation of code from previous code listing.          :          :          RTOSTmrCreate((RTOS\_TMR            \*)&MyTmr2,                     (RTOS\_CHAR           \*)“My Timer #2”,                     (RTOS\_TICK            )10,                     (RTOS\_TICK            )0,                     (RTOS\_OPT             )RTOS\_OPT\_TMR\_ONE\_SHOT,                     (RTOS\_TMR\_CALLBACK\_PTR)MyTmrCallbackFnct2,                     (void              \*)0,                     (RTOS\_ERR            \*)&err);          /\* Check ’err” \*/          RTOSTmrStart ((RTOS\_TMR \*)&MyTmr,                     (RTOS\_ERR \*)&err);          /\* Check ’err” \*/      }  } |

Listing - Creating and Starting a timer - continued

The “second timer” is inserted at the head of the list as shown in the figure below.

RTOSTmrListEntries

RTOSTmrListPtr

RTOSTmrTickCtr

RTOSTmrNextPtr

RTOSTmrPrevPtr

RTOSTmrTickCtr

0

OS\_TMR

RTOSTmrNextPtr

RTOSTmrPrevPtr

RTOSTmrTickCtr

OS\_TMR

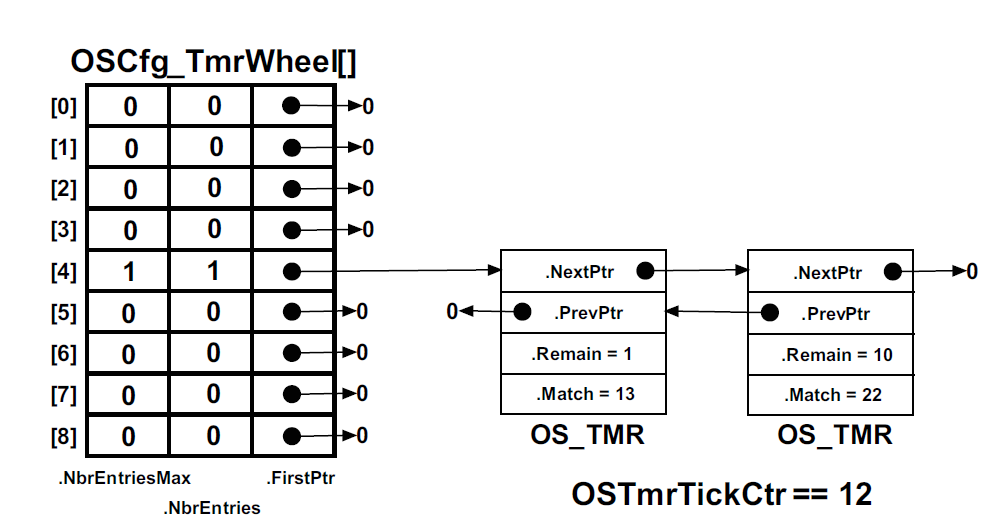
Figure - Inserting a second timer in the tick list

When the timer task executes, it starts by incrementing RTOSTmrTickCtr and goes through the list (linearly) and checks each of the RTOSTmrMatch fields is equal the OSTmtTickCtr. When equal, the timer manager executes the callback function associated with the timer.

When inserting the timer in the link list RTOSTmrMatch field is calculated as OSTmtTickCtr + RTOSTmrDly. If the timer is set to periodic, reloads the RTOSTmrMatch is calculated as OSTmtTickCtr + RTOSTmrPeriod. If the timer is configured as a one-shot timer, the timer is removed from the list upon expiration.

Timer management occurs at the task level. The list must be protected using an internal mutual exclusion semaphore (mutex) or, by locking the scheduler. It’s recommend that you use (and thus enable) mutexes because locking the scheduler impacts task responsiveness of other, higher priority tasks in your application.

Your timer management module might need to literally maintain hundreds of timers, so it needs to be implemented such that it does not take too much of CPU time to update the timers. So you may need to design a hash table of link lists to keep the length of the link lists short. You may want to use the value of (OSTmtTickCtr + RTOSTmrDly) % N to calculate the index of the hash table.



RTOSTmrTickCtr is incremented by RTOSTmrTask() every time the tick ISR signal the task.

Timers are inserted in the timer list by calling RTOSTmrSTart().

You need to implement the following Timer APIs for the tasks that you’re the kernel timer:

|  |
| --- |
| /\* |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*                                                   CREATE A TIMER |
| \* |
| \* Description: This function is called by your application code to create a timer. |
| \* |
| \* Arguments  : dly           Initial delay. |
| \*                            If the timer is configured for ONE-SHOT mode, this is the timeout used |
| \*                            If the timer is configured for PERIODIC mode, this is the first timeout to wait for |
| \*                               before the timer starts entering periodic mode |
| \* |
| \*              period        The 'period' being repeated for the timer. |
| \*                               If you specified 'RTOS\_TMR\_OPT\_PERIODIC' as an option, when the timer expires, it will |
| \*                               automatically restart with the same period. |
| \* |
| \*              opt           Specifies either: |
| \*                               RTOS\_TMR\_OPT\_ONE\_SHOT       The timer counts down only once |
| \*                               RTOS\_TMR\_OPT\_PERIODIC       The timer counts down and then reloads itself |
| \* |
| \*              callback      Is a pointer to a callback function that will be called when the timer expires.  The |
| \*                               callback function must be declared as follows: |
| \* |
| \*                               void MyCallback (RTOS\_TMR \*ptmr, void \*p\_arg); |
| \* |
| \*              callback\_arg  Is an argument (a pointer) that is passed to the callback function when it is called. |
| \* |
| \*              pname         Is a pointer to an ASCII string that is used to name the timer.  Names are useful for |
| \*                               debugging. |
| \* |
| \*              perr          Is a pointer to an error code.  '\*perr' will contain one of the following: |
| \*                               RTOS\_ERR\_NONE |
| \*                               RTOS\_ERR\_TMR\_INVALID\_DLY     you specified an invalid delay |
| \*                               RTOS\_ERR\_TMR\_INVALID\_PERIOD  you specified an invalid period |
| \*                               RTOS\_ERR\_TMR\_INVALID\_OPT     you specified an invalid option |
| \*                               RTOS\_ERR\_TMR\_NON\_AVAIL       if there are no free timers from the timer pool |
| \* |
| \* Returns    : A pointer to an RTOS\_TMR data structure. |
| \*              This is the 'handle' that your application will use to reference the timer created. |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*/ |
| RTOS\_TMR  \*RTOSTmrCreate (INT32U           dly, |
| INT32U           period, |
| INT8U            opt, |
| RTOS\_TMR\_CALLBACK  callback, |
| void            \*callback\_arg, |
| INT8U           \*pname, |
| INT8U           \*perr) |

|  |
| --- |
| /\* |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*                                                   DELETE A TIMER |
| \* |
| \* Description: This function is called by your application code to delete a timer. |
| \* |
| \* Arguments  : ptmr          Is a pointer to the timer to stop and delete. |
| \* |
| \*              perr          Is a pointer to an error code.  '\*perr' will contain one of the following: |
| \*                               RTOS\_ERR\_NONE |
| \*                               RTOS\_ERR\_TMR\_INVALID        'ptmr'  is a NULL pointer |
| \*                               RTOS\_ERR\_TMR\_INVALID\_TYPE   'ptmr'  is not pointing to an RTOS\_TMR |
| \*                               RTOS\_ERR\_TMR\_INACTIVE       if the timer was not created |
| \*                               RTOS\_ERR\_TMR\_INVALID\_STATE  the timer is in an invalid state |
| \* |
| \* Returns    : RTOS\_TRUE       If the call was successful |
| \*              RTOS\_FALSE      If not |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*/ |
| BOOLEAN  RTOSTmrDel (RTOS\_TMR  \*ptmr, |
| INT8U   \*perr) |

|  |
| --- |
| /\* |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*                                             GET THE NAME OF A TIMER |
| \* |
| \* Description: This function is called to obtain the name of a timer. |
| \* |
| \* Arguments  : ptmr          Is a pointer to the timer to obtain the name for |
| \* |
| \*              pdest         Is a pointer to pointer to where the name of the timer will be placed. |
| \* |
| \*              perr          Is a pointer to an error code.  '\*perr' will contain one of the following: |
| \*                               RTOS\_ERR\_NONE               The call was successful |
| \*                               RTOS\_ERR\_TMR\_INVALID\_DEST   'pdest' is a NULL pointer |
| \*                               RTOS\_ERR\_TMR\_INVALID        'ptmr'  is a NULL pointer |
| \*                               RTOS\_ERR\_TMR\_INVALID\_TYPE   'ptmr'  is not pointing to an RTOS\_TMR |
| \*                               RTOS\_ERR\_TMR\_INACTIVE       'ptmr'  points to a timer that is not active |
| \*                               RTOS\_ERR\_TMR\_INVALID\_STATE  the timer is in an invalid state |
| \* |
| \* Returns    : The length of the string or 0 if the timer does not exist. |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*/ |
|  |
| INT8U  RTOSTmrNameGet (RTOS\_TMR   \*ptmr, |
| INT8U   \*\*pdest, |
| INT8U    \*perr) |

|  |
| --- |
| /\* |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*                                    GET HOW MUCH TIME IS LEFT BEFORE A TIMER EXPIRES |
| \* |
| \* Description: This function is called to get the number of ticks before a timer times out. |
| \* |
| \* Arguments  : ptmr          Is a pointer to the timer to obtain the remaining time from. |
| \* |
| \*              perr          Is a pointer to an error code.  '\*perr' will contain one of the following: |
| \*                               RTOS\_ERR\_NONE |
| \*                               RTOS\_ERR\_TMR\_INVALID        'ptmr' is a NULL pointer |
| \*                               RTOS\_ERR\_TMR\_INVALID\_TYPE   'ptmr'  is not pointing to an RTOS\_TMR |
| \*                               RTOS\_ERR\_TMR\_INACTIVE       'ptmr' points to a timer that is not active |
| \*                               RTOS\_ERR\_TMR\_INVALID\_STATE  the timer is in an invalid state |
| \* |
| \* Returns    : The time remaining for the timer to expire.  The time represents 'timer' increments.  In other words, if |
| \*              RTOSTmr\_Task() is signaled every 1/10 of a second then the returned value represents the number of 1/10 of |
| \*              a second remaining before the timer expires. |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*/ |
|  |
|  |
| INT32U  RTOSTmrRemainGet (RTOS\_TMR  \*ptmr, |
| INT8U   \*perr) |

|  |
| --- |
| /\* |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*                                    FIND OUT WHAT STATE A TIMER IS IN |
| \* |
| \* Description: This function is called to determine what state the timer is in: |
| \* |
| \*                  RTOS\_TMR\_STATE\_UNUSED     the timer has not been created |
| \*                  RTOS\_TMR\_STATE\_STOPPED    the timer has been created but has not been started or has been stopped |
| \*                  RTOS\_TMR\_COMPLETED        the timer is in ONE-SHOT mode and has completed it's timeout |
| \*                  RTOS\_TMR\_RUNNING          the timer is currently running |
| \* |
| \* Arguments  : ptmr          Is a pointer to the desired timer |
| \* |
| \*              perr          Is a pointer to an error code.  '\*perr' will contain one of the following: |
| \*                               RTOS\_ERR\_NONE |
| \*                               RTOS\_ERR\_TMR\_INVALID        'ptmr' is a NULL pointer |
| \*                               RTOS\_ERR\_TMR\_INVALID\_TYPE   'ptmr'  is not pointing to an RTOS\_TMR |
| \*                               RTOS\_ERR\_TMR\_INACTIVE       'ptmr' points to a timer that is not active |
| \*                               RTOS\_ERR\_TMR\_INVALID\_STATE  if the timer is not in a valid state |
| \* |
| \* Returns    : The current state of the timer (see description). |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*/ |
|  |
| INT8U  RTOSTmrStateGet (RTOS\_TMR  \*ptmr, |
| INT8U   \*perr) |

|  |
| --- |
| /\* |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*                                                   START A TIMER |
| \* |
| \* Description: This function is called by your application code to start a timer. |
| \* |
| \* Arguments  : ptmr          Is a pointer to an RTOS\_TMR |
| \* |
| \*              perr          Is a pointer to an error code.  '\*perr' will contain one of the following: |
| \*                               RTOS\_ERR\_NONE |
| \*                               RTOS\_ERR\_TMR\_INVALID |
| \*                               RTOS\_ERR\_TMR\_INVALID\_TYPE    'ptmr'  is not pointing to an RTOS\_TMR |
| \*                               RTOS\_ERR\_TMR\_INACTIVE        if the timer was not created |
| \*                               RTOS\_ERR\_TMR\_INVALID\_STATE   the timer is in an invalid state |
| \* |
| \* Returns    : RTOS\_TRUE    if the timer was started |
| \*              RTOS\_FALSE   if an error was detected |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*/ |
|  |
|  |
| BOOLEAN  RTOSTmrStart (RTOS\_TMR   \*ptmr, |
| INT8U    \*perr) |

|  |
| --- |
| /\* |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*                                                   STOP A TIMER |
| \* |
| \* Description: This function is called by your application code to stop a timer. |
| \* |
| \* Arguments  : ptmr          Is a pointer to the timer to stop. |
| \* |
| \*              opt           Allows you to specify an option to this functions which can be: |
| \* |
| \*                               RTOS\_TMR\_OPT\_NONE          Do nothing special but stop the timer |
| \*                               RTOS\_TMR\_OPT\_CALLBACK      Execute the callback function, pass it the callback argument |
| \*                                                        specified when the timer was created. |
| \*                               RTOS\_TMR\_OPT\_CALLBACK\_ARG  Execute the callback function, pass it the callback argument |
| \*                                                        specified in THIS function call |
| \* |
| \*              callback\_arg  Is a pointer to a 'new' callback argument that can be passed to the callback function |
| \*                               instead of the timer's callback argument.  In other words, use 'callback\_arg' passed in |
| \*                               THIS function INSTEAD of ptmr->RTOSTmrCallbackArg |
| \* |
| \*              perr          Is a pointer to an error code.  '\*perr' will contain one of the following: |
| \*                               RTOS\_ERR\_NONE |
| \*                               RTOS\_ERR\_TMR\_INVALID         'ptmr' is a NULL pointer |
| \*                               RTOS\_ERR\_TMR\_INVALID\_TYPE    'ptmr'  is not pointing to an RTOS\_TMR |
| \*                               RTOS\_ERR\_TMR\_INACTIVE        if the timer was not created |
| \*                               RTOS\_ERR\_TMR\_INVALID\_OPT     if you specified an invalid option for 'opt' |
| \*                               RTOS\_ERR\_TMR\_STOPPED         if the timer was already stopped |
| \*                               RTOS\_ERR\_TMR\_INVALID\_STATE   the timer is in an invalid state |
| \*                               RTOS\_ERR\_TMR\_NO\_CALLBACK     if the timer does not have a callback function defined |
| \* |
| \* Returns    : RTOS\_TRUE       If we stopped the timer (if the timer is already stopped, we also return RTOS\_TRUE) |
| \*              RTOS\_FALSE      If not |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*/ |
|  |
|  |
| BOOLEAN  RTOSTmrStop (RTOS\_TMR  \*ptmr, |
| INT8U    opt, |
| void    \*callback\_arg, |
| INT8U   \*perr) |

|  |
| --- |
| /\* |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*                                      SIGNAL THAT IT'S TIME TO UPDATE THE TIMERS |
| \* |
| \* Description: This function is typically called by the ISR that occurs at the timer tick rate and is used to signal to |
| \*              RTOSTmr\_Task() that it's time to update the timers. |
| \* |
| \* Arguments  : none |
| \* |
| \* Returns    : RTOS\_ERR\_NONE         The call was successful and the timer task was signaled. |
| \* |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
| \*/ |
|  |
|  |
| INT8U  RTOSTmrSignal (void) |

You need to demonstrate how you implement the above mentioned APIs. That means, when implementing the above APIs, there are other internal functions which are not publicly available to the users, but nonetheless, they are required for the timer management and API implementation.

1. The timer task , RTOSTmr\_Task(),
2. Any supporting functions you may need for initialization of free list RTOSTmr\_init() and start of the RTOSTmr\_Task(), The OS will call this timer initialization function as part of OSInit() during the start up.
3. Any supporting functions and all link list management required to support your design.

You may assume the following kernel functions available to you by the kernel. Use the Pthread library to simulate the RTOS functions as described below:

|  |
| --- |
| * 1. RTOSTaskCreate() creates a task. **You may use pthread library pthreat\_create().**   2. RTOSSemCreate(0u) initializes a semaphore to have the value of 0. Semaphores are used when a task wants exclusive access to a resource, needs to synchronize its activities with an ISR or a task, or is waiting until an event occurs. You would use a semaphore to signal the occurrence of an event to one or multiple tasks, and use mutexes to guard share resources. However, technically, semaphores allow for both. Example:   RTOS\_SEM RTOSTmrSignal;  RTOSTmrSemSignal      = RTOSSemCreate(0u);  **You can use pthread\_mutex\_init() to implement this. (If you use pthread\_cond\_wait() or pthread\_cond\_signal () to implement the RTOSSemPend and RTOSSemPost, you don’t need to implement this function and can be left blank (see below).**     * 1. RTOSSemPend (RTOS\_SEM \*timerSemPtr);   This is a semaphore Pend. If semaphore is taken it will wait for signal indicating time to update timers. Example:  RTOSSemPend (RTOSSemSignal , &err);  **You can use either pthread\_mutex\_lock() or alternatively pthread\_cond\_wait() to implement this**. |
| * 1. RTOSSemPost (RTOS\_SEM \*timerSemPtr);   This is the semaphore post operation. It will send a signal to the task that is waiting (pending) indicating that semaphore is released.Example:  RTOSSemPost (RTOSSemSignal);  **You can use either pthread\_mutex\_lock() or alternatively pthread\_cond\_signal () to implement this.**   * 1. RTOSShedLock()   This is used to prevent OS from context switching. In pthread library this can be left blank. You don’t need to prevent scheduling. |

* 1. RTOSShedUnlock()

This is used to resume the OS context switching which was previously halted by RTOSShedLock(). In pthread library this can be left blank. You don’t need to prevent scheduling.

To test your project, you need to write a program that prints the current time when timers are created. Then it creates 3 timers that register handles “function1”, “function2” and “function3”, respectively.

Timer 1 gets invoked every 5 seconds and runs function1 which prints “this is is function 1 and time is = <print current time>

Timer 2 gets invoked every 3 seconds and runs function1 which prints “this is is function 2 and time is = <print current time>

Timer 3 gets invoked only once 10 seconds after it was created and runs function3 which prints “this is is function 3 and time is = <print current time>