36 Timer Class and Module (timer.hhf)

The HLA Timer module provides a set of routines that let you time events with millisecond precision.

Note: Like documentation for most standard library modules that are based on an HLA class, this document does not provide examples of low-level calls to the timer functions. If you're interested in making low-level machine instruction calls to the methods in the timer class, please consult the HLA documentation concerning classes and objects.

A Note About Thread Safety: The timer module maintains various values within each object. If you attempt to manipulate the same object from different threads in a multi-threaded application, you may get inconsistent results. Therefore, you should only call the procedures and methods for a particular timer object from one thread or you must explicitly control access to those methods to prevent concurrent execution of the same object's methods from different threads. Note that you may call the methods for *different* timer objects from different threads.

36.1 Timer Module

To use the timer functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "timer.hhf" )
or
#include( "stdlib.hhf" )
```

36.2 Timer Class/Data Structure

The Timer module is actually a class with the following definition:

```
timer_t:  class

            var
                Accumulated:    qword;

                DateStarted:    date.daterec;
                TimeStarted:    time.timerec;
                msStarted:      uns32;

                DateStopped:    date.daterec;
                TimeStopped:    time.timerec;
                msStopped:      uns32;

                Running:     boolean;
                Valid:       boolean;

            procedure create;   external;

            method start;       external;
            method restart;     external;
            method stop; @returns( "edx:eax" );  external;
            method checkPoint; @returns( "edx:eax" ); external;

        endclass;
```

Don't forget that the *timer_t* class, like all class objects, will modify the values of the ESI and EDI registers whenever you call a class procedure or method. So don't expect values in ESI or EDI to be preserved across the calls in this module.

36.3 Timer Operation

The timer_t class maintains an accumulation of time. When you create a class object, or when you call the *timer_t.start* method, the system initializes this 64-bit unsigned integer value to zero. When you call the timer_t.start method, the system notes the point at which you called the method so it can compute the amount of accumulated time when you call *timer_t.stop* or *timer_t.checkPoint* at some point in the future. It is important that you realize that the class' *timer_t.Accumulated* field does *not* contain a real-time representation of the elapsed time. When you call *timer_t.stop*, the object will compute the amount of elapsed time since the call to *timer_t.start* and will update *timer_t.Accumulated* with this value. So to time a simple sequence of events, you would first call *timer_t.start*, do whatever it is that you want to time, and then call *timer_t.stop* when you're finished with the events you want to time. On return from timer_t.stop, the EDX:EAX register pair will contain the 64-bit elapsed time value, or you can retrieve the value from the object's *timer_t.Accumulated* field.

If you would like to compute the current elapsed time during some timing sequence, but you do not want to stop the timing operation, you can call the *timer_t.checkPoint* method. This method will update the *timer_t.Accumulated* field with the elapsed time up to that point without stopping the timer operation. The *timer_t.checkPoint* function call will also return the total accumulated time in the EDX:EAX register pair. The timer will continue running until you call the *timer_t.stop* method at some point in the future. Note that you may call *timer_t.checkPoint* as many times as you like between the *timer_t.start* and *timer_t.stop* method calls. Note, however, that you may only call *timer_t.checkPoint* while the timer is actually running.

For more complex timing applications, it is possible to start, stop, and restart the timer without resetting the accumulated value to zero. Restarting the timer after calling *timer_t.stop* is possible by calling the *timer_t.restart* method. The *timer_t.restart* method is functionally equivalent to *timer_t.start* except that it doesn't zero out the *timer_t.Accumulated* field. When you call *timer_t.stop* after a *timer_t.restart* method invocation, the *timer_t.accumulated* field is updated with the sum of its previous value plus the measured time between the *timer_t.restart* and *timer_t.stop* calls. Of course, you can make multiple calls to the *timer_t.restart/timer_t.stop* methods to accumulate time over longer periods.

36.4 Timer Class Fields

timer_t.Accumulated

This field contains the computed time in milliseconds. This field is only valid if the *timer_t.Valid* field contains true. If *timer_t.Running* contains true, then the timer is still running and the *timer_t.Accumulated* field contains the number of milliseconds at the last *timer_t.checkPoint* or *timer_t.restart* operation.

timer_t.DateStarted
timer_t.TimeStarted
timer_t.msStarted
timer_t.DateStopped
timer_t.TimeStopped
timer_t.msStopped

These are internal variables to the class. You should not modify their values nor should you read their values and use them for anything.

timer_t.Running

This boolean variable indicates that the timer object is currently timing some event. You may read this variable but you should not modify its value.

36.5 Timer Procedures and Methods

```
procedure timer_t.create; @returns( "esi" );
```

This is the constructor for the class. If you call it via "*someObjectName*.create();" then this static class procedure will initialize the fields of the specified object. If you call it via "timer_t.create();" then *timer_t.create* will dynamically allocate storage for a timer_t object and initialize that storage. This call will return a pointer to the new object in the ESI register.

```
HLA high-level calling sequence examples:

  timer_t.create();
  mov( esi, timerPtrVar );

  timerClassVar.create();
```

method timer_t.start;

This method will initialize the timer so it can begin timing some sequence. Note that this call will set the *timer_t.Running* field to true and the *timer_t.Valid* field to false. Use the *timer_t.stop* method call to stop the timing operation. This call will also initialize the *timer_t.Accumulated* field to zero. Calling this method on a timer that is already running will reset the accumulated time to zero. See *timer_t.restart* if you want to start the timer running without clearing the *timer_t.Accumulated* field.

```
HLA high-level calling sequence examples:

  timer_t.create();
  mov( esi, timerPtrVar );
      .
      .
      .
  timerPtrVar.start();
      .
      .
      .
  timerPtrVar.stop();
  mov( edx:eax, qwordTimerValue );// Accumulated time
```

method timer_t.stop; @returns("edx:eax");

This method will stop the timer accumulation and returns the accumulated time in EDX:EAX. This call sets *timer_t.Valid* to true and *timer_t.Running* to false.

method timer_t.restart;

This method restarts the timer after you've stopped the timing via *timer_t.stop*. Note that the result accumulated will be the sum of the previous accumulation plus the new time.

method timer_t.checkPoint;

This computes the current time in *timer_t.Accumulated* without stopping the timer. That is, *timer_t.Valid* will be set to true and *timer_t.Running* will remain true.