TAsyncTimer Component

Properties Methods Events Exceptions

TAsyncTimer is an asynchronous timer component for Delphi. Unlike Delphi's standard timer component, it uses an additional thread to implement timer functionality. Timer event taker (method assigned to OnTimer event) is executed in its own thread asynchronously to VCL's one. *TAsyncTimer* is not a real-time timer, an actual interval between timer events is *RequestedInterval* + *C1* + *C2*, where *RequestedInterval* is value assigned to Interval property, *C1, C2* -- variables representing Windows' mood. Normally C1 + C2 is 1-5 millisecond(s), but the value grows when number of active tasks increases. Anyway, tests show that *TAsyncTimer* is more stable than timers implemented using windows' timer services, and can handle smaller intervals (you can use test program included to asynctim.zip to analyze timer's behavior).

TAsyncTimer creates two threads: one is used internally, and never leaves timer. Timer event taker is executed in the second one. Use TimerThreadPriority and TakerThreadPriority to control thread priorities (defaults are *tpTimeCritical* for timer's thread, and *tpHigher* for taker's one). If timer event occurs when the taker is busy handling previous event, then timer ignores it and calls any method attached to OnTimingFault event to signal error. The Enabled property can be used to enable/disable timer.

Because *TAsyncTimer* events come asynchronously to VCL, you should be more careful writing timer event taker (and do not forget to make sure that the timer is disabled before taker becomes invalid).

Source

TAsyncTimer Properties

TAsyncTimer Methods

Timer
TimingFault

TAsyncTimer Events

OnTimer
OnTimingFault

TAsyncTimer Exceptions

EAsyncTimerError

TAsyncTimer.Enabled

property Enabled :Boolean; (R/W)

TAsyncTimer.Interval

property Interval :Longint; (R/w)

Interval between timer events in milliseconds

TAsyncTimer.Timer

protected procedure Timer; dynamic;

calls any method attached to OnTimer event.

TAsyncTimer.TimingFault

protected procedure TimingFault; dynamic;

calls any method assigned to OnTimingFault event.

TAsyncTimer.OnTimer

property OnTimer: TNotifyEvent;

Description

The OnTimer event is used to execute code at regular intervals. Place the code you want to execute within the OnTimer event handler.

The Interval property of a timer component determines how frequently the OnTimer event occurs. Each time the specified interval passes, the OnTimer event occurs.

TAsyncTimer.OnTimingFault

property OnTimingFault :TNotifyEvent;

OnTimingFault handler is called if timer event occurs when event taker is busy handling previous event

EAsyncTimerError

EAsyncTimerError = class(Exception);

TAsyncTimer raises *EAsyncTimerError* if any error occurred while initializing timer.

TAsyncTimer Source

```
(*

  TAsyncTimer Component for Delphi 2.0
  by Glen Why

  No rights reserved

  File version 1.00.00

  Version history

  1.00.00 - first one

*)

unit AsyncTimer;

interface

uses
  Windows, Classes, SysUtils;

const

AsyncTimer_DefTimerThreadPriority = tpTimeCritical;
AsyncTimer_DefTakerThreadPriority = tpHigher;
AsyncTimer_DefInterval = 100;
AsyncTimer_DefEnabled = false;

type

  EAsyncTimerError = class( Exception );

  TAsyncTimer = class(TComponent)
  private
    FTimerThreadPriority :TThreadPriority;
    FTakerThreadPriority :TThreadPriority;
    FOnTimer :TNotifyEvent;
    FOnTimingFault :TNotifyEvent;
    FInterval :Longint;
    FTimerThread :THandle;
    FTimerThreadID :THandle;
    FTakerThread :THandle;
    FTakerThreadID :THandle;
    FEnabled :Boolean;
    FTakerActive :Boolean;
    FFinished :Boolean;
    procedure InitTimerThread;
    procedure DoneTimerThread;
    procedure SetTimerThreadPriority( NewPriority :TThreadPriority );
    procedure SetTakerThreadPriority( NewPriority :TThreadPriority );
    procedure SetEnabled( NewState :Boolean );
```

```
    procedure UpdateTimerThreadPriority;
    procedure UpdateTakerThreadPriority;
    procedure InitTakerThread;
    procedure DoneTakerThread;
  protected
    procedure Timer; dynamic;
    procedure TimingFault; dynamic;
    procedure Loaded; override;
  public
    constructor Create( AnOwner :TComponent ); override;
    destructor Destroy; override;
  published
    property Enabled :Boolean
      read FEnabled write SetEnabled
      default AsyncTimer_DefEnabled;
    property Interval :Longint
      read FInterval write FInterval
      default AsyncTimer_DefInterval;
    property OnTimer :TNotifyEvent
      read FOnTimer write FOnTimer;
    property OnTimingFault :TNotifyEvent
      read FOnTimingFault write FOnTimingFault;
    property TimerThreadPriority :TThreadPriority
      read FTimerThreadPriority write SetTimerThreadPriority
      default AsyncTimer_DefTimerThreadPriority;
    property TakerThreadPriority :TThreadPriority
      read FTakerThreadPriority write SetTakerThreadPriority
      default AsyncTimer_DefTakerThreadPriority;
  end;

implementation

const TimerThreadStackSize = $1000;

procedure TakerThreadProc( Timer :TAsyncTimer ); stdcall;
begin
  while not Timer.FFinished do
   begin
     Timer.FTakerActive := true;
     Timer.Timer;
     Timer.FTakerActive := false;
     SuspendThread( Timer.FTakerThread );
   end;
end;


procedure TimerThreadProc( Timer :TAsyncTimer ); stdcall;
begin
  while Timer.FInterval > 0 do
   begin
     if Timer.FTakerThread <> 0 then
       if Timer.FTakerActive then Timer.TimingFault
        else ResumeThread( Timer.FTakerThread );
     sleep( Timer.FInterval );
   end;
end;
```

```
{ TAsyncTimer }

constructor TAsyncTimer.Create( AnOwner :TComponent );
begin
 inherited Create( AnOwner );
 FInterval := AsyncTimer_DefInterval;
 FTimerThreadPriority := AsyncTimer_DefTimerThreadPriority;
 FTakerThreadPriority := AsyncTimer_DefTakerThreadPriority;
 FOnTimer := Nil;
 FOnTimingFault := Nil;
 FTimerThread := 0;
 FTakerThread := 0;
 FTakerActive := false;
 FFinished := false;
 FEnabled := AsyncTimer_DefEnabled;
end;

destructor TAsyncTimer.Destroy;
begin
  DoneTimerThread;
  DoneTakerThread;
  inherited Destroy;
end;

procedure TAsyncTimer.Loaded;
begin
 inherited Loaded;
 InitTakerThread;
 InitTimerThread;
end;

procedure TAsyncTimer.SetTimerThreadPriority(
 NewPriority :TThreadPriority );
begin
 if ( NewPriority <> FTimerThreadPriority ) then
   begin
    FTimerThreadPriority := NewPriority;
    UpdateTimerThreadPriority;
   end;
end;

procedure TAsyncTimer.SetTakerThreadPriority(
 NewPriority :TThreadPriority );
begin
 if ( NewPriority <> FTakerThreadPriority ) then
   begin
    FTakerThreadPriority := NewPriority;
    UpdateTakerThreadPriority;
   end;
end;

procedure TAsyncTimer.SetEnabled( NewState :Boolean );
begin
 if ( FEnabled xor NewState ) then
   begin
    if ( ( [ csDesigning, csReading ] - ComponentState ) <> [] ) then
     if NewState
```

```
            then ResumeThread( FTimerThread )
            else SuspendThread( FTimerThread );
          FEnabled := NewState;
         end;
      end;


      procedure TAsyncTimer.InitTimerThread;
      var CreationFlags :Longint;
      begin
        if not ( csDesigning in ComponentState ) then { create thread at run-
time only }
          begin
           CreationFlags := 0;
           if not FEnabled then CreationFlags := CREATE_SUSPENDED;
           FTimerThread := CreateThread( Nil, TimerThreadStackSize,
            @TimerThreadProc, Self, CreationFlags, FTimerThreadID );
           if ( FTimerThread = 0 ) then
            raise EAsyncTimerError.Create( 'Thread creation error' );
           UpdateTimerThreadPriority;
          end;
      end;

      procedure TAsyncTimer.DoneTimerThread;
      begin
       if ( FTimerThread <> 0 ) then
         begin
           FInterval := -1;
           ResumeThread( FTimerThread );
           WaitForSingleObject( FTimerThread, INFINITE );
           FTimerThread := 0;
         end;
      end;

      const

        Priorities: array [TThreadPriority] of Integer =
         (THREAD_PRIORITY_IDLE, THREAD_PRIORITY_LOWEST,
THREAD_PRIORITY_BELOW_NORMAL,
          THREAD_PRIORITY_NORMAL, THREAD_PRIORITY_ABOVE_NORMAL,
          THREAD_PRIORITY_HIGHEST, THREAD_PRIORITY_TIME_CRITICAL);


      procedure TAsyncTimer.UpdateTimerThreadPriority;
      begin
        SetThreadPriority( FTimerThread, Priorities[ FTimerThreadPriority ] );
      end;

      procedure TAsyncTimer.UpdateTakerThreadPriority;
      begin
        SetThreadPriority( FTakerThread, Priorities[ FTakerThreadPriority ] );
      end;

      procedure TAsyncTimer.Timer;
      begin
       if assigned( FOnTimer ) then FOnTimer( Self );
      end;
```

```
      procedure TAsyncTimer.InitTakerThread;
      begin
       if not ( csDesigning in ComponentState ) then { create thread at run-
time only }
         begin
          FTakerActive := false;
          FTakerThread := CreateThread( Nil, 0, @TakerThreadProc,
           Self, CREATE_SUSPENDED, FTakerThreadID );
          if ( FTakerThread = 0 ) then
            raise EAsyncTimerError.Create( 'Timer event taker thread creation
error' );
          UpdateTakerThreadPriority;
        end;
      end;

      procedure TAsyncTimer.DoneTakerThread;
      begin
       if ( FTakerThread <> 0 ) then
         begin
           FFinished := true;
           ResumeThread( FTakerThread );
           WaitForSingleObject( FTakerThread, INFINITE );
           FTakerThread := 0;
         end;
      end;

      procedure TAsyncTimer.TimingFault;
      begin
       if assigned( FOnTimingFault ) then FOnTimingFault( Self );
      end;


      end.
```

TAsyncTimer.TimerThreadPriority

property TimerThreadPriority :TThreadPriority;

TAsyncTimer.TakerThreadPriority

property TakerThreadPriority :TThreadPriority;