

## Simple timer for games

This article presents a simple timer class suitable for games. This class uses CIdle to generate the timing events.

The CIdle class is an [active object](#) that runs whenever there are no other active objects with higher priority ready to run. Different from other approaches found in the Symbian OS API, CIdle does not use a predefined time interval to generate events.

### Defining the class

Here is the class declaration:

```
class CCustomTimer : public CBase
{
public:

    // Creates an instance of this class. MTimerListener
    // is an object interested in handling the timer event.
    static CCustomTimer* NewL (MTimerListener & aListener);

    ~CCustomTimer ();

    // Starts the timer.
    void Start ();

    // Cancels the timer.
    void Cancel ();

private:

    // Callback used internally to notify the listener.
    // Required by CIdle.
    static TInt IdleCallBack (TAny* aPtr);

private:

    // Constructor, stores the reference to the listener
    // object.
    CCustomTimer (MTimerListener & aListener)
    : iListener (& aListener)
    {}

    // Second part of the two-phase constructor.
    void ConstructL ();

private:

    MTimerListener * iListener; // The object that handles the timer event.
    CIdle* iIdle;
};
```

The MTimerListener interface is defined as follows:

```
class MTimerListener
```

```

{
    public:

        virtual void OnTimer () = 0;
};

```

The CCustomTimer class implementation is defined this way:

```

void CCustomTimer::ConstructL ()
{
    iIdle = CIdle::NewL (CIdle::EPriorityIdle);
}

CCustomTimer::~CCustomTimer ()
{
    if (iIdle)
    {
        iIdle->Cancel ();
        delete iIdle;
    }
}

void CCustomTimer::Start ()
{
    if (!iIdle->IsActive () )
        iIdle->Start (TCallback (IdleCallback, this) );
}

void CCustomTimer::Cancel ()
{
    iIdle->Cancel();
}

TInt CCustomTimer::IdleCallback (TAny* aPtr)
{
    CCustomTimer* me = ((CCustomTimer*)aPtr);

    me->iListener->OnTimer ();
    return ETrue;
}

```

## Usage

```

// Suppose we plan the game to run at 30 frames per second. Then, we
// need to calculate the required time step to meet this requisite.
// (time in microseconds)
const TInt KTargetTimeStep = 1000000 / 30;

void MyGame::OnTimer ()
{
    // Handle input.
    ProcessInput ();
}

```

```
// Now we are going to calculate how many steps the game should process,  
// so it is not late.  
  
// Keep current time.  
TTime time;  
time.HomeTime();  
  
// Calculate elapsed time since the last update.  
// The iStartTime variable is a member of the MyGame class.  
TInt elapsed = I64LOW (time.MicroSecondsFrom (iStartTime).Int64());  
  
// Update total game time. The iTotatTime variable  
// is also a member of the MyGame class.  
iTotalTime += elapsed;  
  
// Calculate how many updates should be processed. Notice that the  
// division is an integer division (it means  $5 / 2 = 2$ ).  
elapsed = iTotatTime / KTargetTimeStep;  
  
for (TInt i = 0; i < elapsed; ++i)  
    Update ();  
  
// Subtract the time steps processed.  
iTotalTime -= elapsed * KTargetTimeStep;  
iStartTime = time.Int64();  
  
// Draw.  
Render ();  
}
```