Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Your Own MP3 Duration Calculator

January 26, 2010
URL:http://www.drdobbs.com/embedded-systems/your-own-mp3-duration-calculator/222500141

Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems. He is currently trying to build intelligent machines inspired by the brain at Numenta (www.numenta.com).

My wife recently completed certification as a spinning instructor -- you know, riding real hard on a stationary bike -- and began teaching classes. It turns out that music is very important for spinning, and instructors have to prepare CDs with proper tracks for warm-up, main work-out, and cool down. Each part is measured and you have to find the right tracks with the proper combined length. Of course, you could do this manually by looking at the duration of each track and try to add them together, but this approach is tedious and error-prone. A more serious problem is that the duration of MP3 files is recorded as an ID3 tag in the file header. There is no guarantee that this number is correct (and often it is not). This situation is simply screaming for a software solution.

The solution I present in this article -- the "MP3 Duration Calculator" -- lets you select MP3 files from a directory and calculate the total duration of the selected files on-the-fly. The complete source code and related files are available here. To calculate the duration of each track, the software actually opens the file in a media player and gets an accurate duration number. Figure 1 shows the folder selection dialog where you choose the folder that contains the MP3 files.

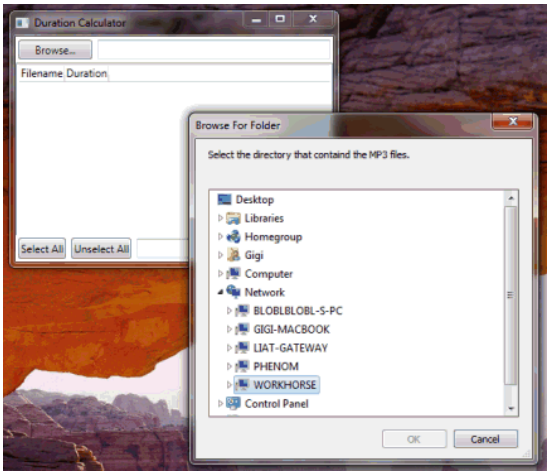

Figure 1

Figure 2 shows the main window of the MP3 Duration Converter.

Figure 2

The program is a WPF application implemented in C# over the .NET 3.5 framework. It demonstrates a few interesting aspects such as:

- Dynamically loading MP3 files
- Working with the **MediaElement** component
- Asynchronous execution of long operations while keeping the UI responsive
- On-the-fly resizing of GridView columns (more obscure then it should be)
- Data binding and custom conversion
- Application state persistence

An Introduction to WPF

WPF (Windows Presentation Foundation) is the user interface/display system of Windows. It is built on top of DirectX and can take advantage of hardware acceleration. It is more advanced than previous UI APIs (including the .NET Windows Forms) all based on GDI/GDI+ and the User32 API. WPF offers a declarative UI (XAML), animation, multi-media support, rich text model, web-like layout model, browser-like page-based navigation, rich drawing model (including 3D) based on primitives and not pixels, totally customizable look-and-feel for controls (via templates and styling), and more.

Silverlight, Microsoft's rich web client platform, is based on the same design and is almost compatible, which really helps if you want to multi-target your application to the desktop and the browser. At this writing, WPF is not supported by Mono (and probably won't in the near future), so if you need to target multiple platforms your best bet is still the Windows Forms API.

XAML

XAML, short for "eXtensible Application Markup Language," is an XML dialect for instantiating .NET objects and hooking them up. It can be used for many purposes, but currently its primary use is to define WPF user interfaces. It provides the following capabilities:

- Full namespace support
- Resource definition
- Animation definition
- Control template definition
- Hooking up event handlers
- Data binding
- Design time integration with Visual Studio 2008

This is a rich feature set for a UI description markup language. It should be noted that XAML is not required and you can code any WPF construct programmatically. However, many things are easier with XAML, especially if you use Visual Studio as your IDE.

MP3 Duration Calculator Design

The design is straightforward. There are four classes:

- **App** (the application class)
- **TrackDurations** (engine)
- **DurationConverter** (a helper class)
- **MainWindow** (GUI).

The **App** and **MainWindow** classes are a code-behind classes for the corresponding XAML files: App.xaml and MainWindow.xaml.

The App Class

The **App** class is generated by Visual Studio if you use it (I do). The code is split across two files: One of them is called App.g.cs and you can find it in the Debug/obj folder, but don't modify it. The other file is called App.xaml.cs and you can find it in the project folder. In this file you can add application-level event handlers like **OnStartup()** and **OnExit()**. In this case it is pretty trivial:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Windows;

namespace MP3DurationCalculator
{
  /// <summary>
  /// Interaction logic for App.xaml
  /// </summary>
  public partial class App : Application
  {
  }
}
```

The two files are merged to a single class by the virtue of the partial class feature of C#. In addition to the code files there is also the application XAML file called App.xaml. This file contains a couple of namespaces, identifies the application class (**MP3DurationCalculator.App** and the URI of the main windows's XAML file. WPF takes it from here and knows how to instantiate the App and display the main window.

```
<Application x:Class="MP3DurationCalculator.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

The TrackDurations Class

The **TrackDurations** class is the workhorse that does the heavy lifting of accurately measuring the duration of each track. It utilizes the useful MediaElement class, which wraps the Windows Media Player engine (At least the Microsoft Windows Media Player 10 OCX for media playback).

Before diving into the **TrackDurations** class, let me introduce the **TrackInfo** class. This is basically a pair of properties: **Name** (a string) and **Duration**. A collection of **TrackInfo** object is the main data structure that travels from the **TrackDurations** class to the **MainWindow** class. The fact that **Name** and **Duration** are properties allows the UI to bind to them directly. More about it later:

```
class TrackInfo
{
  public string Name { get; set; }
  public TimeSpan Duration { get; set; }
}
```

The **TrackDurations** class uses a delegate to notify the UI whenever it discovers the duration of another track. After processing all the tracks it sends a null object to let the UI know it's done. Here is the signature:

```
delegate void TrackDurationsDelegate(TrackInfo ti);
```

A delegate is really a callback function. In this case a **TrackInfo** object is sent with each call.

TrackDurations implements the **IDisposable** interface, which allows explicitly disposing of resources by calling its **Dispose()** method:

```
class TrackDurations : IDisposable
{
  ...
}
```

The class keeps references to a media element and the delegate and stores a list of files in a stack:

```
  MediaElement _me;
  TrackDurationsDelegate _delegate;
  Stack<string> _files;
```

The media element is not managed by **TrackDurations** (it doesn't create or own it). It is passed in the constructor. The reason for this design is that a **TrackDurations** object is instantiated every time a new folder needs to be scanned, but a single media element can be used because it is relatively expansive to instantiate one. Also, there is never more than one scan going on at the same time, so there is no need to worry about conflicts.

The constructor accepts a media element, a delegate and a list of file names. It stores them (the list of files in a stack that can be popped). It sets the **LoadedBehavior** of the media element to "Manual", which means programmatic control on play/pause/stop, attaches two event handlers for **MediaOpened** and **MediaFailed**, sets the **Source** property to the first file in the list, and tells the media element to play it:

```
  public TrackDurations(
    MediaElement me,
    IEnumerable<string> files,
    TrackDurationsDelegate d)
  {
    Debug.Assert(me != null);
    Debug.Assert(d != null);
    _delegate = d;
    _files = new Stack<string>(files);
    Debug.Assert(_files.Count >  0);
    _me = me;
    _me.LoadedBehavior = MediaState.Manual;
    _me.MediaOpened += _onMediaOpened;
    _me.MediaFailed += _onMediaFailed;
    _me.Source = new System.Uri(_files.Peek());
    _me.Play();
  }
```

The result of all these operations is that the media element will open the first file in order to play it. If the open operation succeeds the **_onMediaOpened()** event handler will be called; otherwise **_onMediaFailed()** will be called.

The **Dispose()** method simply detaches the two event handlers:

```
  public void Dispose()
  {
    _me.MediaOpened -= _onMediaOpened;
    _me.MediaFailed -= _onMediaFailed;
  }
```

The **_getNextTrack()** method is used by both event handlers to move on after the current track has been handled. It reads the next file from the stack (processed files are removed by the event handlers) and tries to play it. If there are no more files the delegate is called with a null object to signal end of processing:

```
  void _getNextTrack()
  {
    if (_files.Count == 0)
    {
      _delegate(null);
    }
    else
    {
      // Get the next file
      _me.Source = new System.Uri(_files.Peek());
      _me.Play();
    }
  }
```

When the media element opens successfully the current file as a result of a **Play()** call the **MediaOpened** event is fired. That causes the **_onMediaOpened()** event handler to be called. At this point the correct duration is available as the **NaturalDuration** property of the media element. The event handler pauses the media element to avoid accidental playing, creates a new **TrackInfo** object, stops the media element completely, invokes the delegate and finally gets the next track.

```
private void _onMediaOpened(object sender, RoutedEventArgs e)
{
  _me.Pause();
  Debug.Assert(_me.NaturalDuration.HasTimeSpan);
  TimeSpan duration = _me.NaturalDuration.TimeSpan;
  var ti = new TrackInfo
  {
    Name = System.IO.Path.GetFileName(_files.Pop()),
    Duration = duration
  };

  _me.Stop();
  _delegate(ti);

  _getNextTrack();
}
```

If for some reason, the media element failed to open the track the **MediaFailed** event is fired and the **_onMediaFailed()** event handler is called. All it does, is pop the bad file from the files list and get the next track:

```
private void _onMediaFailed(object sender, RoutedEventArgs e)
{
  // Get rid of the bad file
  _files.Pop();
  _getNextTrack();
}
```

The MainWindow Class

The **MainWindow** class is responsible for displaying the main window and also functions as the controller. In a larger system I would probably split it into multiple files and maybe use some application framework, but in such a small utility I felt okay with just managing everything in a single class. The UI is defined declaratively using XAML in MainWindow.xaml and the controller code is in the code-behind class MainWindow.xaml.cs.

Let's start with the UI. The root element is naturally the **<Window>**. It contains the usual XML namespaces for XAML and the local namespace for the MP3DurationCalculator. It also contains the title of the window and its initial dimensions. There is also a **<Window.Resources>** sub-element that contains resources that can be shared by all elements in the window. In this case it's the **DurationConverter** class. I'll talk more about it later. Here is the XAML for **<Window>** element:

```
<Window x:Class="MP3DurationCalculator.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:MP3DurationCalculator"
  Title="Duration Calculator" Height="293" Width="376">
  <Window.Resources>
    <local:DurationConverter x:Key="DurationConverter"></local:DurationConverter>
  </Window.Resources>
  ...
  </Window>
```

The **Window** element corresponds to the WPF **Window** class, which is a content container. That means it can hold just one item. That sounds pretty limited, but in practice this single item can be a layout container that itself can hold many items. This is how WPF layout works. You nest containers within containers to get the exact desired layout. The most common and flexible layout container is the grid. WPF Layout is a big topic that warrants its own article (or more). In this article, I try to get away with the minimal amount of explanations necessary to understand the layout of the MP3 Duration Converter. The grid element contains two elements: a media element and a dock panel. The media element is not displayed actually because we use it only to play audio. The dock panel contains the rest of the UI. Here is a collapsed view of the grid:

```
  <Grid>
    <MediaElement
        Height="0"
        Width="0"
        Name="mediaElement"
        LoadedBehavior="Manual" />
    <DockPanel LastChildFill="True">
      ...
    </DockPanel>
  </Grid>
```

The dock panel is another layout container, which contains three stripes arranged vertically. The top stripe contains a browse button for selecting a folder and a text box to display the selected folder name (see Figure 3).

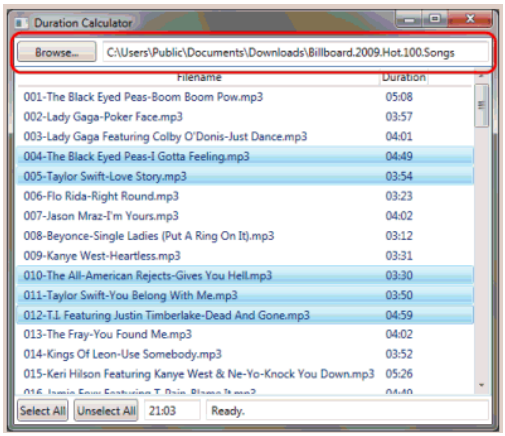

Figure 3

The middle stripe contains a list view that displays the MP3 files in the selected folder (see Figure 4).

Figure 4

The bottom stripe contains a couple of buttons for selecting all or none of the files and a text box to display the total duration of the selected songs (see Figure 5).

Figure 5

The top stripe is a dock panel that has several attributes. The **DockPanel.Dock** attribute is actually an attached attribute and it applies to outer dock panel (the one that contains all three stripes). It has the value **Top**, which means that the top strip will be docked to the top part of outer dock panel. The height is defined to be 30 and the vertical alignment is set to **Stretch**, which means that the top stripe will stretch to fit the entire width of the outer dock panel. The **LastChildFill** attribute is set to **True**, which means that that last child laid out will fill the remaining area. The top stripe contains two controls a button and a text box. Both has attached **DockPanel.Dock** properties that apply to the top stripe itself. The button is docked to the left and the text box is docked to the right. In addition, since the top stripe has a **LastChildFill ="True"** the text box will stretch to fill the remaining area of the top stripe after the button is laid out. The margin attribute of the button and text box ensures that there will be some spacing between them, so they are not squished together. The button has an event handler called **Click** that calls the **btnSelectDir_Click** method when the button is clicked. This method is defined in the code-behind class. Here is XAML for the top stripe:

```
<DockPanel
      DockPanel.Dock="Top"
```

```
        LastChildFill="True"
        Height="30"
        Margin="0" VerticalAlignment="Stretch"
  >
        <Button DockPanel.Dock="Left"  Height="22" Name="btnSelectDir" Width="84" Margin="3" Click="btnSelectDir_Click">Browse...</Button>
        <TextBox Height="22" Name="tbTargetFolder" MinWidth="258" Margin="3" TextChanged="tbTargetFolder_TextChanged"></TextBox>
  </DockPanel>
```

The second stripe is actually the bottom stripe and not the middle stripe. The reason for this supposed inconsistency is that I want the middle stripe to fill the remaining area of the outer stripe after the top and bottom have been laid out (according to the **LastChildFill** attribute), so it must be laid out last. I actually find it a little disorienting. I would prefer if you could simply set one of the **DockPanel.Dock** of one of the children to **Fill** instead of making sure it appears last in the XAML file. Anyway, that's the the choice the WPF designers made, so the second stripe is the bottom stripe. It is very similar to the top stripe. It contains two buttons for selecting all files or unseelcting all files and two text boxes. The **total** text box contains the total duration of all the selected files and the **status** text box shows various status messages. Named elements like **total** and **status** can be accessed using their name in the code-behind class. The text of the "total" text box is data bound to the **total.Text** property of the **MainWindow** object. More on that later. Here is the XAML for the bottom stripe:

```
<DockPanel
        DockPanel.Dock="Bottom"
        LastChildFill="True"
        Height="30"
        Margin="0" VerticalAlignment="Stretch"
  >
        <Button DockPanel.Dock="Left" Margin="3" Click="SelectAll_Click">Select All</Button>
        <Button DockPanel.Dock="Left" Margin="3" Click="UnselectAll_Click">Unselect All</Button>
        <TextBox DockPanel.Dock="Left" Height="22" Name="total" Width="60" Margin="3"
        Text="{Binding Path=Self, Converter={StaticResource DurationConverter}}"></TextBox>
        <TextBox Height="22" Name="status" MinWidth="100" Margin="3" />
  </DockPanel>
```

The middle stripe is a list view. It has no **DockPanel.Dock** attached property because it is the last child and thus just fills the area left between the top stripe and the bottom stripe. When the Window is resized the top and bottom stripe keep their fixed height and the list view is resize to accommodate the new height of the window. The list view has a name ("files") because it is accessed programmatically from the **MainWindow** class. It also has an event handler for the **SelectionChanged** event. The list view also has a view property, which is a grid view in this case (the list view supports several view types including custom views). The grid view has two columns, which are bound to the **Name** and **Duration** properties of its item object. The items that populate the list must be object that have a **Name** and **Duration** properties. The **Duration** property is converted using the **DurationConverter** to a more display-friendly format. Here is the XAML for the list view:

```
  <ListView MinHeight="223" Name="files" MinWidth="355" Background="White" SelectionChanged="files_SelectionChanged">
        <ListView.View>
          <GridView>
                <GridView.Columns>
                  <GridViewColumn
                        Header="Filename"
                        DisplayMemberBinding="{Binding Path=Name}"
                        Width="Auto"
                  />
                  <GridViewColumn
                        Header="Duration"
                                DisplayMemberBinding="{Binding Path=Duration, Converter={StaticResource DurationConverter}}"
                        Width="Auto"
                  />
                </GridView.Columns>
          </GridView>
        </ListView.View>
  </ListView>
```

Let's move on to the **MainWindow** code-behind class. This class oversees the following actions: browse for a new folder that contains MP3 files, collect track info about every MP3 file, select files from the current folder, calculate and display the total duration of all the selected files.

The state the class works with the following member variables:

_folderBrowserDialog . This is a **System.Windows.Forms** component used to display a dialog for browsing the file system and selecting a folder. WPF doesn't have its own component, but using the **Windows.Forms** component is just as easy.

_trackDurations. This is an instance of our very own **TrackDurations** class described earlier

_results. This is an observable collection of **TrackInfo** objects received from the TrackDurations class whenever the selection of MP3 files is changed.

_fileCount. Just an integer that says how many files are in the current selected directory.

_maxLength. Just an integer that measures the length of the longest track name. It is used to properly resize the list view columns to make sure track names are not truncated.

```
FolderBrowserDialog _folderBrowserDialog;
TrackDurations _trackDurations;
ObservableCollection<TrackInfo> _results;
int _fileCount = 0;
double _maxLength = 0;
```

In addition to these member variables defined in code the **MainWindow** class also has additional member variables, which are all the named elements in the XAMNL file like the **files** list view, the **total** and **status** text boxes.

The constructor calls the mandatory **InitializeComponent()** method that reads the XAML and builds all the UI and then instantiates the folder browser dialog and initializes its properties so it starts browsing from the **Downloads** directory (folder) under the current user's desktop directory. It also instantiates **_results** to an empty collection of **TrackInfo** objects. The most important action is binding the 'files' list view to **_results** object by assigning **_results** to **files.ItemsSource**. This ensures

that the **files** list view will always display the contents of the **_results** object.

```
    public MainWindow()
    {
      InitializeComponent();
      _folderBrowserDialog = new FolderBrowserDialog();
      _folderBrowserDialog.Description =
        "Select the directory that containd the MP3 files.";
      // Do not allow the user to create new files via the FolderBrowserDialog.
      _folderBrowserDialog.ShowNewFolderButton = false;
      _folderBrowserDialog.RootFolder = Environment.SpecialFolder.DesktopDirectory;
      var dt = Environment.GetFolderPath(Environment.SpecialFolder.DesktopDirectory);
      var start = System.IO.Path.Combine(dt, "Downloads");
      _folderBrowserDialog.SelectedPath = start;
      _results = new ObservableCollection();
      files.ItemsSource = _results;
    }
```

The action is usually triggered by selecting a folder to browse. When you click the 'Browse...' button the **BrowseFolderDialog** pops up and lets you select a target directory. If the user selected a folder and clicked 'OK' the text of the **tbTargetFolder** text box is set to the selected path. If the user clicked 'Cancel' nothing happens.

```
void btnSelectDir_Click(object sender, RoutedEventArgs e)
{
  DialogResult r = _folderBrowserDialog.ShowDialog();
  if (r == System.Windows.Forms.DialogResult.OK)
  {
      tbTargetFolder.Text = this._folderBrowserDialog.SelectedPath;
  }
}
```

Another way to initiate the action is to directly type a folder path into the **tbTargetFolder** text box. In both cases the **TextChanged** event of the text box will fire and corresponding event handler will be called. It will check if the text constitutes a valid folder path and if so calls the **collectTrackInfo()** method.

```
    private void tbTargetFolder_TextChanged(object sender,
                                                    TextChangedEventArgs e)
    {
      if (Directory.Exists(tbTargetFolder.Text))
        collectTrackInfo(tbTargetFolder.Text);
    }
```

The **collectTrackInfo()** method is pretty central so I'll explain it in detail. First of all, it disables the browse button and the target folder text box to ensure that the user doesn't try to go to a different folder while the collection is in progress. This prevents a whole class of race condition and synchronization issues.

```
    void collectTrackInfo(string targetFolder)
    {
      btnSelectDir.IsEnabled = false;
      tbTargetFolder.IsEnabled = false;
```

The next part is getting all the MP3 files in the target folder. I used a LINQ expression that reads almost like English: "From the files in the target folder select all the files whose extension is ".mp3":

```
var mp3_files = from f in Directory.GetFiles(targetFolder)
                    where System.IO.Path.GetExtension(f) == ".mp3"
                    select f;
```

The collection of mp3 files is returned in the reverse order for some reason, so I reverse them back.

```
      mp3_files = mp3_files.Reverse();
```

Now, the **_fileCount** member variable is updated and the **_results** collection is cleared:

```
      _fileCount = mp3_files.Count();
      _results.Clear();
```

If **_fileCount** is 0 it means no mp3 files were found and there is no need to collect any track information. The status text box is updated and the browse button and the target folder text box are enabled.

```
      if (_fileCount == 0)
      {
        status.Text = "No MP3 files in this folder.";
        btnSelectDir.IsEnabled = true;
        tbTargetFolder.IsEnabled = true;
      }
```

If **_fileCount** is greater than 0, then a new instance of **_trackDurations** is created and receives the media element, the collection of mp3 files in the target folder and the **onTrackInfo()** callback.

```
      else
        _trackDurations = new TrackDurations(mediaElement,
                                             mp3_files,
                                             onTrackInfo);
```

The **onTrackInfo()** callback is called by **TrackDurations** every time the information about one of the tracks is collected and once more in the end (with a null

TrackInfo). If **ti** (the **TrackInfo** object) is null it means we are done with the current directory. The **_maxLength** variable is reset to 0, the **_trackDurations** object is disposed of, the status text box displays "Ready." and the selection controls are enabled again.

```
void onTrackInfo(TrackInfo ti)
{
  if (ti == null)
  {
    _maxLength = 0;
    _trackDurations.Dispose();
    status.Text = "Ready.";
    btnSelectDir.IsEnabled = true;
    tbTargetFolder.IsEnabled = true;
  }
```

If **ti** is not null it means a new **TrackInfo** object was received asynchronously. First of all the **TrackInfo** object is added to the **_results** collection. As you recall (or not) the **_results** collection is data-bound to the list view, so just adding it to **_results** make the new track info show up in the list view.

```
  else
  {
    _results.Add(ti);
```

The next step is to make sure the new filename fits in the first list view column. This is a little cumbersome and involves first creating a **FormattedText** object with the proper font of the list view and then checking if the size of this formatted text object is greater than the current **_maxWidth**. If it is greater than it becomes the new **_maxWidth** and the width of first column of the list view is set to the new **_maxWidth**.

```
    // Make sure the new filename fits in the column
    var ft = new FormattedText(
      ti.Name,
      CultureInfo.GetCultureInfo("en-us"),
      System.Windows.FlowDirection.LeftToRight,
      new Typeface(files.FontFamily,
                   files.FontStyle,
                   files.FontWeight,
                   files.FontStretch),
      files.FontSize,
      Brushes.Black);

    if (ft.Width > _maxLength)
    {
      _maxLength = ft.Width;
      var gv = (GridView)files.View;
      var gvc = gv.Columns[0];
      var curWidth = gvc.Width;

      // Reset to a specific width before auto-sizing
      gvc.Width = _maxLength;
      // This causes auto-sizing
      gvc.Width = Double.NaN;

    }
```

The last part of the **onTrackInfo()** method is updating the status line with current count of track info object out of the total number of MP3 files.

```
    // Update the status line
    var st = String.Format("Collecting track info {0}/{1} ...",
                           _results.Count,
                           _fileCount);
    status.Text = st;
  }
}
```

After all the information has been collected the user may select files in the list view using the standard Windows selection conventions (click/space to select, ctrl+click/space to toggle selection and shift+click/space to extend selection). Whenever the selected files change the **SelectionChanged** event is fired and the event handler calculates the total duration of all the currently selected files and update 'total' text box.

```
private void files_SelectionChanged(object sender,
                                    SelectionChangedEventArgs e)
{
  var tp = new TimeSpan();
  foreach (var f in files.SelectedItems)
  {
    tp += ((TrackInfo)f).Duration;
  }

  var d = new DateTime(tp.Ticks);
  string format = "mm:ss";
  if (tp.Hours > 0)
    format = "hh:mm:ss";

  total.Text = d.ToString(format);
}
```

There are two buttons called 'Select All' and 'Unselect All' that are hooked to corresponding event handlers and simply select or unselect all the files in list view when clicked. This results of course in a **SelectionChanged** event handled by the files **_SelectionChanged** event handler described above.

```
private void SelectAll_Click(object sender, RoutedEventArgs e)
{
  files.SelectAll();
}
private void UnselectAll_Click(object sender, RoutedEventArgs e)
```

```
  {
    files.UnselectAll();
  }
```

The DurationConverter Class

There is one last small class that demonstrates an interesting feature of WPF data binding called "data conversion." When binding a data source to a target UI element you may often want to format it a little differently. You can do it of course by creating a new class that consumes the original data object and formats it and then bind the UI element to the new object. But, WPF provides a standardized solution in the form of the **IValueConverter** interface. To use it you implement a value converter class that implements the interface (a **Convert()** and **ConvertBack()** methods), decorate it with the **ValueConversion** attribute and you are good to go. The **ConvertBack()** method is useful in two-way (or write-only) binding scenarios where you want to update the source object when you modify the value in the UI. The **DurationConverter** class is used for one-way binding only so the **ConvertBack()** method just throws a **NotImplementedException**. The **Convert()** method takes a **TimeSpan** value and converts it to a string that displays minutes and seconds (if less than an hour) or hours, minutes and seconds. This is more human-readable and there is no need for a finer resolution for the purpose of the MP3 Duration Converter:

```
[ValueConversion(typeof(TimeSpan), typeof(string))]
public class DurationConverter : IValueConverter
{
  public object Convert(object value,
                        Type targetType,
                        object parameter,
                        CultureInfo culture)
  {
    var duration = (TimeSpan)value;
    var d = new DateTime(duration.Ticks);
    string format = "mm:ss";
    if (duration.Hours > 0)
      format = "hh:mm:ss";

    return d.ToString(format);
  }

  public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
  {
     throw new NotImplementedException();
  }
}
```

The **DurationConverter** is used in the XAML for the text property of the **total** text box:

```
      <TextBox DockPanel.Dock="Left"
               Height="22"
               Name="total"
               Width="60"
               Margin="3"
                       Text="{Binding Path=Self,
                              Converter={StaticResource DurationConverter}}"
      >
      </TextBox>
```

Conclusion

This article showcased the MP3 duration converter and demonstrated various WPF techniques and practices that cover a lot of ground: UI, asynchronous programming, data binding and multimedia control. In the next article I will tell you what happend when I released the program to my wife (hint: not a lot of gratitude).