WinInteropUtils

WinInteropUtils is a C# library written in .NET 8 that provides managed P/Invoke wrappers for tons of functions you can use, from reading file attributes to interacting with COM interfaces.

The base namespace is `FireBlade.WinInteropUtils`.

> **WARNING**
>
> WinInteropUtils is still in beta, so bugs may occur. If you experience any bugs, please report them on the Issues page.

Features

- WindowsFile file info wrapper
- Full enums (1600+ values) for HRESULTs and Win32 error codes
- Dialog boxes
- Useful utility User32 functions (SendMessage, SetWindowLongPtr...)
- Constants and Macros
- COM interfaces
- Icon management (DestroyIcon, ...)

Getting Started

WinInteropUtils provides many wrappers for P/Invoke functions, from simple functions to COM interfaces. This guide will show you how to install and use this library.

> **WARNING**
>
> WinInteropUtils is still in beta, so bugs may occur. If you experience any bugs, please report them on the Issues page.

Usage Examples

Reading file attributes

This example reads the attributes of a file named `C:\test.txt` and checks if the file is hidden.

```
// Get the WindowsFile
var winFile = Shell32.GetFileInfo(@"C:\test.txt");

if (winFile != null)
{
    // Always make sure to dispose WindowsFiles when you're done using them
    // to avoid leaks on GDI handles!
    using WindowsFile file = winFile;

    // Check the attributes
    Console.WriteLine(file.Attributes.HasFlag(WindowsFileAttributes.Hidden)
    ? "File is hidden!"
     : "File is visible...");
}
else
{
    Console.WriteLine("File not found or read error");
}
```

Showing dialog boxes

The following examples show how to display common dialog boxes.

```
// ShellAbout dialog box (for an example run the winver command)
// icon: either a System.Drawing.Icon or nint GDI icon handle

Shell32.ShellAbout(hWnd, "My app name", "Some additional info", myAppIcon);
```

```
Shell32.ShellAbout("My app name", "Some additional info"); // without an icon and hwnd

// Icon picker dialog (PickIconDlg)

Shell32.ShowPickIconDialog(@"%SYSTEMROOT%\System32\shell32.dll", 16, out Icon ic);
Shell32.ShowPickIconDialog(@"%SYSTEMROOT%\System32\imageres.dll", 48, out string path, out
int idx);

// File properties dialog (doesn't support hwnd because that's how Windows works)

Shell32.ShowFileProperties(@"C:\test.txt");
```

Using COM interfaces

This example shows how to use COM interfaces.

TIP

For more info, see Using COM.

```
using FireBlade.WinInteropUtils.ComponentObjectModel; // make sure to add this for COM!

HRESULT hr = COM.Initialize(CoInit.ApartmentThreaded);

if (Macros.Succeeded(hr))
{
    hr = COM.CreateInstance<IFileOpenDialog>(
        new Guid("84bccd23-5fde-4cdb-aea4-af64b83d78ab"),
        null,
        CreateInstanceContext.InprocServer,
        new Guid("84bccd23-5fde-4cdb-aea4-af64b83d78ab"),
        out IFileOpenDialog dlg);

    if (Macros.Succeeded(hr))
    {
        dlg.SetTitle("My File Dialog - Open a File");

        hr = dlg.Show(hWnd);

        if (Macros.Succeeded(hr))
        {
            Console.WriteLine("Dialog accepted!");
        }
    }

    COM.Uninitialize();
}
```

API

For more documentation, visit the API page.

Using COM

COM (Component-Object Model) is a Windows framework that allows for reusable components in your code. This guide will go over how to use COM in WinInteropUtils.

> **WARNING**
>
> **COM** is for advanced users, so there's manual memory management and other advanced things. Wrappers for COM interfaces will come in future releases, but for now this is the only option.

Basic Setup

To begin using COM, you need to import the appropirate COM namespaces into your script:

```
using FireBlade.WinInteropUtils.ComponentObjectModel; // for base COM class and additional future COM non-interface wrappers
using FireBlade.WinInteropUtils.ComponentObjectModel.Interfaces; // for COM interfaces

using FireBlade.WinInteropUtils; // base WinInteropUtils namespace, required for Macros (useful for COM because we need to check a lot of HRESULTs)
```

Now we're ready to use COM. Over the course of this guide we will build up a sample of a file dialog. Before using COM interfaces, we need to initialize COM:

```
// if you're using v0.11 or earlier the function is named COM.CoInitialize instead
// if you're using v0.2 or earlier the enum is named CoInit instead (don't blame me for the compatibility and migration issues; that's what you get for using a library in beta)
HRESULT hr = COM.Initialize(COMInitOptions.ApartmentThreaded); // or MultiThreaded if inside a non-GUI/non-STA app
```

Now, after initializng COM, we need to check if the initialization succeeded:

```
if (Macros.Succeeded(hr))
{
    // init succeeded: proceeded with COM

    // our COM code...
}
```

Creating and Interacting with a COM instance

Now that COM is initialized, we need to actually create the COM instance, and once again check for success:

```
hr = COM.CreateInstance<IFileOpenDialog>(
    new Guid("DC1C5A9C-E88A-4dde-A5A1-60F82A20AEF7"),
    null,
    COM.CreateInstanceContext.InprocServer,
    new Guid("d57c7288-d4ad-4768-be02-9d969532d960"),
    out IFileOpenDialog dlg);

if (Macros.Succeeded(hr))
{
    // creation succeeded: proceed with interaction
}
```

And we can now proceed with interacting with the interface, which is a lot easier. In the following example, we configure the file dialog with a custom title and 2 filters:

NOTE

This code defines the `COMDLG_FILTERSPEC` struct itself, because WinInteropUtils doesn't support it yet.

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct COMDLG_FILTERSPEC
{
    public string pszName;
    public string pszSpec;
}


COMDLG_FILTERSPEC[] filters = new COMDLG_FILTERSPEC[]
{
    new COMDLG_FILTERSPEC { pszName = "Text Files", pszSpec = "*.txt" },
    new COMDLG_FILTERSPEC { pszName = "All Files", pszSpec = "*.*" }
};

// Allocate unmanaged memory (SetFileTypes requires pointer)
IntPtr ptr = Marshal.AllocHGlobal(Marshal.SizeOf<COMDLG_FILTERSPEC>() * filters.Length);

try
{
    // Copy each struct into unmanaged memory
    for (int i = 0; i < filters.Length; i++)
    {
        IntPtr structPtr = IntPtr.Add(ptr, i * Marshal.SizeOf<COMDLG_FILTERSPEC>());
        Marshal.StructureToPtr(filters[i], structPtr, false);
    }

    // ptr now points to the unmanaged array
    dlg.SetFileTypes((uint)filters.Length, ptr);
}
finally
{
    Marshal.FreeHGlobal(ptr);
}

dlg.SetTitle("Cool File Dialog");
```

And now we can actually SHOW the file dialog and read its results:

```
    hr = dlg.Show(hWnd);

    if (Macros.Succeeded(hr))
    {
        Console.WriteLine("Dialog accepted!");

        hr = dlg.GetResult(out nint iptr);

        if (Macros.Succeeded(hr))
        {
            IShellItem item = (IShellItem)Marshal.GetTypedObjectForIUnknown(iptr,
    typeof(IShellItem));

            hr = item.GetDisplayName(
                SIGDN.SIGDN_FILESYSPATH,
                out nint pptr);

            if (Macros.Succeeded(hr))
            {
                string path = Marshal.PtrToStringUni(pptr);

                Console.WriteLine("Chosen path: " + path);

                Marshal.FreeCoTaskMem(pptr);
            }
    }
}
```

Cleanup

Once you're done using COM, you need to deinitialize the COM library (if the initialization succeeded) and release any COM interfaces, using the Release method:

```
dlg.Release();
COM.Uninitialize();
```

Final Example

So now, our final example code looks like this:

```
using FireBlade.WinInteropUtils.ComponentObjectModel;
using FireBlade.WinInteropUtils.ComponentObjectModel.Interfaces;
using FireBlade.WinInteropUtils;

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct COMDLG_FILTERSPEC
{
    public string pszName;
    public string pszSpec;
}

HRESULT hr = COM.Initialize(COMInitOptions.ApartmentThreaded);

if (Macros.Succeeded(hr))
{
    hr = COM.CreateInstance<IFileOpenDialog>(
        new Guid("DC1C5A9C-E88A-4dde-A5A1-60F82A20AEF7"),
        null,
        COM.CreateInstanceContext.InprocServer,
        new Guid("d57c7288-d4ad-4768-be02-9d969532d960"),
        out IFileOpenDialog dlg);

    if (Macros.Succeeded(hr))
    {
        COMDLG_FILTERSPEC[] filters = new COMDLG_FILTERSPEC[]
        {
            new COMDLG_FILTERSPEC { pszName = "Text Files", pszSpec = "*.txt" },
            new COMDLG_FILTERSPEC { pszName = "All Files", pszSpec = "*.*" }
        };
        // Allocate unmanaged memory (SetFileTypes requires pointer)
        IntPtr ptr = Marshal.AllocHGlobal(Marshal.SizeOf<COMDLG_FILTERSPEC>()
* filters.Length);

        try
        {
            // Copy each struct into unmanaged memory
            for (int i = 0; i < filters.Length; i++)
            {
                IntPtr structPtr = IntPtr.Add(ptr, i * Marshal.SizeOf<COMDLG_FILTERSPEC>());
                Marshal.StructureToPtr(filters[i], structPtr, false);
            }

            // ptr now points to the unmanaged array
            dlg.SetFileTypes((uint)filters.Length, ptr);
        }
        finally
        {
            Marshal.FreeHGlobal(ptr);
        }

        dlg.SetTitle("Cool File Dialog");

        hr = dlg.Show(hWnd);

        if (Macros.Succeeded(hr))
        {
            Console.WriteLine("Dialog accepted!");

            hr = dlg.GetResult(out nint iptr);

            if (Macros.Succeeded(hr))
            {
                IShellItem item = (IShellItem)Marshal.GetTypedObjectForIUnknown(iptr,
typeof(IShellItem));

                hr = item.GetDisplayName(
                    SIGDN.SIGDN_FILESYSPATH,
                    out nint pptr);

                if (Macros.Succeeded(hr))
                {
                    string path = Marshal.PtrToStringUni(pptr);

                    Console.WriteLine("Chosen path: " + path);
                    Marshal.FreeCoTaskMem(pptr);
                }

                item.Release();
            }
        }

        dlg.Release();
    }

    COM.Uninitialize();
}
```

If we run this sample code, a file dialog should appear and, after picking a file, its path should be outputted.

Basic COM Interfaces

This article lists the basic COM interfaces.

NOTE

This article lists only COM interfaces defined by WinInteropUtils.
For interfaces for the Common Item Dialog, see File Dialog COM Interfaces.

IUnknown

API
The base interface that all COM interfaces inherit from. This interface defines the base Release and AddRef methods.

IShellItem

API
Exposes methods that retrieve information about a Shell item, such as a file or directory. Third parties do not implement this interface; only use the implementation provided with the system.

IModalWindow

API
This interface represents a modal window, such as a file dialog. This interface only exposes the Show method.

File Dialog COM Interfaces

This article lists the COM interfaces for the Common Item Dialog, the modernized common file dialog introduced in Windows Vista.

ⓘ **NOTE**

This article lists only COM interfaces defined by WinInteropUtils.
For basic interfaces that are not associated with the Common Item Dialog, see Basic COM Interfaces.

IFileDialog

API

The base interface for file dialogs. This interface exposes methods that initialize, show, and get results from the common file dialog, and inherits from IModalWindow.

IFileOpenDialog

API

This interface exposes methods for the Open file dialog. The only methods exposed in this interface are GetResults and GetSelectedItems. It inherits from IFileDialog.

IFileSaveDialog

API

This interface exposes methods for the Save file dialog. It extends the IFileDialog interface by adding methods specific to the save dialog, which include those that provide support for the collection of metadata to be persisted with the file.

IFileDialogCustomize

API

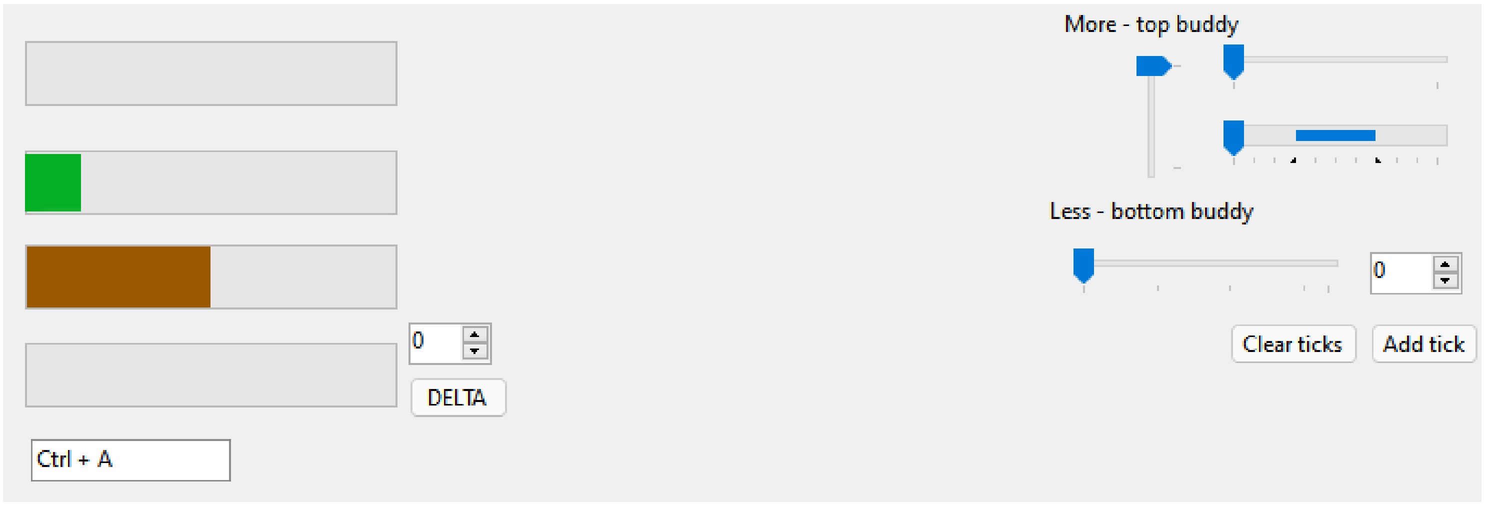
This interface exposes methods that allow adding controls and customizing a file dialog.

⚠ **WARNING**

Controls are added to the dialog before the dialog is shown. Their layout is implied by the order in which they are added. Once the dialog is shown, controls cannot be added or removed, but the existing controls can be hidden or disabled at any time. Their labels can also be changed at any time.

WinInteropUtils.WinForms

`WinInteropUtils.WinForms` is a sub-package of WinInteropUtils that provides Windows Forms (WinForms) wrappers for Win32 controls.



Controls

`WinInteropUtils.WinForms` currently has the following control wrappers:

- Slider (Win32: `Track Bar`)
- ProgressBarEx (Win32: `ProgressBar`)
- HotKeyBox (Win32: `HotKey`)

What is WinInteropUtils.WinForms for?

`WinInteropUtils.WinForms` allows you to make better apps with new Win32 controls that are not built-in by default with WinForms or extended versions of the built-in controls that support all the features of the native Win32 control.

Install WinInteropUtils.WinForms

`WinInteropUtils.WinForms` is inside the same branch and repository on Git as the main `WinInteropUtils` package. To use it, you can go to the same releases page as the main `WinInteropUtils` library, but get the `WinInteropUtils.WinForms.dll` file instead of the base `WinInteropUtils.dll` file. Alternatively, you can install the package on NuGet.

Add the controls to the Toolbox

If the controls don't show up in your WinForms designer's toolbox after installation, you need to do the following steps:

1. Inside the Toolbox, right-click on the header where you'd like the `WinInteropUtils.WinForms` controls to be located, and click **Choose Items...** .

2. Inside the dialog, find and check the controls to add. If you need all the controls, make sure you check:

   - Slider
   - ProgressBarEx
   - HotKeyBox

   Make sure that the namespace(s) of the controls you checked are all `FireBlade.WinInteropUtils.WinForms`.

   > **TIP**
   >
   > You can use the **Filter** box to search through the list to find the controls easier.

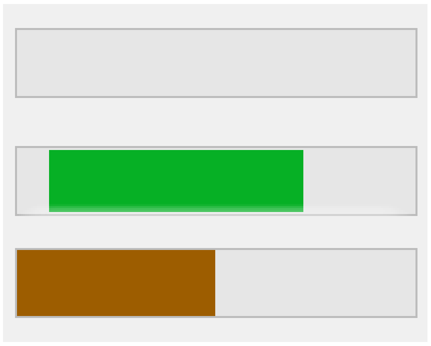
3. Once done, press **OK**. The checked controls should now show up in the toolbox.

Additional info

The base namespace for all `WinInteropUtils.WinForms` controls is `FireBlade.WinInteropUtils.WinForms`.

ProgressBarEx control

The `ProgressBarEx` control is an extended version of the WinForms `ProgressBar` control that provides all the features of the Win32 `ProgressBar`.



Range

To set the range of the progress bar, use the `Maximum` and `Minimum` properties. To set the value of the progress bar, set the `Value` property:

```
progressBarEx1.Minimum = 0;
progressBarEx1.Maximum = 250;
progressBarEx1.Value = 125;
```

Progress Bar Types

A progress bar type is the type of movement of the progress bar. It can be one of the following values:

- `Marquee`
- `Blocks`
- `Smooth`

`Marquee` makes the progress bar act like a marquee, where the progress portion of the bar repeatedly moves along the bar. The `Blocks` type moves the progress bar in small blocks until it reaches the destination value. `Smooth` smoothly moves the progress bar to the target value.

To set the progress bar type, use the `Type` property on the `ProgressBarEx` control:

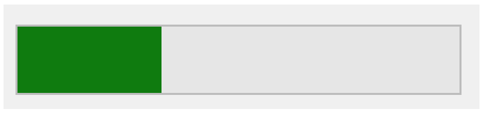
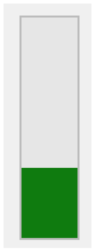
```
progressBarEx1.Type = ProgressBarExType.Marquee;
```

Reverse on Backward

If you're using a `ProgressBarEx` type of either `Smooth` or `Blocks`, by default, the bar will snap to the target value if the new value is smaller than the previous value. If you want to make the progress bar perform the normal movement animation even when going backwards, set the `ReverseOnBackward` property to `true` to re-enable the behavior.

Orientation

A `ProgressBarEx` control can be in one of 2 orientations:

- Horizontal
  
- Vertical
  

To set the orientation, use the `Orientation` property:

```
progressBarEx1.Orientation = Orientation.Vertical;
```

Style

WARNING

Requires visual styles. Make sure that Application.EnableVisualStyles is inside your Program.Main function.

A `ProgressBarEx` can be styled for a normal, paused or error bar. To set the style, use the `Style` property:

```
progressBarEx1.Style = ProgressBarExStyle.Error;
```

Colors

NOTE

Most visual styles override these colors, meaning that these changes may not apply if your app uses visual styles.

You can customize the bar color using the `ForeColor` property and the background color using the `BackColor` property.

Steps

You can step a `ProgressBarEx` control by a specified amount. The `PerformStep` function will step the progress bar by the amount of the `Step` property:

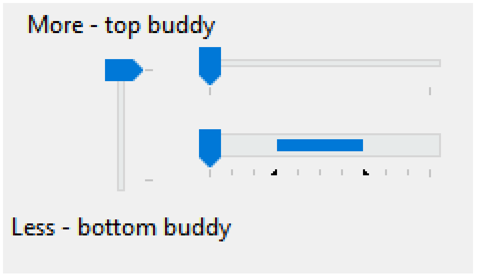
```
progressBarEx1.Step = 10;

// steps the progress bar by the step amount and wraps it around if it exceeds the range
progressBarEx1.PerformStep();
```

On the other hand, the `StepBy` function will step the progress bar by a specific amount:

```
// note: does not wrap around!
progressBarEx1.StepBy(25);
```

Slider control

The `Slider` control allows the user to pick a numerical value by dragging a thumb on a slider. Commonly also referred to as a `TrackBar` in Win32 programming.



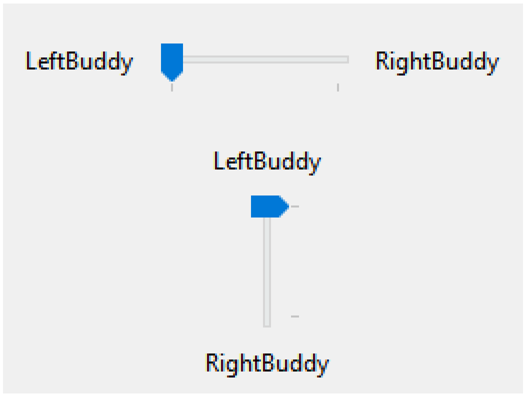
Slider Range

To set the range of the slider, use the `Maximum` and `Minimum` properties. To set the value of the slider, set the `Value` property:

```
// example: allow millisecond input, 1 ms - 1 s
slider1.Minimum = 1;
slider1.Maximum = 1000;
slider1.Value = 500;
```

Buddy Controls

A `Slider` can have buddy controls that can appear on the left or right if the `Slider` is horizontal or on the top or bottom when the `Slider` is vertical.



To set the buddy controls, use the `LeftBuddy` and `RightBuddy` properties:

```
slider1.LeftBuddy = label1;
slider1.RightBuddy = label2;
```

TIP

You can assign any control as the buddy of a slider, including another slider! The only exceptions to this are Forms and the slider that you're setting the buddy on itself.

Orientation

A `Slider` can be set to `Horizontal` or `Vertical` orientation:

```
slider1.Orientation = Orientation.Vertical;
```

Reverse Sliders

You can reverse a slider to make it so that smaller numbers mean higher and larger numbers mean lower.

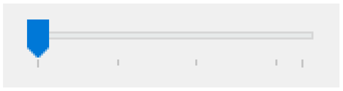
```
slider1.Reversed = true;
```

You can also reverse the input of the slider separately, by setting the `ReverseInput` property:

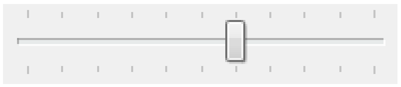
```
slider1.ReverseInput = true;
```

Thumb

The thumb on the `Slider` is the actual handle you drag. In the screenshot below, the blue part is the thumb:



The thumb may look different in different visual styles. For example, this is how a `Slider` may look like under the `Aero` visual style in Windows Vista or Windows 7. Note the fact that the thumb under this visual style is a silver color:



To hide the thumb, use the `IsThumbVisible` property:

```
slider1.IsThumbVisible = false;
```

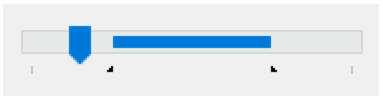
By default, the thumb is automatically sized to fit the `Slider`. To override the width, set the `FixedLength` property to `true` and use the `ThumbLength` property to specify the desired length.

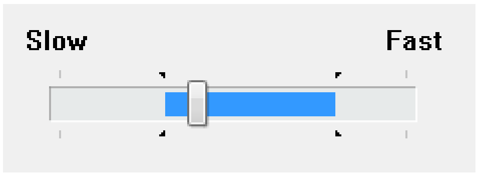
```
slider1.FixedLength = true;
slider1.ThumbLength = 10;
```

To get the current `Rectangle` size of the thumb, use the `ThumbRect` property.

Slider Selection Range

The selection range allows you to define a range that is highlighted by the `Slider`:





The slider will also show triangular ticks marks at the start and end of the range. To set the selection range, set the `ShowSelectionRange` property to `true` and use the `SelectionRangeStart` and `SelectionRangeEnd` properties to set the desired range:

```
slider1.ShowSelectionRange = true;
slider1.SelectionRangeStart = 25;
slider1.SelectionRangeEnd = 75;
```

To clear the selection range, call the `ClearSelectionRange` method.

NOTE

The slider's selection range does not affect its functionality in any way. It is up to the application to implement the range. You might do this in one of the following ways:

- Use a selection range to enable the user to set maximum and minimum values for some parameter. For example, the user could move the slider to a position and then click a button labeled **Max**. The application then sets the selection range to show the values chosen by the user.
- Limit the movement of the slider to a predetermined subrange within the control, by handling the ValueChanging event and disallowing any movement outside the selection range. You might do this, for example, if the range of values available to the user can change because of other choices the user has made, or according to available resources.

Ticks

The `Slider` can be in one of three tick modes: `None`, `Auto`, or `Custom`. `None` indicates no ticks. `Auto` indicates ticks will show up on every single value in the `Slider`'s available range of values. `Custom` indicates that ticks will show up every `X` values in the slider's range, where `X` is the value of the `TickFrequency` property.

To set the tick mode, set the `TickMode` property to the desired mode:

```
slider1.TickMode = SliderTickMode.None;
```

To use a custom tick amount, set the `TickMode` property to `SliderTickMode.Custom`, and set the `TickFrequency` property to the desired frequency:

```
slider1.TickMode = SliderTickMode.Custom;
slider1.TickFrequency = 10;
```

When using a `TickMode` of `Custom`, you can also manually add ticks individually:

```
slider1.AddTick(5);
```

NOTE

Once a tick is added using `AddTick`, you cannot remove it. If you want to remove a certain tick, you must call the `ClearTicks` method to clear the ticks and re-add the ticks back:

```
// add some ticks...
slider1.AddTick(10);
slider1.AddTick(12);
slider1.AddTick(17);

// to remove the tick for 12:
// clear the ticks...
slider1.ClearTicks();
// ...and add back the ticks we want and exclude the ones we don't want
slider1.AddTick(10);
slider1.AddTick(17);
```

This is a **Windows limitation**.

To get the current tick positions, call the `GetTickPosition`, `GetTickCount`, or `GetTickPositions` methods.

Large and Small Change (page/line size)

The small change (also called the "line size") is the amount the `Slider` moves after performing an action such as moving the slider with the arrow keys. To set the small change, use the `SmallChange` property:

```
slider1.SmallChange = 1;
```

The large change (also called the "page size") is the amount the `Slider` moves after performing an action such as moving the slider with the `PageUp` or `PageDown` keys. To set the large change, use the `LargeChange` property:

```
slider1.LargeChange = 5;
```

Tooltips

The `Slider` can provide tooltips when the user hovers over the thumb to show the current value. To show the tooltip, use the `ShowToolTip` property:

```
slider1.ShowToolTip = true;
```

You can also customize the side where the tooltip appears. To set the tooltip side, use the `ToolTipSide` property:

```
slider1.ToolTipSide = SliderToolTipSide.Left; // or: Top, Bottom, Right
```

Validation

You can validate the value the user drags the slider to and accept or block it. To validate the `Slider`'s value, subscribe to the `ValueChanging` event. The `ValueChanging` event fires before the slider's value is updated, allowing you to validate or cancel the change.

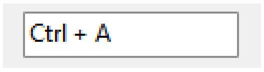
```
slider1.ValueChanging += Slider1_ValueChanging;
```

In the event, perform the neccessary validation checks and set `e.Cancel` to `true` if you want to block the movement operation.

```
private void Slider1_ValueChanging(object sender, SliderValueChangingEventArgs e)
{
    // peform validation checks...
    e.Cancel = true;
}
```

HotKeyBox control

The `HotKeyBox` control enables the user to enter a combination of keystrokes to be used as a hot key. A hot key is a key combination that the user can press to perform an action quickly. For example, a user can create a hot key that undoes the last action performed in the application and assign it to `Ctrl + Z`. The hot key box control displays the user's choices and ensures that the user selects a valid key combination. The following screen shot shows how a hot key box control appears after the user enters the Ctrl + A key combination.



Keys

To get or set the currently inputted keys in the control, use the `Keys` property:

```
hotKeyBox1.Keys = Keys.Control | Keys.Z;
```

The control also allows setting the extended key. To get or set the extended key, use the `IsExKey` property.

```
hotKeyBox1.IsExKey = true;
```

Rules

The hot key box control allows setting rules. Rules determine which modifiers are counted as invalid and are not able to be used as a key combination. For example, you may block the `Ctrl` key to make combinations like `Ctrl + Z` invalid. To set the rules, set the `Rules` property to a bitwise combination of `HotKeyBoxRules` values:

```
hotKeyBox1.Rules = HotKeyBoxRules.Ctrl | HotKeyBoxRules.ShiftCtrl;
```

When a user enters an invalid key combination, as defined by rules, the system uses the bitwise-OR operator to combine the keys entered by the user with the flags specified in the `FallbackValue` property. The resulting key combination is converted into a string and then displayed in the hot key control.

```
// Example: fall back to Ctrl+Alt if user enters invalid combination
hotKeyBox1.FallbackValue = Keys.Control | Keys.Alt;
```

`HotKeyChanged` event

The `HotKeyChanged` event fires when the inputted hot key changes in the control. Note that this event fires for every keystroke, even if that keystroke is a modifier key, so the event will still fire before the user types an actual non-modifier key.

Once the event fires, invoke the getter of the `Keys` property on the `HotKeyBox` to get the pressed keys.