```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Runtime.InteropServices;

namespace USBrelayDeviceNET
{
    /// <summary>
    /// Provides control of HID compliant USB Relay Devices
    /// The class uses native function calls to standard Windows APIs
    /// for communicating with the USB Relay devices.
    /// </summary>
    /// <remarks>
    /// Native Libraries used: hid.dll, setupapi.dll and kernel32.dll
    /// .NET Framework: 4.5, Build platform: AnyCPU
    /// Libarary has been tested in applications compiled for 32 bit and 64 bit CPUs in Windows 7/10/11
    /// </remarks>
    public sealed class USBrelayDevice: IDisposable
    {
        #region class constructer/destructor

        /// <summary>
        /// Initializes a new instance of USBrelayDevice class
        /// </summary>
        public USBrelayDevice()
        {
            pDevice = new IntPtr();
            IsDisposed = false;
            GetDevices(true);
        }

        /// <summary>
        /// Initializes a new instance of USBrelayDevice class
        /// and opens a USB Relay device matching the device_ID
        /// </summary>
        /// <param name="device_ID">ID/serial number of device to open</param>
        /// <remarks>
        /// typically used when multiple device are attached to the system and
        /// a specific device should be opened when the class is initialized
        /// </remarks>
        public USBrelayDevice(string device_ID)
        {
            pDevice = new IntPtr();
            IsDisposed = false;
            GetDevices(true);
            if (USBrelayDevicesFound)
            {
                OpenDevice(device_ID);
            }
        }

        /// <summary>
        /// Initializes a new instance of USBrelayDevice class
        /// and opens the first USB Relay Device found
        /// </summary>
        /// <param name="open_First">true to open the device, false if no device should be opened</param>
        /// <remarks>
        /// typically used when only one device is attached to the system
        /// </remarks>
        public USBrelayDevice(bool open_First)
        {
            pDevice = new IntPtr();
            IsDisposed = false;
            GetDevices(true);
            if (USBrelayDevicesFound & open_First)
            {
                OpenDevice(0);
            }
        }

        /// <summary>
        /// Class finalizer
        /// </summary>
        ~USBrelayDevice()
        {
            Dispose(false);
        }

        /// <summary>
        /// Releases all resources used by USBrelayDevice.
        /// </summary>
        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        /// <summary>
        /// Releases the unmanaged resources used by USBrelayDevice and optionally releases the managed resources.
        /// </summary>
        /// <param name="disposing">true to release both managed and unmanaged resources; false to release only unmanaged 
        resources.</param>
        private void Dispose(bool disposing)
        {
            if (IsDisposed) return;
            if (disposing)
            {
                if (DeviceOpen | pDevice != IntPtr.Zero) CloseDevice();
                localDeviceList.Clear();
        }
        pDevice = IntPtr.Zero;
        DeviceOpen = false;
        USBrelayDevicesFound = false;
        IsDisposed = true;
    }

    #endregion

    #region Error Event Handler

    /// <summary>
    /// Error event handler
    /// </summary>
    /// <param name="e">error event arguments</param>
    private void OnError(USBrelayEventArgs e)
    {
        var handler = USBrelayError;
        if (handler != null)
        {
            handler(this, e);
        }
    }

    /// <summary>
    /// USBrelayDevice class Error Event
    /// </summary>
    public event EventHandler<USBrelayEventArgs> USBrelayError;

    /// <summary>
    /// Common method to fire Error event
    /// </summary>
    /// <param name="error">error to report</param>
    /// <param name="source">method name with error</param>
    /// <param name="exception">if error is Exception, then exception thrown, else null</param>
    private void Error_Handler(string error, string source, object exception)
    {
        var _error = new USBrelayEventArgs
        {
            ExceptionThrown = (Exception)exception,
            Error = error,
            Source = source,
            IsException = (exception == null)
        };

        OnError(_error);
    }

    #endregion

    #region Public Properties

    /// <summary>Contains USB Relay device information</summary>
    public struct DeviceInfo
    {
        /// <summary> device serial number</summary>
        public string device_ID;
        /// <summary> system path to device</summary>
        public string device_path;
        /// <summary> number of relay channels</summary>
```

```
            public int relay_Count;
            /// <summary>Device description</summary>
            public string product_name;
            /// <summary>list index value</summary>
            public int index;
        }

        /// <summary>true when class has been disposed</summary>
        public bool IsDisposed { get; private set; }


        /// <summary>true when USB Relay devices are found and available for connection</summary>
        public bool USBrelayDevicesFound { get; private set; }

        /// <summary>true when a USB Relay device is connected</summary>
        public bool DeviceOpen { get; private set; }

        #endregion

        #region Private Member Declarations

        private struct LocalDeviceInfo
        {
            public string device_path;
            public string device_ID;
            public int relay_Count;
        }

        private List<LocalDeviceInfo> localDeviceList; //Device Information list of all USB Relay Devices attached to system
        private int relay_count; //relay count of current open device
        private IntPtr pDevice; //handle of current open device

        #endregion

        #region Public Methods

        /// <summary>
        /// Finds all USB Relay Devices attached to the system and
        /// generates a list containing device information
        /// for each of the attched devices.
        /// </summary>
        /// <returns>list of device info structures</returns>
        /// <remarks>This method uses most all of the NativeMethod function calls and structures</remarks>
        public List<DeviceInfo> GetDevices(bool closeOpenDevice)
        {
            //Close device if open
            if (pDevice != IntPtr.Zero & closeOpenDevice)
            {
                CloseDevice();
            }

            //constants used to validate a specific HID device
            const int PID = 0x05df; // USB Relay Device Product ID
            const int VID = 0x16c0; // USB Relay Device Vendor ID

            //constants used by SetupDiGetClassDevs function
            const int DIGCF_PRESENT = 0x02;
            const int DIGCF_DEVICEINTERFACE = 0x10;

            //initialize public and private class members set within method
            var DeviceInfoList = new List<DeviceInfo>(); //return List
            localDeviceList = new List<LocalDeviceInfo>(); //internal class list
            USBrelayDevicesFound = false;

            //pointer declarations
            var pDeviceList = new IntPtr(); //pointer to device list
            var pDeviceInterfaceDetailData = new IntPtr(); //pointer to SP_DEVICE_INTERFACE_DETAIL_DATA structure
            var pHandle = new IntPtr(); //pointer to device handle
            var pProductName = new IntPtr(); //pointer to device product name

            //structure declarations
            var deviceInterfaceData = new NativeMethods.SP_DEVICE_INTERFACE_DATA(); //required for setup api calls, members not used
            var deviceAttributes = new NativeMethods.HIDD_ATTRIBUTES(); //HID device attributes

            //Set the size parameter for the SP_DEVICE_INTERFACE_DATA structure
            deviceInterfaceData.Size = (uint)Marshal.SizeOf(typeof(NativeMethods.SP_DEVICE_INTERFACE_DATA));

            //loop variable declarations
            uint devIndex = 0; //device enumeration interface indexer
            var index = 0; //device info list indexer

            try
            {
                // Method reference Microsoft page "Finding and Opening a HID Collection"
                // https://learn.microsoft.com/en-us/windows-hardware/drivers/hid/finding-and-opening-a-hid-collection
                // Note that the SP_DEVICE_INTERFACE_DETAIL_DATA structure is not used directly, instead only the pointer
                // to the structure is used to get the device path that is used to open the device.
                // For reference, SP_DEVICE_INTERFACE_DETAIL_DATA Structure definition:
                // https://learn.microsoft.com/en-us/windows/win32/api/setupapi/ns-setupapi-sp_device_interface_detail_data_a

                // Get HID GUID
                Guid gHid;
                NativeMethods.HidD_GetHidGuid(out gHid);

                // Get a pointer to a list of all HID devices
                pDeviceList = NativeMethods.SetupDiGetClassDevs(ref gHid, null, IntPtr.Zero,
                    DIGCF_DEVICEINTERFACE | DIGCF_PRESENT);

                while (true) //loop through all found HID devices and find all HID Devices matching defined pid and vid values
                {
                    try
                    {
                        //Get a pointer to device information from the HID device List at device index
                        var result = NativeMethods.SetupDiEnumDeviceInterfaces(pDeviceList, IntPtr.Zero,
                            ref gHid, devIndex, ref deviceInterfaceData);

                        //if fails stop device enumeration, occurs when there are no more HID devices
                        if (!result) break;

                        //advance device enumeration index for next iteration
                        devIndex++;

                        //call Device Interface Detail to get size of device interface detail structure, ignore return value
                        uint size;
                        NativeMethods.SetupDiGetDeviceInterfaceDetail(pDeviceList, ref deviceInterfaceData,
                            IntPtr.Zero, 0, out size, IntPtr.Zero);

                        //if failed to get size for Device Interface Detail Data, go to next device
                        if (size == 0) continue;
```

```
//intialize Device Interface Details Data pointer
pDeviceInterfaceDetailData = Marshal.AllocCoTaskMem((int)size);

// set the cbSize property of DeviceInterfaceDetailData structure for 32 bit / 64 bit compatability
Marshal.WriteInt32(pDeviceInterfaceDetailData, (IntPtr.Size == 4) ? (4 + Marshal.SystemDefaultCharSize) : 8);

//set default size parameter for SetupDiGetDeviceInterfaceDetail function
var nBytes = size;

//get a pointer to Device Interface Details data, if fails skip device
result = NativeMethods.SetupDiGetDeviceInterfaceDetail(pDeviceList, ref deviceInterfaceData,
    pDeviceInterfaceDetailData, nBytes, out size, IntPtr.Zero);
if (!result | pDeviceInterfaceDetailData == IntPtr.Zero) continue;

//get path to the device from Device Interface Details data pointer(4 bytes from head address), if fails skip device
var devicePath = Marshal.PtrToStringAuto(pDeviceInterfaceDetailData + 4);
if (string.IsNullOrEmpty(devicePath)) continue;

//using found path to device, open device, returns a handle pointer to opened device, if fails skip device
pHandle = OpenHIDdevice(devicePath, true);
if (pHandle == IntPtr.Zero) continue;

//Get the device attributes to be used for device validation, if fails skip device
deviceAttributes.Size = (uint)Marshal.SizeOf(typeof(NativeMethods.HIDD_ATTRIBUTES));
result = NativeMethods.HidD_GetAttributes(pHandle, ref deviceAttributes);
if (!result) continue;

//Get device Vid and Pid values
var pid = deviceAttributes.ProductID;
var vid = deviceAttributes.VendorID;

//check device matches defined vid and pid values, if fails skip device
if (pid != PID | vid != VID)  continue;

/* Specified Device found -
 * following code used for USB Relay Devices
 * get Product Name and ID and update device info lists***
 */

//get product name, if fails skip device
var product_name = "";
pProductName = Marshal.AllocCoTaskMem(sizeof(char) * 128);
result = NativeMethods.HidD_GetProductString(pHandle, pProductName, sizeof(char) * 128);
if (result) product_name = Marshal.PtrToStringAuto(pProductName);
if (string.IsNullOrEmpty(product_name) || !product_name.ToUpper().Contains("USBRELAY")) continue;

//Get relay count of device (last char of Product Name string), if fails skip device
var rly_count = Convert.ToInt32(product_name.Remove(0, 8));
if(rly_count <= 0 | rly_count > 8) continue;

//get device ID string
var idStr = "error";
var idBuffer = new byte[10];
result = NativeMethods.HidD_GetFeature(pHandle, idBuffer, 9);
if (result)
{
    var str_arry = new char[5];
    Array.Copy(idBuffer, 1, str_arry, 0, 5);
    idStr = new string(str_arry);
}
```

```
                //create device info structures
                var devInfo = new DeviceInfo
                {
                    device_path = devicePath,
                    device_ID = idStr,
                    product_name = product_name,
                    relay_Count = rly_count,
                    index = index
                };

                var localDev = new LocalDeviceInfo
                {
                    device_path = devicePath,
                    device_ID = idStr,
                    relay_Count = rly_count
                };

                //update lists
                DeviceInfoList.Add(devInfo);
                localDeviceList.Add(localDev);
                index++; //advance list index for next USB Relay Device
            }
            catch (Exception ex)
            {
                var err = NativeMethods.GetLastError();
                Error_Handler("Exception ", "GetDevices() - internal loop - " + err, ex);
            }
            finally
            {
                //free resources and reset pointers for next iteration
                if (pHandle != IntPtr.Zero)
                {
                    NativeMethods.CloseHandle(pHandle);
                    pHandle = IntPtr.Zero;
                }

                if (pDeviceInterfaceDetailData != IntPtr.Zero)
                {
                    Marshal.FreeCoTaskMem(pDeviceInterfaceDetailData);
                    pDeviceInterfaceDetailData = IntPtr.Zero;
                }

                if (pProductName != IntPtr.Zero)
                {
                    Marshal.FreeCoTaskMem(pProductName);
                    pProductName = IntPtr.Zero;
                }
            }
        }
    }
    catch (Exception ex)
    {
        var err = NativeMethods.GetLastError();
        Error_Handler("Exception ", "GetDevices() - internal loop - " + err, ex);
    }
    finally
    {
        //free resource
        if(pDeviceList != IntPtr.Zero) NativeMethods.SetupDiDestroyDeviceInfoList(pDeviceList);
    }

    USBrelayDevicesFound = (DeviceInfoList.Count > 0);
    return DeviceInfoList;
}

/// <summary>
/// Opens a USB Relay Device using list index
/// </summary>
/// <param name="device_index">index of device in device listto open</param>
/// <returns>true if success, false if failed</returns>
public bool OpenDevice(int device_index)
{
    //if device is open, close device
    if (pDevice != IntPtr.Zero)
    {
        CloseDevice();
    }

    //initialize device open flag and relay count
    DeviceOpen = false;
    relay_count = 0;

    //verify device available to open
    if (localDeviceList.Count == 0 | device_index >= localDeviceList.Count) return false;

    //open device using device path and set device handle pointer
    pDevice = OpenHIDdevice(localDeviceList[device_index].device_path, false);

    //set device open flag
    DeviceOpen = (pDevice != IntPtr.Zero);

    //set device relay count
    if(DeviceOpen)
        relay_count = localDeviceList[device_index].relay_Count;

    return DeviceOpen;
}

/// <summary>
/// Opens a USB Relay Device using device id
/// </summary>
/// <param name="device_ID">USB Relay ID/serial number</param>
/// <returns>true if success, false if failed</returns>
public bool OpenDevice(string device_ID)
{
    //if device is open, close device
    if (pDevice != IntPtr.Zero)
    {
        CloseDevice();
    }

    //initialize device open flag and relay count and search index
    DeviceOpen = false;
    relay_count = 0;
    var index = -1;

    //verify device available to open
    if (localDeviceList.Count == 0) return false;
    //search for device with matching ID string
    for (var i = 0; i < localDeviceList.Count; i++)
    {
        if (!localDeviceList[i].device_ID.Equals(device_ID)) continue;
        index = i;
        break;
    }

    if (index == -1) //if device not found fire error event
    {
        var error = new USBrelayEventArgs
        {
            ExceptionThrown = null,
            Error = "Device ID not found.",
            Source = "OpenDevice()",
            IsException = false
        };
        OnError(error);
        return false;
    }

    //open device using device path and set device handle pointer
    pDevice = OpenHIDdevice(localDeviceList[index].device_path, false);

    //set device open flag
    DeviceOpen = (pDevice != IntPtr.Zero);

    //set device relay count
    if (DeviceOpen)
        relay_count = localDeviceList[index].relay_Count;

    return DeviceOpen;
}

/// <summary>
/// Closes connected USB Relay device
/// </summary>
public void CloseDevice()
{
    if (pDevice != IntPtr.Zero)
    {
        try
        {
            NativeMethods.CloseHandle(pDevice);
        }
        catch (Exception ex)
        {
            Error_Handler("Exception", "CloseDevice()", ex);
            var error = new USBrelayEventArgs
            {
                ExceptionThrown = ex,
                Error = "Exception",
                Source = "CloseDevice()",
                IsException = true
            };

            OnError(error);
        }
    }
    pDevice = IntPtr.Zero;
```

```
            DeviceOpen = false;
        }

        /// <summary>
        /// Turns on all relays
        /// </summary>
        /// <returns>true if success, false if failed</returns>
        public bool ALLRelaysON()
        {
            return SetRelayState(true, 0);
        }

        /// <summary>
        /// Turns off all relays
        /// </summary>
        /// <returns></returns>
        public bool ALLRelaysOFF()
        {
            return SetRelayState(false, 0);
        }

        /// <summary>
        /// Turns on a single relay
        /// </summary>
        /// <param name="relay">relay number</param>
        /// <returns>true if success, false if failed</returns>
        public bool RelayON(int relay)
        {
            return SetRelayState(true, relay);
        }

        /// <summary>
        /// Turns off a single relay
        /// </summary>
        /// <param name="relay">relay number</param>
        /// <returns>true if success, false if failed</returns>
        public bool RelayOFF(int relay)
        {
            return SetRelayState(false, relay);
        }

        /// <summary>
        /// Get the current status of all relays
        /// </summary>
        /// <returns>array of relay states, true = ON and false = OFF</returns>
        /// <remarks>return array index 0 = relay 1, index 1 = relay 2...</remarks>
        public bool[] GetRelayStatus()
        {
            //check that a device is open
            if (pDevice == IntPtr.Zero | !DeviceOpen)
            {
                Error_Handler("Device is not connected/open.", "GetRelayStatus()", null);
                return new bool[8];
            }

            //initialize return array
            var status_bits = new bool[8];

            //initialize data buffer
            var data_buffer = new byte[10];
            try
            {
                //send get data command to device
                NativeMethods.HidD_GetFeature(pDevice, data_buffer, 9);

                //transfer value from index 8 of data buffer to status byte
                var status = data_buffer[8];

                //create bit array from ststus byte value
                var bitarry = new BitArray(new int[] { status });

                //check bit array size
                if (bitarry.Count < 8)
                {
                    var err = NativeMethods.GetLastError();
                    Error_Handler("Index count mismatch", "GetRelayStatus() - " + err, null);
                    return status_bits;
                }

                //convert bit array to bool array having 8 elements
                var bits = new bool[bitarry.Count];
                bitarry.CopyTo(bits, 0);
                Array.Copy(bits, status_bits, 8);
            }
            catch (Exception ex)
            {
                var err = NativeMethods.GetLastError();
                Error_Handler("Exception", "GetRelayStatus() - " + err, ex);
            }
            return status_bits;
        }

        /// <summary>
        /// Gets the name and version number of library
        /// </summary>
        /// <returns>returns string array, index [0] is the Libarary name and index [1] is the version number</returns>
        public static string[] GetLibraryInfo()
        {
            var info = new string[2];
            var attributes = Assembly.GetExecutingAssembly().GetCustomAttributes(typeof(AssemblyFileVersionAttribute), false);
            info[0] = Assembly.GetExecutingAssembly().GetName().Name + ".dll";
            info[1] = attributes.Length == 0 ? "Unknown" : ((AssemblyFileVersionAttribute)attributes[0]).Version;
            return info;
        }

        /// <summary>
        /// Sets the Device ID/serial number
        /// </summary>
        /// <param name="deviceID">value to set, must be 5 characters in length</param>
        /// <returns>true if success, false if failed</returns>
        public bool SetDeviceID(string deviceID)
        {
            //check that a device is open
            if (pDevice == IntPtr.Zero | !DeviceOpen)
            {
                Error_Handler("Device is not connected/open.", "SetDeviceID()", null);
                return false;
            }
            try
            {
                if (deviceID.Length < 5) //check id string length (must be 5 or greater, only first five char will be used)
                {
                    Error_Handler("Device ID must be 5 characters in length.", "SetDeviceID()", null);
                    return false;
                }
                //convert deviceID string to char array
                var id = deviceID.ToCharArray(0, 5);

                //initialize data buffer and load data
                var cmd_buffer = new byte[9];
                cmd_buffer[0] = 0x00;
                cmd_buffer[1] = 0xfa;
                cmd_buffer[2] = (byte)id[0];
                cmd_buffer[3] = (byte)id[1];
                cmd_buffer[4] = (byte)id[2];
                cmd_buffer[5] = (byte)id[3];
                cmd_buffer[6] = (byte)id[4];
                cmd_buffer[7] = 0x00;
                cmd_buffer[8] = 0x00;

                //send data buffer to device to change ID
                var result = NativeMethods.HidD_SetFeature(pDevice, cmd_buffer, 9);

                if (!result) //check for error
                {
                    var _error = NativeMethods.GetLastError();
                    Error_Handler("Error setting Device ID. error code: " + _error, "SetDeviceID()", null);
                }

                return result;
            }
            catch (Exception ex)
            {
                Error_Handler("Exception ", "SetDeviceID()", ex);
            }

            return false;
        }

        #endregion

        #region Private Methods

        /// <summary>
        /// Method called by all methods requesting to change relay states
        /// </summary>
        /// <param name="state">true to turn relay(s) on, false to turn the relay(s) off</param>
        /// <param name="relay_num">index of relay to turn on (0 = All relays, otherwise value = relay number)</param>
        /// <returns>true if success</returns>
        private bool SetRelayState(bool state, int relay_num)
        {
            //check that a device is open
            if (pDevice == IntPtr.Zero | !DeviceOpen)
            {
                Error_Handler("Device is not connected/open.", "SetRelayState()", null);
                return false;
            }
            //check that the relay number is valid
            if (relay_num > relay_count)
            {
                Error_Handler("Relay number exceeds relay count of device.", "SetRelayState()", null);
                return false;
            }

            //initialize validation array
            var validate = new bool[8];

            //initialize command buffer
            var cmd_buffer = Enumerable.Repeat((byte)0x00, 10).ToArray();

            //set values for command buffer
            if (relay_num == 0) //All Relays
            {
                if (state)
                {
                    cmd_buffer[1] = 0xFE;
                    for (var i = 0; i < relay_count; i++)
                    {
                        validate[i] = true;
                    }
                }
                else
                {
                    cmd_buffer[1] = 0xFC;
                }
                cmd_buffer[2] = 0x00;
            }
            else //Individual Relay
            {
                if (state)
                {
                    cmd_buffer[1] = 0xFF;
                }
                else
                {
                    cmd_buffer[1] = 0xFD;
                }
                cmd_buffer[2] = (byte) relay_num;
                validate = GetRelayStatus();
                validate[relay_num - 1] = state;
            }

            //check if relay states are already set
            var check = GetRelayStatus();
            var result = validate.SequenceEqual(check);
            if (result) return true;

            //send command buffer to device
            try
            {
                result = NativeMethods.HidD_SetFeature(pDevice, cmd_buffer, 9);
                if (result)
                {
                    //check relay states set properly
                    check = GetRelayStatus();
                    result = validate.SequenceEqual(check);
                    if (result) return true;
                    Error_Handler("Error setting relay state.", "SetRelayState()", null);
                }
                else
                {
                    var _error = NativeMethods.GetLastError();
                    Error_Handler("Error setting relay state.", "SetRelayState() - " + _error, null);
                }
                return false;
            }
            catch (Exception ex)
            {
                var err = NativeMethods.GetLastError();
                Error_Handler("Exception", "SetRelayState() - " + err, ex);
                return false;
            }
        }

        /// <summary>
        /// Gets pointer/handle to HID device
        /// </summary>
        /// <param name="devicePath">path to HID device</param>
        /// <param name="suppressError">true to suppress exception reporting</param>
        /// <returns>on success returns pointer, if fails returns 0</returns>
        private IntPtr OpenHIDdevice(string devicePath, bool suppressError)
        {
            //File Flags
            const uint GENERIC_READ = 0x80000000;
            const uint GENERIC_WRITE = 0x40000000;
            const uint FILE_SHARE_READ = 0x00000001;
            const uint FILE_SHARE_WRITE = 0x00000002;
            const uint OPEN_EXISTING = 3;

            try
            {
                var pHandle = NativeMethods.CreateFile(devicePath, GENERIC_READ | GENERIC_WRITE,
                    FILE_SHARE_READ | FILE_SHARE_WRITE, IntPtr.Zero,
                    OPEN_EXISTING, 0, IntPtr.Zero);

                return pHandle;
            }
            catch (Exception ex)
            {
                DeviceOpen = false;
                if (!suppressError) Error_Handler("Exception", "OpenHIDdevice()", ex);
            }
            return IntPtr.Zero;
        }

        #endregion

        #region Native API call Definitions

        /// <summary>
        /// Class provides function prototypes and strcture definitions for native API calls
        /// </summary>
        internal static class NativeMethods
        {
            #region Structures
```

```
        //reference: https://learn.microsoft.com/en-us/windows/win32/api/setupapi/ns-setupapi-sp_device_interface_data
        [StructLayout(LayoutKind.Sequential, Pack = 1)]
        public struct SP_DEVICE_INTERFACE_DATA
        {
            public uint Size;
            private readonly Guid InterfaceClassGuid;
            private readonly int Flags;
            private readonly IntPtr Reserved;
        }

        //reference: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/hidsdi/ns-hidsdi-_hidd_attributes
        [StructLayout(LayoutKind.Sequential)]
        public struct HIDD_ATTRIBUTES
        {
            public uint Size;
            public readonly ushort VendorID;
            public readonly ushort ProductID;
            private readonly ushort VersionNumber;
        };

        #endregion

        #region Native function prototypes

        /// <summary>
        /// Function returns a handle to a device information set that contains requested device information elements for a local computer.
        /// </summary>
        /// <param name="classGuid">A pointer to the GUID for a device setup class or a device interface class.</param>
        /// <param name="enumerator">Specification string, optional and can be null</param>
        /// <param name="hwndParent">Handle to be used for a user interface device instance in the device information set. Optional and can 
be NULL.</param>
        /// <param name="flags">Specifies control options that filter the device information elements.</param>
        /// <returns>returns a handle to a device information set</returns>
        /// <remarks>ref: https://learn.microsoft.com/en-us/windows/win32/api/setupapi/nf-setupapi-setupdigetclassdevsw</remarks>
        [DllImport("setupapi.dll", SetLastError = true)]
        public static extern IntPtr SetupDiGetClassDevs(
            ref Guid classGuid,
            [MarshalAs(UnmanagedType.LPStr)] string enumerator,
            IntPtr hwndParent,
            uint flags);

        /// <summary>
        /// Function enumerates the device interfaces that are contained in a device information set.
        /// </summary>
        /// <param name="deviceInfoSet">pointer to a device information set that contains the device interfaces</param>
        /// <param name="deviceInfoData">pointer to an SP_DEVINFO_DATA structure. Optional can be set to zero</param>
        /// <param name="interfaceClassGuid">pointer to a GUID that specifies the device interface class.</param>
        /// <param name="memberIndex">A zero-based index into the list of interfaces in the device information set.</param>
        /// <param name="deviceInterfaceData">On successful return, a completed SP_DEVICE_INTERFACE_DATA structure</param>
        /// <returns>returns TRUE if the function completed without error.</returns>
        /// <remarks>ref: https://learn.microsoft.com/en-us/windows/win32/api/setupapi/nf-setupapi-setupdienumdeviceinterfaces</remarks>
        [DllImport("setupapi.dll", SetLastError = true)]
        public static extern bool SetupDiEnumDeviceInterfaces(
            IntPtr deviceInfoSet,
            IntPtr deviceInfoData, //uint deviceInfoData,
            ref Guid interfaceClassGuid,
            uint memberIndex,
            ref SP_DEVICE_INTERFACE_DATA deviceInterfaceData); //ref DEVICE_INTERFACE_DATA oInterfaceData);

        /// <summary>
        /// Function returns details about a device interface.
        /// </summary>
        /// <param name="deviceInfoSet">pointer to the device information set, typically returned by SetupDiEnumDeviceInterfaces.</param>
        /// <param name="deviceInterfaceData">reference to a SP_DEVICE_INTERFACE_DATA structure</param>
        /// <param name="deviceInterfaceDetailData">pointer to a SP_DEVICE_INTERFACE_DETAIL_DATA structure to receive information
        about the specified interface.</param>
        /// <param name="deviceInterfaceDetailDataSize">The size of the DeviceInterfaceDetailData buffer. Must be zero if
        DeviceInterfaceDetailData is NULL</param>
        /// <param name="requiredSize">variable that receives the required size of the DeviceInterfaceDetailData buffer.</param>
        /// <param name="deviceInfoData">pointer to a buffer that receives information about the device</param>
        /// <returns>returns TRUE if the function completed without error.</returns>
        /// <remarks>ref:
        https://learn.microsoft.com/en-us/windows/win32/api/setupapi/nf-setupapi-setupdigetdeviceinterfacedetaila</remarks>
        [DllImport("setupapi.dll", CharSet = CharSet.Auto, SetLastError = true)]
        public static extern bool SetupDiGetDeviceInterfaceDetail(
           IntPtr deviceInfoSet,
           ref SP_DEVICE_INTERFACE_DATA deviceInterfaceData,
           IntPtr deviceInterfaceDetailData,
           uint deviceInterfaceDetailDataSize,
           out uint requiredSize,
           IntPtr deviceInfoData);

        /// <summary>
        /// Function deletes a device information set and frees all associated memory.
        /// </summary>
        /// <param name="deviceInfoSet">A handle to the device information set to delete.</param>
        /// <returns>returns TRUE if it is successful.</returns>
        /// <remarks>ref: https://learn.microsoft.com/en-us/windows/win32/api/setupapi/nf-setupapi-setupdidestroydeviceinfolist</remarks>
        [DllImport("setupapi.dll", SetLastError = true)]
        public static extern bool SetupDiDestroyDeviceInfoList(IntPtr deviceInfoSet);

        /// <summary>
        /// Creates or opens a file or I/O device.
        /// </summary>
        /// <param name="lpFileName">The name of the file or path to a device to be created or opened.</param>
        /// <param name="dwDesiredAccess">The requested access to the file or devic</param>
        /// <param name="dwShareMode">The requested sharing mode of the file or device</param>
        /// <param name="lpSecurityAttributes">pointer to a SECURITY_ATTRIBUTES structure. Optional can be zero</param>
        /// <param name="dwCreationDisposition">action to take on a file or device that exists or does not exist.</param>
        /// <param name="dwFlagsAndAttributes"> file or device attributes and flags</param>
        /// <param name="hTemplateFile">handle to a template file with the GENERIC_READ access right.</param>
        /// <returns>If the function succeeds, the returns an open handle to the specified file, device.</returns>
        /// <remarks>ref:  https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea</remarks>
        [DllImport("kernel32.dll", SetLastError = true)]
        public static extern IntPtr CreateFile(
           string lpFileName,
           uint dwDesiredAccess,
           uint dwShareMode,
           IntPtr lpSecurityAttributes,
           uint dwCreationDisposition,
           uint dwFlagsAndAttributes,
           IntPtr hTemplateFile);

        /// <summary>
        /// Closes an open object handle.
        /// </summary>
        /// <param name="handle">A valid handle to an open object.</param>
        /// <returns>returns TRUE if it is successful.</returns>
        /// <remarks>ref: https://learn.microsoft.com/en-us/windows/win32/api/handleapi/nf-handleapi-closehandle</remarks>
        [DllImport("kernel32.dll", SetLastError = true)]
        public static extern bool CloseHandle(IntPtr handle);

        /// <summary>
        /// Retrieves the calling thread's last-error code value.
        /// </summary>
        /// <returns>The return value is the calling thread's last-error code.</returns>
        /// <remarks>ref: https://learn.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-getlasterror</remarks>
        [DllImport("kernel32.dll")]
        public static extern uint GetLastError();

        /// <summary>
        /// routine returns the device interface GUID for HIDClass devices.
        /// </summary>
        /// <param name="gHid">Pointer to GUID buffer that the routine uses to return the device interface GUID for HIDClass devices.</param>
        /// <remarks>ref: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/hidsdi/nf-hidsdi-hidd_gethidguid</remarks>
        [DllImport("hid.dll", SetLastError = true)]
        public static extern void HidD_GetHidGuid(out Guid gHid);

        /// <summary>
        /// routine returns the attributes of a specified top-level collection.
        /// </summary>
        /// <param name="hidDeviceObject">Specifies an open handle to a top-level collection.</param>
        /// <param name="attributes">Pointer to a caller-allocated HIDD_ATTRIBUTES structure</param>
        /// <returns>returns TRUE if succeeds; otherwise, it returns FALSE.</returns>
        /// <remarks>ref: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/hidsdi/nf-hidsdi-hidd_getattributes</remarks>
        [DllImport("hid.dll", SetLastError = true)]
        public static extern bool HidD_GetAttributes(IntPtr hidDeviceObject, ref HIDD_ATTRIBUTES attributes);

        /// <summary>
        /// routine returns the embedded string of a top-level collection that identifies the manufacturer's product.
        /// </summary>
        /// <param name="HidDeviceObject">open handle to a top-level collection.</param>
        /// <param name="Buffer">Pointer to a caller-allocated buffer that the routine uses to return the requested product string.</param>
        /// <param name="BufferLength">Specifies the length, in bytes, of a caller-allocated buffer provided at Buffer.</param>
        /// <returns>returns TRUE if succeeds; otherwise, it returns FALSE.</returns>
        /// <remarks>ref: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/hidsdi/nf-hidsdi-hidd_getproductstring</remarks>
        [DllImport("hid.dll", SetLastError = true)]
        public static extern bool HidD_GetProductString(IntPtr HidDeviceObject, IntPtr Buffer, uint BufferLength);

        /// <summary>
        /// routine returns a feature report from a specified top-level collection.
        /// </summary>
        /// <param name="hidDeviceObject">An open handle to a top-level collection.</param>
        /// <param name="lpReportBuffer">caller-allocated HID report buffer to return the specified feature report.</param>
        /// <param name="reportBufferLength">The size of the report buffer in bytes.</param>
        /// <returns>returns TRUE if succeeds; otherwise, it returns FALSE.</returns>
        /// <remarks>ref: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/hidsdi/nf-hidsdi-hidd_getfeature</remarks>
        [DllImport("hid.dll", SetLastError = true)]
        public static extern bool HidD_GetFeature(IntPtr hidDeviceObject, byte[] lpReportBuffer, int reportBufferLength);

        /// <summary>
        /// routine sends a feature report to a top-level collection.
        /// </summary>
        /// <param name="hidDeviceObject">An open handle to a top-level collection.</param>
        /// <param name="lpReportBuffer">caller-allocated feature report buffer that the caller uses to specify a HID report ID.</param>
        /// <param name="reportBufferLength">The size of the report buffer in bytes.</param>
        /// <returns>returns TRUE if succeeds; otherwise, it returns FALSE.</returns>
        /// <remarks>ref: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/hidsdi/nf-hidsdi-hidd_setfeature</remarks>
        [DllImport("hid.dll", SetLastError = true)]
        public static extern bool HidD_SetFeature(IntPtr hidDeviceObject, byte[] lpReportBuffer, int reportBufferLength);
```

```
            #endregion
        }

        #endregion
    }

    #region Error Argument Class

    /// <summary>
    /// Error Event Argument
    /// </summary>
    public class USBrelayEventArgs : EventArgs
    {
        /// <summary>Error Description</summary>
        public string Error { get; set; }

        /// <summary>Method throwing error</summary>
        public string Source { get; set; }

        /// <summary>true if error is an exception type</summary>
        public bool IsException { get; set; }

        /// <summary>if error is exception type, is the exception thrown else null</summary>
        public Exception ExceptionThrown { get; set; }
    }

    #endregion
}
```