

# C# vJoy Feeder/Receptor SDK

Version 2.1.5 – March 2015

## Table of Contents

C# vJoy Feeder SDK.....	1
File listing:.....	2
Fundamentals:.....	3
Reccomended Practices:.....	4
Test vJoy Driver:.....	4
Test Interface DLL matches vJoy Driver:.....	4
Test vJoy Virtual Devices:.....	5
Acquire the vJoy Device:.....	6
Feed vJoy Device:.....	6
Relinquish the vJoy Device:.....	9
Detecting Changes.....	9
Receptor Unit.....	10
Interface Function Reference:.....	12
General driver data.....	12
Write access to vJoy Device.....	12
vJoy Device properties.....	13
Robust write access to vJoy Devices.....	15
FFB Functions.....	16
FFB Helper Functions.....	16
Build & Deploy:.....	21
Location of vJoyInterface.dll.....	21
Logging [2.0.5].....	22
Start/Stop Logging.....	22
Log File.....	23

This SDK includes all that is needed to write a feeder for vJoy version 2.0.1

Check for the latest [SDK](#).

## ***File listing:***

<b>c#</b>	C# SDK (This folder)
<b>x86</b>	Library folder (x86)
<b>x86\vJoyInterface.dll</b>	vJoy Interface DLL file (32-bit version)
<b>x86\vJoyInterface.pdb</b>	Program Database – Use it for debugging (32-bit version)
<b>x86\vJoyInterfaceWrap.dll</b>	vJoy c# wrapper DLL
<b>x64</b>	Library folder (x64)
<b>x64\vJoyInterface.dll</b>	vJoy Interface DLL file (64-bit version)
<b>x64\vJoyInterface.pdb</b>	Program Database – Use it for debugging (64-bit version)
<b>x64\vJoyInterfaceWrap.dll</b>	vJoy c# wrapper DLL
<b>FeederDemoCS</b>	Demo Feeder Project (Visual Studio 2008 Express)
<b>FeederDemoCS\Program.cs</b>	C# code that demonstrates writing a simple feeder
<b>FeederDemoCS\FeederDemoCS.csproj</b>	Demo Feeder Project file (Visual Studio 2008 Express)
<b>FeederDemoCS\FeederDemoCS.sln</b>	Demo Feeder solution file (Visual Studio 2008 Express)
<b>FeederDemoCS\Properties\AssemblyInfo.cs</b>	Demo Feeder properties file (Visual Studio 2008 Express)

## Fundamentals:

This interface and example will enable you to write a C# vJoy feeder.

To write a C/C++ refer to ReadMe file in parent folder.

Features introduced in version 2.0.5 are marked with [2.0.5]

It is advisable to start your feeder from the supplied example and make the needed changes. Here are the five basic steps you might want to follow:

<b>Test Driver:</b>	Check that the driver is installed and enabled. Obtain information about the driver. An installed driver implies at least one vJoy device. [2.0.5] Test if driver matches interface DLL file (vJoyInterface.dll)
<b>Test Virtual Device(s):</b>	Get information regarding one or more devices. Read information about a specific device capabilities: Axes, buttons and POV hat switches.
<b>Device acquisition:</b>	Obtain status of a vJoy device. Acquire the device if the device status is <i>owned</i> or is <i>free</i> .
<b>Updating:</b>	Inject <u>position data</u> to a device (as long as the device is owned by the feeder). Position data includes the position of the axes, state of the buttons and state of the POV hat switches.
<b>Relinquishing the device:</b>	The device is <i>owned</i> by the feeder and cannot be fed by another application until relinquished.

### Notes:

1. The interface library file (vJoyInterface.dll) and the wrapper library (vJoyInterfaceWrap.dll) must be placed together.
2. The feeder must use **using** directive:  
`using vJoyInterfaceWrap;`
3. Wrapper only class is vJoy: Start your application by creating a **vJoy** object:  
`joystick = new vJoy();`

## ***Reccomended Practices:***

### ***Test vJoy Driver:***

Before you start, check if the vJoy driver is installed and check that it is what you expected:

```
joystick = new vJoy();

// Get the driver attributes (Vendor ID, Product ID, Version Number)
if (!joystick.vJoyEnabled())
{
    Console.WriteLine("vJoy driver not enabled: Failed Getting vJoy attributes.\n");
    return;
}
else
    Console.WriteLine("Vendor: {0}\nProduct :{1}\nVersion Number:{2}\n",
        joystick.GetvJoyManufacturerString(),
        joystick.GetvJoyProductString(),
        joystick.GetvJoySerialNumberString());
```

### ***Test Interface DLL matches vJoy Driver:***

**[2.0.5]**

Before you start, check if file vJoyInterface.dll that you link to matches the vJoy driver that is installed. It is recommended that their version numbers will be identical.

```
// Test if DLL matches the driver
UInt32 DllVer = 0, DrvVer = 0;
bool match = joystick.DriverMatch(ref DllVer, ref DrvVer);
if (match)
    Console.WriteLine("Version of Driver Matches DLL Version ({0:X})\n", DllVer);
else
    Console.WriteLine("Version of Driver ({0:X}) does NOT match DLL Version ({1:X})\n",
        DrvVer, DllVer);
```

If you are not interested in the actual values of the respective version numbers, you can simplify your code by passing NULL to both function parameters.

## *Test vJoy Virtual Devices:*

Check if device is installed and what its state:

```
// Get the state of the requested device
VjdStat status = joystick.GetVJDStatus(id);

switch (status)
{
    case VjdStat.VJD_STAT_OWN:
        Console.WriteLine("vJoy Device {0} is already owned by this feeder\n", id);
        break;
    case VjdStat.VJD_STAT_FREE:
        Console.WriteLine("vJoy Device {0} is free\n", id);
        break;
    case VjdStat.VJD_STAT_BUSY:
        Console.WriteLine(
            "vJoy Device {0} is already owned by another feeder\nCannot continue\n", id);
        return;
    case VjdStat.VJD_STAT_MISS:
        Console.WriteLine(
            "vJoy Device {0} is not installed or disabled\nCannot continue\n", id);
        return;
    default:
        Console.WriteLine("vJoy Device {0} general error\nCannot continue\n", id);
        return;
};
```

Now make sure that the axes, buttons (and POV hat switches) are as expected:

```
///// vJoy Device properties
int nBtn = joystick.GetVJDButtonNumber(id);
int nDPov = joystick.GetVJDDiscPovNumber(id);
int nCPov = joystick.GetVJDContPovNumber(id);
bool X_Exist = joystick.GetVJDAxisExist(id, HID_USAGES.HID_USAGE_X);
bool Y_Exist = joystick.GetVJDAxisExist(id, HID_USAGES.HID_USAGE_Y);
bool Z_Exist = joystick.GetVJDAxisExist(id, HID_USAGES.HID_USAGE_Z);
bool RX_Exist = joystick.GetVJDAxisExist(id, HID_USAGES.HID_USAGE_RX);

prt = String.Format("Device[{0}]: Buttons={1}; DiscPOVs:{2}; ContPOVs:{3}", \
                    id, nBtn, nDPov, nCPov);
Console.WriteLine(prt);
```

### *Acquire the vJoy Device:*

Until now you just made inquiries about the system and about the vJoy device status. In order to change the position of the vJoy device you need to Acquire it (if it is not already owned):

```
///// Write access to vJoy Device - Basic
VjdStat status;
status = joystick.GetVJDStatus(id);
// Acquire the target
if ((status == VjdStat.VJD_STAT_OWN) ||
    ((status == VjdStat.VJD_STAT_FREE) && (! joystick.AcquireVJD(id))))
    prt = String.Format("Failed to acquire vJoy device number {0}.", id);
else
    prt = String.Format("Acquired: vJoy device number {0}.", id);
Console.WriteLine(prt);
```

### *Feed vJoy Device:*

The time has come to do some real work: feed the vJoy device with position data.

There are two approaches:

1. **Efficient:** Collect position data, place the data in a position structure then finally send the data to the device.
2. **Robust:** Reset the device once then send the position data for every control (axis, button,POV) at a time.

The first approach is more efficient but requires more code to deal with the position structure that may change in the future.

The second approach hides the details of the data fed to the device at the expence of exessive calls to the device driver.

## Efficient:

```
while (true)
{
    // Feed the device id
    iReport.bDevice = (byte)id;
    // Feed position data per axis
    iReport.AxisX = X;
    iReport.AxisY = Y;
    iReport.AxisZ = Z;
    iReport.AxisZRot = ZR;
    iReport.AxisXRot = XR;
    // Set buttons one by one
    iReport.Buttons = (uint)(0x1 << (int)(count / 20));

    if (ContPovNumber>0)
    {
        // Make Continuous POV Hat spin
        iReport.bHats = (count*70);
        iReport.bHatsEx1 = (count*70)+3000;
        iReport.bHatsEx2 = (count*70)+5000;
        iReport.bHatsEx3 = 15000 - (count*70);
        if ((count*70) > 36000)
        {
            iReport.bHats = 0xFFFFFFFF; // Neutral state
            iReport.bHatsEx1 = 0xFFFFFFFF; // Neutral state
            iReport.bHatsEx2 = 0xFFFFFFFF; // Neutral state
            iReport.bHatsEx3 = 0xFFFFFFFF; // Neutral state
        };
    }
    else
    {
        // Make 5-position POV Hat spin
        pov[0] = (byte)(((count / 20) + 0) % 4);
        pov[1] = (byte)(((count / 20) + 1) % 4);
        pov[2] = (byte)(((count / 20) + 2) % 4);
        pov[3] = (byte)(((count / 20) + 3) % 4);

        iReport.bHats =
            (uint)(pov[3]<<12) | (uint)(pov[2]<<8) | (uint)(pov[1]<<4) | (uint)pov[0];
        if ((count) > 550)
            iReport.bHats = 0xFFFFFFFF; // Neutral state
    };

    /** Feed the driver with the position packet */
    joystick.UpdateVJD(id, ref iReport)
    System.Threading.Thread.Sleep(20);
    count++;
    if (count > 640) count = 0;

    X += 150; if (X > maxval) X = 0;
    Y += 250; if (Y > maxval) Y = 0;
    Z += 350; if (Z > maxval) Z = 0;
    XR += 220; if (XR > maxval) XR = 0;
    ZR += 200; if (ZR > maxval) ZR = 0;

}; // While
```

if the structure changes in the future then the code will have to change too.

## Robust:

```
joystick.ResetVJD(id); // Reset this device to default values
while (true) // Feed the device in endless loop
{
    // Set position of 4 axes
    res = joystick.SetAxis(X, id, HID_USAGES.HID_USAGE_X);
    res = joystick.SetAxis(Y, id, HID_USAGES.HID_USAGE_Y);
    res = joystick.SetAxis(Z, id, HID_USAGES.HID_USAGE_Z);
    res = joystick.SetAxis(XR, id, HID_USAGES.HID_USAGE_RX);
    res = joystick.SetAxis(ZR, id, HID_USAGES.HID_USAGE_RZ);

    // Press/Release Buttons
    res = joystick.SetBtn(true, id, count / 50);
    res = joystick.SetBtn(false, id, 1 + count / 50);

    // If Continuous POV hat switches installed - make them go round
    // For high values - put the switches in neutral state
    if (ContPovNumber > 0)
    {
        if ((count * 70) < 30000)
        {
            res = joystick.SetContPov(((int)count * 70), id, 1);
            res = joystick.SetContPov(((int)count * 70) + 2000, id, 2);
            res = joystick.SetContPov(((int)count * 70) + 4000, id, 3);
            res = joystick.SetContPov(((int)count * 70) + 6000, id, 4);
        }
        else
        {
            res = joystick.SetContPov(-1, id, 1);
            res = joystick.SetContPov(-1, id, 2);
            res = joystick.SetContPov(-1, id, 3);
            res = joystick.SetContPov(-1, id, 4);
        }
    };

    // If Discrete POV hat switches installed - make them go round
    // From time to time - put the switches in neutral state
    if (DiscPovNumber > 0)
    {
        if (count < 550)
        {
            joystick.SetDiscPov((((int)count / 20) + 0) % 4, id, 1);
            joystick.SetDiscPov((((int)count / 20) + 1) % 4, id, 2);
            joystick.SetDiscPov((((int)count / 20) + 2) % 4, id, 3);
            joystick.SetDiscPov((((int)count / 20) + 3) % 4, id, 4);
        }
        else
        {
            joystick.SetDiscPov(-1, id, 1);
            joystick.SetDiscPov(-1, id, 2);
            joystick.SetDiscPov(-1, id, 3);
            joystick.SetDiscPov(-1, id, 4);
        }
    };

    System.Threading.Thread.Sleep(20);
} // While (Robust)
```



This code is readable and does not rely on any specific structure. However, the driver is updated with every *SetAxis()* and every *SetBtn()*.

### ***Relinquish the vJoy Device:***

You must relinquish the device when the driver exits:

```
joystick.RelinquishVJD(iInterface);
```

## ***Detecting Changes***

[2.0.5]

It is sometimes necessary to detect changes in the number of available vJoy devices.

You may define a callback function that will be called whenever such a change occurs. In order for it to be called, the user-defined callback function should first be registered by calling function *RegisterRemovalCB* as in the following example:

```
joystick.RegisterRemovalCB(ChangedCB, label2);
```

Where *ChangedCB* is the user-defined callback function and *label2* is some C# object.

An example to an implementation of the user-defined callback function *ChangedCB*:

```
void CALLBACK ChangedCB(bool Removed, bool First, object userData)
{
    Label l = userData as Label;
    int id = 1;
    int nBtn = joystick.GetVJDButtonNumber(id);

    // Final values after the last arrival
    if (!removal && !first)
    {
        prt = String.Format("Device[{0}]: Buttons={1}" id, nBtn);
        l.Text = prt;
    }

    // Temporary message during intermediate states
    else
    {
        prt = String.Format("Device[{0}]: Wait ...", id);
        l.Text = prt;
    }
}
```

This function is called when a process of vJoy device removal starts or ends and when a process of vJoy device arrival starts or ends. The function must return as soon as possible.

- When a process of vJoy device removal starts, Parameter *Removed*=TRUE and parameter *First*=TRUE.
- When a process of vJoy device removal ends, Parameter *Removed*=TRUE and parameter *First*=FALSE.
- When a process of vJoy device arrival starts, Parameter *Removed*=FALSE and parameter *First*=TRUE.
- When a process of vJoy device arrival ends, Parameter *Removed*= FALSE and parameter *First*=FALSE .

Parameter *userData* is always an object registered as second parameter of function *RegisterRemovalCB*.

## Receptor Unit

### [2.1.5]

To take advantage of vJoy ability to process **Force Feedback** (FFB) data, you need to add a **receptor** unit to the feeder.

The receptor unit receives the FFB data from a **source application**, and processes the FFB data. The data can be passed on to another entity (e.g. a physical joystick) or processed in place.

The Receptor is activated by **Acquiring** one or more vJoy devices (if not acquired yet), then **Starting** the devices' FFB capabilities and finally **registering** a single user-defined FFB callback function.

Once registered, the user-defined FFB callback function is called by a vJoy device every time a new FFB packet arrives from the **source application**. This function is called in the application thread and is **blocking**. This means that you must return from the FFB callback function ASAP – never wait in this function for the next FFB packet!

The SDK offers you a wide range of FFB helper-functions to process the FFB packet and a demo application that demonstrates the usage of the helper-functions. The helper-functions are efficient and can be used inside the FFB callback function.

Start a vJoy device' FFB capabilities by calling function *FfbStart()*.

Register a user-defined FFB callback function by calling *FfbRegisterGenCB()*.

```
public FfbInterface(TesterForm dialog)
{
    dlg = dialog;
    joystick = dialog.joystick;
    // Start FFB Mechanism
    if (!joystick.FfbStart(id))
        throw new Exception(String.Format("Cannot start Forcefeedback on device {0}", id));

    // Convert Form to pointer and pass it as user data to the callback function
    GCHandle h = GCHandle.Alloc(dialog);
    IntPtr parameter = (IntPtr)h;

    // Register the callback function & pass the dialog box object
    joystick.FfbRegisterGenCB(OnEffectObj, dialog);
}
```

The FFB callback function is defined by the user. The function interface is as follows:

```
private static void OnEffectObj(IntPtr data, object userData)
```

Where *OnEffectObj* is the name of the user-defined callback function. Parameter *data* is a pointer to a C-language data packet (Type FFB\_DATA) arriving from the vJoy device. Parameter *userData* is a user-defined object. You are not required to understand the structure of the FFB\_DATA structure – just pass it to the various FFB helper-functions.

### Structure FFB\_DATA:

Normally, you are not required to understand this structure as it is usually passed to the various helper function. However, you might want to access the raw FFB packet.

```
typedef struct _FFB_DATA {
    ULONG size;
    ULONG cmd;
    UCHAR *data;
} FFB_DATA;
```

FFB\_DATA Fields:

**size:** Size of FFB\_DATA structure in bytes

**cmd:** Reserved

**data:** Array of size-8 bytes holding the FFB packet.

### **FFB Helper Functions:**

These functions receive a pointer to FFB\_DATA as their first parameter and return a **uint** status. The returned value is either **ERROR\_SUCCESS** on success or other values on failure.

Use these functions to analyze the FFB data packets avoiding direct access to the raw FFB\_DATA structure.

## Interface Function Reference:

### General driver data

The following functions return general data regarding the installed vJoy device driver. It is recommended to call them when starting your feeder.

```
bool vJoyEnabled();
```

Returns **true** if vJoy version 2.x is installed and enabled.

```
short GetvJoyVersion();
```

Return the version number of the installed vJoy. To be used only after `vJoyEnabled()`

```
string GetvJoyProductString();
string GetvJoyManufacturerString();
string GetvJoySerialNumberString();
```

To be used only after `vJoyEnabled()`

[2.0.5]

```
bool DriverMatch(ref UInt32 DllVer, ref UInt32 DrvVer);
```

Returns **TRUE** if vJoyInterface.dll file version and vJoy Driver version are identical. Otherwise returns **FALSE**.

Optional output parameter *DllVer*: If a reference to 32-bit unsigned integer is passed then the value of the **DLL file** version will be written to this parameter (e.g. 0x205).

Optional output parameter *DrvVer*: If a reference to 32-bit unsigned integer is passed then the value of the **Driver** version will be written to this parameter (e.g. 0x205).

[2.0.5]

```
void RegisterRemovalCB(RemovalCbFunc cb, object data);
```

This function registers a user-defined **cb** callback function that is called everytime a vJoy device is added or removed. Parameter *cb* is a reference to the user-defined callback function.

Parameter *data* is a pointer to a user-defined object. The callback function receives this object as its third parameter.

The user-defined callback function type definition:

```
void RemovalCbFunc(bool complete, bool First, object userData);
```

More in section [Detecting Changes](#).

### Write access to vJoy Device

The following functions access the virtual device by its ID (rID). The value of rID may vary between 1 and 16. There may be more than one virtual device installed on a given system.

VJD stands for *Virtual Joystick Device*.

```
VjdStat GetVJDStatus(UInt32 rID);
```

Returns the status of the specified device

The status can be one of the following values:

- VJD\_STAT\_OWN // The vJoy Device is owned by this application.
- VJD\_STAT\_FREE // The vJoy Device is NOT owned by any application (including this one).
- VJD\_STAT\_BUSY // The vJoy Device is owned by another application.  
// It cannot be acquired by this application.
- VJD\_STAT\_MISS // The vJoy Device is missing. It either does not exist or the driver is disabled.
- VJD\_STAT\_UNKN // Unknown

```
bool AcquireVJD(UInt32 rID);
```

Acquire the specified device.

Only a device in state VJD\_STAT\_FREE can be acquired.

If acquisition is successful the function returns TRUE and the device status becomes VJD\_STAT\_OWN.

```
void RelinquishVJD (UInt32 rID);
```

Relinquish the previously acquired specified device.

Use only when device is state VJD\_STAT\_OWN.

State becomes VJD\_STAT\_FREE immediately after this function returns.

```
bool UpdateVJD (UInt32 rID, ref JoystickState pData);
```

Update the position data of the specified device.

Use only after device has been successfully acquired.

Input parameter is a reference to structure of type JoystickState that holds the position data.

Returns true if device updated.

### *vJoy Device properties*

The following functions receive the virtual device ID (rID) and return the relevant data.

The value of rID may vary between 1 and 16. There may be more than one virtual device installed on a given system.

The return values are meaningful only if the specified device exists

VJD stands for Virtual Joystick Device.

```
int GetVJDButtonNumber (uint rID);
```

Returns the number of buttons in the specified device.

If function succeeds, returns the number of buttons in the specified device. Valid values are 0 to 128

**[2.0.5]** If function fails, returns a negative error code:

- NO\_HANDLE\_BY\_INDEX
- BAD\_PREPARED\_DATA
- NO\_CAPS
- BAD\_N\_BTN\_CAPS
- BAD\_BTN\_CAPS
- BAD\_BTN\_RANGE

```
int GetVJDDiscPovNumber (uint rID);
```

Returns the number of discrete-type POV hats in the specified device

Discrete-type POV Hat values may be North, East, South, West or neutral

Valid values are 0 to 4 (in version 2.0.1)

```
int GetVJDContPovNumber (uint rID);
```

Returns the number of continuous-type POV hats in the specified device

continuous-type POV Hat values may be 0 to 35900

Valid values are 0 to 4 (in version 2.0.1)

```
bool GetVJDAxisExist (UInt32 rID, HID_USAGES Axis);
```

Returns TRUE if the specified axis exists in the specified device

Axis values can be:

HID_USAGES.HID_USAGE_X	// X Axis
HID_USAGES.HID_USAGE_Y	// Y Axis
HID_USAGES.HID_USAGE_Z	// Z Axis
HID_USAGES.HID_USAGE_RX	// Rx Axis
HID_USAGES.HID_USAGE_RY	// Ry Axis
HID_USAGES.HID_USAGE_RZ	// Rz Axis

```
HID_USAGES.HID_USAGE_SL0      // Slider 0
HID_USAGES.HID_USAGE_SL1      // Slider 1
HID_USAGES.HID_USAGE_WHL // Wheel
```

## Robust write access to vJoy Devices

The following functions receive the virtual device ID (rID) and return the relevant data.

These functions hide the details of the position data structure by allowing you to alter the value of a specific control. The downside of these functions is that you inject the data to the device serially as opposed to function *UpdateVJD()*. The value of rID may vary between 1 and 16. There may be more than one virtual device installed on a given system.

```
bool ResetVJD (UInt32 rID) ;
```

Resets all the controls of the specified device to a set of values.

These values are hard coded in the interface DLL and are currently set as follows:

- Axes X,Y & Z: Middle point.
- All other axes: 0.
- POV Switches: Neutral (-1).
- Buttons: Not Pressed (0).

```
bool ResetAll () ;
```

Resets all the controls of the all devices to a set of values.

See function Reset VJD for details.

```
bool ResetButtons (UInt32 rID) ;
```

Resets all buttons (To 0) in the specified device.

```
bool ResetPovs (UInt32 rID) ;
```

Resets all POV Switches (To -1) in the specified device.

```
bool SetAxis (Int32 Value, UInt32 rID, HID_USAGES Axis) ;
```

Write Value to a given axis defined in the specified VDJ.

Axis values can be:

```
HID_USAGES.HID_USAGE_X           // X Axis
HID_USAGES.HID_USAGE_Y           // Y Axis
HID_USAGES.HID_USAGE_Z           // Z Axis
HID_USAGES.HID_USAGE_RX          // Rx Axis
HID_USAGES.HID_USAGE_RY          // Ry Axis
HID_USAGES.HID_USAGE_RZ          // Rz Axis
HID_USAGES.HID_USAGE_SL0         // Slider 0
HID_USAGES.HID_USAGE_SL1         // Slider 1
HID_USAGES.HID_USAGE_WHL         // Wheel
```

```
bool SetBtn (bool Value, UInt32 rID, uint nBtn) ;
```

Write Value (true or false) to a given button defined in the specified VDJ.

nBtn can range 1-32

```
bool SetDiscPov (Int32 Value, UInt32 rID, uint nPov) ;
```

Write Value to a given discrete POV defined in the specified VDJ

**Value** can be one of the following:

- 0: North (or Forwards)
- 1: East (or Right)
- 2: South (or backwards)
- 3: West (or left)
- 1: Neutral (Nothing pressed)

**nPov** selects the destination POV Switch. It can be 1 to 4

```
bool SetContPov(Int32 Value, UInt32 rID, uint nPov);
```

Write Value to a given continuous POV defined in the specified VDJ

**Value** can be in the range: -1 to 35999. It is measured in units of one-hundredth a degree. -1 means Neutral (Nothing pressed).

**nPov** selects the destination POV Switch. It can be 1 to 4

## FFB Functions

The following functions are used for accessing and manipulating Force Feedback data.

```
void FfbRegisterGenCB(FfbCbFunc cb, object data);
```

Register a FFB callback function that will be called by the driver every time a FFB data packet arrives. For additional information see [Receptor Unit section](#).

```
bool FfbStart(UInt32 rID);
```

Enable the FFB mechanism of the specified VDJ.

Return **true** on success. Otherwise return **false**.

```
bool FfbStop(UInt32 rID);
```

Disable the FFB mechanism of the specified VDJ.

## FFB Helper Functions

```
UInt32 Ffb_h_DeviceID(IntPtr Packet, ref int DeviceID);
```

Get the origin of the FFB data packet.

If valid device ID was found then returns ERROR\_SUCCESS and sets the ID (Range 1-15) in **DeviceID**.

If Packet is NULL then returns ERROR\_INVALID\_PARAMETER. DeviceID is undefined.

If Packet is malformed or Device ID is out of range then returns ERROR\_INVALID\_DATA. DeviceID is undefined.

```
UInt32 Ffb_h_Type(IntPtr Packet, ref FFBPType Type);
```

Get the type of the FFB data packet.

**Type** may be one of the following:

```
// Write
PT_EFFREP      // Usage Set Effect Report
PT_ENVREP      // Usage Set Envelope Report
PT_CONDREP     // Usage Set Condition Report
PT_PRIDREP     // Usage Set Periodic Report
PT_CONSTREP    // Usage Set Constant Force Report
PT_RAMPREP     // Usage Set Ramp Force Report
PT_CSTMREP     // Usage Custom Force Data Report
PT_SMPLREP     // Usage Download Force Sample
PT_EFOPREP     // Usage Effect Operation Report
PT_BLKFRREP    // Usage PID Block Free Report
PT_CTRLREP     // Usage PID Device Control
PT_GAINREP     // Usage Device Gain Report
PT_SETCREP     // Usage Set Custom Force Report

// Feature
PT_NEWEFFREP   // Usage Create New Effect Report
PT_BLKLDREP    // Usage Block Load Report
PT_POOLREP     // Usage PID Pool Report
```

If valid Type was found then returns ERROR\_SUCCESS and sets **Type**.

If Packet is NULL then returns ERROR\_INVALID\_PARAMETER. Feature is undefined.

If Packet is malformed then returns ERROR\_INVALID\_DATA. Feature is undefined.



```
UInt32 Ffb_h_Packet(IntPtr Packet, ref UInt32 Type, ref Int32 DataSize, ref Byte[] Data);
```

Extract the raw FFB data packet and the command type (Write/Set Feature).

If valid Packet was found then returns ERROR\_SUCCESS and -

Sets **Type** to IOCTL value (Expected values are IOCTL\_HID\_WRITE\_REPORT and IOCTL\_HID\_SET\_FEATURE).

Sets **DataSize** to the size (in bytes) of the payload data (FFB\_DATA.data ).

Sets **Data** to the payload data (FFB\_DATA.data ) - this is an array of bytes.

If Packet is NULL then returns ERROR\_INVALID\_PARAMETER. Output parameters are undefined.

If Packet is malformed then returns ERROR\_INVALID\_DATA. Output parameters are undefined.

```
UInt32 Ffb_h_EBI(IntPtr Packet, ref Int32 Index);
```

Get the Effect Block Index

If valid Packet was found then returns ERROR\_SUCCESS and sets **Index** to the value of Effect Block Index (if applicable). Expected value is '1'.

If Packet is NULL then returns ERROR\_INVALID\_PARAMETER. Output parameters are undefined.

If Packet is malformed or does not contain an Effect Block Index then returns ERROR\_INVALID\_DATA. Output parameters are undefined.

```
UInt32 Ffb_h_Eff_Const(IntPtr Packet, ref FFB_EFF_CONST Effect);
```

Get parameters of an Effect of type Constant (PT\_EFFREP)

Effect structure (FFB\_EFF\_CONST) definition:

```
public struct FFB_EFF_CONST
{
    [FieldOffset(0)]
    public Byte EffectBlockIndex;
    [FieldOffset(4)]
    public FFBEType EffectType;
    [FieldOffset(8)]
    public UInt16 Duration; // Value in milliseconds. 0xFFFF means infinite
    [FieldOffset(10)]
    public UInt16 TrigerRpt;
    [FieldOffset(12)]
    public UInt16 SamplePrd;
    [FieldOffset(14)]
    public Byte Gain;
    [FieldOffset(15)]
    public Byte TrigerBtn;
    [FieldOffset(16)]
    public bool Polar; // How to interpret force direction Polar (0-360°)
    //or Cartesian (X,Y)
    [FieldOffset(20)]
    public Byte Direction; // Polar direction: (0x00-0xFF correspond to 0-360°)
    [FieldOffset(20)]
    public Byte DirX; // X direction: Positive values are To the right
    // of the center (X); Negative are Two's complement
    [FieldOffset(21)]
    public Byte DirY; // Y direction: Positive values are below the center (Y);
    // Negative are Two's complement
}
```

If Constant Effect Packet was found then returns ERROR\_SUCCESS and fills structure **Effect**

If Packet is NULL then returns ERROR\_INVALID\_PARAMETER. Output parameters are undefined.

If Packet is malformed then returns ERROR\_INVALID\_DATA. Output parameters are undefined.

```
UInt32 Ffb_h_Eff_Ramp(IntPtr Packet, ref FFB_EFF_RAMP RampEffect);
```

Get parameters of an Effect of type Ramp (PT\_RAMPREP)

Effect structure (FFB\_EFF\_RAMP) definition:

```
public struct FFB_EFF_RAMP
{
    public Byte EffectBlockIndex;
    public Byte Start; // The Normalized magnitude at the start of the effect
    public Byte End;   // The Normalized magnitude at the end of the effect
}
```

If Ramp effect Packet was found then returns ERROR\_SUCCESS and fills structure Effect.

If Packet is NULL then returns ERROR\_INVALID\_PARAMETER. Output parameters are undefined.

If Packet is malformed then returns ERROR\_INVALID\_DATA. Output parameters are undefined.

```
UInt32 Ffb_h_EffOp(IntPtr Packet, ref FFB_EFF_OP Operation);
```

Get parameters of an Effect of type Operation (PT\_EFOPREP) that describe the effect operation (Start/Solo/Stop) and loop count.

Effect structure (FFB\_EFF\_OP) definition:

```
public struct FFB_EFF_OP
{
    [FieldOffset(0)]
    public Byte EffectBlockIndex;
    [FieldOffset(4)]
    public FFBOP EffectOp;
    [FieldOffset(8)]
    public Byte LoopCount;
}
```

If Operation Effect Packet was found then returns ERROR\_SUCCESS and fills structure Operation- this structure holds Effect Block Index, Operation(Start, Start Solo, Stop) and Loop Count.

If Packet is NULL then returns ERROR\_INVALID\_PARAMETER. Output parameters are undefined.

If Packet is malformed then returns ERROR\_INVALID\_DATA. Output parameters are undefined.

```
UInt32 Ffb_h_Eff_Period(IntPtr Packet, ref FFB_EFF_PERIOD Effect);
```

Get parameters of an Effect of type Periodic (PT\_PRIDREP) that describe the periodic attribute of an effect.

Effect structure (FFB\_EFF\_PERIOD) definition:

```
public struct FFB_EFF_PERIOD
{
    [FieldOffset(0)]
    public Byte EffectBlockIndex;
    [FieldOffset(1)]
    public Byte Magnitude;
    [FieldOffset(2)]
    public Byte Offset;
    [FieldOffset(3)]
    public Byte Phase;
    [FieldOffset(4)]
    public UInt16 Period;
}
```

If Periodic Packet was found then returns ERROR\_SUCCESS and fills structure Effect – this structure holds Effect Block Index, Magnitude, Offset, Phase and period.

If Packet is NULL then returns ERROR\_INVALID\_PARAMETER. Output parameters are undefined.  
If Packet is malformed then returns ERROR\_INVALID\_DATA. Output parameters are undefined.

```
UInt32 Ffb_h_Eff_Cond(IntPtr Packet, ref FFB_EFF_COND Condition);
```

Get parameters of an Effect of type Conditional (PT\_CONDREP).

Effect structure (FFB\_EFF\_COND) definition:

```
public struct FFB_EFF_COND
{
    public Byte EffectBlockIndex;
    [FieldOffset(4)]
    public bool isY;
    [FieldOffset(8)]
    public Byte CenterPointOffset; // CP Offset: Range -10000 to 10000
    [FieldOffset(9)]
    public Byte PosCoeff; // Positive Coefficient: Range -10000 to 10000
    [FieldOffset(10)]
    public Byte NegCoeff; // Negative Coefficient: Range -10000 to 10000
    [FieldOffset(11)]
    public Byte PosSatur; // Positive Saturation: Range 0 - 10000
    [FieldOffset(12)]
    public Byte NegSatur; // Negative Saturation: Range 0 - 10000
    [FieldOffset(13)]
    public Byte DeadBand; // Dead Band: : Range 0 - 10000
}
```

If Condition Packet was found then returns ERROR\_SUCCESS and fills structure Condition - this structure holds Effect Block Index, Direction (X/Y), Centre Point Offset, Dead Band and other conditions.

If Packet is NULL then returns ERROR\_INVALID\_PARAMETER. Output parameters are undefined.

If Packet is malformed then returns ERROR\_INVALID\_DATA. Output parameters are undefined.

```
UInt32 Ffb_h_Eff_Envlp(IntPtr Packet, ref FFB_EFF_ENVLP Envelope);
```

Get parameters of an Effect of type Envelope (PT\_ENVREP).

Effect structure (FFB\_EFF\_ENVLP) definition:

```
public struct FFB_EFF_ENVLP
{
    [FieldOffset(0)]
    public Byte EffectBlockIndex;
    [FieldOffset(1)]
    public Byte AttackLevel;
    [FieldOffset(2)]
    public Byte FadeLevel;
    [FieldOffset(4)]
    public UInt16 AttackTime;
    [FieldOffset(6)]
    public UInt16 FadeTime;
}
```

If Envelope Packet was found then returns ERROR\_SUCCESS and fills structure Envelope

If Packet is NULL then returns ERROR\_INVALID\_PARAMETER. Output parameters are undefined.

If Packet is malformed then returns ERROR\_INVALID\_DATA. Output parameters are undefined.

```
UInt32 Ffb_h_EffNew(IntPtr Packet, ref FFBType Effect);
```

Get the type of the next effect. Parameter **Effect** can get one of the following values:

```
ET_NONE      =      0      //      No Force
```

```

ET_CONST      = 1    // Constant Force
ET_RAMP       = 2    // Ramp
ET_SQR        = 3    // Square
ET_SINE       = 4    // Sine
ET_TRNGL      = 5    // Triangle
ET_STUP       = 6    // Sawtooth Up
ET_STDN       = 7    // Sawtooth Down
ET_SPRNG      = 8    // Spring
ET_DMPR       = 9    // Damper
ET_INRT       = 10   // Inertia
ET_FRCTN      = 11   // Friction
ET_CSTM       = 12   // Custom Force Data

```

If valid Packet was found then returns ERROR\_SUCCESS and sets the new **Effect** type

If Packet is NULL then returns ERROR\_INVALID\_PARAMETER. Output parameters are undefined.

If Packet is malformed then returns ERROR\_INVALID\_DATA. Output parameters are undefined.

```

UInt32 Ffb_h_DevCtrl(IntPtr Packet, ref FFB_CTRL Control);

```

Get device-wide control instructions. **Control** can get one of the following values:

```

CTRL_ENACT      = 1    // Enable all device actuators.
CTRL_DISACT     = 2    // Disable all the device actuators.
CTRL_STOPALL    = 3    // Stop All Effects Issues a stop on every running effect.
CTRL_DEVRST     = 4    // Device Reset
                    // Clears any device paused condition,
                    // enables all actuators and clears all effects from memory.

CTRL_DEVPAUSE   = 5    // Device Pause
                    // All effects on the device are paused
                    // at the current time step.

CTRL_DEVCONT    = 6    // Device Continue
                    // All effects that running when the
                    // device was paused are restarted from their last time step.

```

```

UInt32 Ffb_h_DevGain(IntPtr Packet, ref Byte Gain);

```

Get device Global gain in parameter **Gain**.

If valid Packet was found then returns ERROR\_SUCCESS and gets the device global gain.

If Packet is NULL then returns ERROR\_INVALID\_PARAMETER. Output parameters are undefined.

If Packet is malformed then returns ERROR\_INVALID\_DATA. Output parameters are undefined.

**Build & Deploy:**

The quickest way to build your project is to start from the supplied demo project written in C# under Visual Studio 2008 Express. It will compile as-is for x86/x64 target machines.

When you deploy your feeder, don't forget to supply the user with files vJoyInterface.dll and vJoyInterfaceWrap.dll of the correct bitness. They should be located on the target machine's DLL search path. Usually meaning the same directory as your feeder.

*Location of vJoyInterface.dll*

[2.0.5]

vJoy folders are pointed at by registry Entries located under key:  
HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\{8E31F76F-74C3-47F1-9550-E041EEDC5FBB}\_is1

Entry	Default Value	Notes
InstallLocation	C:\Program Files\vJoy\	vJoy root folder: Location of vJoy driver installer and uninstaller
DllX64Location	C:\Program Files\vJoy\x64	<ul style="list-style-type: none"><li>• Location of 64-bit utilities and libraries</li><li>• Only on 64-bit Machines</li></ul>
DllX86Location	C:\Program Files\vJoy\x86	<ul style="list-style-type: none"><li>• Location of 32-bit utilities and libraries</li><li>• On 32-bit and 64-bit Machines</li></ul>

Note that on 64-bit machine you are capable of developing both 32-bit and 64-bit feeders.  
You can assume that DLL files are located in sub-folders x64 and x32 under vJoy root folder.

## Logging [2.0.5]

Logging of vJoyInterface.dll activity into a log file is an option.

Use this feature for debugging purposes only. It accumulates data into the log file and generally slows down the system.

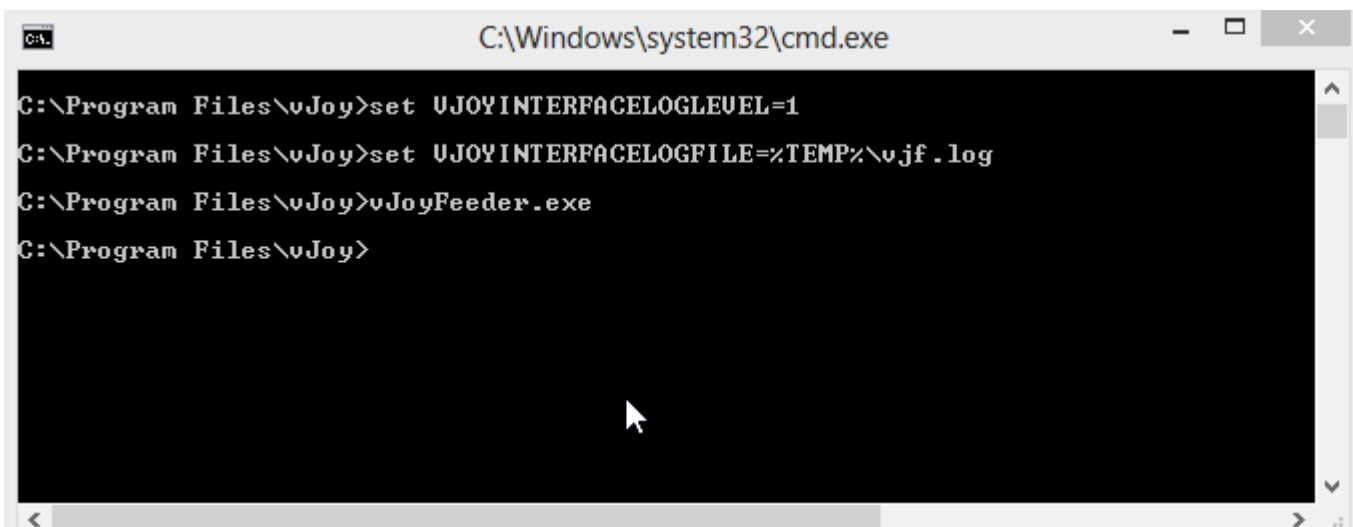
This feature is intended both for helping you develop your feeder and to collect data at the user's location – provided the user is willing to trigger logging for you. By default, logging state is OFF.

### Start/Stop Logging.

To start logging, there are one or two system environment variables that have to be changed before the feeder (Or any other application calling vJoyInterface.dll) is started.

- VJOYINTERFACELOGLEVEL:  
Any positive value will trigger logging.  
Set to 0 to stop logging.
- VJOYINTERFACELOGFILE (Optional):  
If set, this is the full path to the log file.  
Default Path: %TEMP%\vJoyInterface.log

### Example:



```
C:\Windows\system32\cmd.exe

C:\Program Files\vJoy>set VJOYINTERFACELOGLEVEL=1
C:\Program Files\vJoy>set VJOYINTERFACELOGFILE=%TEMP%\vjf.log
C:\Program Files\vJoy>vJoyFeeder.exe
C:\Program Files\vJoy>
```

### Notes:

- This session of vJoyFeeder will log into the given file.
- If the file exists, it will append the new data to the existing file.
- To stop logging, kill vJoyFeeder and then close this window.

### Limitations:

- Logging begins on the application's first call to function AcquireVJD()
- If VJOYINTERFACELOGFILE is not defined, all applications that call AcquireVJD() will write to the same default output file.

## Log File

The log file contains information about vJoyInterface.dll values, states and functions. It is mainly useful in conjunction with the code.

Here is a snippet of a log file:

```
[04988]Info: GetHandleByIndex(index=3) - Starting

[04988]Info: GetHandleByIndex(index=3) - Exit OK (Handle to \\?\hid#hidclass&col01#1&2d595ca7&db&0000#{4d1e55b2-f16f-11cf-88cb-001111000030})

[03088]Process:"D:\WinDDK\vJoy-2.0.5\apps\vJoyFeeder\x64\Release\vJoyFeeder.exe"

[03088]Info: OpenDeviceInterface(9) - DevicePath[0]=\\?\{d6e55ca0-1a2e-4234-aaf3-3852170b492f}\vjoyrawpdo#1&2d595ca7&db&vjoyinstance00#{781ef630-72b2-11d2-b852-00c04fad5101}\device_001

[03088]Info: isRawDevice(9) - Compare \\?\{d6e55ca0-1a2e-4234-aaf3-3852170b492f}\vjoyrawpdo#1&2d595ca7&db&vjoyinstance00#{781ef630-72b2-11d2-b852-00c04fad5101}\device_001 with 001(d=1)

[03088]Info: OpenDeviceInterface(9) - DevicePath[1]=\\?\{d6e55ca0-1a2e-4234-aaf3-3852170b492f}\vjoyrawpdo#1&2d595ca7&db&vjoyinstance00#{781ef630-72b2-11d2-b852-00c04fad5101}\device_002

[03088]Info: isRawDevice(9) - Compare \\?\{d6e55ca0-1a2e-4234-aaf3-3852170b492f}\vjoyrawpdo#1&2d595ca7&db&vjoyinstance00#{781ef630-72b2-11d2-b852-00c04fad5101}\device_002 with 002(d=2)

[03088]Info: OpenDeviceInterface(9) - DevicePath[2]=\\?\{d6e55ca0-1a2e-4234-aaf3-3852170b492f}\vjoyrawpdo#1&2d595ca7&db&vjoyinstance00#{781ef630-72b2-11d2-b852-00c04fad5101}\device_003

[03088]Info: isRawDevice(9) - Compare \\?\{d6e55ca0-1a2e-4234-aaf3-3852170b492f}\vjoyrawpdo#1&2d595ca7&db&vjoyinstance00#{781ef630-72b2-11d2-b852-00c04fad5101}\device_003 with 003(d=3)
```

You can see the end of one process (Process ids are in brackets) and the beginning of a second process. The first line referring the second project is **highlighted**, and it indicates the command this process is carrying out.

Every line in the log file starts with the process id and followed by an error level string such as **Info** and a column.

The next string is usually the name of the **function** (e.g. `isRawDevice`) and its significant parameters.

For full understanding of the printout you should refer to the source file.