

Prepar3D - Managed SimConnect in vb.Net

Introduction	3
What's in this?.....	3
Which versions of Prepar3D does this apply to?	3
Connecting to the SimConnect Server	4
The Solution	4
frmError	4
frmMain	4
Modules	5
Reading and Displaying Default Values	7
The Solution	7
frmMain	7
mdConversions.vb	8
mdEnum.vb	8
mdFuncs.vb	9
mdStruct.vb	9
mdDataDefinitions	9
mdSimconnect.vb.....	10
mdDisplay.vb	11
Masking Keyboard Inputs.....	13
The Solution	13
frmMain	13
mdEnum.vb	15
mdSimconnect.vb.....	15
Client – Server Communications	17
The Solution	18
frmMain	18
mdEnum.vb	19
mdFuncs.vb	19
mdSimconnect.vb.....	19

Introduction

I'm old-school; I came to the dotnet framework from Visual Basic (now known as VB Classic). Consequently, my choice of language was always going to be vb.Net even though I programmed in C/C++ because C# didn't really resemble C (or C++ for that matter), whereas vb.Net at least had a similar syntax and GUI-building ability when compared to VB Classic. When I first decided that I really ought to learn how to use SimConnect, I found that the C/C++ side of it wasn't too difficult and that the C# side would not be difficult either, *if* I was a C# programmer. Why? Because the P3D SDK has multiple C++ and C# examples, but just one miserable vb.Net example that reads just four variables (aircraft name plus lat-lon-alt). I even bought a copy of Petzold ("Programming Windows: Fifth Edition") to try to learn to design GUIs in Win32 rather than dotNet. That was a nightmare on its own. In short, I struggled.

So I hacked and hacked at that one poor vb.Net example until eventually the light went on. I made a note then that I really should provide a few more complex vb.Net examples to help those that, like me, prefer to use vb.Net rather than C# to create SimConnect GUIs. I hadn't intended to write a tutorial, but then I hadn't intended to write the sd2gau tutorials either. So... here we go again.

What's in this?

The initial subjects that I will be dealing with are setting up the SimConnect link with Prepar3D, reading and displaying variables, client-server communications and keyboard masking. As I have developed a set of WinForms templates for vb.Net development, you are going to see the same code repeated in the areas of startup/connect and disconnect/shutdown. Once we have discussed a subject in one project (e.g. how the connection works) and that code appears in another project, that subject will not be discussed again in the subsequent project. The basic connect/disconnect will be the first one we look at.

Everything we deal with here will be stored in the \Source Code folder in a ready-to-run **debug** solution. Each project builds on either the one before it (or on top of the basic connection example) to give a coherent build series and each project is heavily commented to explain what is happening at that point. The one thing that you will need to change in all projects is the path that references the SimConnect dll in the SDK. As ever, be aware of line wraps when code is posted into the tutorial.

Like sd2gau, this is my way of doing it which, in the end, may not suit you at all. Once you know how things work you will inevitably produce your own methods of working. Unlike sd2gau though, I am assuming that you really do know your way around Visual Studio and are reasonably competent with vb.Net.

Which versions of Prepar3D does this apply to?

Short answer: all of the 64-bit versions. When I wrote this, VS2022 was the current version of Visual Studio.

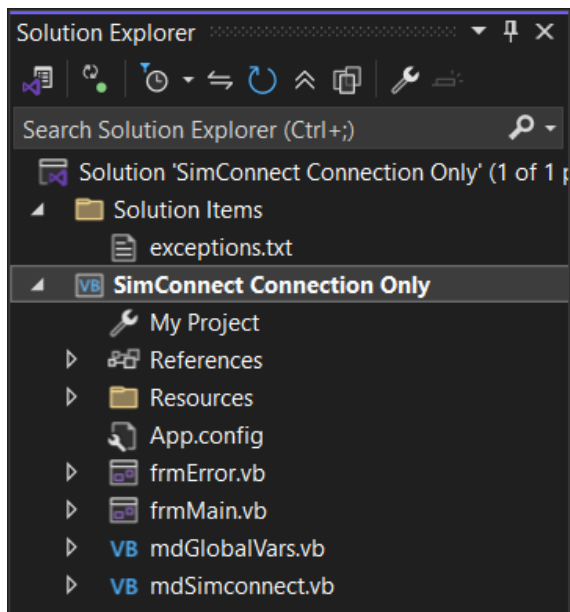
-Dai

Connecting to the SimConnect Server

Solution Name: SimConnect Connection Only

This starting project sets up a client connection to the main SimConnect server in P3D and allows you to disconnect when the connection has been made.

The Solution

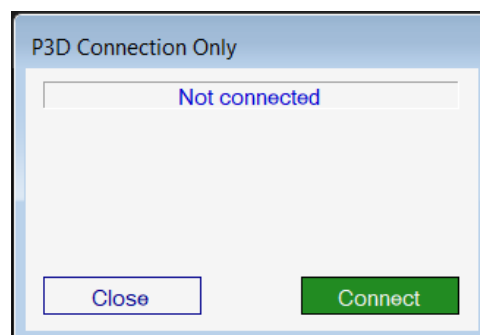


frmError

One thing I do with every WinForms application I create is to add a form that displays any exception thrown by SimConnect. It helps with debugging as it gives you the exception number, the exception description and the extended description (if any). frmError exists as a separate form on my PC and gets loaded into each project as an existing item. You will also need to list the exceptions.txt file as a resource in each new project. Loading the exceptions into the application is done in the Form Load procedure of frmMain where the text file is read into an array (`arrExceptions()`) for faster processing.

frmMain

The heavy lifting for the connect/disconnect is done here. It is my long-standing habit to call the main GUI 'frmMain', even if there is a splash screen to precede it.



frmMain has a label for information and two buttons – that's it.

All of the examples provided with the SDK make the same assumption; that is, Prepar3D is already running before you launch your SimConnect add-on. For a number of reasons this may not be the case (P3D crash, network error if your application is running remotely, etc.). Clicking on 'Connect' sets off a sequence of events:

- The GUI changes state to indicate that a connection to the server has been requested
- A timer is started that pings the server for a response

There is an intermediate state between request and response that you may not see if P3D is running. 'Connect' will change to say 'Abort' and the status label will say 'Waiting...'. To see this, start the application while P3D is not running. Clicking on 'Abort' may take a short time to respond because it has to wait for the connect request timer to time out. The connection request code looks like this:

```
Try
    ' Ping the simconnect server (P3D) for a response
    p3d_simconnect = New SimConnect(simconnect_name, Me.Handle, WM_USER_SIMCONNECT, Nothing, 0)
    ' The server has responded, so load the simconnect message handlers
    addHandlers()
    ' Change the UI to show that a connection has been made
    lblStatusDisplay.BackColor = Color.LimeGreen
    lblStatusDisplay.ForeColor = Color.Black
    lblStatusDisplay.Text = "Connected"
    btnConnect.BackColor = Color.DarkRed
    btnConnect.Text = "Disconnect"
    ' Disable the close button to prevent any possible exceptions
    ' caused by a shutdown with an active connection
    btnClose.Enabled = False
    ' Force a screen refresh
    Me.Refresh()
    ' Tell the app that a connection is active
    bConnectState = True
    ' Shut down the connection request because we have a good connection
    tmrConnect.Enabled = False
Catch ex As Exception
    ' If we're here then we didn't get a response from the server
    ' so allow the timer to restart and try again
End Try
```

It will keep trying to make a connection until you either click on 'Abort' or the server responds. If you start the application and then start P3D, eventually the server will respond and the application will go into listening (connected) mode.

Modules

Apart from frmMain and frmError, there are two vb.Net modules included in the project.

- mdSimconnect.vb at the moment only contains two Simconnect message handlers: p3d_simconnect_OnRecvException() and p3d_simconnect_OnRecvQuit(). Both of these will be loaded when the request connection timer gets a response – they are called from the addHandlers() sub-routine:
 - o OnRecvQuit() will trigger if P3D shuts down before your application.

- OnRecvException() traps and displays any exception messages received from the server. Note that OnRecvException() discards any exception 7 messages (SIMCONNECT_EXCEPTION_NAME_UNRECOGNISED) because this message may come in before the client-server link is fully established.
- mdGlobalVars.vb is exactly what is says; it contains any variables that need to be seen across all pages and modules.

Please compile and run the project. Test it with and without Prepar3D active.

Reading and Displaying Default Values

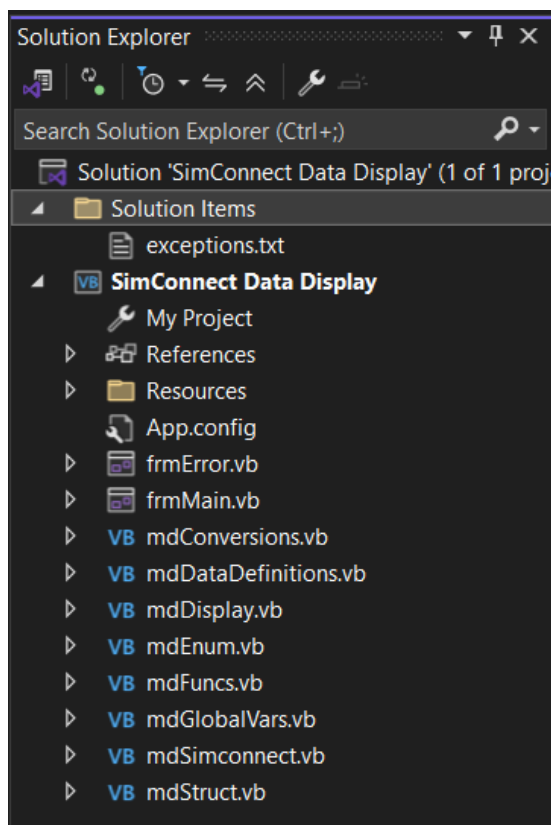
Solution Name: SimConnect Data Display

Information that comes from P3D via SimConnect is ephemeral; that is, data that exists on one cycle of P3D will not exist on the next. Like reading SimConnect data in C/C++, we need to grab that data and paste it into a structure so that we can work on it while P3D goes away and does something else. This is where we hit our first gotcha; in both C/C++ and vb.Net you can create a structure and then declare copies of it, after which you can then store different data in each copy. You **cannot** do that with the SimConnect dotNet wrapper; for example, if you have multiple engines you will need to declare identical structures for each engine, but each structure must have a unique name.

In this project we will read the following:

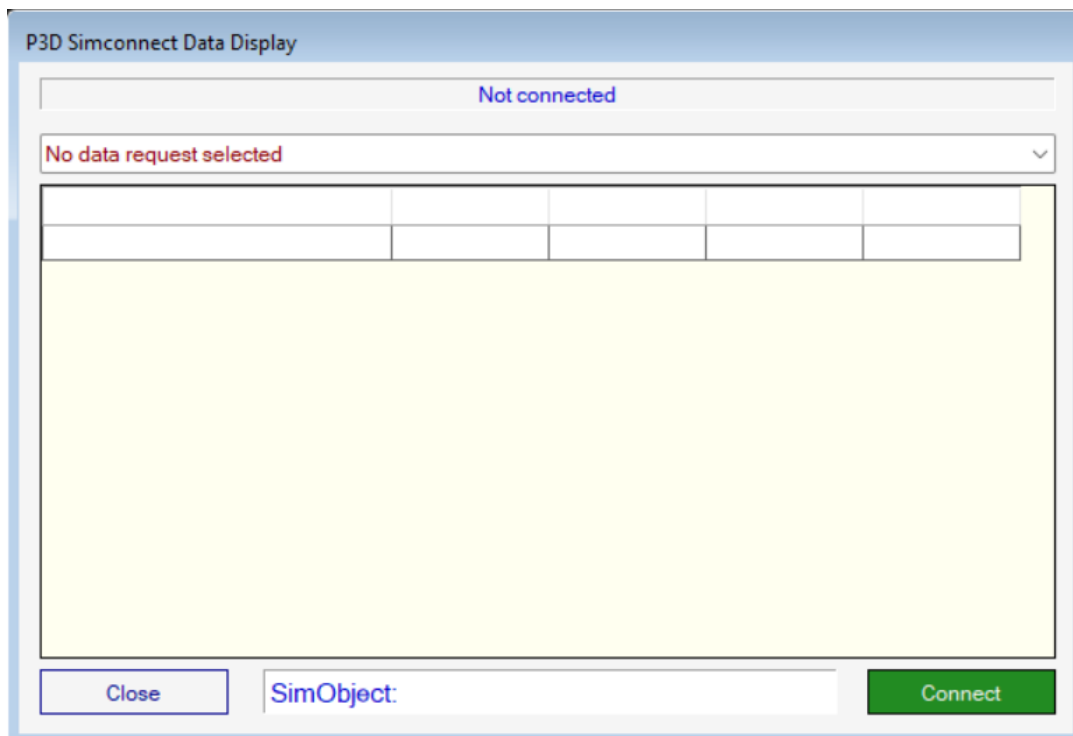
- aircraft instrumentation
- reciprocating engine data
- propeller data

The Solution



frmMain

frmMain has been expanded to include a datagridview to display the information provided by SimConnect, a combobox that lists the types of data available for reading (listed above) and a label that displays the name of the SimObject being queried.



As I've stated in the SimConnect section of the sd2gau series, I like to break projects up into easily readable/reusable chunks. We still have the same four files that existed in the first project but now we've gained six more modules:

- mdConversions.vb
- mdDataDefinitions.vb
- mdDisplay.vb
- mdEnum.vb
- mdFunc.vb
- mdStruct.vb

We'll take a brief look at each of the newcomers and at how the existing modules have expanded in this project:

mdConversions.vb

This module contains functions for converting one data type to another data type e.g. converting degrees Rankine to degrees Celsius or PSF to PSI. It's the full version of the one that I use.

mdEnum.vb

The dotNet wrapper requires that you enumerate all your requests and defines in a similar fashion to SimConnect in C/C++. This is a fairly simple setup; we have Requests and Defines for the instrumentation data and for the reciprocating engines and propellers. We also have an Event enumeration – CLIENT_PAUSE – which we will use to prevent any display updates if the main sim is paused. My habit is to force enumerations to start from 1 (one) as this can make debugging easier.

mdFuncs.vb

As you'd expect from the name, this one contains reusable functions that help control the data display and SimConnect setup. There are only three functions and all of them could have been coded into frmMain, but that would have defeated the object of making the code as easy as possible to debug.

mdStruct.vb

Within the dotNet structures you have to marshall every string. It's not good enough to marshall the first string in a series; if you try this you will get crud written to the screen. As part of the instrumentation data, we also get the aircraft type and ATC details.

```
(...)  
Public smoke_available As Double  
Public smoke_enabled As Double  
<MarshalAs(UnmanagedType.ByValTStr, SizeConst:=256)> Public atc_type As String  
<MarshalAs(UnmanagedType.ByValTStr, SizeConst:=256)> Public atc_model As String  
<MarshalAs(UnmanagedType.ByValTStr, SizeConst:=256)> Public atc_id As String  
<MarshalAs(UnmanagedType.ByValTStr, SizeConst:=256)> Public atc_airline As String  
<MarshalAs(UnmanagedType.ByValTStr, SizeConst:=256)> Public atc_flight_number As String  
Public eng_type As Double  
(...)
```

If you know how long the string is likely to be, you can modify the SizeConst value at the end of the marshall declaration to be 8, 32, 64, 128, 256 or 260 (default MAX_PATH). If you do change it, ensure that the SIMCONNECT_DATATYPE variable in the corresponding data definition is also the same size.

mdDataDefinitions

mdDataDefinitions is divided into a series of subroutines, all of which are called in order from InitDataRequest() in mdFuncs.vb. InitDataRequest() is called after a SimConnect connection is established by tmrConnect() in frmMain. Until these subroutines have been executed and the SimConnect stream fully established, nothing will happen.

Sub addDataDefinitions()

Each data definition MUST be suffixed with a unique identifying number. They don't have to be in numeric order; they just have to be unique. If you do repeat a number, the compiler will warn you. You also have to provide a full path to the enumeration i.e. the enumeration structure title followed by the enum itself e.g.

```
p3d_simconnect.AddToDataDefinition(P3Data.DEFINE_AIRCRAFT_DATA, "Title", "",  
SIMCONNECT_DATATYPE.STRING256, 0, 0)  
p3d_simconnect.AddToDataDefinition(P3Data.DEFINE_AIRCRAFT_DATA, "NUMBER OF ENGINES", "number",  
SIMCONNECT_DATATYPE.FLOAT64, 0, 1)
```

where:-

- p3d_simconnect is the name of this SimConnect instance as defined by you
- In the first field P3Data is the enum structure name (again, the name is defined by you) and DEFINE_AIRCRAFT_DATA is an enumeration group for the instrumentation information
- The second field is the required data ("Title", "NUMBER OF ENGINES", etc. as defined in the SDK)

- The third field is the data type as defined in the SDK. Note that this field **is always blank** if you are returning a string
- The fourth field is the data type definition as discussed above
- Field five is normally set to zero. More information can be found in the SDK
- Finally, you have the unique data definition number.

Unlike the C/C++ version you can't get an HRESULT through the SimConnect wrapper. If your project compiles but fails to run, inspect the failing data definition very carefully.

Sub registerDefs()

This is where you assign the enumeration group to the structure it needs to fill. As the note in the code says:-

```
' IMPORTANT: you must register definitions with the simconnect managed wrapper marshaller.
' If you skip this step, you will only receive an int in the .dwData field!!!!
```

e.g.

```
p3d_simconnect.RegisterDataDefineStruct(Of Aircraft_Data)(P3Data.DEFINE_AIRCRAFT_DATA)
```

Sub addEvents()

This contains the single event that we need to listen for – is the sim paused or not. This one is a ‘subscription’ in that we subscribe to be notified when that system event changes.

```
' Sim is paused signal
p3d_simconnect.SubscribeToSystemEvent(P3Data.CLIENT_PAUSE, "Pause")
```

Sub startUpdates()

Unless you tell SimConnect that you want to receive data at a specified interval, you won't get anything to display. We are using `RequestDataOnSimObject()` because that allows us to specify which SimObject we want to hear from. Normally this would be SimObject zero (`SIMCONNECT_SIMOBJECT_TYPE.USER` - your vehicle) but see the SDK for information on how to locate and specify the SimObject you want to talk to. For instrumentation you would want to call the update as quickly as possible (`SIMCONNECT_PERIOD.VISUAL_FRAME`), but for something like fuel consumption `SIMCONNECT_PERIOD.SECOND` would be adequate. See the SDK for how to use `SIMCONNECT_DATA_REQUEST_FLAG` and the three default fields at the end of the request.

```
p3d_simconnect.RequestDataOnSimObject(P3Data.REQUEST_AIRCRAFT_DATA,
P3Data.DEFINE_AIRCRAFT_DATA, SIMCONNECT_SIMOBJECT_TYPE.USER, SIMCONNECT_PERIOD.VISUAL_FRAME,
SIMCONNECT_DATA_REQUEST_FLAG.DEFAULT, 0, 0, 0)
```

mdSimconnect.vb

mdSimconnect has been expanded with two more handlers: `OnReceiveSimObjectData()` and `OnReceiveEvent()`.

- `OnReceiveEvent()` picks up the `CLIENT_PAUSE` subscription when the simulator is paused or unpaused
- `OnReceiveSimObjectData()` is all the information that we requested with the `AddToDataDefinition` calls.

No data can be passed to `mdDisplay.vb` until `P3Data.REQUEST_AIRCRAFT_DATA` has been called at least once because we need to know how many engines and what type of engines are on the `SimObject`. This part of the code is also responsible for checking for a change of `SimObject`. The rest of the code fills up the structures with the incoming data from `SimConnect`. Note that we have two `Select Case` statements: one for the aircraft data so that we can read the engine type and number of engines and one for the engine and prop data. It's necessary to do it this way because we can't guarantee that `P3Data.REQUEST_AIRCRAFT_DATA` will be the first data to be received.

The `Select Case` statement for the engines and the props also has a guard clause on it:

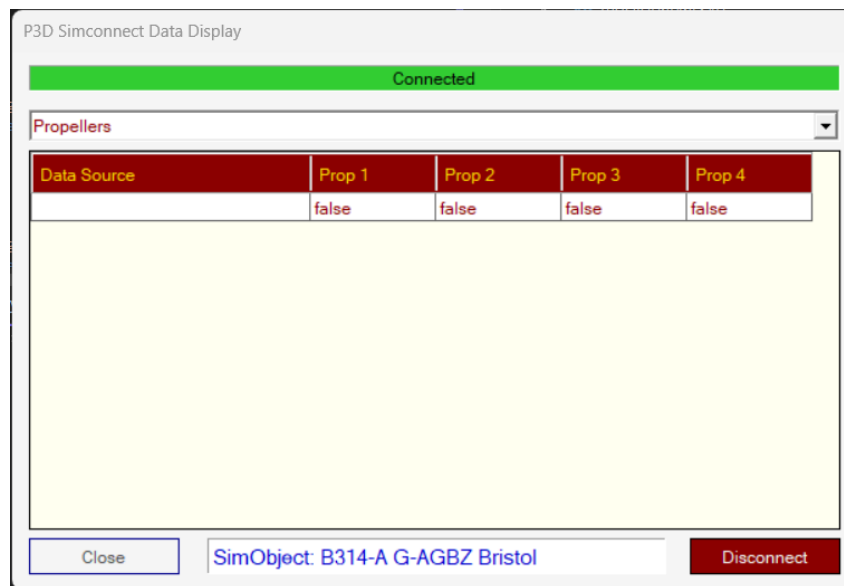
```
if eng_type = engRecip Then
```

It's not actually needed here as this is such a small app and only deals with reciprocating engines, but this type of guard clause can speed up execution by avoiding code that is irrelevant to the current requirement. If the simulator is paused then no data will be passed to `mdDisplay.vb` until the sim is released again.

mdDisplay.vb

`mdDisplay.vb`, as its name suggests, takes the incoming data in the structure from `mdSimconnect.vb` and writes it to screen along with the source name of the respective data. The data is displayed in a `DataGridView`; the code is straightforward and should be easy to understand. Note that only engine 1 and prop 1 fill the Source column.

Please compile and run the project. Change aircraft and aircraft types in the sim and watch the application respond. If you get a page freeze like this when changing `SimObject`:



select another page and come back to the frozen one. I haven't added any trap code as figuring out exactly when the SimObject has finished loading creates a level of complexity that I don't want in this tutorial.

Masking Keyboard Inputs

Solution Name: SimConnect Keyboard Masking

This was the one that caused me the most grief. There are two major deviations from the C/C++ equivalent code:

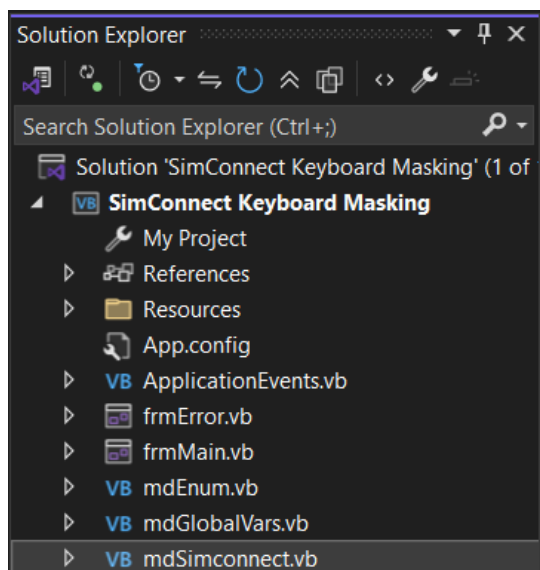
- the optional parameters in C++ are **not** optional in the dotNet wrapper
- consequently, you will **always** receive both the key down and key up actions, so you will need to mask/reject one or other of those

Additionally:

- the SDK leads you to believe that you always need to provide a sim variable name in the masking field (you don't)
- key inputs will only be read when Prepar3D has focus.

This last point is also shared by the C++ version.

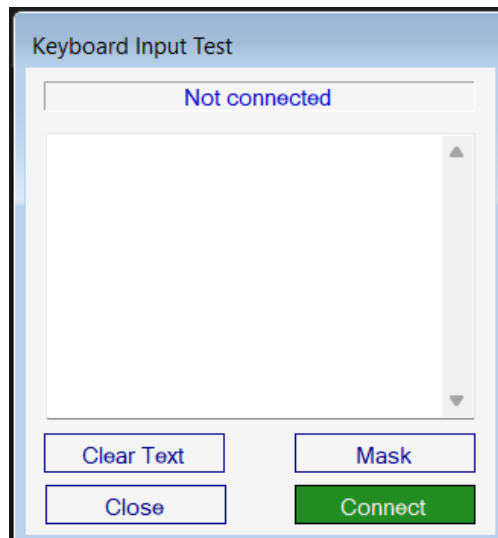
The Solution



We have reverted back to the SimConnect Connection Only solution to build this project on to. The only addition to the connection project is mdEnum.vb, but there are changes to both mdGlobalVars.vb and mdSimConnect.vb.

frmMain

frmMain has been expanded to include a text box that will display the key press and a message relating to its action.



- 'Clear Text' clears the information from the textbox
- 'Mask' tells SimConnect to mask/unmask the key inputs. 'Mask' is a designer placeholder as the wording on the button will change with its function.

Once the mask/unmask button has been pressed, we need to *immediately* return focus to Prepar3D. To do this we use a p/invoke call to user32.dll and a function that uses the p/invoke:

```
Public Declare Function SetForegroundWindow Lib "user32.dll" (ByVal hwnd As Integer) As Integer
Public Declare Auto Function FindWindow Lib "user32.dll" (ByVal lpClassName As String, ByVal lpWindowName As String) As Integer
```

(...)

```
Function setForegroundApp(ByVal app As String)

    ' Sets the named application to the foreground.
    For Each process As Process In Process.GetProcessesByName(app)
        Dim wHandle As IntPtr = FindWindow(Nothing, process.MainWindowTitle)
        If wHandle <> IntPtr.Zero Then
            SetForegroundWindow(CInt(wHandle))
        End If
    Next

    Return 0

End Function
```

To return the focus to Prepar3D, call the function as the last action in the mask/unmask button subroutine with the name of the simulator as the parameter:

```
' Return focus to P3D
setForegroundApp("prepar3d")
```

Using `appActivate()` is another possibility but this function is more flexible. If Lockheed-Martin change the process title in the main window (e.g. a new version number), `appActivate()` will fail until the code is changed to match, but this function will not (unless they change the name of the executable, which is highly unlikely).

mdEnum.vb

As this project demonstrates both masked and unmasked events, mdEnum.vb contains groupings for both the unmasked event (key E – external brake request) and the masked events (keys A and D).

mdSimconnect.vb

Compared to the simple connection only project, mdSimconnect.vb has been expanded with one more handler: OnReceiveEvent(). It also has a new subroutine addKeyInputs() which is called from initDataRequest() on frmMain after the handlers have been initiated.

Sub addKeyInputs()

addKeyInputs() is where we set up the keyboard and masking. As I found this area of the dotNet wrapper to be very frustrating, I'm going to step through it line-by-line.

First of all, we'll assign the unmasked input event to key E:

```
p3d_simconnect.MapClientEventToSimEvent(EVENT_ID.EVENT_KEY_E, "brakes")
```

We now add that event (key E) to the notification group GROUP_NO_MASK. A notification group is simply a convenient way of being able to set the priority for multiple events in one go:

```
p3d_simconnect.AddClientEventToNotificationGroup(GROUP_ID.GROUP_NO_MASK, EVENT_ID.EVENT_KEY_E, False)
```

Next, add the input event to a client event. This is where it differs from the C++ version in that we must have both keydown and keyup events:

```
p3d_simconnect.MapInputEventToClientEvent(INPUT_ID.INPUT_KEY_E, "e", EVENT_ID.EVENT_KEY_E, 0, EVENT_ID.EVENT_KEY_E, 0, False)
```

Refer to the SDK for further information on the parameter following the key action. For us, zero is fine. The final parameter – False – is telling SimConnect that this key event is not to be masked.

Now we assign a priority to our notification group. As we don't want the key inputs to be swamped by events inside SimConnect, we assign the highest possible priority:

```
p3d_simconnect.SetNotificationGroupPriority(GROUP_ID.GROUP_NO_MASK, SimConnect.SIMCONNECT_GROUP_PRIORITY_HIGHEST)
```

Finally, we turn the input group on. For this project we won't be turning it off again as we always want to see the input from the E key.

```
p3d_simconnect.SetInputGroupState(INPUT_ID.INPUT_KEY_E, SIMCONNECT_STATE.ON)
```

The code for the input keys A and D is very similar, but with three changes:

- there is no sim event mapped to the client key in the MapClientEventToSimEvent() lines (it was 'brakes' in the unmasked example)

- in the `AddClientEventToNotificationGroup()` lines, the final parameter is set to `True` to indicate that these key presses can be masked
- finally, the initial state of the `SetInputGroupState()` is `OFF`

```
p3d_simconnect.MapClientEventToSimEvent(EVENT_ID.EVENT_KEY_A, "")
p3d_simconnect.MapClientEventToSimEvent(EVENT_ID.EVENT_KEY_S, "")
p3d_simconnect.AddClientEventToNotificationGroup(GROUP_ID.GROUP_KEYS, EVENT_ID.EVENT_KEY_A,
True)
p3d_simconnect.AddClientEventToNotificationGroup(GROUP_ID.GROUP_KEYS, EVENT_ID.EVENT_KEY_S,
True)
p3d_simconnect.MapInputEventToClientEvent(INPUT_ID.INPUT_KEY_AD, "a", EVENT_ID.EVENT_KEY_A, 0,
EVENT_ID.EVENT_KEY_A, 0, True)
p3d_simconnect.MapInputEventToClientEvent(INPUT_ID.INPUT_KEY_AD, "s", EVENT_ID.EVENT_KEY_S, 0,
EVENT_ID.EVENT_KEY_S, 0, True)
p3d_simconnect.SetNotificationGroupPriority(GROUP_ID.GROUP_KEYS,
SimConnect.SIMCONNECT_GROUP_PRIORITY_HIGHEST_MASKABLE)
p3d_simconnect.SetInputGroupState(INPUT_ID.INPUT_KEY_AD, SIMCONNECT_STATE.OFF)
```

Switching the input group state is done in `btnMask` on `frmMain()`.

Sub `p3d_simconnect_OnRecvEvent()`

When a key is pressed, the action is seen as an event in this handler. However, we have the problem that because the dotNet wrapper insists on having both the keydown and keyup events, we need to ignore one of them. In this case, I've ignored the keyup event.

```
Static key_status As Integer = 0

If key_status = 0 Then ' Key down event

    Select Case data.uEventID

        (...)

    End Select

    key_status = 1 ' Prevent the key up event
Else
    key_status = 0 ' Reset the key event lock
End If
```

As before, please correct paths, compile and run. This type of functionality could be used as an instructor board to switch `SimObject` failures on and off.

Client – Server Communications

Definitions: *Server* means Prepar3D and *Client* is any SimConnect application that talks to Prepar3D. You can have multiple clients and servers in a networked Prepar3D setup.

Solution Name(s): SimConnect LVar dotNet and SimConnect LVar Gauge

There are two solutions to this part of the tutorial because we need Prepar3D to send some data to the SimConnect application. In its current iteration SimConnect **cannot** send string data, only numeric data.

SimConnect LVar Gauge

The C++ gauge is set up to send data from an L:var. Although the SDK doesn't mention being able to use L:vars as data variables in SimConnect client-server, it seemed logical that it could do so as the `get_named_variable_value(Lvar_name)` is simply returning the value of the data held in the registered variable. A full description of how to set up the server side of client-server communications can be found in `sd2gau`. Regardless, the solution is provided in the `\source` folder and the code is commented throughout.

Enter the following two lines in the `panel.cfg` file of the aircraft of your choice:

```
gaugeNN=LVar Gauge!lvarswitch,          N,N,50,50
gaugeNN=LVar Gauge!simconnect_interface, 0,0,0,0
```

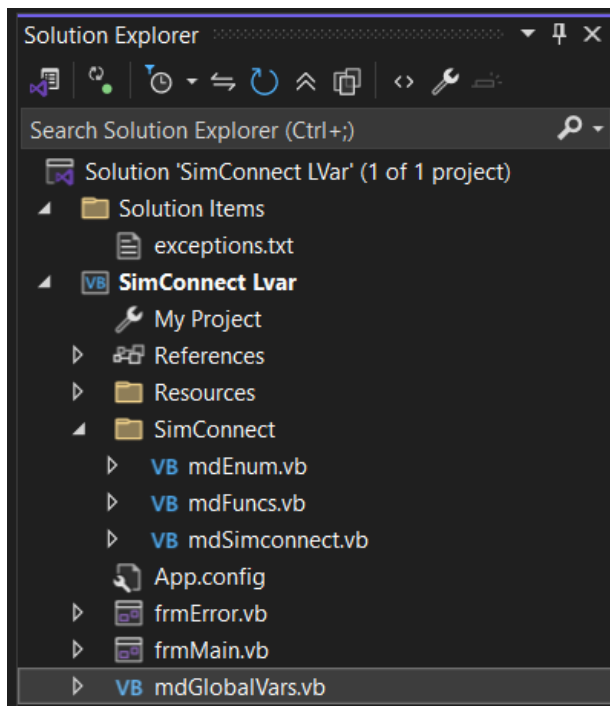
then copy the `Lvar Gauge.dll` to either that aircraft's `\panel` folder or to the main `Prepar3D\gauges` folder. This will place a switch and an indicator lamp on the panel.



If you recognise the background, yes; it's the poor unfortunate Cessna C172 being abused again. A left-click on the switch will light the indicator and send data to the dotNet app to say the light is on. The indicator will stay lit until the dotNet app tells it to turn off.

SimConnect LVar dotNet

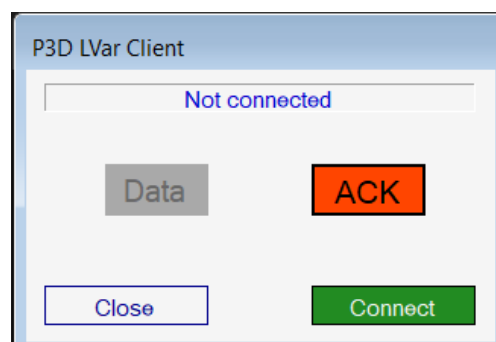
The Solution



For this project I elected to place all the SimConnect files under a SimConnect filter, but this is not necessary.

frmMain

The basic Connection Only frmMain has two new controls; a label ('Data') to say that a SimConnect input has been detected and a button ('ACK') to acknowledge the receipt of the data and to tell the gauge to turn the indicator off.



You may have spotted a potential problem with the acknowledgement button; pressing the button will shift focus away from Prepar3D and disrupt any keypresses directed at the sim. Like the Masking Keyboard Inputs project above, the last action in the keypress subroutine is to force focus back to Prepar3D.

mdEnum.vb

The contents of this module set up the receive and transmit groupings.

```
' Adding event names to strings makes it easy to change them
' as they are listed in one place only
Friend rc16000 As String = "#0x16000" ' Receive Data
Friend tr16001 As String = "#0x16001" ' Transmit ACK

Enum GROUPS

    GROUP_DATA

End Enum

Enum LVARs

    DATA = 16000
    ACK

End Enum
```

If you open up the gauge solution and look at the `simconnect_lvars.h` file, you will find the same transmit and receive identifiers listed there. The difference is in usage; in the gauge solution, identifier `#0x16000` is being used to transmit data and identifier `#0x16001` is being used to receive data. In the dotNet solution their function is the other way round. If the transmit and receive identifiers are not identical you will **never receive any data**. It's my preference to set up the enumerations to have the same values as the identifiers because this makes debugging so much easier when trying to figure out which enumeration is attached to which identifier.

mdFuncs.vb

The only code in here is a short function that is a wrapper around the `SimConnect TransmitClientEvent()` call. It's really a piece of 'lazy code' because it reduces the amount of typing needed to instigate a data transmission. For client-server communications, the event flag is always going to be `GROUPID_IS_PRIORITY` because otherwise Prepar3D would assign the event to a much lower priority. In the worst case, it may never be passed to the receiver at all because it keeps getting reassigned.

mdSimconnect.vb

addClientListener()

Here we set up the app to listen for incoming data by associating the identifier with its event enumeration and then assigning that to a group. Note the final notification group parameter is 'false'; that is, the incoming event is not being masked.

```
' Unlike C/C++ this cannot be converted to a function because enum is a Type and the
' dotnet framework won't allow you to pass Types as parameters
p3d_simconnect.MapClientEventToSimEvent(LVARs.DATA, rc16000)
p3d_simconnect.AddClientEventToNotificationGroup(GROUPS.GROUP_DATA, LVARs.DATA, False)
```

addClientTransmitter()

This simply assigns the transmitter identifier to an event enumeration.

```
p3d_simconnect.MapClientEventToSimEvent(LVARS.ACK, tr16001)
```

Sub p3d_simconnect_OnRecvEvent()

Data sent from the gauge is received in the OnRecvEvent subroutine. Although we know that in this case the only event will be the incoming notification that the indicator light is on, the Case statement has deliberately been written as if the incoming data associated with that event enumeration could vary in its numeric value.

```
Case LVARS.DATA  
    If data.dwData = 1 Then  
        data_in = data.dwData  
        frmMain.lblData.BackColor = Color.DarkGreen  
        frmMain.lblData.ForeColor = Color.White  
    End If
```

This will illuminate the Data symbol. The state of the variable `data_in` also controls the actions of the ACK button.

Correct paths, compile and run. Have fun turning the indicators on and off.

Copyright 2025 Dai Griffiths, Dragonflight Design.