

Prepar3D - Managed SimConnect in vb.Net (v2.0)

Revision List.....	4
Introduction	5
What's in this?.....	5
Which versions of Prepar3D does this apply to?	5
Connecting to the SimConnect Server	6
The Solution	6
frmError	6
frmMain	6
Modules	7
Reading and Displaying Default Values	9
The Solution	9
frmMain	9
mdConversions.vb.....	10
mdEnum.vb	10
mdFuncs.vb	11
mdStruct.vb.....	11
mdDataDefinitions	11
mdSimconnect.vb.....	12
mdDisplay.vb	13
Masking Keyboard Inputs.....	14
The Solution	14
frmMain	14
mdEnum.vb	16
mdSimconnect.vb.....	16
Client – Server Communications	18
The Solution	19
frmMain	19
mdEnum.vb	20
mdFuncs.vb	20
mdSimconnect.vb.....	20
Setting Data On SimObjects	22
The Solution	22
frmMain	22
mdEnum.vb	23
mdDataDefinitions.vb	23
mdDisplay.vb	24
Creating SimObjects from a Text File	25
frmMain	26
mdEnum.vb	27

mdDataDefinitions.vb	27
mdDisplay.vb	28
mdFuncs.vb	28
mdSimconnect.vb.....	28
The Initialisation/Waypoints File.....	29

Revision List

25 th Aug 2025	Initial release 1.0
26 th Aug 2025	Squashed bug in all samples when loading the exceptions.txt file Fixed the occasional DataGridView non-display in the SimConnect Data Display sample
28 th Aug 2025	Version 2.0 Added 'Setting Data On SimObjects' topic Added 'Creating SimObjects from a Text File' topic

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 been 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.

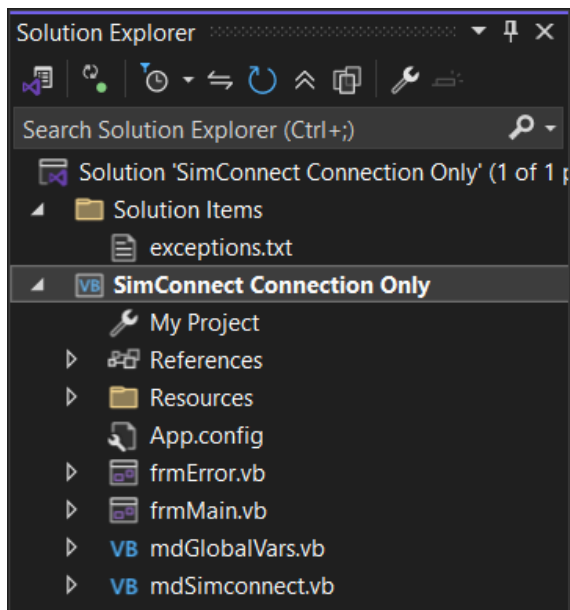
-Dai

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

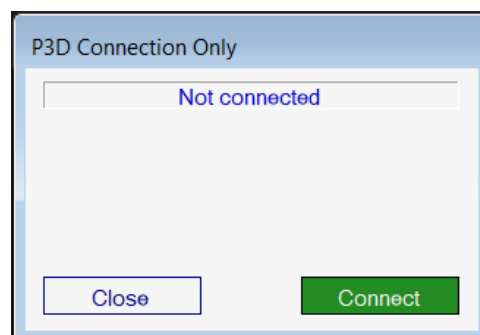


frmError

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

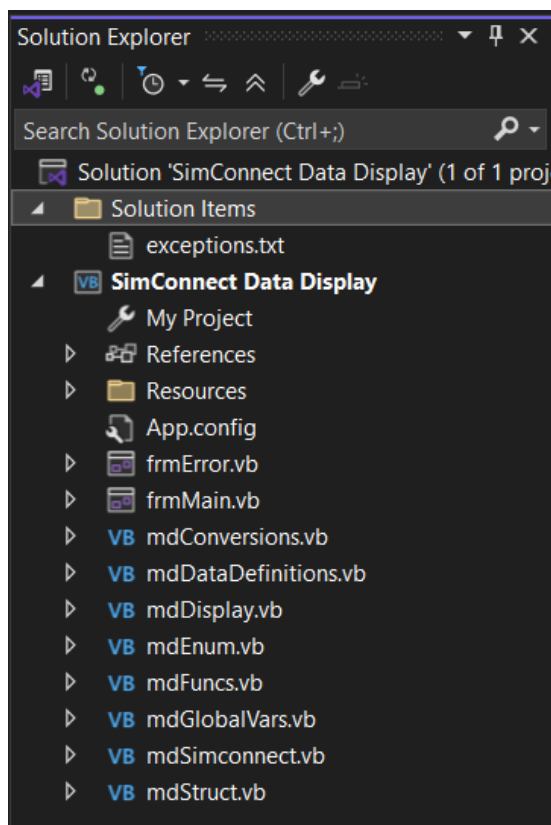
Solution Name: SimConnect Data Display

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:

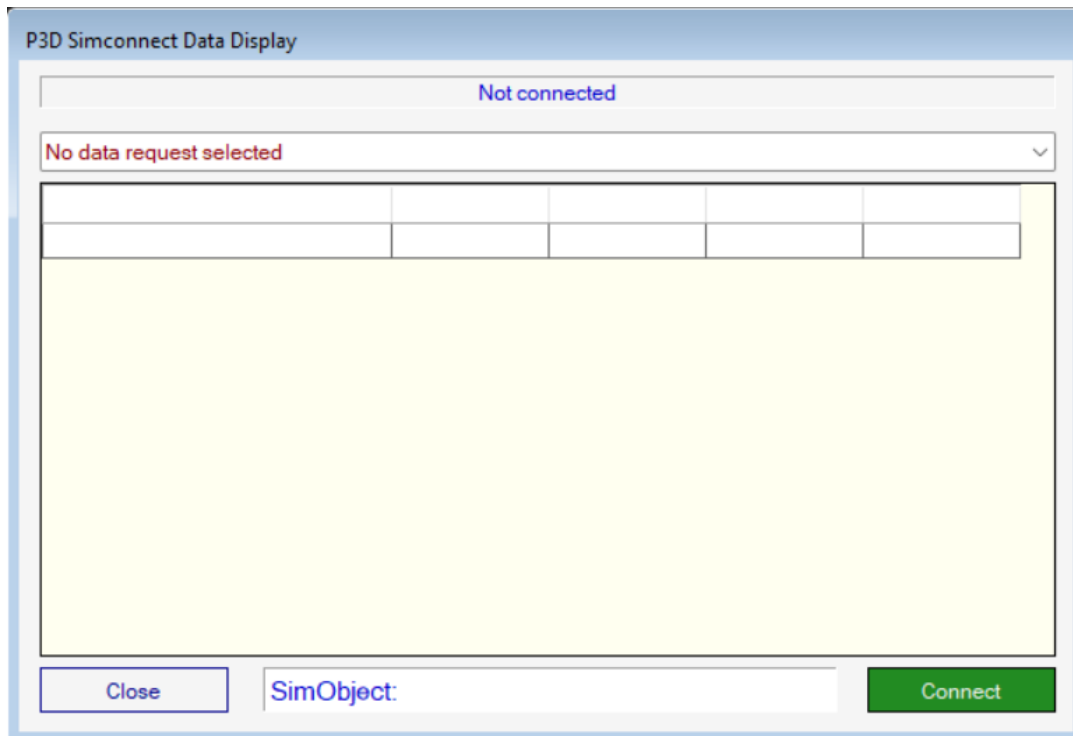
- aircraft instrumentation
- reciprocating engine data
- propeller data

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.

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:

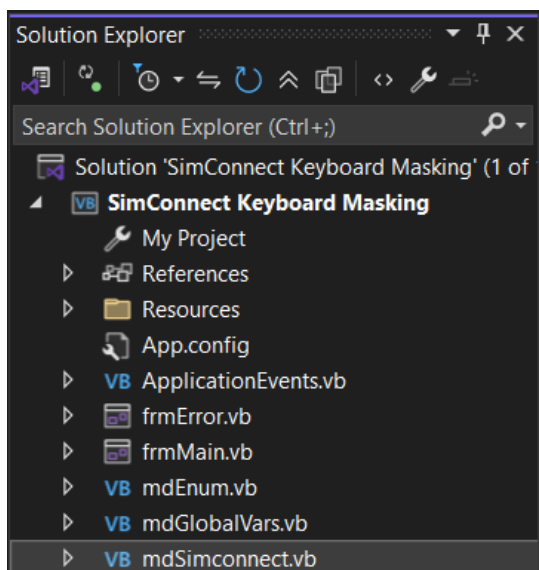
- the optional parameters in C++ are **not** optional in the dotNet wrapper
- consequently, you will **always** receive both the key down and key up actions, so you will need to mask/reject one or other of those

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.

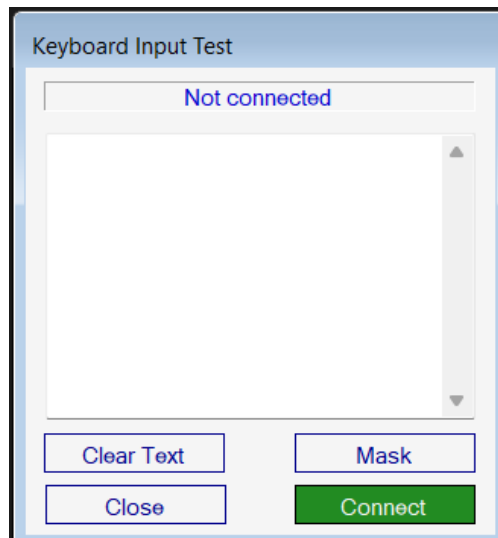
The Solution



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
```

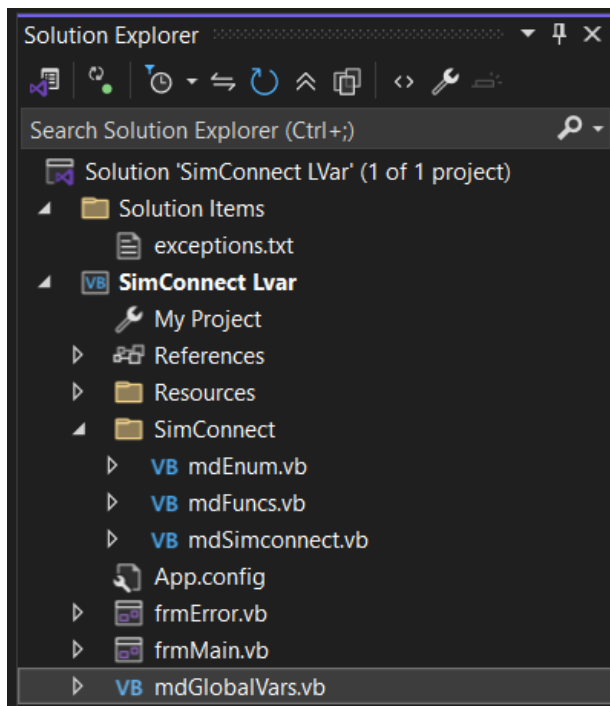
then copy the `Lvar Gauge.dll` to either that aircraft's `\panel` folder or to the main `Prepar3D\gauges` folder. This will place a switch and an indicator lamp on the panel.



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

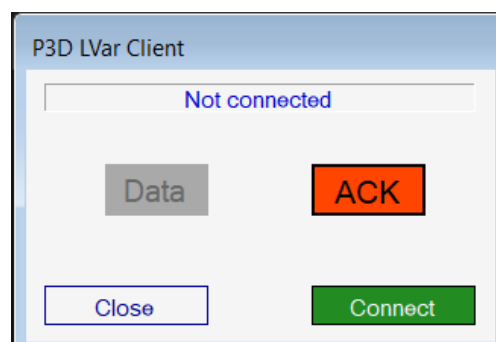
The Solution



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.

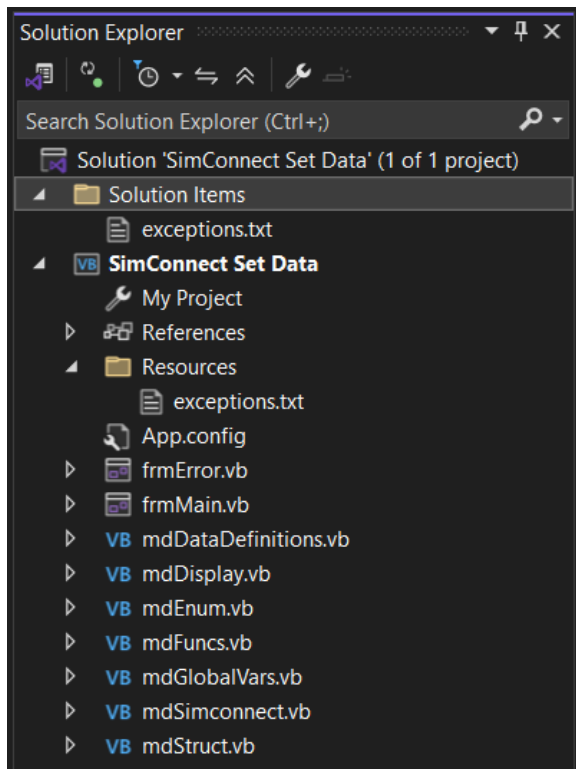
Correct paths, compile and run. Have fun turning the indicators on and off.

Setting Data On SimObjects

Solution Name: SimConnect Set Data

The ability to be able to interface directly with a SimObject is quite a useful one. In this sample we will control the throttle of the aircraft via a slider. The application will automatically recognise the number of engines and will provide sliders and data to suit.

The Solution



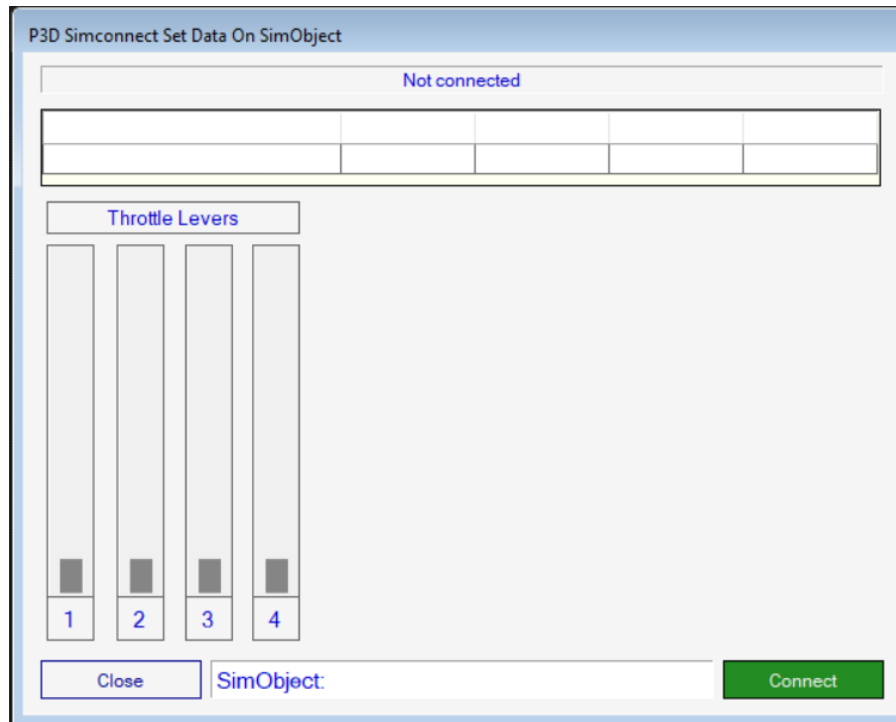
The starting point for this solution is a severely cut-down version of SimConnect Data Display. All that remains of the data collection from that solution are the aircraft title and the number of engines. The collection of that data and the display of the throttle position is done in exactly the same way.

frmMain

frmMain still retains the DataGridView as we need to see the percentage of throttle selected. I have added four sliders, one to control each individual throttle lever. A change here will be reflected in the sim and a change in the sim will cause the relevant slider to be updated to match. For some odd reason the sliders won't return below 9% but that annoyance aside, it doesn't affect the code or the structure of the application.

To save you from being driven to distraction on how I created the sliders, they are simply VScrollBars with a panel added on top and sized to remove the up and down buttons from view. There is a bit of a gotcha with the vbScrollBar: zero is at the top of the scrollbar and vScrollBar.Maximum is at the bottom.

To provide a more familiar percentage display, here the scrollbars start from minus 100 and end at zero.



mdEnum.vb

The enumeration for this project is short and simple; we need a request and define to receive the data from Prepar3D, a CLIENT_PAUSE event and four individual throttle enumerations to set the data on the sim.

```
Enum P3Data

    REQUEST_AIRCRAFT_DATA = 1
    DEFINE_AIRCRAFT_DATA

    SET_THROTTLE1
    SET_THROTTLE2
    SET_THROTTLE3
    SET_THROTTLE4

    CLIENT_PAUSE

End Enum
```

mdDataDefinitions.vb

addDataDefinitions()

The structure of `addDataDefinitions` is pretty much the same as it is in the Display Data sample. The only change is that whereas for a read action we can assign multiple variables to one enumeration, for a write action each variable has to have its own enumerator.

```

' Read data
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)
p3d_simconnect.AddToDataDefinition(P3Data.DEFINE_AIRCRAFT_DATA, "GENERAL ENG THROTTLE
LEVER POSITION:1", "percent", SIMCONNECT_DATATYPE.FLOAT64, 0, 2)
p3d_simconnect.AddToDataDefinition(P3Data.DEFINE_AIRCRAFT_DATA, "GENERAL ENG THROTTLE
LEVER POSITION:2", "percent", SIMCONNECT_DATATYPE.FLOAT64, 0, 3)
p3d_simconnect.AddToDataDefinition(P3Data.DEFINE_AIRCRAFT_DATA, "GENERAL ENG THROTTLE
LEVER POSITION:3", "percent", SIMCONNECT_DATATYPE.FLOAT64, 0, 4)
p3d_simconnect.AddToDataDefinition(P3Data.DEFINE_AIRCRAFT_DATA, "GENERAL ENG THROTTLE
LEVER POSITION:4", "percent", SIMCONNECT_DATATYPE.FLOAT64, 0, 5)

' Write throttle data
p3d_simconnect.AddToDataDefinition(P3Data.SET_THROTTLE1, "GENERAL ENG THROTTLE LEVER
POSITION:1", "percent", SIMCONNECT_DATATYPE.FLOAT64, 0, 6)
p3d_simconnect.AddToDataDefinition(P3Data.SET_THROTTLE2, "GENERAL ENG THROTTLE LEVER
POSITION:2", "percent", SIMCONNECT_DATATYPE.FLOAT64, 0, 7)
p3d_simconnect.AddToDataDefinition(P3Data.SET_THROTTLE3, "GENERAL ENG THROTTLE LEVER
POSITION:3", "percent", SIMCONNECT_DATATYPE.FLOAT64, 0, 8)
p3d_simconnect.AddToDataDefinition(P3Data.SET_THROTTLE4, "GENERAL ENG THROTTLE LEVER
POSITION:4", "percent", SIMCONNECT_DATATYPE.FLOAT64, 0, 9)

```

addEvents()

A pause event will cause the throttle sliders to be disabled. This is to prevent a sudden jump in the sim when it is unpaused if the throttle sliders have been moved.

mdDisplay.vb

The functionality of mdDisplay.vb is identical to the functionality as discussed in SimConnect Data Display. A change in the throttle in the sim will be reflected in a change of the relevant slider; as discussed above, the vbScrollBar starts from negative 100 to zero, so we have to invert the incoming data to provide a recognisable on-screen display. There is no equivalent to the C/C++ ‘-abs’ library function so there is a local function (`minusAbs()`) in the mdFuncs module which does the same job.

Setting the new data on the sim

Actually setting data on the sim is done directly from the vscrollbar.Scroll subroutines in frmMain. When you move the scrollbar the new position is stored in `e.NewValue`. Because the value is negative, first it has to be passed through `Maths.Abs` to invert it to a positive figure and then sent via SimConnect to the sim:

```

Dim data As Double = 0

' We can't apply the absolute new value directly as
' dotNet throws an exception so we go through this intermediate step
data = Maths.Abs(e.NewValue)

' Set the new throttle position
p3d_simconnect.SetDataOnSimObject(P3Data.SET_THROTTLE1, SIMCONNECT_SIMOBJECT_TYPE.USER,
SIMCONNECT_DATA_SET_FLAG.DEFAULT, data)

```

Compile, correct and play.

Creating SimObjects from a Text File

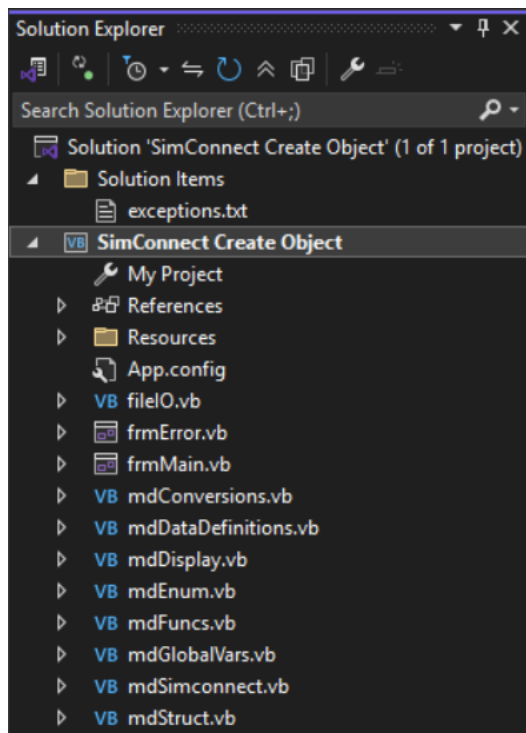
Solution Name: SimConnect Create Object

Effectively this sample is a vb.Net conversion of the C# Managed_AI_Waypoints sample included in the SDK, except that I've moved the starting airport from KSEA to EGAC. I have provided a scenario in the source code but you can just as easily load any other scenario and move to EGAC as it defaults to the end of runway 33 anyway.

With this example we're going to step away from hard-coding information and build it as a blackbox application that reads all of its required data from an external text file. However, this approach requires some discipline with the enumeration that lists all of the requests; the enumerator for the object request has to be named `REQUEST_OBJECT` (or any other name as long as you know what it is and use it consistently across your project). This arises because an enum is a type and you can't pass a type as parameter to a function. What we can do is to create a copy of the enumeration that contains `REQUEST_OBJECT` outside the function (it's done within `mdGlobalVars.vb`) and then refer to the copy within the function. This at least means that you don't have to retain the same enum name from project to project. There is an alternative; you use the same enum with a single enumerator across all projects which use the `initialiseObject()` function. This means that you can use the enumerator directly in the creation function.

One thing you *must* remember about creating SimObjects is that they will only appear in Prepar3D as long as the SimConnect connection is active.

The Solution

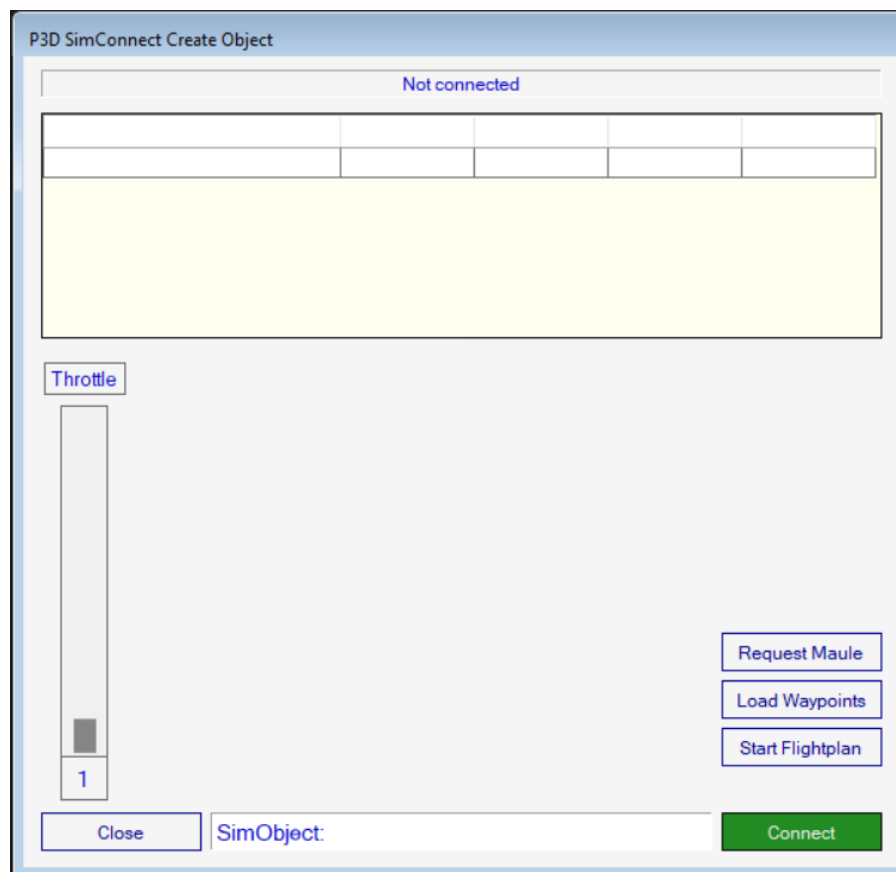


This sample is built on a reduced version of the SimConnect Set Data solution; reduced in that we're dealing with a single-engine aircraft (the Maule M7) so throttles 2 – 4 have been removed. We've also added the conversions module back in so that we can display latitude and longitude in degrees. In total, we will be displaying data for latitude, longitude, altitude, airspeed, heading and throttle percentage. We've also gained an extra module – fileRW.vb – which is simply a wrapper around the dotNet StreamReader and StreamWriter functions, but with a couple of extra tricks and the ability to read and write ini files as well.

frmMain

We have added three buttons when compared against the SimConnect Set Data GUI.

- 'Request Maule' will ask SimConnect to create a Maule M7 at the end of runway 33 at EGAC. Once the Maule is created, you can control the engine via the slider.
- 'Load Waypoints' will create a flightplan and apply it to the Maule.
- 'Start Flightplan' applies the flightplan to the Maule and sets the throttle to automatic control



Each of the buttons will only be enabled when the previous action is completed. In this example the waypoints will be created externally in a text file, hence the need for the fileRW.vb module. By doing it this way we avoid hardcoding the waypoints; when you get to the code you will see that it has been black-boxed into a reusable function.

Request Maule

This makes a direct call to `initialiseObject()` where all the work is done (`mdFuncs.vb`). `initialiseObject()` has two parameters; the name of the SimConnect instance and the name of the file containing the initialisation data.

Load Waypoints

This also calls a function in `mdFuncs.vb` (`buildWaypoints()`). Like `initialiseObject()`, it requires two parameters; the name of the SimConnect instance and the path to the waypoint file.

Start Flightplan

If the Maule has been created and the Load Waypoints function has succeeded, this button will apply the flightplan to the Maule and it will start its take-off run. It calls a `SetDataOnSimObject()` function which applies the flightplan, but the usual `SIMCONNECT_SIMOBJECT_TYPE.USER` for the ObjectID has been replaced with the ID issued by the sim when requesting the Maule (variable `newObjectID`).

```
' Start the flightplan
p3d_simconnect.SetDataOnSimObject(P3Data.DEFINE_WAYPOINTS, newObjectID,
SIMCONNECT_DATA_SET_FLAG.DEFAULT, objWpts)
```

mdEnum.vb

We need a request for the Maule (`REQUEST_OBJECT`) and a define for the new waypoints (`DEFINE_WAYPOINTS`). Both the request and the define are created on-the-fly in code i.e. they are not added to the normal `addDataDefinitions()` subroutine.

```
Enum P3Data

    REQUEST_AIRCRAFT_DATA = 1
    REQUEST_OBJECT
    DEFINE_AIRCRAFT_DATA
    DEFINE_WAYPOINTS
    SET_THROTTLE1
    CLIENT_PAUSE

End Enum
```

Also included is the `NewObject` enum. It doesn't need commenting out because the enumeration is not used if you use a copy of `P3Data`.

```
' Use if you don't want to make a copy of P3Data
' for use in the initialiseObject() function
Enum NewObject

    REQUEST_OBJECT

End Enum
```

mdDataDefinitions.vb

By now this module shouldn't have any nasty surprises in it.

mdDisplay.vb

As above, this also shouldn't have any nasty surprises in it. The only thing you might want to look at is the data for latitude, longitude and heading. Each of these is returned in radians, so it's put through the `Rad_to_Deg` (radians to degrees) function in `mdConversions.vb`.

mdFuncs.vb

`mdFuncs.vb` has gained three new functions; `convLatLon()`, `buildWaypoints()` and `initialiseObject()`.

`convLatLon()`

`convLatLon()` converts latitude and longitude from Prepar3D format to a decimal format usable by SimConnect.

`initialiseObject()`

`initialiseObject()` does a *lot* of work on sanity checking the data in the [init] section of `waypoints.txt`. What cannot be checked are the name of the object and its registration number.

Once the sanity checks have been passed and the data read into a `SIMCONNECT_DATA_INITPOSITION` structure, we then create the object. Note that we are using the `AICreateNonATCAircraft` version of the create functions. This is also where that copy of the enumeration that contains `REQUEST_OBJECT` is used:

```
instanceName.AICreateNonATCAircraft(data, temp, initObject, p3dat.REQUEST_OBJECT)
```

If you are using the copy of the `P3Data` enum, when the project is compiled you will get a BC42025 warning:

'Access of shared member, constant member, enum member or nested type through an instance; qualifying expression will not be evaluated'

As we are not evaluating anything the warning can be safely ignored. `initialiseObject()` has almost a comment per line, so it should be easy to follow exactly what it is doing.

`buildWaypoints`

`buildWaypoints()` also does sanity checks on all the information in the waypoints file (many of which are the same as those done in the `initialiseObject()` function) before actually constructing the flightplan. The final action is to convert the waypoints to a polymorphic array (don't ask – look it up) that contains each waypoint definition along with all the data for that waypoint. `buildWaypoints()` has also almost a comment per line.

mdSimconnect.vb

`mdSimconnect.vb` has gained a new handler – `OnRecvAssignedObjectId`. When you successfully initialise the Maule, the `newObjectId` assigned by SimConnect is retrieved here. We need this ID to set the flightplan data on the Maule.

```

Select Case data.dwRequestID

    ' Use with the NewObject enum and comment out the Case line below
    'Case NewObject.REQUEST_OBJECT

    ' Use with the copy of P2Data
    Case P3Data.REQUEST_OBJECT

        newObjectID = data.dwObjectID

        If newObjectID <> 0 Then
            ' Show the GUI when we get assigned an ObjectID
            frmMain.pnlThrottle1.Visible = True
            frmMain.lbl1.Visible = True
            frmMain.lblLever.Visible = True
            frmMain.dgvData.Visible = True
            ' Start gathering the aircraft data
            startUpdates()
        End If

    End Select

```

As you can see, I have also added (but commented out) the equivalent Case code if you want to use the NewObject enumeration.

The Initialisation/Waypoints File

The file must be placed in the same folder as the application executable unless you change the pathing code in the frmMain_Load procedure. Although the file in the example is called 'waypoints.txt', you can give it any name you like simply by changing the file name attached to the string variable 'waypointscfg' in the mdGlobalVars.vb module. If you want to have a separate initialisation and waypoint file, change the code in the frmMain_Load procedure to add another file path and also add a new global variable in mdGlobalVars.vb for the initialisation file.

The text file is in an ini layout. Four rules must be followed to make it a valid file:

- in latitude and longitude, the major figure MUST be followed by a comma i.e. xnn,nn.nn
- waypoint numbering is zero-based
- for waypoints all five key pairs have to be there
- for initialisation all ten key pairs must be there

The [init] section in the waypoints.txt to initialise the simobject (the Maule) is laid out as follows:

```

[init]
Airspeed=0
Altitude=15.0
Latitude=N54,36.73
Longitude=W5,52.78
Pitch=0.0
Bank=0.0
Heading=35
OnGround=1
Object=Maule M7 260C paint3
Reg=G-DAIG

```

The waypoints in the example file provided for the Maule at EGAC are as follows:

```
[wpt.0]
flags=&H4
altitude=500
latitude=N54,37.46
longitude=W5,51.90
airspeed=100
```

```
[wpt.1]
flags=&H4
altitude=500
latitude=N54,38.12
longitude=W5,51.72
airspeed=100
```

The `SIMCONNECT_WAYPOINT_` flags are listed in the sample waypoint file for convenience. Each flag is a hex figure and they must be added together to provide a valid `flags=` line. Fortunately it's a decimal-style addition! The number must be preceded by the Hex notation - in the example we are passing the `SPEED_REQUESTED` flag only. Altitude is in feet and airspeed is in knots.

Correct, compile and try both versions of the enumeration.