

HOW TO CORRECTLY USE SIMCONNECT IN STANDALONE APPLICATIONS IN C++

By AlexShag, October, 25, 2013

alxshag@gmail.ru

alexey_shag@yahoo.com

Introduction

The SimConnect SDK is the programmer's interface to Lockheed Martin® Prepar3D® or Microsoft® Flight Simulator X. Many people who want to programming for these simulators begin their work with examples provided with the SDK. However in a little while, they are faced with the fact that these examples are short programs designed to demonstrate the function calls in a small console application. Moving programming techniques which are based on these examples into traditional Windows GUI application everyone meet some difficulties.

In this article we describe a correctly way to write a simple Windows GUI application which uses the SimConnect functions into C++ classes. At that moment there is not such pattern as it can be seen from Internet search. It is important to note that in this article we do not describe the basics of the SimConnect programming for the simulator. There are standard guides describing the process of add-ons creation the main of which are the [Lockheed Martin® Prepar3D® SDK](#) and the [Microsoft® ESP™ SDK](#). Without going into detail describing functions, we will provide links to the appropriate description of where it is needed.

An important thing in SimConnect programming

And so, we have to write an application which will communicate with the simulator through SimConnect module. It will be simpler standalone Windows GUI-application, not console. Such programs usually call the SimConnect client since SimConnect library plays a server role in the communication process. It provides us information about events occurring in the simulator. The client opens up communications with the server and then requests that certain events and certain object information are passed to it. The client then waits for the information to be received from the server, and then processes it appropriately. It is very important that the SimConnect client is represented as a separated thread in the application process that is working in the shade of GUI interface. User can navigate on GUI interface while the second thread is communicating with the simulator. Such second thread is named as working thread. However the main thread should control this secondary thread via a pointer to the application object. Having this pointer we can call

any member function of the application object in the SimConnect working thread. Moreover, we cannot send messages to the SimConnect message loop from the window. It may be done by registering a user message for the simulator. However the SimConnect client thread can send ordinary window messages to the main application window.

Setting of the problem

Once you start doing multi-threaded programs, the first thing you know, you are faced with the problem of multiple threads accessing the same resource.

The cycle of SimConnect message handling should be created as a separated second thread. This thread is independent of main GUI interface thread. The problem is that the main thread and the working threads are not synchronized. They both live their own independent life. The main application window receives messages from its own thread while SimConnect is working according its own messages dispatch procedure algorithm.

The access violation error may be occurring when some resource is no longer available for the working thread. Often such resource is a disk file or an object in memory of the main thread. This situation most often occurs during the terminating of the application. When the user closes main application window working thread knows nothing about the event and it is still working. The second thread still works and tries to send requests to simulator by using objects of the application, but these objects may be deleted during GUI interface closing. The method of stopping the second thread must be found. The second thread should be correctly terminated before main window will be closed and before the objects will be deleted. We offer the plan in the next form.

About source code of demo program

To illustrate the article a program **SimConnectDemo** was created. This program shows a window where messages from the simulator are displayed. Moreover the program gets some flight data from the simulator and outputs these data on the simulator screen and writes to a disk file. This file may be written with different separators signs and with different extensions. It may be opened in such programs as Microsoft® Excel and other similar to it. Displaying of these data is controlled by the user settings. To illustrate the text of the article many debug macros have been inserted into the source code. You can load the solution in Visual Studio, set a breakpoint and see debug messages in the Output window. These messages help to understand the discussing ideas. This program was created for studying purposes only. This program is not suitable for direct use in the simulator with the gaming or any other purpose. It is possible that this program has bugs and is not working properly.

The executable file absents in the archive due to license restrictions. You can build it from the source code. There is a project for Visual Studio 2010 in attached archive. By default all constants and paths in the project are set to build for Prepar3D® SDK. However it is not a trouble to make these values in correspondingly with compilation for the Microsoft® Flight Simulator. See file “SimConnectDemo Description” for detail information.

Solution of the problem

At the first we have to wrap conventional Windows GUI functions into C++ objects. It can be performed by many ways. Let's suppose MFC-like style for classes naming. We should minimize using global variables as it described in many C++ guides. There is only one global variable in our case CApplication* g_pApp that represents the application. For example, it can be done by the next way:

(Note that this code is not for copying see an original program source code for more detail information.)

```
class CApplication
{
public:
    CApplication();
    ~CApplication();

    // Interaction with the simulator
    CSimThread* m_pSimThread;           // Client's working thread
    CSimConnect* m_pSimConnect;         // Instance of SimConnect class
    DWORD SimThreadProc();              // Simulator messages handler

    // Windows message Handlers
    void OnCreate();                    // WM_CREATE
    void OnClose();                     // WM_CLOSE
    void OnDestroy();                  // WM_DESTROY
};
```

Second problem is issue about SimConnect data structures. These are represented of enumerations and structures providing information on simulator events or transferred data. We suppose that target of the programming is an aircraft object. In any case the process of interaction with the simulator is just procedure of receiving data of the object and sending information to the simulator. As it well knows the [SimConnect RequestDataOnSimObject](#) function is used to request when the SimConnect client is to receive data values for a specific object. These data received in a SIMCONNECT_RECV_SIMOBJECT_DATA structure. The dwData member of this structure is a data array containing information on a specified object in 8-byte (double word) elements. So, we can write a structure which represents parameters of current flight:

```
typedef struct _FLIGHTDATA
{
    char* DatumName;    // Name of the simulation variable
    char* UnitsName;    // Units of the variable
    double Value;        // Value of the variable
}FLIGHTDATA;

// This pointer is returned in dwData of SIMCONNECT_RECV_SIMOBJECT_DATA
// structure
typedef double(*PDOUBLE_DATA_ARRAY)[25];
```

I define some macros to make the program text shorter and readable. May be these macros will be usefully for you (see the attached source code for more detailed description). Look at these macros and decide are they more convenient than original variables or not? In these designations the required variables of the simulator may be written as follows.

```

BEGIN_SIMDATA(FLIGHT_DATA_NUMBER)
    SIMDATA("Incidence Alpha", "Degrees")
    SIMDATA("Incidence Beta", "Degrees")
    SIMDATA("Airspeed Indicated", "Knots")
END_SIMDATA

```

And now we write down an object which wraps the SimConnect library handle and functions of its. We don't set a goal to encapsulate all these functions into C++ objects but only frequently used. And of course this is not full record of the object but only a part which is needed for our narration.

```

class CSimConnect
{
public:
    CSimConnect();
    ~CSimConnect();
    HANDLE m_hSimConnect;

    FLIGHTDATA* m_pFlightData; // pointer to the data array
    DWORD m_dwDataNumber;      // total number of the elements
    void InitFlightDataArray(DWORD dwTotal);

    void UpdateFlightData(LPVOID lpParams); // Simulator Variable Update
};

```

Let show how to use the pointer to flight data array mentioned above. When new data from the simulator are received we should get this pointer from the argument of message handler and to copy data into the structure holding information about the flight. We can write down the procedure in next form:

```

void CALLBACK MyDispatchProc(SIMCONNECT_RECV* pData, DWORD cbData, void
*pContext)
{
    // Create a local copy of the CSimConnect object
    CSimConnect* pSimConnect = static_cast<CSimConnect*>(pContext);
    switch(pData->dwID)
    {
    case SIMCONNECT_RECV_ID_SIMOBJECT_DATA:
        {
            SIMCONNECT_RECV_SIMOBJECT_DATA* pObjData
                =
static_cast<SIMCONNECT_RECV_SIMOBJECT_DATA*>(pData);
            switch(pObjData->dwRequestID)
            {
            case REQUEST_FLIGHT_DATA:
                {
                    SIMCONNECT_RECV_SIMOBJECT_DATA *pObjData
                        =
static_cast<SIMCONNECT_RECV_SIMOBJECT_DATA*>(pData);
                    pSimConnect->UpdateFlightData(&pObjData->dwData);
                }
                break;
            }
            ...Other code here...
        }
    }
}

```

This function is called when the data received. Take an address of the PDOUBLE_DATA_ARRAY array and copy the data into the FLIGHTDATA structure.

```
void CSimConnect::UpdateFlightData(LPVOID lpParams)
{
    PDOUBLE_DATA_ARRAY pDA = static_cast<PDOUBLE_DATA_ARRAY>(lpParams);
    for (UINT i = 0; i < m_dwDataNumber; i++)
        m_pFlightData[i].Value = (*pDA)[i];

    // TODO: Process flight data
}
```

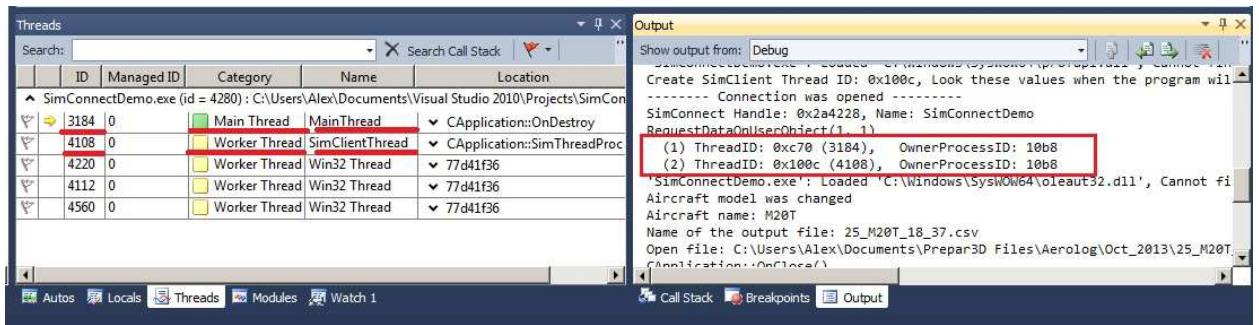
The application object has a pointer to the CSimThread object that allow it to manipulate the child thread through the members of this object. The main thread of our application creates a child thread which performs all work on communication with the server i.e. with the simulator. We should write a global function **MyThreadProc** that represents this working thread. Main thread may control this secondary thread via a pointer to the application object. Having this pointer we can call any member function of the application object in the SimConnect working thread. Here is a code fragment, which is illustrating the idea.

```
DWORD WINAPI MyThreadProc(LPVOID lpThreadParameter)
{
    // Seconds thread gets a pointer to the application object
    CApplication* pApp = static_cast<CApplication*>(lpThreadParameter);
    pApp->SimThreadProc();
    return 0;
}
```

We must have a function that enumerates all thread running in the current process. Such function can be found [here](#). The **ListProcessThreads** function takes a snapshot of the currently executing threads in the system using **CreateToolhelp32Snapshot**, and then it walks through the list.

You can *visualize* these two threads in Visual Studio debugger. There is the [SetThreadName](#) function for this. It allows setting a name which will be displayed in the Threads window of Visual Studio debugger. Note that the name is never really attached to the thread by a windows API call. If you run your application without a debugger then setting a thread name has no effect, therefore you can't retrieve the name.

On the image below you can see the "Threads" window of Visual Studio debugger where these two threads were named by the **SetThreadName** function on the right side and the output from the **ListProcessThreads** function on the left side that show the same threads.



Because we are creating a multi-thread application, we must create a synchronization object which set an interrelation between the main and child threads. To synchronize access to a resource, use one of the [synchronization objects](#) in one of the [wait functions](#). The state of a synchronization object is either *signaled* or *nonsignaled*. The wait functions allow a thread to block its own execution until a specified nonsignaled object is set to the signaled state. There are three methods of the thread synchronization on different processes. These are event, mutex and semaphore. We select an **event object** whose state can be explicitly set to signaled by use of the [SetEvent](#) function as it defined in MSDN. The event object is useful in sending a signal to a thread indicating that a particular event has occurred. Any number of waiting threads, or threads that subsequently begin wait operations for the specified event object by calling one of the wait functions, can be released while the object's state is signaled. A single thread can specify different event objects in several simultaneous overlapped operations, then use one of the multiple-object [wait functions](#) to wait for the state of any one of the event objects to be signaled. For example, it can be done by the next way.

When the working thread is created the "event" object of synchronization is created with it in the global namespace. Initially this object is in non signaled state.

```

BOOL CSimThread::CreateThread(LPCTSTR lpStartAddress, LPVOID
lpParameter)
{
    DWORD dwThreadId;
    m_hEventExit = ::CreateEvent(NULL, TRUE, FALSE, _T("ExitEvent"));
    m_hThread = ::CreateThread(NULL, 0,
                             lpStartAddress, lpParameter, 0, &dwThreadId);

    return TRUE;
}

```

When user closes the main window of GUI interface the main thread informs of the second thread that is time to terminate work with the aid of setting the event in "signaled" state.

```

void CApplication::OnClose()
{
    if(m_pSimThread->m_hThread)
        m_pSimThread->SendSignal();
}

```

The working thread of SimConnect client continuously inspects the event object state. Having found the object in "signaled" state the thread performs needed operations and

when it's done sends back to main thread a reporting message on its readiness to be destroyed. This procedure allows the working thread be terminated by natural way.

This function allows determining message was received or not.

```
BOOL CSimThread::IsTerminated()
{
    return (::WaitForSingleObject(m_hEventExit, 0) ==
        WAIT_OBJECT_0)?TRUE:FALSE);
};
```

The highlighted lines shows messages which the working thread send back to the main thread to destroy main application window.

```
DWORD CApplication::SimThreadProc()
{
    if(m_pSimConnect->Open(APP_NAME))
    {
        m_pSimConnect->Initialize();

        while(!m_pSimConnect->m_bEndSession)
        {
            m_pSimConnect->CallDispatch(MyDispatchProc, m_pSimConnect);
            Sleep(1); // Introduce a delay for a lower resolution.
            if(m_pSimThread->IsTerminated())
                m_pSimConnect->m_bEndSession = TRUE;
        }

        m_pSimConnect->Close();
        ::SendMessage(m_hWndMain, WM_DESTROY, 0, 0);
        return 0;
    }
    else
    {
        Error(IDS_ERROR_CONNECT_TO_SIM);
        ::SendMessage(m_hWndMain, WM_DESTROY, 0, 0);
        return 0;
    }
}
```

Having received the reporting message from the working thread main thread fearlessly may destroy all objects used by the working thread. The next code fragment demonstrates as the main window process the message.

```
void CApplication::OnDestroy()
{
    if(m_pSimThread->m_hThread)
        m_pSimThread->ExitThread();
}
```

Due to the fact that many debug functions were inserted in the demo program you can see the sequence of events in the Visual Studio debugger window.

Conclusion

We can summarise of ideas which are described above. To correctly using SimConnect we must create a multi-threading application in which at least two threads should be working simultaneously. Moreover we must create a synchronization object which set an interrelation between these two threads. The sequence of events when the application should be terminated looks as follows.

- 1) When the working thread is created the "event" object of synchronization is created with it in the global namespace. Initially this object is in non signaled state.
- 2) When user closes the main window of GUI interface the main thread informs of the second thread that is time to terminate work with the aid of setting the event in "signaled" state.
- 3) The working thread of SimConnect client continuously inspects the event object state. Having found the object in "signaled" state the thread performs needed operations and when it's done sends back to main thread a reporting message on its readiness to be destroyed. This procedure allows the working thread be terminated by natural way.
- 4) Having received the reporting message from the working thread main thread fearlessly may destroy all objects used by the working thread. The next code fragment demonstrates as the main window process the message.

In this article we gave only a small example of using SimConnect functions in C++ code. We don't claim comprehensive and profound review of encapsulating all functions of the SimConnect library into C++ objects. We have only demonstrated basic tricks of using these functions in multi-thread applications.

If the author could not explain that either accessible manner, the best way to understand the text of this article is to view the attached source code.

For the author all your responses and wishes about this program are important. To make a suggestion or report a bug or another feature of this product, write to alxshag@gmail.ru or alexey_shag@yahoo.com

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.