### **NatNetSDK Installation and Linking**

1. Visit <https://optitrack.com/support/downloads/developer-tools.html> to download the NatNetSDK developer installer.
2. Open SampleClient project file.

A screenshot of a computer

Description automatically generated

1. Navigate to the SDK folder in your file explorer, and locate the “inc” and “lib” folders

A screenshot of a computer

Description automatically generated

1. In the Property Pages of the Visual Studio, navigate to C/C++ > General, then “Additional Include Directories”
2. Paste the file path of the “inc” folder, which may look the same as this example path: *C:\Users\egrstudent\Documents\OptiTrack\NatNetSDK\include*
3. In the Property Pages of the Visual Studio, navigate to C/C++ > Preprocessor > Preprocessor Definitions. Then, add “\_CRT\_SECURE\_NO\_WARNINGS”
4. Now navigate to the Linker section for the library requirements
5. Go to Linker > General, then “Additional Library Directories”
6. Paste the file path of the “lib” folder, which may look the same as this example path: *C:\Users\egrstudent\Documents\OptiTrack\NatNetSDK\lib\x64*
7. Then, go to Linker > Input, then “Additional Dependencies”. Then, add “NatNetLib.lib”
8. Then go to the directory ( *..\projectXX\_name/projectXX\_name*) in the Visual Studio project and copy the “NatNetLib.dll” file into the directory, which should be found in the following location:

|  |
| --- |
| Dear Steven,  Thank you very much for your response. Your email was incredibly helpful in clarifying the structure of the C++ example.  I have attempted to modify the OutputFrameQueueToConsole function. Within this function, I created an array to collect x, y, and z labeled marker positions, passing them as arguments. My goal is to access this array from LabVIEW.  In the C++ example, it seems that DisplayQue contains several frame data at each iteration. I was wondering how I could calculate the number of frames stored in the queue to dynamically allocate the array size needed to store the x, y, and z positions from all frames.  I was able to display a fraction of streaming data to LabVIEW, but it often crashes due to memory issue..  **[Reply]**  From the code, it looks like the networkQueue is used to collect frames before transferring them into the displayQueue. You might be able to get a count from one of these queues to help determine how large your array needs to be in order to store all the data you wish to transmit. It should be a simple ".size()" in order to get the count. |

**LabView DLL**

In addition to the C++ based program used in the original project, a DLL was constructed to allow LabView to control the motors and query their information so a comprehensive program can be fully made in LabView. This was primarily achieved through a “wrapper”, in which the functions to be converted to LabView were written in C++ like usual, then wrapped around a C based function so LabView could correctly interpret it (since LabView only accepts C based user generated libraries). The DLL was made using Visual Studio’s DLL template and tutorials online pertaining to creating DLL’s using Visual Studio [3], and revolves around 2 important files: Source.cpp, and Header.h. An excerpt each file is shown below:

A screen shot of a computer program

Description automatically generated

The top image is a section from the “Header.h” file, and primarily serves to list all the function signatures of the soon to be converted functions. The return types of each function must be compatible with C, and each function is marked with the “extern “C”” keyword to preserve the function names after conversion. Then in the “Source.cpp” file, the actual function body is written, though it is important to make sure that the signatures between these 2 files match for each function. There should be the same number of function signatures in both of these files as they are 1 to 1 matches. Once all the function are written, one build the project in Visual Studio to produce the DLL. Then, this DLL and the corresponding header file that has the function signatures can be imported into LabVIEW, where each function will be its own newly created VI.

After successfully building the wrapper code, go to import menu in the LabVIEW and do the importing process. Also, copy and paste the OptitrackWrapper.dll under user.lib>OptitrackWrapper folder (see below).

A screenshot of a computer

Description automatically generated

**How to pass LabVIEW's array in DLL** [3]

The following figures illustrate the concept

A computer code with text

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

***Take a note, that you is responsible for range control, attempt to write array’s elements outside of the allocated memory will (may) caused exception (crash).***

In my code, the size of the array passed from LabVIEW (float, 4 bytes per element) is determined by the formula (number of markers \* 4) + 1. Each marker consists of its ID and three coordinates, with one additional element used for the timestamp. Within the function, a new array named vec is created using std::vector. The marker coordinates are stored in this vec array, and its contents are then copied to the passed argument (arr) at the end of the function (see the code snippet below).

It is crucial to ensure that the range of operations does not exceed the size of vec, as doing so will result in a crash. Since the exact number of frames stored in displayQueue is unknown, the initial size of vec was arbitrarily set to 10,000. For example, if the OutputFrameQueueToConsole function is called at every 100 ms, more than 10 frames per function call might be stored, which will cause the size of vec to grow. For example, 8 markers \* 4+ 1 =33, and 10 frames result in 333=10\*33. However, the size = 10,000 does not necessarily refer to the estimated size. That’s the reason I made up to a big value in order to avoid crash. In the event of a crash, you should increase the initial size of vec to accommodate the higher number of frames.

A computer screen with text

Description automatically generated

A black screen with white text

Description automatically generated

A diagram of a computer

Description automatically generated

**About LabVIEW Program**

The LabVIEW program was tested under the conditions of two rigid bodies and eight markers, with four markers assigned to each rigid body. The intended output format for the data is [Time, marker ID, x, y, z, marker ID, x, y, z, … ], resulting in a total of 33 elements (1 for time + 4 attributes per marker × 8 markers).

Within the while loop, the OutputFrameQueueToConsole function is called, followed by a 5 ms pause. During this pause, frame data continues to accumulate in the displayQueue object (due to background thread programming). Consequently, the total amount of data retrieved depends on the number of frames stored during the interval.

For example:

* If 2 frames are stored within 5 ms, the total number of elements is 2 × 33 = 66.
* If only 1 frame is stored, the total is 1 × 33 = 33.

If the pause duration increases to 50 ms, the number of frames stored can range between 6 and 7. In this case, the size of the array passed to LabVIEW must be appropriately adjusted. However, since the exact number of frames stored during a given interval is unpredictable, passing an array size of only 33 for a 50 ms interval would result in losing all but the first frame’s data. To handle this, the array size could be increased to accommodate multiple frames (e.g., 10 × 33 = 330). But, if only 7 frames are queued, any unused array elements (e.g., from index 232 to 330) would be filled with the initialized zeros.

In the current LabVIEW program, the pause interval is set to 5 ms. According to my testing, the queue size during this interval is typically 1 or 0. Therefore, in cases where the OutputFrameQueueToConsole function outputs zero (indicating no data), no assignment is made to the output array.

**Developing Notes:**

This project consists of three separate project files: Project1, Project2, and OptitrackWrapper.

* **Project1**: Executes the SampleClient.cpp file with minor modifications.
* **Project2**: Runs a version of the code parsed and modified from SampleClient.cpp.
* **OptitrackWrapper**: Compiles a DLL designed for use with LabVIEW.

**References**

[1] <https://learn.microsoft.com/en-us/cpp/build/walkthrough-creating-and-using-a-dynamic-link-library-cpp?view=msvc-170>

[2] Wrapper DLL code

<https://github.com/biokimcom/OptitrackWrapper>

[3] how to pass LabVIEW’s array in DLL

<https://blog.dmitriev.de/labview/dll/labview_array_in_dll>

1. **Appendix: C++ code**

/\*

\* The MotorManager import. MotorManager is a custom class centered around the control of the dual software controlled servo motors

\* in the robot, and heavily uses the Teknic ClearView SDK.

\*/

#include "MotorManager.h"

/\*

\* The pubSysCls import. This is the main header file from the ClearView SDK, and is only header file required to manually import aside from

\* the dependecies added through the project properties window.

\*/

#include "pubSysCls.h"

#include <conio.h>

/\*

\* VERY IMPORTANT:

\* This demo expects two motors in the robot configuration, and will not operate under any other general conditions. This demo is

\* specifically for robot purposes, and not for general motor demos.

\*

\* The program expects the motors to be zero'd already. This must be done externally using the ClearView diagnostic program.

\*/

int main()

{

//CREATE SYSTEM AND PORT OBJECTS FOR MOTOR CONTROL

sFnd::SysManager\* myMgr = sFnd::SysManager::Instance();

std::vector<std::string> comHubPorts;

try {

size\_t portCount = 0;

sFnd::SysManager::FindComHubPorts(comHubPorts);

printf("Found %zd SC Hubs\n", comHubPorts.size());

for (portCount = 0; portCount < comHubPorts.size() && portCount < NET\_CONTROLLER\_MAX; portCount++) {

myMgr->ComHubPort(portCount, comHubPorts[portCount].c\_str()); //define the first SC Hub port (port 0) to be associated

// with COM portnum (as seen in device manager)

}

if (portCount < 0) {

printf("Unable to locate SC hub port\n");

}

myMgr->PortsOpen(portCount); //Open the port

for (size\_t i = 0; i < portCount; i++) {

sFnd::IPort& myPort = myMgr->Ports(i);

printf(" Port[%d]: state=%d, nodes=%d\n",

myPort.NetNumber(), myPort.OpenState(), myPort.NodeCount());

}

}

catch (sFnd::mnErr& theErr) {

printf("Failed to disable Nodes n\n");

//This statement will print the address of the error, the error code (defined by the mnErr class),

//as well as the corresponding error message.

printf("Caught error: addr=%d, err=0x%08x\nmsg=%s\n", theErr.TheAddr, theErr.ErrorCode, theErr.ErrorMsg);

}

if (comHubPorts.size() > 0) {

//Create MotorManager with actual system object, port, and the two motors found.

MotorManager mtrManager(myMgr, myMgr->Ports(0), myMgr->Ports(0).Nodes(0), myMgr->Ports(0).Nodes(1));

//THIS IS WHERE THE DEMO OCCURS, CONTROLLED VIA A WHILE LOOP

// Keep looping until 'Q' is pressed

char userInput;

//THIS IS THE CENTER POINT OF THE HANDLE WORKSPACE IN ABSOLUTE COORDINATES (origin at motor shaft)

//This is for reference, and not actively used in the demo.

std::pair<double, double> workOrigin(0, 22.5); //IN INCHES

//THE WORKSPACE IS 18 INCHES IN THE X DIRECTION AND 15 INCHES IN Y DIRECTION

//THESE BOUNDS ARE IN REFERENCE TO THE MOTOR SHAFTS, LOOKING FORWARD FROM BEHIND ROBOT.

//FOR EXAMPLE, TO THE RIGHT AND FURTHER FROM THE ROBOT IS THE UPPER RIGHT BOUND.

//These are for reference, and not actively used in the demo.

std::pair<double, double> upperRightBound(9, 30);

std::pair<double, double> lowerRightBound(9, 15);

std::pair<double, double> upperLeftBound(-9, 30);

std::pair<double, double> lowerLeftBound(-9, 15);

//VERY IMPORTANT: The motors are enabled during this demo, meaning free movement is not available.

mtrManager.enableMotor(mtrManager.motorTheta);

mtrManager.enableMotor(mtrManager.motorGamma);

//BUILD THE NORMAL AND UNIT VECTORS ONCE USING A B AND C

double a = 0;

double b = 1;

double c = -14;

double gain = 0.15;

std::pair<double, double> normVector = mtrManager.getNormVector(a, b, c);

std::pair<double, double> unitVector = mtrManager.getUnitVector(a, b, c);

// Keep looping until 'end' is typed

while (true) {

//Motor Movement Code using MotorManager

/\*queryMotor: This function simply returns the count value of the given motor.MotorManager has 2 internal motor objects :

\* motorTheta, and motorGamma. You can provide the queryMotor function and any other function from MotorManager with these

\* motor porperties to fulfill the motor object argument.

\*/

double thetaCounts = mtrManager.queryMotor(mtrManager.motorTheta);

double gammaCounts = mtrManager.queryMotor(mtrManager.motorGamma);

/\*getHandlePos: This function takes in the 2 motor counts separately (theta first, then gamma) and returns a pair of doubles that

\* represents the real-life handle position.

\*/

std::pair<double, double> handlePos = mtrManager.getHandlePos(thetaCounts, gammaCounts);

std::pair<double, double> forceVector = mtrManager.buildForceVector(gain, handlePos.first, handlePos.second, normVector, unitVector, a, b, c);

std::cout << "\t\t\t\tForce X: " << forceVector.first << std::endl;

std::cout << "\t\t\t\tForce Y: " << forceVector.second << std::endl;

//Carry out the commanded torque

std::pair<double, double> currentCounts(thetaCounts, gammaCounts);

mtrManager.activateSpringWall(currentCounts, forceVector.first, forceVector.second);

// Check if a key has been pressed

if (\_kbhit()) {

// Get the pressed key

userInput = \_getch();

// Check if the pressed key is 'Q' or 'q'

if (userInput == 'Q' || userInput == 'q') {

std::cout << "Exiting demo." << std::endl;

//Once the demo is over, disable the motors

mtrManager.disableMotor(mtrManager.motorTheta);

mtrManager.disableMotor(mtrManager.motorGamma);

break; // Exit the loop

}

}

}

}

else {

std::cout << "No motors detected" << std::endl;

}

return 0;

}