The DLL component – Getting started, Part 1: Setting up a Netbeans Project

November 13, 2014

The DLL (Dynamic Link Library) component is arguably the most powerful component in Flowstone, it allows us to extend Flowstones functionally (albeit in a limited way) in ways previously not possible. It also enables us to do processing that would be just too slow in Ruby or Green.

You may be thinking because it is named “DLL component” that this means we can load any DLL, this is not the case. Flowstone is expecting a very specific type of function with specific parameters, which any old DLL will not have, hence they cannot be used directly. You are going to have to get your hands dirty writing your own DLL, which means writing C++ and using an IDE and compiler. This I think is why the DLL component doesn’t get much love over on the Flowstone forums. Most of us are using Flowstone to avoid having to write or learn C++! But if you want to really push the boundaries of what is possible in Flowstone then learning C++ and the DLL component is a must.

In this article I will give instruction on how to set up a Flowstone DLL project in the Netbeans IDE.

Netbeans IDE

Firstly, why Netbeans? Well this is just my personal preference because I use it for Java projects and it also supports C++. It helps that it is also a really great IDE and free .

There is a bit of work involved in getting Netbeans set up but hopefully this won’t be too much hassle for you.

Firstly you need to download the Netbeans IDE . Choose either the “C/C++” version or “All” , “All” includes many things that may not be of interest to you, if so just select the “C/C++” version. Also it is important to note that the IDE needs Java installed to work.

Getting Netbeans Ready for C++

We have a bit more work to do to set the IDE up to work with C++. The Netbeans IDE doesn’t come with a C++ compiler so this has to be installed separately.

The compiler I use is MinGW – (Minimalist GNU for Windows) download it and follow these steps (some of this is copy pasted from the Netbeans site).

1. Run the installer.
2. Accept the default install location C:\MinGW, don’t change it.
3. In the MinGW installer select the following components to install:

- mingw-developer-toolkit
- mingw32-base
- msys-base
- mingw32-gcc-g++

4. Select Installation > Apply Changes from the main menu. Click Apply to confirm the installation and wait while the installer downloads the components.
5. Click Close when the packages installation is completed.

Now you must add the paths to the the binaries for MinGW and MSYS tools to your PATH. If you installed to the default location the paths are C:\MinGW\bin and C:\MinGW\MSYS\1.0\bin.

To edit your PATH environment variable in Windows:

1. Open the Environment Variables window:
    1. On Windows XP and Windows 2000: Right-click My Computer > Properties > Advanced tab > Environment Variables button.
    2. On Windows Vista and Windows 7: Right-click My Computer > Properties > Advanced System Settings link > Environment Variables button.
    3. On Windows 8: Right-Click My-PC > Properties> Advanced System Settings > Environment Variables button.
2. In the Environment Variables window, select the Path variable in the Systems Variable section and click Edit.
3. At the end of the path, insert a semi-colon and add the paths to the executables for MinGW and MSYS. Use semi-colons between the paths and do not use any spaces. Be careful not to remove anything already on your PATH or your computer might not work correctly.

    When you are finished, your path should look similar to the following:

```
%SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem;
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0;C:\MinGW\bin;C:\MinGW\MSYS\1.0\bin
```

4. Click OK in the Environment Variables window.

Setting up the Flowstone DLL project

So now if you followed the previous steps correctly Netbeans should be set up to work with C++ projects.

Now open Netbeans (if it fails to open you may not have Java installed) and follow these steps.

1. Click File > New Project.
2. In the Categories you should see “C/C++” click that category.
3. In projects select “C/C++ Dynamic Library” and click Next.
4. Enter a project name and then click Finish.

Now in the left corner you should see your project with folders such as Header Files, Resource Files ect.

* * * MISSING GRAPHIC * * * *

Now right click the “Source Files” folder select New > C++ Source File. Give the file the name “function” and click Finish.

The new file should automatically open, now just copy and paste in this bit of code…

```
////////////////////////////////////////////////////////////////////////////////

// Helper Macros - use as you please
#define GETFLOAT(p) *((float*)&p)
#define GETBOOL(p) *((bool*)&p)
#define GETINT(p) p
#define GETSTRING(p) *((char**)&p)
#define GETFLOATARRAY(p) p ? ((float*)p+1) : 0
#define GETINTARRAY(p) p ? ((int*)p+1) : 0
#define GETSTRINGARRAY(p) p ? ((char**)p+1) : 0
#define GETARRAYSIZE(p) p ? *((int*)p) : 0
#define GETFRAME(p) p ? ((float*)p+1) : 0
#define GETFRAMESIZE(p) p ? *((int*)p) : 0
#define GETBITMAPWIDTH(p) p ? *((int*)p) : 0
#define GETBITMAPHEIGHT(p) p ? *((int*)p+1) : 0
#define GETBITMAPCHANNELS(p) p ? *((int*)p+2) : 0
#define GETBITMAPDATA(p) p ? ((BYTE*)p+12) : 0
#define GETBITMAPBYTES(p) p ? *((int*)p) * *((int*)p+1) * *((int*)p+2) : 0
#define NEWINTARRAY(p,n) if(n>0) { *((int**)&p)=new int[n+1]; ((int*)p)[0]=n; }
#define NEWFLOATARRAY(p,n) if(n>0) { *((float**)&p)=new float[n+1]; ((int*)p)[0]=n; }
#define NEWSTRINGARRAY(p,n) if(n>0) { *((char***)&p)=new char*[n+1]; ((int*)p)[0]=n;
}
#define DELETESTRING(p) if(p) { delete *((char**)&p); p=0; }
#define DELETEINTARRAY(p) if(p) { delete *((int**)&p); p=0; }
#define DELETEFLOATARRAY(p) if(p) { delete *((float**)&p); p=0; }
#define DELETESTRINGARRAY(p) if(p) { for( int j=0; j<*((int*)p); j++ ) { if( ((char**
)p+1)[j] ) delete ((char**)p+1)[j]; } delete *((char***)&p); p=0; }

////////////////////////////////////////////////////////////////////////////////

extern "C" __declspec(dllexport) void myFunction( int nParams, int* pIn, int* pOut )
{
}
```

Now select File > Save all to save your project.

All the defines are just handy methods for casting to certain types and creating/deleting Arrays. Trust me you will need these! Some of those methods are pretty ugly and confusing. The reason why they are needed is because “pIn” and “pOut” are int arrays so only int’s can be passed to the function (although we can actually pass many types)

these int's are the memory addresses of each type with the exception of int which is just passed "as is". I will explain all this in more detail in the next article.

Dependencies

To avoid dependencies in your resulting DLL we can set a couple of linker options, this will ensure your DLL works on other PCs that could be missing some libraries. Follow these steps…

1. Right click your project name, select properties > Linker.
2. Click the box with the three dots next to libraries then click the Add option button.
3. Click OK to select "Static bindings"
4. Click Add option button again and select "Other option"
5. Add "-static-libgcc" (without quotes) press OK. Click OK again then Click Apply.

Now you need to do the same again for the release build (A Pain I know)

1. Click "Configuration" and select Release.
2. Follow the previous 5 steps again.

All setup

Phew… that is it now I promise. Now you can get on with some coding!
In the next article on the DLL component I will introduce some basic C++ code in a few functions which will highlight everything you need to know when creating a Flowstone DLL.

The DLL Component – Getting Started, Part 2: Creating a basic DLL

November 17, 2014

In this article I'll show you the creation of a basic Flowstone DLL that implements a few simple functions that hopefully will demonstrate most things you need to know about creating a Flowstone DLL (Note I won't be explaining C++ in any real depth because that is beyond the scope of this tutorial).

I am also assuming you have followed the previous article and have a Netbeans project already set up and ready to go, although you should be able to follow this tutorial in any IDE.

The function

First let's take a look at the definition of a function that Flowstone is expecting:

```
extern "C" __declspec(dllexport) void myFunction( int nParams, int* pIn, int* pOut )
{
}
```

You should recognize that function as the one in your project. All functions in your DLL have to be written exactly like that. We can change the name of the function to anything we like though.

If you want to understand what " extern "C" __declspec(dllexport) " really means then just do a Google search on that, in simple terms it is allowing us to export the function to be called outside the DLL and with the C calling convention, which prevents Name mangling.

The rest of the function is simple enough; "void" is the return type, which means it literally returns nothing. "myFunction" is the function name, this is the only part of the declaration that can be changed. "nParams" is an integer value giving the number of parameters. "pIn" is an int Pointer to an int array containing the memory addresses for all our inputs to the DLL component; and of course "pOut" is pointing to an int array holding all our outputs.

Now I will show you a few basic algorithms and how they can be achieved using the DLL component. In the next article (part 3) I will show you how to set everything up and call the functions from within Flowstone.

Writing a reverse int array function

OK so let's get straight on with writing our first function. This function will simply take an int array reverse it and then send it to the output. Simple, but will shed some light on a couple of things.

```
extern "C" __declspec(dllexport) void reverseIntArray( int nParams, int* pIn, int* pO
ut )
{
    int* intArrayIn = GETINTARRAY(pIn[0]); //Input 0 is an int array
    int size = GETARRAYSIZE(pIn[0]);

     //Here we create a new output array if needed
     if(pOut[0]) //check if pOut[0] exists (not NULL)
     {
         if(size != GETARRAYSIZE(pOut[0])) //Check if size of input/output arrays are
the same
         {
            DELETEINTARRAY(pOut[0]); //Delete old one
            NEWINTARRAY(pOut[0], size); //Create with new size
         }
     }else{
        NEWINTARRAY(pOut[0], size); //No output array so create one.
     }

     int* intArrayOut = GETINTARRAY(pOut[0]); //Get a pointer to the output array

     int j=0;
     for(int i=size-1; i >= 0; i--) //Loop backwards starting from "size"
     {
         intArrayOut[j] = intArrayIn[i];
         j++; //j increments forwards as i goes backwards
     }
}
```

So this function is showing us the usage of some of those funky defines! Firstly we are using GETINTARRAY to get an int array from the first input, then we use GETARRAYSIZE to get its size.

Now we check if an output array needs to be created. If pOut[0] is NULL we create a new array for the output, if it's not NULL we check the input array size against the output array size, if they do not match(meaning input array has changed size since last time the function was called) then we delete the old array and create a new one so the sizes match.

Now we use GETINTARRAY to get a pointer to the output array.
The rest of the code should be pretty self-explanatory to anyone with basic programming knowledge.

So what this simple function is showing us is that we need to use those defines to delete and create new arrays ect. When we pass an array in Flowstone to the DLL component this array is copied (behind the scenes) which prevent us

from modifying the original. We then are passed the address of this copied array. We then have to get this address into a form we can use, hence the GETINTARRAY function which gives us an int pointer which points to the beginning of the array. Once we have the pointer we can treat it like any array in C++.

In Flowstone when you assign an input on the DLL component, a corresponding output is created. So if pIn[0] is an int array then so is pOut[0], only we have to manually create the output array ourselves and then copy our data into that, as shown in the example above.

Writing a Mem section function

This function is essentially just an array section function (output a section of an array) but for Mem data type. The DLL component doesn't support the Mem data type, but we shall not let that stop us! We can pass a Mem by passing its address and casting that to a float pointer, so here is the function….

```
extern "C" __declspec(dllexport) void memSection( int nParams, int* pIn, int* pOut )
{
    //Main Mem
    float* mem = (float*)pIn[0]; //Simply cast the address to a float pointer
    int size = pIn[1]; //Passing the size on the second input

    //Mem Section
    float* memSection = (float*)pIn[2]; //Mem to copy section too
    int startIndex = pIn[3]; //Where section will start from
    int sectionSize = pIn[4];

     for(int i=0; i < sectionSize; i++)
     {
         if(i+startIndex < size) //Make sure we don't read beyond the size of the Mem
         {
             memSection[i] = mem[i+startIndex];

         }
     }

}
```

Here we are passing two mems, "mem" and "memSection". "mem" could be a wave file, "memSection" is a mem that we need to create in Flowstone using the mem create component. This will be shown in the next article which focuses purely on the Flowstone side of things and will complete all these functions. The main purpose of this code is to show you a simple but rather useful example of how to read and write to the mem data type in a Flowstone DLL.

Writing a basic over drive function

This is just a very basic overdrive algorithm to show how to write effects for streams (audio).
Here is the code…

```
extern "C" __declspec(dllexport) void overDrive( int nParams, int* pIn, int* pOut )
{

    float* audio = GETFRAME(pIn[0]); //A frame is just a float array
    int frameSize = GETFRAMESIZE(pIn[0]);

    float gain = GETFLOAT(pIn[1]);
    float in=0;

    for(int i =0; i < frameSize; i++)
    {
        in = audio[i]*gain; //Apply gain

        audio[i] = in/((in*in*0.25)+1); //Saturation
    }
}
```

Again this should be pretty simple stuff to anyone with a bit of programming experience. Audio is passed to the function in frames. Within Flowstone or an exe the size of the frame is determined by the buffer size of your audio driver (ASIO, Direct sound etc.). Within a VST host the frame size is determined by the host and can differ to that of the audio driver. The frame size can even change on the fly. You can never rely on a fixed sized frame, so always use GETFRAMESIZE and never try hard coding the frame size.

The other interesting thing about frames is that they are treated a bit differently to the other data types. You may have noticed in the code that there are no references to any pOuts (outputs), hence nothing is explicitly passed to an output. We perform the algorithm on the input frame and then assign that back to the very same input frame, then as if by magic it is passed to the output. This assignment back to the input frame does not actually modify the original input frame (as passed from Flowstone), because in the DLL the input frame is the output frame! You can test this by changing pIn[0] for pOut[0] in the code and it will work exactly the same. So what is going on here? Flowstone is copying the original contents of the input frame to another frame, this copy is then assigned to both the input and output (pIn[0] and pOut[0]), hence we can read and write from either.

Building the DLL

Now we have three simple functions to call from inside Flowstone, the next step is to build the DLL and then we can open Flowstone and get started on creating the schematic.

To build the DLL simply right click the project name and click “Build” then if successful you will see this message in the output window “BUILD SUCCESSFUL (total time: xxs)”, and that is it!

In the next article I will show how to set the DLL component up in Flowstone to call our functions.

The DLL Component – Getting Started

Part 3: The Flowstone Schematic

November 21, 2014

In this part you will learn about the DLL component in Flowstone with some basic examples.

Here is a schematic that demonstrates all the functions from part 2 in action DLLExamplesV1.fsm . Inside you will find some comments that explain somethings that may be unclear.

I think examples speak larger volumes than words so I have restrained from doing a step by step guide to setting things up because the DLL component is pretty self-explanatory, the examples are also simple enough to understand. Instead I will offer a few hints and best practices when working with this component.

Creating and labelling inputs

We create inputs by click and dragging the little rectangle with 6 dots down until we have enough inputs set. Each input will take the type of the input before it. Right click an input to change its type.

Naming inputs is slightly more awkward. Click on the DLL component to highlight it. The inputs names and short description will show for each input. Now you have to click one of them as though you were going to edit it and then you can press TAB to tab down to the inputs you just created and then name each one. You can do a short name (which shows on the front of the component) and a longer description which shows when highlighted. Do this by adding a \n (newline) , for example (without quotes) “Gain\n Apply gain to the input signal. Range 1 -4 ” .

Execute

The DLL component needs to be triggered via one of the “exec” inputs before it can do anything. There are two, one for “Green” events and one for Ruby events. Both can be used at the same time. It is important to know that these inputs are processed in different threads(processes) so triggering the Ruby input will not send triggers through green outputs (but values are updated) and vice a versa.

In the over drive example in the schematic you can see the Ruby exec input being triggered by the output of a mono to frame, this gets triggered in perfect sync with the audio stream.

Best Practices

Top level DLL locator

In the example schematic there is a top level module used to locate your DLL and send its file location to all other DLL components. This is a best practice I recommend to follow. It also includes a button for resetting and unloading, this is essential if you have many components using the same DLL. Whilst debugging and making improvements to the DLL it is likely that you will have Flowstone open to test any changes straight away; the DLL will not build unless it has been unloaded from Flowstone first. With the top level reset button you can unload from all DLL components at once and then complete your build. The DLL will reload automatically when triggering “exec”.

Avoid embedding until necessary

The DLL component comes with a “boolean input” to embed the DLL into the component. My advice is to not manually embed until you absolutely need too. For instance when sharing a schematic or exporting to VST or EXE. In the case of exporting we can choose the option to embed any DLLs in the final product, so there is no need to do this during development. If you are sharing a schematic but don’t want share the actual DLL then embed the DLL just before sharing. In the example schematic there is a button to embed the DLL. It is handy to have a “embed” button in one place so you can embed all DLLs at once before saving and sharing the schematic.

It may be desirable to not embed the DLL in some cases, for instance you may want to update the DLL separately from your application or you may want to share the same DLL among instances of your app, saving memory usage. If neither of the scenarios applies to you then do your end users a favor and embed the DLL.

Create the DLL first then the schematic

To some people it may seem to make more sense to create the schematic first and set the DLL component up with the inputs you think you will need. I did this at first but found it was always better to create the DLL first without even opening Flowstone. The reason being that algorithms and design can change, you may think you will need this or that input/output but until you actually write the algorithm you cannot be 100 percent sure. It is better to finish the function first so you know exactly what inputs you need from Flowstone. Otherwise you may find yourself going backwards and forwards adding or removing inputs you thought you need but actually don’t. But I digress this is only really a problem if you are expecting lots of inputs, very simple stuff would be OK doing the schematic first.

That is it for now!

Hopefully this brief series has helped you to get started with creating and using your own DLLs with Flowstone. Studying the code examples as well as the schematic should teach you all the basics you need to know about working with this component.

Comments:

kohugaly - November 21, 2014 - 11:21 pm
Just a little correction… When you trigger the DLL component with either green or ruby exec input it will update the values in all outputs, however green triggers will be send only when green input was used and ruby triggers only when ruby input was used.

> **Exo - November 21, 2014 - 11:50 pm**
> Thanks I made a little edit

Youlean - March 31, 2015 - 2:56 pm
Thanks Exo, with this guide I was able to build my first DLL file.
BTW, should I do “Static bindings” procedure when compiling with Visual Studio 2012?

> **Exo - March 31, 2015 - 6:05 pm**
> Great, glad it helped you.
> Yes still do that. The DLL will work for you without it but may not work on another version of windows. It is to ensure compatibility on other PCs.