Documentation

Version 1.0.0

Ethan Codron

February 13, 2026

Contents

1 Introduction

Welcome to the official documentation for Network Dynamics eXperimentation Library (NDXL), a software package for simulating vertex dynamics in complex networks. This document will cover everything from acquiring the software, to importing it into your preferred working environment, to understanding how the library objects and functions work, to setting up and simulating various networks.

Note: this documentation is for NDXL version 1.0.0 ONLY, and we cannot guarantee that future iterations of NDXL will function if used in the way outlined in this document.

1.1 How to use the documentation

If you are a first-time user of NDXL, we highly recommend reading the documentation right-through like a book; it explains everything from what objects and functions are available, to how they work and what the library expects from the user every step of the way. The documentation also covers how to import NDXL into a C/C++ project, and the code examples show how to use the C++ wrapper to setup and simulate networks.

If you are comfortable with importing libraries in C/C++, you can skip that section and go straight to the objects and functions.

The code examples at the end are also designed to give an intuitive sense of how the library works for those who want to jump straight in, but it does help to know why certain parameters or function signatures look the way they do, which is why we recommend at least skimming through the functional documentation first.

1.2 What is NDXL?

NDXL, at its core, is a collection of objects and functions that have been programmed to simulate complex networks. In essence, what this means is NDXL provides the user with a means for setting up a network made of vertices (nodes) who have (potentially unique) differential equations describing how their state evolves over time. The user may then specify how these vertices

are connected via edges; together, the vertices and edges form the network, which is what gets simulated at the end of the day.

1.3 Which version of NDXL do I use?

Regardless of version, NDXL is always packaged as a library file in two variants, namely, static and dynamic. Static libraries are built into the final executable at compile time, so they are faster but make the executable larger. Dynamic libraries are linked at run time, so they are slower but do not increase the size of the executable as much, and can also be used by multiple programs or processes at once.

On Windows, the relevant files are NDXL.lib and NDXL.dll, whereas on Unix-like operating systems, the relevant files are NDXL.a and NDXL.so. Whether you use the static or dynamic library versions is really up to you, and this documentation will cover how to use both.

1.4 What languages does NDXL support?

NDXL is language-invariant, as it is compiled using only the C standard library. What this means is, once you have the library files (recall NDXL.lib and NDXL.dll on Windows, and NDXL.a and NDXL.so on Unix), you can import the objects and functions within those library files into whatever language you may be using, as long as the names and signatures match exactly. You can then create your own wrappers for those objects and functions if you so wish.

By default, NDXL ships with a C++ wrapper layer, and in future we will add official support for other languages like Python, Julia, and so forth. The process of importing a library varies between languages, which makes adding support a lengthy process, but we will show towards the end of the documentation how such a task might be achieved.

1.5 What operating systems does NDXL support?

NDXL officially supports Microsoft Windows and Unix-like operating systems such as Linux. At the time of writing, we are working on adding support for macOS as well.

2 Acquiring the library

As mentioned earlier, NDXL ships as a compiled library for both Windows and Unix. You will most likely acquire NDXL as a zipped folder (unzip it) containing the following sub-folders:

- include
- lib

The include sub-folder contains the C header files necessary to import NDXL; they contain the object and function definitions for everything inside the library, which effectively tells the compiler what signatures to look for (it is these function and object names that must be defined if you want to use NDXL from another language). These files also contain the C++ wrappers.

The lib sub-folder then has another two sub-folders inside it, namely, Linux and x64. Both of these sub-folders have another two sub-folders inside, called Dynamic and Static, within which you will find the actual library files:

```
lib
├── Linux
│   ├── Dynamic
│   │   └── libNDXL.so
│   └── Static
│       └── libNDXL.a
└── x64
    ├── Dynamic
    │   ├── NDXL.dll
    │   └── NDXL.lib
    └── Static
        └── NDXL.lib
```

Notice that the file NDXL.lib appears twice – in both the Dynamic and Static subfolders for x64. **This is not a mistake, and these files are NOT identical.** The reason for the seemingly duplicate files is the way Windows handles imports for dynamics libraries; the "dynamic" NDXL.lib provides the linker with the names and locations of the exported symbols in NDXL.dll, whereas the "static" NDXL.lib contains those exported symbols directly.

2.1 Including the library on Unix

Incorporating NDXL is fairly straightforward on traditionally command-line interface (CLI) compilers (e.g. gcc, MinGW, Cygwin, clang, etc), as it merely requires a couple extra parameters in the compilation command itself. For Unix users, we will assume you are compiling with gcc, and that your folder structure looks as follows:

```
My Project
├── src
│   ├── main.cpp
│   └── other header files
├── other folders
└── other files
```

Copy the extracted NDXL folders into your project directory; we recommend copying the folders DIRECTLY into your working directory:

```
My Project
├── src
│   ├── main.cpp
│   └── other header files
├── include
│   └── NDXL headers...
├── lib
│   ├── Linux
│   │   ├── Dynamic
│   │   │   └── libNDXL.so
│   │   └── Static
│   │       └── libNDXL.a
│   └── x64
│       ├── Dynamic
│       │   ├── NDXL.dll
│       │   └── NDXL.lib
│       └── Static
│           └── NDXL.lib
├── other folders
└── other files
```

From a terminal opened inside "My Project", you will then compile your executable as normal, but with the added -l and -L flags (what libraries to

include, and where to find them). Of course, you may need to modify the compilation command depending on any other dependencies you may have, but the basics are:

```
g++ -w -o path/to/output -Llib/Linux/Static -lNDXL -lm
```

We specify `-lm` as you will most likely be working with math functions found in the standard math library (`libm`) throughout your project. If you want to compile with the dynamic version of `NDXL` instead of static, you simply need to change the `-L` flag to point to the dynamic library directory.

2.2 Including the library on Windows

On Windows, you will most likely be using Visual Studio, and going forward that is what we will assume. Similar to the `gcc` layout, we assume your folder structure looks something like this:

```
My Solution
├─My Project
│  ├─src
│  │  └─main.cpp
│  ├─My Project.vcxproj
│  ├─My Project.vcxproj.filters
│  └─My Project.vcxproj.user
└─My Solution.sln
```

Then as with `gcc`, we recommend copying the `NDXL` folders DIRECTLY into your project directory:

```
My Solution
├─My Project
│  ├─src
│  │  └─main.cpp
│  ├─include
│  │  └─NDXL headers...
│  ├─lib
│  │  └─NDXL libraries...
│  ├─My Project.vcxproj
│  ├─My Project.vcxproj.filters
│  └─My Project.vcxproj.user
└─My Solution.sln
```

The process for linking the library is sadly a bit more involved, and warrants extra attention to your folder structure; we will assume your folder structure is exactly as laid out above, so adapt as necessary.

In Visual Studio, navigate to the solution explorer in the top right, and search for your project. Right-click it, then select "Properties". This will bring up the properties window for the project in the center of the screen (see figure 1).

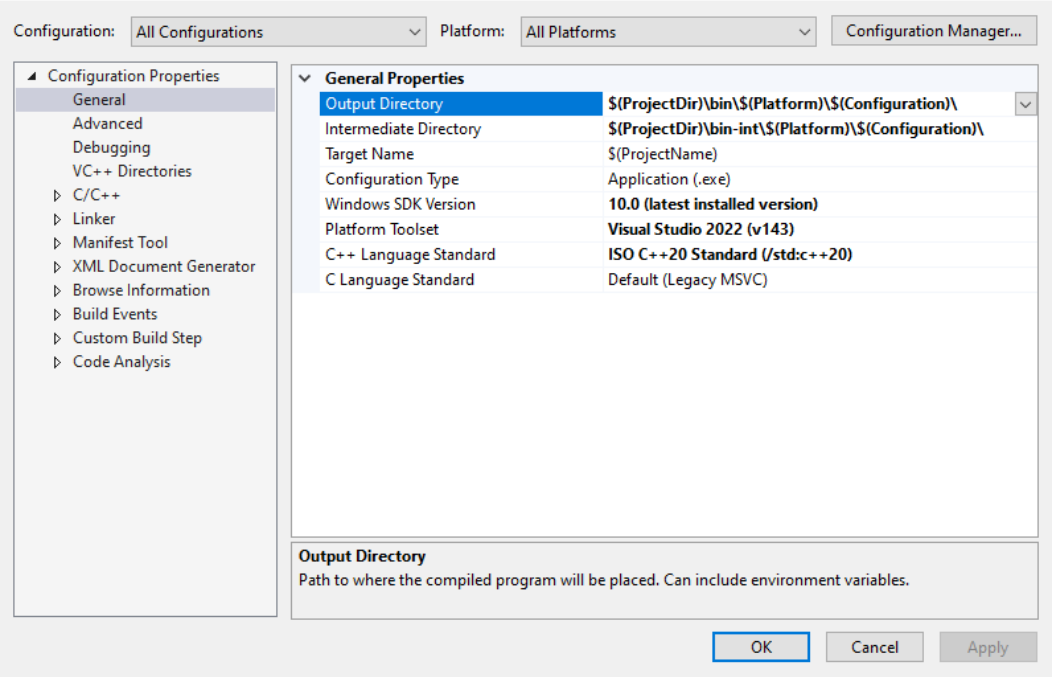

Fig. 1 Properties window default view when first opening. Note that if no source files are part of your project, some of the categories on the left may be missing, so it is a good idea to have at least a `main.cpp` in your project before opening this window.

Ensure the "Configuration" drop-down is set to "All Configurations", and the "Platform" drop-down is set to "All Platforms". We recommend changing the output and intermediate directories to something like a "bin" or "build" path, such as shown in figure 1. Use Visual Studio's macros to retrieve the path relative to your solution (`.sln`) or project (`.vcxproj`) files.

Next, navigate to the "Linker" tab, and go to "General". Roughly halfway down you should see a field titled "Additional Library Directories" (see figure 2), where you will need to specify the path to your library files. Assuming your folder structure is as described above, we can again exploit some Visual Studio macros to get the path to the correct library files regardless of whether

we are building with the static or dynamic version of `NDXL`. You will need to decide at this point, however, whether you want to use the static or dynamic version of `NDXL`. You could, in theory, set up more configurations and macros to automate the process of switching between builds, but that is beyond the scope of this documentation. For more information on how to set up libraries in Visual Studio, you can visit the official documentation for Visual Studio.

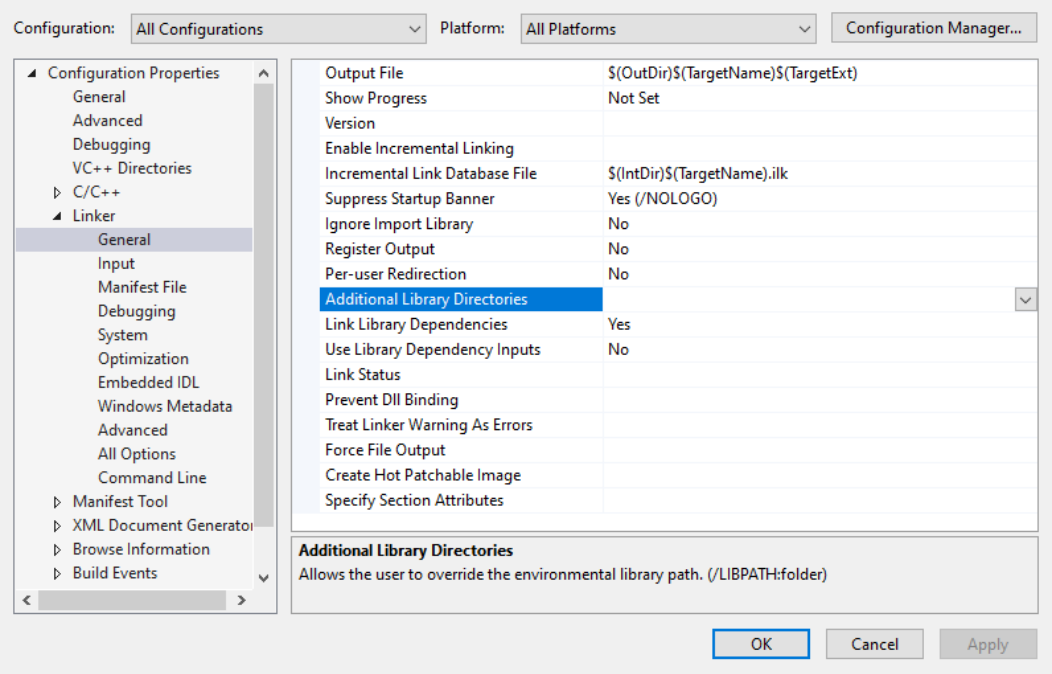

Fig. 2 "General" window for the "Linker" section, with the "Additional Library Directories" box highlighted.

In the "Additional Library Directories" field, paste the following if you are linking with the **static** version of `NDXL`...

```
$(ProjectDir)\lib\$(Platform)\Static\
```

...**or** this, if you are linking with the **dynamic** version of `NDXL`:

```
$(ProjectDir)\lib\$(Platform)\Dynamic\
```

We now need to tell Visual Studio what files to look for in those directories. Recall that both the static and dynamic builds of `NDXL` had a file called `NDXL.lib` inside them; it is this file we need to link in the linker. Just under "General", click on "Input" – in the new pane, at the very top, will be a field called "Additional Dependencies", and by default will have one of the following

values:

```
$(CoreLibraryDependencies);%(AdditionalDependencies)
```

```
<different options>
```

If the text is "<different options>", delete it, otherwise leave it as-is and append a semicolon to the end, then add:

```
NDXL.lib
```

to the field, as shown in figure 3.

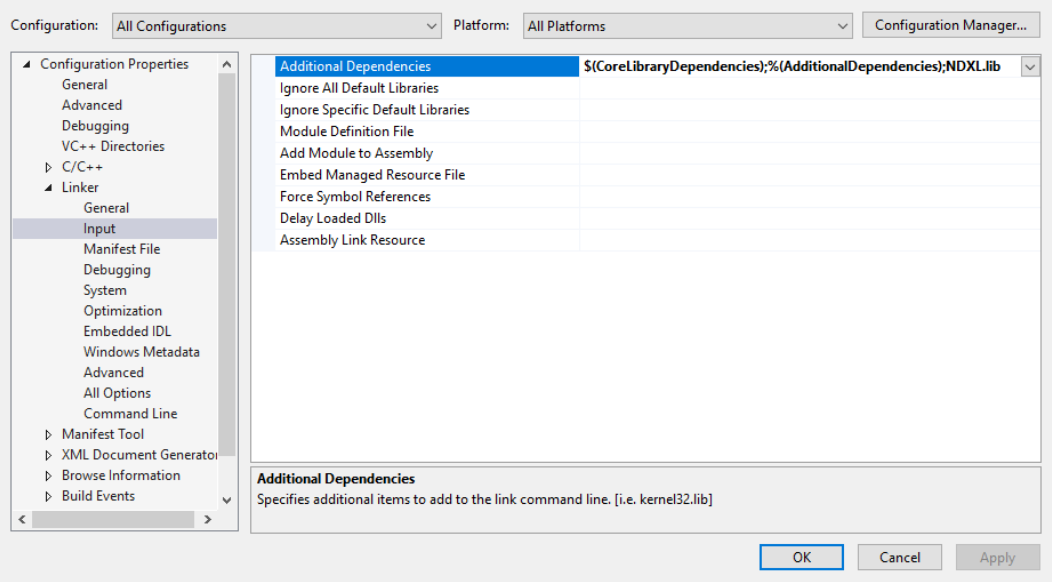

Fig. 3 Adding `NDXL.lib` as a dependency.

At this point, we have a couple more steps to follow that are specific to the static and dynamic versions of `NDXL`. For static linking, continue reading the next section, otherwise, skip to the dynamic linking section.

2.3 Linking statically on Windows

Do NOT follow this section if you are using the DYNAMIC version of `NDXL`!. Navigate to the "Preprocessor" tab under "C/C++". The first field listed should be "Preprocessor Definitions", where you must add `NDXL_STATIC` as shown in figure 4.

It is **CRUCIAL** that `NDXL_STATIC` ONLY be defined if linking against the **static**

version of NDXL. Once the preprocessor definition has been applied, you can set the "Configuration" to "All Configurations" again – the "Preprocessor Definitions" field will change to **<different options>**, indicating that NDXL_STATIC is not defined for all configurations, only for "Static" configuration.

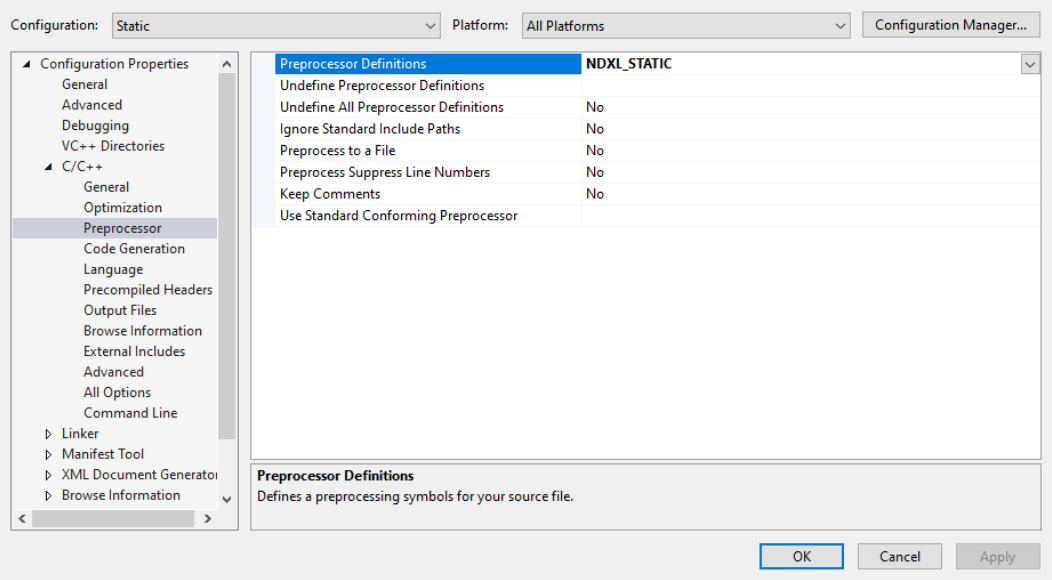

Fig. 4 Adding NDXL_STATIC as a preprocessor definition when linking against the static library.

2.4 Linking dynamically on Windows

Do NOT follow this section if you are using the STATIC version of NDXL!. Navigate to the "Preprocessor" tab under "C/C++" (see figure 4). The first field listed should be "Preprocessor Definitions", and must be EMPTY except if there are definitions there used by other libraries in your project.

Next, navigate to the "Build Events" tab (third from the bottom) in the properties menu, and click on "Post-Build Event". The first field should be called "Command Line", and if you click on the field, a button with a downwards facing arrow will appear on the right (see figure 5). Click the arrow, then click "<Edit...>", which will bring up a smaller sub-window as shown in figure 6.

Inside this window, under "Command Line", we want to tell Visual Studio to copy NDXL.dll from the "Dynamic" directory to the output directory, so that our executable can find it at runtime (this is standard practice for any dynamic library). We will do this using the xcopy command, as shown in figure 6, to

copy `NDXL.dll` to the output directory using the `$(TargetDir)` macro.

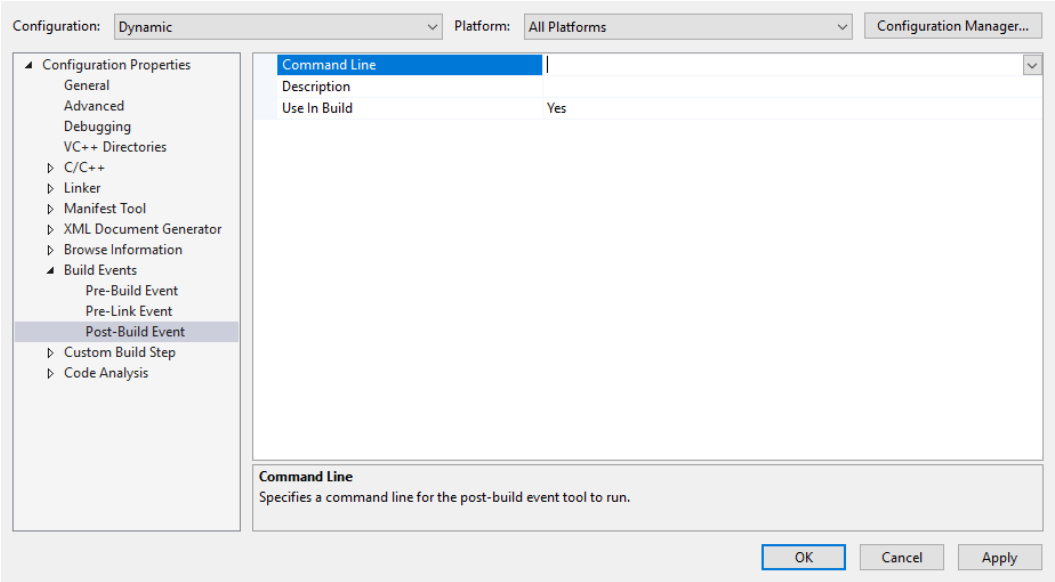

Fig. 5 Post-build events screen with the configuration set to "Dynamic".

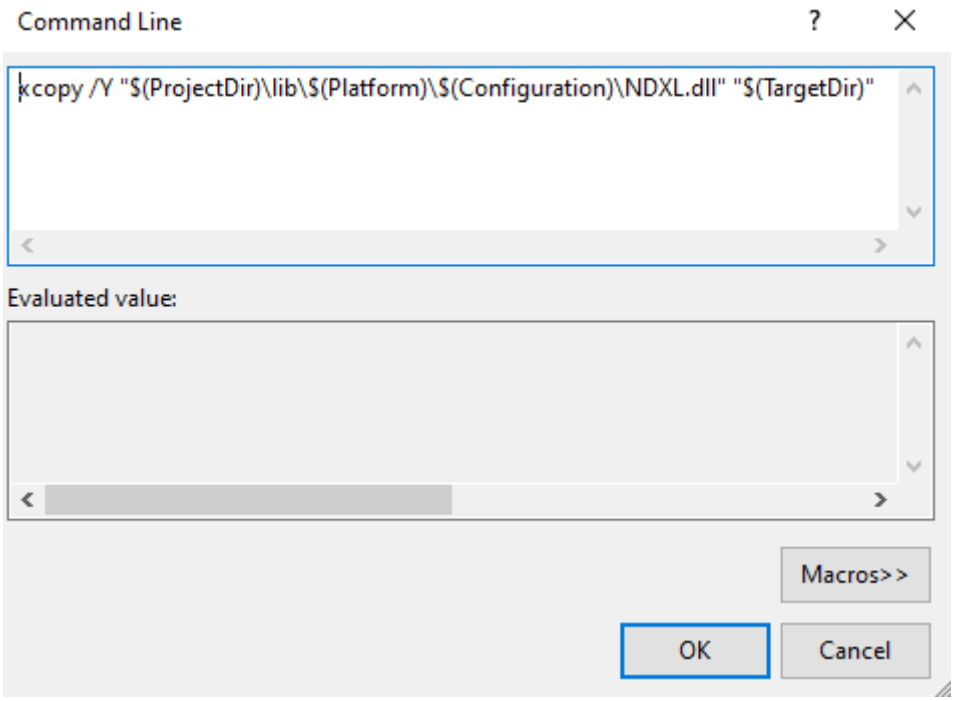

Fig. 6 "Dynamic" configuration command to copy `NDXL.dll` to the output directory.

As before, assuming your file structure is the same as illustrated above, you can simply copy this command into the "Command Line":

```
xcopy /Y "$(ProjectDir)\lib\$(Platform)\Dynamic\NDXL.dll"
"$(TargetDir)"
```

Otherwise, change the first path to wherever you put `NDXL.dll`. When you are done, click "OK", then click "Apply" in the bottom-right of the properties window. You have now linked `NDXL` either statically or dynamically, and are ready to start simulating networks!

Remember to set the "Configuration" back to "All Configurations" before clicking "OK" in the bottom-right of the properties window to exit it. Double-check the linker inputs, additional library directories, preprocessor definitions, build events, and output paths, lest you run into "file not found" errors later.

3 Using the library in code

Recall that the folder structure we are working off of is as follows:

```
My Solution
├─My Project
│  ├─src
│  │  └─main.cpp
│  ├─include
│  │  ├─ndxl
│  │  │  └─ndxl headers...
│  │  ├─unicpp
│  │  │  └─unicpp headers...
│  │  ├─ndxl.h
│  │  └─ndxlcore.h
│  ├─lib
│  │  └─NDXL libraries...
│  ├─My Project.vcxproj
│  ├─My Project.vcxproj.filters
│  └─My Project.vcxproj.user
└─My Solution.sln
```

The important thing for now is the path from `main.cpp` to `ndxl.h`, with the latter being the file you need to include in your project to access the `NDXL` functions and objects. From the diagram, we can see that the relative path to `ndxl.h`, starting at `main.cpp`, as a step-wise process, is:

- Go back one folder
- Go to `include`
- See `ndxl.h` inside `include`

Inside `main.cpp`, we need to include `ndxl.h` using the #include preprocessor macro with the relative path at the top of the file (before `int main()`):

```
#include "../include/ndxl.h"
```

There are other ways of including the library files, such as specifying "Additional Include Directories" in the property pages of the project, but using the relative path is easier, more verbose, and does not require additional command-line arguments when using compilers like `gcc`.

3.1 `ndxlInit()`

Before you proceed any further, now would be a good time to do a sanity check and ensure everything is set up and linked correctly. In `main.cpp`, we will set up a minimalist program:

```
#include "../include/ndxl.h"

int main(int argc, char** argv)
{
    ndxlInit();
    return 0;
}
```

But what is that function, `ndxlInit()`? Do we need it? When must we call it? What does it do?

Behind the scenes, `NDXL` keeps track of every network it creates, and each network keeps track of the vertices and edges it creates. When the program ends, `NDXL` automatically cleans up any memory it allocated that was not explicitly freed up by the user. Without this functionality, simulations could have massive memory leaks if not cleaned up properly. However, to instantiate this functionality correctly, `ndxlInit()` MUST be called BEFORE ANY OTHER `NDXL` CALLS. The main thing `ndxlInit()` does in the background, is register several cleanup functions using the `atexit()` function in `C` – without these, automatic garbage collection would never happen, leading to those memory leaks we just talked about.

At any point the user is free to manually destroy objects created with `NDXL`; `ndxlInit()` just ensures that anything not manually freed is still properly taken care of at the end. **Note that the automatic memory cleanup performed by `NDXL` ONLY affects objects created by `NDXL` i.e., any calls to functions like `malloc()` and `realloc()` must still be freed by the user.**

Once you have the program setup, compile and run it. If any linking errors occur, go back and double check all steps and compilation flags. If all runs well, you are finally ready to start setting up a network of your own. Feel free to jump straight into the code examples, as they are set up to cover the basic concepts, but we recommend reading the function and object descriptions first more information on what is available in `NDXL`.

3.2 `NDXL` objects

To take full advantage of the capabilities of `NDXL`, you will need to familiarise yourself with its three core objects (`structs`), which follow the naming scheme of `NDXL` + `StructName`.

- `NDXLNetwork`
- `NDXLVertex`
- `NDXLEdge`

All these objects are implemented as opaque types, which has two important consequences. Firstly, when creating import headers for `NDXL`, you need only specify that the objects exist, and do not need to worry about implementation details. Secondly, all functions working with these objects will take pointers to the object, instead of values or references.

To create an object, you need to use its constructor function along with any parameters it may require; the constructor follows the naming scheme `ndxlCreate` + `StructName`, so for example if you want to create a network, you would use:

```
NDXLNetwork* network = ndxlCreateNetwork();
```

This constructor does not take any parameters, as networks in `NDXL` are constructed by appending vertices and edges to an existing `NDXLNetwork*` object. To destroy an object, its destructor function may be called, which follows the naming scheme `ndxlDestroy` + `StructName`, and always takes in ONLY a pointer to that object:

```
NDXLNetwork* network = ndxlCreateNetwork();
ndxlDestroyNetwork(network);
network = 0;
```

Notice we reset the `network` variable to point to null after destroying it; this ensures we cannot accidentally reuse `network` after it has been destroyed, but is not strictly necessary. We will dive into more detail on each object next, along with its constructor and destructor, in the order you should create them.

3.2.1 `NDXLNetwork`

`NDXLNetwork` is the main object in `NDXL` that gets simulated at the end of the day. As we saw above, its constructor takes no parameters, as a "network" in `NDXL` is just a host object for creating vertices and edges, which will define the network's topology. A `NDXLNetwork` MUST therefore be created BEFORE any other `NDXL` objects, and you can have more than one network at a time.

`Constructor`

```
NDXLNetwork* ndxlCreateNetwork();
```

`Destructor`

```
void ndxlDestroyNetwork(NDXLNetwork* _Network);
```

3.2.2 `NDXLVertex`

`NDXLVertex` is the object representing a vertex or node in the network. It keeps track of its own simulation time relative to when it was first created, meaning you can have vertices in the network that are younger or older than others; we will show later how to query the current time of a particular `NDXLVertex`.

Each `NDXLVertex` also has an initial state vector V, which we shall denote V_i for the state of the i^{th} vertex. This state is allowed to evolve according to some dynamical evolution function, specified as an ordinary differential equation (ODE); mathematically, we write this as:

$$\frac{dV_i}{dt} = f(V_i,\ V_{j\to i},\ E_{j\to i},\ t,\ p\ldots) \tag{1}$$

where $V_{j\to i}$ and $E_{j\to i}$ represent respectively the vertices and edges explicitly connected to i, t is time, and $p\ldots$ is an extra parameter list. Because of the way `NDXL` handles information transfer, implementing eq. (1) in code is a lot simpler than it may look, and the dependence of $\frac{dV_i}{dt}$ on arguments like $E_{j\to i}$ is given explicitly by the user in the evolution function. This function must have the signature:

```
void vertexFunction(
    NDXLVertex* vertex,
    double* outState,
    double* inState,
    double time,
    const void* params
);
```

Here, `outState` and `inState` are arrays corresponding to the final and initial state vectors of `vertex`, respectively; that is to say, $\frac{dV}{dt}$ = `outState`, and V = `inState`. The initial state, and how many elements are within, are specified when creating the vertex. Finally, the extra parameter list p ... is represented in code by `params`.

Constructor

```
NDXLVertex* ndxlCreateVertex(
    NDXLNetwork* hostNetwork,
    const long stateCount,
    const double* initialState,
    const void* params,
    NDXLVertexFunction vertexFunction,
    NDXLVertexNoise noiseFunction,
    NDXLVertexCallback callbackFunction
);
```

Destructor

```
void ndxlDestroyVertex(NDXLVertex* _Vertex);
```

Notice the first parameter in the constructor is the hose `NDXLNetwork`, which is why it is necessary to create a `NDXLNetwork` object BEFORE creating a `NDXLVertex`. `stateCount` tells `NDXL` how many elements are in the state of the vertex, and `params` is the optional extra parameter list p ... .

We already saw how to specify the `vertexFunction` above, and the definition `NDXLVertexFunction` is just a `typedef` to simplify function signatures. Similarly, `NDXLNoiseFunction` and `NDXLVertexCallback` are functions whose signatures are given respectively by:

```
void vertexNoise(
    NDXLVertex* vertex,
    double* outNoise,
    double time,
    const void* params
);

void vertexCallback(
    NDXLVertex* vertex,
    double* state,
    double time,
    const void* params
);
```

In the case of the noise function, `outNoise` is an array of length `stateCount` i.e., the noise at each index gets added to the state at each corresponding index AFTER solving the differential equation for that time step. The vertex callback passes the state directly i.e., one modifies the state of the vertex in-place by modifying `state`. Note that the vertex, noise, and callback functions all pass a `NDXLVertex` as their first parameter, which corresponds to the vertex that was created with those functions – it is effectively a `C`-style way of mimicking the `this` keyword in `C++`, or `self` in `Python`.

3.2.3 `NDXLEdge`

`NDXLEdge` is the object representing a connection (edge) between two vertices in a network, and in general has two main properties, namely, a state and a delay. The state of an edge refers to all (potentially time-dependent) parameters that define the physical and mathematical properties of the edge; if one were modeling cables in an electrical power grid, the state of each edge would contain information about its conductivity, temperature, radius, length, capacitance, and so forth.

Mathematically, it is often convenient to summarise the properties of an edge by a property called the edge weight, often denoted W or K; the edge weight is a single value that represents a scaled version of all the physical properties considered in the edge. That is to say, an edge weight of 2 is meaningless except in the context of another edge with a weight of 1; in this case, one would surmise that the vertices joined by the first edge are twice as strongly

connected – whatever that may mean physically – than the vertices joined by the second edge.

In `NDXL`, an edge is assigned a state in the exact same way as a vertex; you could therefore choose to model all relevant properties separately within the state vector of an edge, or you could create a state vector of length 1 whose value is the edge weight. The ODE for an edge is as follows:

$$\frac{dE_{ij}}{dt} = g(E_{ij},\ V_i,\ V_j,\ t,\ q\ldots) \tag{2}$$

Similarly to eq. (1), the state of the edge at the next time step can depend on the state of the edge at the previous time step, as well as the states of the vertices it connects, and $q\ldots$ is the extra parameter list for the edge. In code, this is again far simpler to implement than it may initially appear:

```
void edgeFunction(
    NDXLEdge* edge,
    double* outState,
    double* inState,
    double time,
    const void* params
);
```

Note that the vertices are not explicitly provided as arguments to the function, since `NDXL` provides functions to query the `edge` for the vertices it connects.

Constructor

```
NDXLVertex* ndxlCreateEdge(
    NDXLVertex* vertex1,
    NDXLVertex* vertex2,
    const double* initialState
    const long stateCount,
    const double delay,
    const void* params,
    const unsigned char directed,
    NDXLEdgeFunction edgeFunction,
    NDXLEdgeNoise noiseFunction,
    NDXLEdgeCallback callbackFunction
);
```

Destructor

```
void ndxlDestroyEdge(NDXLEdge* _Edge);
```

In contrast to constructing a `NDXLVertex`, we do not specify a parent network when creating a `NDXLEdge`; instead, the host network is deduced by querying the parents of `vertex1` and `vertex2` – if they are not the same, the function fails to create an edge, and an error is raised internally. `directed` is a Boolean value indicating whether the edge is directed or not; directed edges will only allow information to flow from `vertex1` to `vertex2`, but not vice versa.

The noise and callback functions for an edge have the same signatures as for a vertex, except that the first parameter is a `NDXLEdge*` instead of a `NDXLVertex*`:

```
void edgeNoise(
    NDXLEdge* edge,
    double* outNoise,
    double time,
    const void* params
);

void edgeCallback(
    NDXLEdge* edge,
    double* state,
    double time,
    const void* params
);
```

4 Callbacks

Callbacks are user-defined functions that are used by objects or other functions to implement externally defined behaviour. An easy way to conceptualise callbacks is to think about an application with several buttons on the interface. Each button neads to call a different function when clicked, for example, the "Save" button should only "save" the user's work, and the "Load" button should only "load" a different workspace – the functions of either button are not linked.

However, each button is an object, which was created from the same "button" template. They will thus have the same functions, including the one that fires when the button has been clicked – an "on clicked" event, so to say. If each button fires the same "on clicked" function when it is clicked, how can the function do different things for each button?

The answer is callbacks; the user writes a separate function for each button that they want to trigger when the button is clicked, then they pass that function as a callback to the button's interface. When the button's "on clicked" function is called, it calls the callback function, which is different for each button. If the user wants nothing to happen when the button is clicked, they simply pass no callback ("null" or "nothing" or 0) as the callback for the button's "on clicked" event.

In sum, callbacks are a way for the user to specify how a certain function must behave for a specific instance of an object, even if all objects have the same function. All three objects in `NDXL` have callback functions associated with them, which serve as a way to directly operate on the objects outside of their evolution functions. During simulation, the callbacks for each object are evaluated **before anything else**.

4.1 Call order

`NDXL` is a simulation package, at the end of the day, and the simulation happens in discrete steps. In our paper, we stated that `NDXL` uses static RK4 to solve the vertex and edge dynamics; what this means in Layman's terms is that the user can specify a step size h which corresponds to the "jump" on the time (x) axis before the algorithm is called again, and in general a smaller step size translates to higher accuracy at the cost of speed (though this is a

gross oversimplification of the algorithm).

Each step, several things must happen, but they cannot occur simultaneously (this is just a basic fact of computers). We must therefore introduce a **call order**, which defines when each thing happens in relation to everything else. Below we show the signatures and usage of each callback in `NDXL` (example usage shown in section **??**), along with where in the call order they reside

4.2 Network callback

```
typedef void(*NDXLNetworkCallback)(NDXLNetwork*, double, const
    void*);

void network_callback(
    NDXLNetwork* network,
    double time,
    const void* params
);
```

Call order in simulation: called first before any other callbacks or functions.

Parameters: the parameters are passed as the `NDXLNetwork*` the callback is assigned to, the current simulation time of that network, and the list of extra parameters that was passed to that network during instantiation (see section 6.4.1).

4.3 Vertex callback

```
typedef void(*NDXLVertexCallback)(NDXLVertex*, double*, double,
    const void*);

void vertex_callback(
    NDXLVertex* vertex,
    double* state,
    double time,
    const void* params
);
```

Call order in simulation: each vertex's callback is called before its own evolution function, but both the callback and evolution function are processed before the next vertex is at all processed. All vertices are

processed **after** their host network's callback, but **before** any edges are processed.

Parameters: the parameters are passed as the `NDXLVertex*` the callback is assigned to, the state vector of that vertex (modified in-place), the current simulation time of that vertex, and the list of extra parameters that was passed to that vertex during instantiation (see section 6.5.1).

4.4 Edge callback

```
typedef void(*NDXLEdgeCallback)(NDXLEdge*, double*, double, const
    void*);

void edge_callback(
    NDXLEdge* edge,
    double* state,
    double time,
    const void* params
);
```

Call order in simulation: each edge's callback is called before its own evolution function, but both the callback and evolution function are processed before the next edge is at all processed. All edges are processed **after** their host network's vertices have been processed.

Parameters: the parameters are passed as the `NDXLEdge*` the callback is assigned to, the state vector of that edge (modified in-place), the current simulation time of that edge, and the list of extra parameters that was passed to that edge during instantiation (see section 6.6.1).

5 Error handling

Errors in C are not raised or handled in the same way as many higher level languages; typically, any function that could potentially raise an error returns a specific value – known as a **sentinel value** – if an error occurred; failure to check for this error and react appropriately may lead to a crash later on.

NDXL raises errors internally, the details of which can be queried by the user. Under the hood, NDXL keeps track of the most recently raised error code and message, which can be accessed using the following functions:

```
const char* ndxlGetLastErrorMsg();  // Get most recent error message
const int ndxlGetLastErrorCode();   // Get most recent error code
void ndxlPrintError();              /* Print the most recent error
                                    code and message to the console */
```

All NDXL functions are programmed to return sentinel values in the case of an error, but sometimes the sentinel value is not necessarily enough to deduce if an error has occurred. Consider, for example, trying to get the delay of an edge that does not exist:

```
NDXLEdge* edge;
const double delay = ndxlGetEdgeDelay(0);
```

In this case, we are trying to query the delay of a "null" edge (a pointer that points to nothing); if we just dereferenced the pointer to return the edge delay, the program would crash, since dereferencing a null pointer is an invalid operation. Instead, NDXL will just return zero. However, a delay value of zero is **not an invalid value** – it just implies instant information transfer. Clearly, simply checking if `delay` is zero is not enough, since zero could be returned because the edge was invalid, or because it had a delay of zero. We must therefore further interrogate the most recent error code, to check which scenario occurred:

```
NDXLEdge* edge;
const double delay = ndxlGetEdgeDelay(0);
if (ndxlGetLastErrorCode() == NDXL_ARG_ERR)
    printf("Null edge");
```

Functions that return a pointer will return NULL (0) if pointer creation failed, and in those cases it is sufficient to simply check if the pointer is null or not (failed or succeeded to create, respectively):

```
NDXLEdge* edge = ndxlCreateEdge(blah blah...);
if (!edge)
    printf("Edge creation failed");
```

Lastly, functions that alter properties or perform tasks on a specific object will always return one of two error codes: `NDXL_OK` or `NDXL_ERR`. The former indicates that the function succeeded, while the latter indicates that some error was raised internally, and the user should use one of the error functions to check the details:

```
if (ndxlSetEdgeDelay(edge, -4) == NDXL_ERR)
    printf("Failed to set edge delay");

// ----- OR ----- \\

if (ndxlSetEdgeDelay(edge, -4) == NDXL_ERR)
    ndxlPrintError();
```

Using `ndxlPrintError()` gives the most verbose report on the error, as it tells you exactly what error was raised, by what function, and due to which parameter. For example, the above would print the following to the console:

```
EDGE (SET DELAY): nonpositive delay.
```

5.1 Error codes

We have already seen some error codes in use, but to summarise, these are the error codes and their interpretations used within `NDXL`:

- **`NDXL_OK`:** no errors occurred
- **`NDXL_ERR`:** an error occurred; use `ndxlGetLastErrorCode()` to see what error occurred, use `ndxlGetLastErrorMsg()` to see the error details, and use `ndxlPrintError()` for verbose output
- **`NDXL_ARG_ERR`:** invalid argument e.g., passing a null pointer
- **`NDXL_VALUE_ERR`:** invalid value e.g., setting edge delay to -4

The error codes and messages returned by each function are detailed in section 6 (Functions).

6 Functions

This section covers all functions in NDXL, as well as general rules, naming schemes, parameter expectations, return value interpretations, and so forth.

6.1 General rules

All functions in NDXL– except the error handling functions (see section 5) – operate on at least one of the three NDXL objects (see section 3.2). Additionally, **ANY** function may raise an error internally, **even if the function does not return an error code**, so it is a good idea to periodically use something like `ndxlPrintError()` during development to ensure your code is set up correctly.

All three NDXL objects are handled as "pointer to" types:

`NDXLNetwork*`

`NDXLVertex*`

`NDXLEdge*`

Since pointers are just memory addresses, one may pass 0 in place of a pointer to represent "nothing", for example:

```
ndxlGetVertexOrder(0);
```

While not technically invalid from a language perspective, it makes no logical sense to try and operate on "nothing". In this case, your program will not crash, but NDXL will internally raise an **argument error** (NDXL_ARG_ERR). Just like in regular C, if your argument is not 0, the pointer **must be a valid pointer to the specified type** – C has no mechanism for type-checking or address validation, so it is your responsibility to ensure the pointer is not "invalid".

In general, whenever a function expects a pointer of any kind (object, function, state vector, etc), you may pass 0, and it is not invalid in all cases to do so; for example, passing 0 as the evolution function for an edge just means the state of the edge won't change.

For allocating memory blocks of a given size, NDXL uses `size_t` as the type for the size variable. `size_t` is a `typedef` in C/C++ that corresponds to the type used by the underlying architecture to allocate memory blocks; a good

rule of thumb is that on 64-bit systems, `size_t` is a signed 64-bit integer, and on 32-bit systems it is a signed 32-bit integer, **but this is not always guaranteed to be the case**. It is therefore safer to use `size_t` over the type you expect it to represent.

To indicate the direction of data flow, we will mark each parameter with `in`, `out`, or `inout` to show data going into, out of, or both ways in a function, respectively. All functions are represented as standard `C`-style signatures, with the relevant parameter annotations and `typedef` s.

Before using anything inside `NDXL`, you **MUST** remember to call `ndxlInit()` first, which registers the automatic memory cleanup functions to trigger at the end of the program's execution – without this function, you are susceptible to memory leaks!!!

6.2 Naming schemes

In section 3.2, we saw that all objects in `NDXL` are prefixed with "`NDXL` ", that is to say, they follow the naming scheme:

```
NDXLObject
```

Similarly, all functions in `NDXL` are prefixed with "`ndxl` ", so they follow the naming scheme:

```
ndxlFunctionName
```

Functions in `NDXL` are classified into six distinct types, which we will dive into more detail with next:

- **Constructors**
- **Destructors**
- **Getters**
- **Setters**
- **Operational functions**
- **Error handlers**

6.2.1 Constructors

Constructors are functions that **construct** (create/instantiate) an object, and thus typically have a parameter list resembling the initial values the object must take on. In NDXL, all constructors follow the naming scheme:

```
ndxlCreateObject
```

Where `Object` is the name of any of the three objects from section 3.2 – the parameter lists will vary depending on the object being created. Constructors always return a "pointer to type", for example:

```
NDXLNetwork* ndxlCreateNetwork
```

This pointer is tracked by NDXL in the background, so assuming you remembered to call `ndxlInit()` first, it will be automatically cleaned up at the end of the program. For this reason, **objects created in `NDXL` must NOT be destroyed with `free()`**, as this will interfere with the automatic memory cleanup functions. If you want/need to destroy an object, either for simulation purposes or just to free up memory, you must use its associated **destructor**.

6.2.2 Destructors

Destructors are functions that **destroy** (delete/free) an object, releasing any resources (memory allocations, handles, members, etc) that were associated with that object. In NDXL, they follow the naming scheme:

```
ndxlDestroyObject
```

The destructor **always** takes only one parameter, namely, a pointer to the object you want to destroy, for example:

```
void ndxlDestroyNetwork(NDXLNetwork* network)
```

6.2.3 Getters

Getters are functions that **get** (retrieve/access) the value of a specific property of an object, and typically employ "business logic" to ensure that the object and the property being requested are both valid; this includes everything from checking that the property exists, to having a value, to being allowed to be shown to the user requesting that property. Getters in NDXL follow the naming scheme:

`ndxlGetObjectProperty`

The first parameter in a getter is the object of interest i.e., "which `object` are you trying to retrieve the `property` from?", and the return type depends on the type of the property being accessed:

```
const size_t ndxlGetVertexOrder
```

```
const double* ndxlGetEdgeState
```

6.2.4 Setters

Setters are functions that **set** (modify/update) the value of a specific property of an object. Like getters, they typically employ "business logic" to ensure that the object and the property being modified are both valid, **and, perhaps most importantly, that the new value being assigned to the property is valid for that property**. Setters in `NDXL` follow the naming scheme:

`ndxlSetObjectProperty`

The first parameter in a setter is also the object of interest i.e., "which `object` are you trying to set the `property` of?", and the return type in `NDXL` is **always** in `int` representing whether the function failed or succeeded in updating the property (see section 5.1 for details):

```
const int ndxlSetVertexState
```

```
const int ndxlSetEdgeDelay
```

If the returned value is `NDXL_OK`, then the setter worked and the property of the object has been assigned the new specified value. If the returned value is `NDXL_ERR`, then something went wrong in the setting process – either one of the arguments was invalid, or had an invalid value – and one of the error checking functions (see section 5) should be used to inspect the details.

6.2.5 Operational functions

Operational functions perform **operations** (tasks/actions) on objects. They are the "do something functions" i.e., "now that we have an object, we want it to actually do something". Operational functions do not explicitly get/set object properties, though they will interface with them in order to get the task done. In `NDXL`, they follow the naming scheme:

```
ndxlObjectFunctionName
```

Like getters and setters, the first parameter to an operational function is a pointer to the object that the function must act on. Also similar to setters, operational functions **always** return either `NDXL_OK` or `NDXL_ERR` depending on whether the specific operation executed successfully, or encountered an error along the way.

6.2.6 Error handlers

Error handlers interface with the internal error state stored by `NDXL`. Any function may raise an error if its arguments are invalid, but unlike other programming languages, `C` has no (straightforward) mechanism for "raising", "throwing", or "catching" errors; instead, specific codes (see section 5.1) are used to signal something going wrong, and it is the programmer's job to check for these codes where applicable.

As we have seen in section 5, the functions `ndxlGetLastErrorCode()` and `ndxlGetLastErrorMsg()` can be used to retrieve the most recent error code and error message, respectively. A more verbose output is given by using `ndxlPrintError()` instead, which will print the most recent error code and message to the standard output stream in the following format:

```
OBJECT (OPERATION PROPERTY): Details.
```

Here, "`OPERATION` " will be any of "GET", "SET", "CREATE", "DESTROY", or the name of an operational function, which tells you the kind of function called on the specified `object` that caused the error. For example, if the output is:

```
VERTEX (GET ORDER): null vertex.
```

Then we can deduce that the error came from trying to get the order of a vertex, which would correspond to calling the function:

```
ndxlGetVertexOrder
```

This function, as we will see later, only takes in one parameter, namely, a `NDXLVertex*` (the vertex we want to get the order of). The "null vertex" detail tells us that we passed 0 as the `NDXLVertex*` into the function, which is invalid because one cannot retrieve the order of "nothing" (recall that a 0 pointer indicates "nothing").

6.3 `NDXL` general functions

This section details all functions pertaining to general usage of `NDXL` (setup, error handling, etc).

6.3.1 `ndxlInit`

```
void ndxlInit();
```

Description:

Initialises the library if not already initialised, and registers automatic cleanup functions to execute when the program ends to ensure no memory leaks. This function **MUST** be called before using any other `NDXL` functions.

Parameters:

None.

Returns:

None.

Errors raised:

None.

6.3.2 `ndxlGetLastErrorCode`

```
const int ndxlGetLastErrorCode();
```

Description:

Retrieves the most recently-raised error code (see section 5.1 for interpretation).

Parameters:

None.

Returns:

The most recently-raised error code as an integer.

Errors raised:

None.

6.3.3 `ndxlGetLastErrorMsg`

```
const char* const ndxlGetLastErrorMsg();
```

Description:

Retrieves the most recently-raised error message.

Parameters:

None.

Returns:

The most recently-raised error message as a C-string.

Errors raised:

None.

6.3.4 `ndxlPrintError`

```
void ndxlPrintError();
```

Description:

Prints the most recently-raised error code and message to the console in a verbose format.

Parameters:

None.

Returns:

None, but prints to the console using standard out.

Errors raised:

None.

6.4 `NDXLNetwork` functions

This section details all functions pertaining to `NDXLNetwork`.

6.4.1 `ndxlCreateNetwork`

```
NDXLNetwork* ndxlCreateNetwork(
    in const void* params
);
```

Description:

Creates a new network and registers it with the automatic memory cleanup functions.

Parameters:

params: pointer to arbitrary extra data used by the network. Note that `NDXL` does **NOT** make a copy of this data, so it is up to the user to ensure the data pointed to by `params` remains in-scope for the duration of the network's lifetime. If no extra data are used, this parameter may be 0.

Returns:

A new `NDXLNetwork` pointer.

Errors raised:

None.

6.4.2 `ndxlDestroyNetwork`

```
void ndxlDestroyNetwork(
    in NDXLNetwork* network
);
```

Description:

Destroys all `NDXLVertex` and `NDXLEdge` pointers that were created on `network`, then destroys the network itself.

Parameters:

network: pointer to a `NDXLNetwork` that was created using `ndxlCreateNetwork`. If `network` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

None.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `network` is 0.

6.4.3 `ndxlGetNetworkEdgeCount`

```
const size_t ndxlGetNetworkEdgeCount(
    in NDXLNetwork* network
);
```

Description:

Retrieves the number of edges in `network`.

Parameters:

network: pointer to a `NDXLNetwork` that was created using `ndxlCreateNetwork`. If `network` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

The number of edges in `network`, or 0 if `network` is 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `network` is 0.

6.4.4 `ndxlGetNetworkEdges`

```
void ndxlGetNetworkEdges(
    in NDXLNetwork* network,
    out NDXLEdge** edges
);
```

Description:

Populates `edges` with all `NDXLEdge` pointers created on `network`.

Parameters:

network: pointer to a `NDXLNetwork` that was created using `ndxlCreateNetwork`. If `network` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

edges: array of `NDXLEdge` pointers. It is assumed that `edges` can contain a number of elements at least equal to the value that would be returned by `ndxlGetNetworkEdgeCount`. If `edges` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

None; output data are written in-place to `edges`.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `network` is 0.

Argument error (`NDXL_ARG_ERR`) if `edges` is 0.

6.4.5 `ndxlGetNetworkVertexCount`

```
const size_t ndxlGetNetworkVertexCount(
    in NDXLNetwork* network
);
```

Description:

Retrieves the number of vertices in `network`.

Parameters:

network: pointer to a `NDXLNetwork` that was created using `ndxlCreateNetwork`. If `network` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

The number of vertices in `network`, or 0 if `network` is 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `network` is 0.

6.4.6 `ndxlGetNetworkVertices`

```
void ndxlGetNetworkEdges(
    in NDXLNetwork* network,
    out NDXLVertex** vertices
);
```

Description:

Populates `vertices` with all `NDXLVertex` pointers associated with `network`.

Parameters:

network: pointer to a `NDXLNetwork` that was created using `ndxlCreateNetwork`. If `network` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

vertices: array of `NDXLVertex` pointers. It is assumed that `vertices` can contain a number of elements at least equal to the value that would be returned by `ndxlGetNetworkVertexCount`. If `vertices` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

None; output data are written in-place to `vertices`.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `network` is 0.

Argument error (`NDXL_ARG_ERR`) if `vertices` is 0.

6.4.7 `ndxlNetworkSimulate`

```
void ndxlNetworkSimulate(
    in NDXLNetwork* network,
```

```
    in const double time,
    in const double step,
    in NDXLNetworkCallback callback
);
```

Description:

Simulates `network` for a total duration of `time`, with timestep size `step`.

Parameters:

network: pointer to a `NDXLNetwork` that was created using `ndxl-CreateNetwork`. If `network` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

time: how long to simulate `network` for. Note that a `NDXLNetwork` always starts at 0 simulation time, and repeated calls to this function simulate the network based on its current state, NOT its original state.

step: resolution on which to simulate; the total number of steps is `time` divided by `step`. Generally, accuracy and speed are inversely proportional, and smaller step sizes tend to favour accuracy, but this is not always the case.

callback: a `NDXLNetworkCallback` (see section 4.2). If no callback is desired, this parameter may be 0.

Returns:

None.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `network` is 0.

Value error (`NDXL_VALUE_ERR`) if `time` is less than or equal to 0.

Value error (`NDXL_VALUE_ERR`) if `step` is less than or equal to 0.

6.5 `NDXLVertex` functions

This section details all functions pertaining to `NDXLVertex`.

6.5.1 `ndxlCreateVertex`

```
NDXLVertex* ndxlCreateVertex(
    in NDXLNetwork* network,
    in const double* state,
    in const unsigned char order,
    in const void* params,
    in NDXLVertexFunction func,
    in NDXLVertexNoise noise,
    in NDXLVertexCallback callback
);
```

Description:

Adds a new vertex in `network` and returns the newly created pointer to `NDXLVertex`.

Parameters:

network: the host `NDXLNetwork` that this vertex must be a part of. This parameter **may not be** 0 i.e., you must always have a `NDXLNetwork` to attach vertices and edges to.

state: the initial state vector of this vertex, of length `order`. This parameter **may not be** 0, and its size cannot be altered once the vertex has been created.

order: the number of elements in `state`. Must be greater than 0 and less than 256.

params: pointer to arbitrary extra data used by the vertex. Note that `NDXL` does **NOT** make a copy of this data, so it is up to the user to ensure the data pointed to by `params` remains in-scope for the duration of the vertex's lifetime. If no extra data are used, this parameter may be 0.

func: the evolution function of this vertex (see section 3.2.2 for signature). This parameter may be 0, in which case the state of the vertex will never change (static vertex).

noise: the noise function of this vertex (see section 3.2.2 for signature). This parameter may be 0, in which case there is no noise in the signal of the vertex (quiet vertex).

callback: the callback function of this vertex (see section 4.3 for signature). This parameter may be 0, in which case there is no callback associated with the vertex (no external perturbations).

Returns:

If no errors are raised, returns a new `NDXLVertex` pointer, otherwise returns 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `network` is 0.

Argument error (`NDXL_ARG_ERR`) if `state` is 0.

Value error (`NDXL_VALUE_ERR`) if `order` is less than or equal to 0.

6.5.2 `ndxlCopyVertex`

```
NDXLVertex* ndxlCopyVertex(
    in NDXLVertex* vertex,
    in NDXLNetwork* network,
    in const char copy_state0
);
```

Description:

Creates a new `NDXLVertex` with `network` as the host network, by copying the state and parameters of the `NDXLVertex` pointed to by `vertex`.

Parameters:

vertex: the vertex whose settings must be copied to create the new vertex. This parameter **may not be** 0.

network: the host `NDXLNetwork` that the new vertex must be a part of. This parameter **may not be** 0.

copy_state0: Boolean value. If `true`, the vertex is created by copying the **initial** state of `vertex` as its own initial state; if `false`, the vertex

is created by copying the **current** state of `vertex` as its own initial state.

Returns:

If no errors are raised, returns a new `NDXLVertex` pointer, otherwise returns 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

Argument error (`NDXL_ARG_ERR`) if `network` is 0.

6.5.3 `ndxlDestroyVertex`

```
void ndxlDestroyVertex(
    in NDXLVertex* vertex
);
```

Description:

Destroys the `NDXLVertex` pointed to by `vertex`, freeing all resources used by it, and removing it from its host network.

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

None.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

6.5.4 `ndxlGetVertexConnectivity`

```
const size_t ndxlGetVertexConnectivity(
    in NDXLVertex* vertex
```

```
);
```

Description:

Retrieves the connectivity of `vertex`, defined as **one plus the number of neighbours `vertex` has**.

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

One plus the number of vertices `vertex` is explicitly connected to, or 0 if `vertex` is 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

6.5.5 `ndxlGetVertexDegree`

```
const size_t ndxlGetVertexDegree(
    in NDXLVertex* vertex
);
```

Description:

Retrieves the degree of `vertex`, defined as **the number of neighbours `vertex` has**.

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

The number of vertices `vertex` is explicitly connected to, or 0 if `vertex` is 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

6.5.6 `ndxlGetVertexEdges`

```
void ndxlGetVertexEdges(
    in NDXLVertex* vertex,
    out NDXLEdge** edges,
);
```

Description:

Populates `edges` with all the edges connected to `vertex`.

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

edges: array of `NDXLEdge` pointers. It is assumed that `edges` can contain a number of elements at least equal to the value that would be returned by `ndxlGetVertexDegree`. If `edges` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

None; output data are written in-place to `edges`.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

Argument error (`NDXL_ARG_ERR`) if `edges` is 0.

6.5.7 `ndxlGetVertexFirstEdge`

```
NDXLEdge* ndxlGetVertexFirstEdge(
    in NDXLVertex* vertex
);
```

Description:

Retrieves the first `NDXLEdge` connected to `vertex`. This function is typically used in conjunction with `ndxlGetVertexNextEdge` and `ndxlGetVertexLastEdge` to iterate over all edges connected to `vertex` (see section **??**).

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

The first `NDXLEdge` pointer connected to `vertex`, or 0 if `vertex` is 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

6.5.8 `ndxlGetVertexLastEdge`

```
NDXLEdge* ndxlGetVertexLastEdge(
    in NDXLVertex* vertex
);
```

Description:

Retrieves the last `NDXLEdge` connected to `vertex`. This function is typically used in conjunction with `ndxlGetVertexFirstEdge` and `ndxlGetVertexNextEdge` to iterate over all edges connected to `vertex` (see section **??**).

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

The last `NDXLEdge` pointer connected to `vertex`, or 0 if `vertex` is 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

6.5.9 `ndxlGetVertexNeighbour`

```
NDXLVertex* ndxlGetVertexNeighbour(
    in NDXLVertex* vertex,
    in NDXLEdge* edge
);
```

Description:

Retrieves the neighbouring vertex to `vertex` along `edge`.

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

edge: pointer to a `NDXLEdge` that was created using `ndxlCreateEdge`. If `edge` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

If `edge` is not 0, and is connected to `vertex` (which must also not be 0), returns the other `NDXLVertex` joined by `edge`, otherwise returns 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

Argument error (`NDXL_ARG_ERR`) if `edge` is 0.

6.5.10 `ndxlGetVertexNeighbours`

```
void ndxlGetVertexNeighbours(
    in NDXLVertex* vertex,
    out NDXLVertex** neighbours
);
```

Description:

Populates `neighbours` with all vertices that are explicitly connected to `vertex`.

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

neighbours: array of `NDXLVertex` pointers. It is assumed that `neighbours` can contain a number of elements at least equal to the value that would be returned by `ndxlGetVertexDegree`. If `neighbours` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

None; output data are written in-place to `neighbours`.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

Argument error (`NDXL_ARG_ERR`) if `neighbours` is 0.

6.5.11 `ndxlGetVertexNeighbourState`

```
const double* ndxlGetVertexNeighbourState(
    in NDXLVertex* vertex,
    in NDXLEdge* edge
);
```

Description:

Retrieves the delayed state vector of the vertex that directly neighbours `vertex` along `edge`.

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

edge: pointer to a `NDXLEdge` that was created using `ndxlCreateEdge`. If `edge` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

If `edge` is not 0, and is connected to `vertex` (which must also not be 0), returns the state vector of the other `NDXLVertex` joined by `edge`, as seen by `vertex`, otherwise returns 0. The "relative" state is defined as the state at $t - \tau$, where t is the current time of codevertex, and τ is the delay of `edge`.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

Argument error (`NDXL_ARG_ERR`) if `neighbours` is 0.

6.5.12 `ndxlGetVertexNetwork`

```
NDXLNetwork* ndxlGetVertexNetwork(
    in NDXLVertex* vertex
);
```

Description:

Retrieves the `NDXLNetwork` that `vertex` is a part of.

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

Pointer to `NDXLNetwork` that hosts `vertex`, or 0 if `vertex` is 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

6.5.13 `ndxlGetVertexNextEdge`

```
NDXLEdge* ndxlGetVertexNextEdge(
    in NDXLVertex* vertex,
    in NDXLEdge* current_edge
);
```

Description:

Retrieves the `NDXLEdge` that was added to `vertex` after `current_edge`. This function is typically used in conjunction with `ndxlGetVertexFirstEdge` and `ndxlGetVertexLastEdge` to iterate over all edges connected to `vertex` (see section **??**).

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

current_edge: pointer to a `NDXLEdge` that was created using `ndxlCreateEdge`. If `current_edge` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

The `NDXLEdge` pointer that was added to `vertex` after `current_edge`, or 0 if either `vertex` or `current_edge` are 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

Argument error (`NDXL_ARG_ERR`) if `current_edge` is 0.

Value error (`NDXL_VALUE_ERR`) if `current_edge` is not connected to `vertex`.

6.5.14 `ndxlGetVertexOrder`

```
const size_t ndxlGetVertexOrder(
    in NDXLVertex* vertex
);
```

Description:

Retrieves the order of `vertex`.

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

The number of parameters in the state vector of `vertex`, or 0 if `vertex` is 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

6.5.15 `ndxlGetVertexState`

```
const double* ndxlGetVertexState(
    in NDXLVertex* vertex
);
```

Description:

Retrieves the state vector of `vertex`.

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

The current state vector of `vertex`, or 0 if `vertex` is 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

6.5.16 `ndxlSetVertexState`

```
const int ndxlSetVertexState(
    in NDXLVertex* vertex,
    in const double* state
);
```

Description:

Updates the state vector of `vertex`.

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

state: the new state vector that `vertex` must take on. This parameter **may not be** 0, and must correspond to a vector that is **exactly the length of the order of `vertex`**. Passing an invalid parameter will result in undefined behaviour.

Returns:

`NDXL_OK` if `vertex` and `state` are not 0, otherwise `NDXL_ERR`.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex` is 0.

Argument error (`NDXL_ARG_ERR`) if `state` is 0.

6.6 `NDXLEdge` functions

This section details all functions pertaining to `NDXLEdge`.

6.6.1 `ndxlCreateEdge`

```
NDXLEdge* ndxlCreateEdge(
    in NDXLVertex* vertex1,
    in NDXLVertex* vertex2,
    in const double* state,
    in const unsigned char order,
    in const double delay,
    in const void* params,
    in const char directed,
    in NDXLEdgeFunction func,
    in NDXLEdgeNoise noise,
    in NDXLEdgeCallback callback
);
```

Description:

Adds a new edge between `vertex1` and `vertex2`, and returns the newly created `NDXLEdge`.

Parameters:

vertex1: the first vertex this edge joins. This parameter **may not be** 0, and must have the same host network as `vertex2`.

vertex2: the second vertex this edge joins. This parameter **may not be** 0, and must have the same host network as `vertex1`.

state: the initial state vector of this edge, of length `order`. This parameter **may not be** 0, and its size cannot be altered once the edge has been created.

order: the number of elements in `state`. Must be greater than 0 and less than 256.

delay: how long, in time units, data from `vertex1` take to reach `vertex2` (and vice versa, if the edge is not directed). This parameter may be 0 (delay-less edge), but not negative.

params: pointer to arbitrary extra data used by the edge. Note that

`NDXL` does **NOT** make a copy of this data, so it is up to the user to ensure the data pointed to by `params` remains in-scope for the duration of the edges's lifetime. If no extra data are used, this parameter may be 0.

directed: Boolean value. If `true`, data may only flow from `vertex1` to `vertex2`, and not vice versa; if `false`, data may flow from either vertex to the other one.

func: the evolution function of this edge (see section 3.2.3 for signature). This parameter may be 0, in which case the state of the edge will never change (static edge).

noise: the noise function of this edge (see section 3.2.3 for signature). This parameter may be 0, in which case there is no noise in the signal of the edge (quiet edge).

callback: the callback function of this edge (see section 4.4 for signature). This parameter may be 0, in which case there is no callback associated with the edge (no external perturbations).

Returns:

If no errors are raised, returns a new `NDXLEdge` pointer, otherwise returns 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `vertex1` is 0.

Argument error (`NDXL_ARG_ERR`) if `vertex2` is 0.

Argument error (`NDXL_ARG_ERR`) if `state` is 0.

Value error (`NDXL_VALUE_ERR`) if `order` is less than or equal to 0.

Value error (`NDXL_VALUE_ERR`) if `delay` is less than 0.

Value error (`NDXL_VALUE_ERR`) if `vertex1` and `vertex2` are not part of the same network.

6.6.2 `ndxlCopyEdge`

```
NDXLEdge* ndxlCopyEdge(
    in NDXLEdge* edge,
```

```
    in NDXLVertex* vertex1,
    in NDXLVertex* vertex2,
    in const char directed,
    in const char copy_state0
);
```

Description:

Creates a new `NDXLEdge` joining `vertex1` and `vertex2`, by copying the state and parameters of the `NDXLEdge` pointed to by `edge`.

Parameters:

edge: the edge whose settings must be copied to create the new edge. This parameter **may not be** 0.

vertex1: the first `NDXLVertex` that the new edge must connect. This parameter **may not be** 0, and must have the same host network as `vertex2`.

vertex2: the second `NDXLVertex` that the new edge must connect. This parameter **may not be** 0, and must have the same host network as `vertex1`.

directed: Boolean value. If `true`, data may only flow from `vertex1` to `vertex2`, and not vice versa; if `false`, data may flow from either vertex to the other one.

copy_state0: Boolean value. If `true`, the vertex is created by copying the **initial** state of `vertex` as its own initial state; if `false`, the vertex is created by copying the **current** state of `vertex` as its own initial state.

Returns:

If no errors are raised, returns a new `NDXLEdge` pointer, otherwise returns 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `edge` is 0.

Argument error (`NDXL_ARG_ERR`) if `vertex1` is 0.

Argument error (`NDXL_ARG_ERR`) if `vertex2` is 0.

Value error (`NDXL_VALUE_ERR`) if `vertex1` and `vertex2` are not part of the same network.

6.6.3 `ndxlDestroyEdge`

```
void ndxlDestroyEdge(
    in NDXLEdge* edge
);
```

Description:

Destroys the `NDXLEdge` pointed to by `edge`, freeing all resources used by it, and removing it from its host network.

Parameters:

edge: pointer to a `NDXLEdge` that was created using `ndxlCreateEdge`. If `edge` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

None.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `edge` is 0.

6.6.4 `ndxlGetEdgeDelay`

```
const size_t ndxlGetEdgeDelay(
    in NDXLEdge* edge
);
```

Description:

Retrieves the delay of `edge`.

Parameters:

edge: pointer to a `NDXLEdge` that was created using `ndxlCreateEdge`. If `edge` is 0, the function does nothing, but an invalid pointer will result

in undefined behaviour.

Returns:

The delay of `edge`, or 0 if `edge` is 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `edge` is 0.

6.6.5 `ndxlGetEdgeState`

```
const double* ndxlGetEdgeState(
    in NDXLEdge* edge
);
```

Description:

Retrieves the state vector of `edge`.

Parameters:

vertex: pointer to a `NDXLVertex` that was created using `ndxlCreateVertex`. If `vertex` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

Returns:

The current state vector of `edge`, or 0 if `edge` is 0.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `edge` is 0.

6.6.6 `ndxlSetEdgeDelay`

```
const int ndxlSetEdgeDelay(
    in NDXLEdge* edge,
    in const double delay
);
```

Description:

Updates the delay value of `edge`.

Parameters:

edge: pointer to a `NDXLEdge` that was created using `ndxlCreateEdge`. If `edge` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

delay: the new delay that `edge` must take on. Must be greater than or equal to 0.

Returns:

`NDXL_OK` if `edge` is not 0 and `delay` is not less than 0, otherwise `NDXL_ERR`.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `edge` is 0.

Value error (`NDXL_VALUE_ERR`) if `delay` is less than 0.

6.6.7 `ndxlSetEdgeState`

```
const int ndxlSetEdgeState(
    in NDXLEdge* edge,
    in const double* state
);
```

Description:

Updates the state vector of `edge`.

Parameters:

edge: pointer to a `NDXLEdge` that was created using `ndxlCreateEdge`. If `edge` is 0, the function does nothing, but an invalid pointer will result in undefined behaviour.

state: the new state vector that `edge` must take on. This parameter **may not be** 0, and must correspond to a vector that is **exactly the length of the order of `edge`**. Passing an invalid parameter will result in undefined behaviour.

Returns:

`NDXL_OK` if `edge` and `state` are not 0, otherwise `NDXL_ERR`.

Errors raised:

Argument error (`NDXL_ARG_ERR`) if `edge` is 0.

Argument error (`NDXL_ARG_ERR`) if `state` is 0.