April 16, 2021

LINKÖPING UNIVERSITY

HOPSAN

Tutorial - Writing Component Libraries

Introduction

This tutorial will show how to create your own component libraries from C++ by using the builtin editor in the Hopsan graphical interface. In the first section, the requirements are listed. In the second section, all files in a component library are explained. Finally, creating a library is demonstrated with a step-by-step guide in section three.

Requirements

It is strongly recommended to complete the *Getting Started* tutorial before undertaking this one. Some basic knowledge of programming languages are also recommended. The following tools are also required:

- Hopsan
- Inkscape (optional, for creating component icons)

Hopsan Component Libraries

A component library consists of one or more components that can be loaded from Hopsan. It is written in C++ and then compiled to a shared library file (`.dll` in Windows or `.so` in Linux). A complete library constists of the files listen below. First, there is a *library description file* in `.xml` format, which contains basic information about the library. Second, there is a source code file of the library with the `.cpp` extension.

Component Library
- **MyLibrary.xml** – Library description file
- **MyLibrary.cpp** – Library source code
- Component
  - **MyFirstComponent.hpp**
  - **MyFirstComponent.xml**
  - **MyFirstComponent.svg**
- Component
  - **MySecondComponent.hpp**
  - **MySecondComponent.xml**
  - **MySecondComponent.svg**

Each component in the library consist of three files. The first one has the `.hpp` extension, and contains the source code of the component. The second is an `.xml` file, which contains specifications about the component and its appearance. Finally there is a `.svg` file, which contains the graphical icon.

Library Description File

The library description file contains general information, such as name, version and source files. With this file it is possible to recompile libraries from inside Hopsan. It also works as a project file for editing the library. It can for example look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<hopsancomponentlibrary version="0.3">
    <id>34992911-0c43-485a-abd8-ccf754bac212</id>
    <name>LibName</name>
    <lib>LibName</lib>
    <source>LibName.cpp</source>
    </buildflags>
    <component>Component1.hpp</component>
    <component>Component2.hpp</component>
    <componentxml>MyComponent1.xml</componentxml>
    <componentxml>MyComponent2.xml</componentxml>
</hopsancomponentlibrary>
```

The first line contains general information about the XML format. It always looks the same. The remaining tags are explained below:

- **hopsancomponentlibrary** - Main tag for a component library
  - **version** - Version of the library xml, should be 0.3 with this XML structure
- **id** - Unique identifier for the library. Can be any random text.
- **name** - Name of the library to be shown in Hopsan
- **lib** - Base file name of the library file (without prefix, suffix and extension)
- **source** - Source file used to compile the library
- **buildflags** - Define custom flags for the compilation, unused in this tutorial
- **component** - Source file for a component in the library
- **componentxml** - Appearance `.xml` file for a component in the library

Library Source File

The library source file can be viewed as the wrapper file, that takes the source code for each component and puts them together to a library. It contains code that is used to register you library components in the HopsanCore, and provides general information about the library for Hopsan.

```
#include "Component1.hpp"
#include "Component2.hpp"

#include "ComponentEssentials.h"
using namespace hopsan;

extern "C" DLLEXPORT void register_contents(ComponentFactory* pComponentFactory,
                                            NodeFactory* pNodeFactory)
{
    pComponentFactory->registerCreatorFunction("Component1", Component1::Creator);
    pComponentFactory->registerCreatorFunction("Component2", Component2::Creator);

    HOPSAN_UNUSED(pNodeFactory)
}

extern "C" DLLEXPORT void get_hopsan_info(...)
{
    pHopsanExternalLibInfo->libName = (char*)"LibName";
    pHopsanExternalLibInfo->hopsanCoreVersion = (char*)HOPSANCOREVERSION;
    pHopsanExternalLibInfo->libCompiledDebugRelease = (char*)HOPSAN_BUILD_TYPE_STR;
}
```

First of all, all component code files must be included. This is shown in lines 1 and 2 in the example below. Second, each component needs to be registered in the HopsanCore by the `registerCreatorFunction()`, as shown in lines lines 10 and 11. The first argument must be a unique *type name* that identifies you component. The second argument is the `Creator()` function, which must exist in each component. Finally, the `get_hopsan_info()` function must provide a unique *library name*, here called "LibName" at line 18. The next two lines should be left as they are.

Component Source Files

Components are written in C++ files with the .hpp file extension. We will now go through the fundamental pars of a component file.

```
1  #ifndef LAMINARORIFICE_H
2  #define LAMINARORIFICE_H
3
4  #include "ComponentEssentials.h"
5  #include "ComponentUtilities.h"
6
7  namespace hopsan {
8
9      class LaminarOrifice : public ComponentQ
10     {
11     private:
12         double *mpP1_p1, *mpP1_q1, *mpP1_c1, *mpP1_Zc1;
13         double *mpP2_p2, *mpP2_q2, *mpP2_c2, *mpP2_Zc2, *mpKc;
14         Port *mpP1, *mpP2;
15
16     public:
17         ...
```

The first two lines sets a header guard to avoid including the same code twice. Then we include the essential functions for the component from `ComponentEssentials.h`, and all built-in utilities from `ComponentUtilities.h`. You may also include other external header files if you wish to use functions from external libraries.

On line 9 the component class is declared. We inherit from the `ComponentQ` class, since this is a Q-type component. Similarly we would have used `ComponentC` for C-type or `ComponentSignal` for signal type.

The first contents in the class is the private variables, which will be persistent in the component. In this case we have nine node data pointers and two ports. A pointer is a variable that points to another variable located somewhere else. It is used in components to improve performance. After the private section, the public section begins. Public variables and functions can be accessed by the outside world, while private are only allowed to be accessed from functions belonging to the class.

```
18 static Component *Creator()
19 {
20     return new LaminarOrifice();
21 }
```

In the public part we first define a static creator function, which is used to create instances of the component in the simulation core. Nothing needs to be changed except the name of the class.

```
22 void configure()
23 {
24     mpP1 = addPowerPort("P1", "NodeHydraulic");
25     mpP2 = addPowerPort("P2", "NodeHydraulic");
26     addInputVariable("Kc","Coefficient","m^5/Ns", 1.0e-11, &mpIn);
27 }
```

The second function you need to define is the `configure()` function for the component. This function is run every time a new instance of the component is added to the model. The function is

used to to register ports, input variables, output variables and constants, and to configure default values for persistant variables. First we create the ports used for communication with the surrounding components, in this case two hydraulic power ports, see line 24 and 25. Then on line 26 we register the restrictor coefficient as an input variable with name, description, unit and default value. Input variables can be used either as signal inputs or as parameters. It is also possible to add constant parameters and output variables using the `addConstant()` and `addOutputVariable()` functions.

```
28  void initialize()
29  {
30      mpND_p1 = getSafeNodeDataPtr(mpP1, NodeHydraulic::Pressure);
31      mpND_q1 = getSafeNodeDataPtr(mpP1, NodeHydraulic::Flow);
32      mpND_c1 = getSafeNodeDataPtr(mpP1, NodeHydraulic::WaveVariable);
33      mpND_Zc1 = getSafeNodeDataPtr(mpP1, NodeHydraulic::CharImpedance);
34
35      mpND_p2 = getSafeNodeDataPtr(mpP2, NodeHydraulic::Pressure);
36      mpND_q2 = getSafeNodeDataPtr(mpP2, NodeHydraulic::Flow);
37      mpND_c2 = getSafeNodeDataPtr(mpP2, NodeHydraulic::WaveVariable);
38      mpND_Zc2 = getSafeNodeDataPtr(mpP2, NodeHydraulic::CharImpedance);
39  }
```

The next member function that must be defined is the initialize function. This function is run once before each simulation starts. As this function is run after connections have been establish you can read or write to/from connected components. If needed you can use this information to initialize your component properly. This is also the place to allocate additional memory if needed. In this case we initialize the component by using the `getSafeNodeDataPtr()` function to set the node data pointers to point to the variables in the node. These lines are always the same for hydraulic nodes, and similar for other node types such as mechanical, pneumatic and electric.

```
40  void simulateOneTimestep()
41  {
42      double p1, q1, c1, Zc1, p2, q2, c2, Zc2;
43
44      //Get variable values from nodes
45      c1 = (*mpND_c1);
46      Zc1 = (*mpND_Zc1);
47      c2 = (*mpND_c2);
48      Zc2 = (*mpND_Zc2);
49      Kc = (*mpND_Kc);
50
51      //Orifice equations
52      q2 = Kc*(c1-c2)/(1.0+Kc*(Zc1+Zc2));
53      q1 = -q2;
54      p1 = c1 + q1*Zc1;
55      p2 = c2 + q2*Zc2;
56
57      //Write new variables to nodes
58      (*mpND_p1) = p1;
59      (*mpND_q1) = q1;
60      (*mpND_p2) = p2;
61      (*mpND_q2) = q2;
62  }
```

The next function, `simulateOneTimestep()`, is the most important function. It contains the model equations that are executed each time step. We begin on line 42 by creating a local variable for each node data pointer. The purpose of this is to make the code more readable. The node data pointers could have been used directly, but this would have been difficult to understand. Then on line 45-49 we assign all input variables with the values from their respective node data pointer. Line 52-55 consists of the actual equations. In this case we calculate flow and pressure through the orifice from wave variables and impedance in the neighboring C-type components. We then end by assigning the output node data pointers with the value of their respective local variable.

```
63  void finalize()
64  {
65          //Finalize code
66  }
67
68  void deconfigure()
69  {
70      //Deconfigure code
71  }
```

The next function, `finalize()`, is optional. It is only useful if you want some code to be run after each simulation has finished. This is usually only needed if you want to free memory that was additionally allocated in the initialize function.

The last function, `deconfigure()`, is also optional. This code is run once the component is deleted. Here you can cleanup any memory allocation or similar that you have done in the configure function.

Component Appearance Files

Information about the component for the graphical interface, such as icon, port positions and component name, is stored in a `.xml` file. This file contains information about the component that is not part of the actual simulation code. This information can be changed without the need to recompile the actual component code. A typical appearance file looks like this:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <hopsanobjectappearance version="0.3">
3      <modelobject typename="LaminarOrifice" displayname="Orifice"
4                   sourcecode="LaminarOrifice.hpp">
5          <icons>
6              <icon type="user" path="orifice.svg"
7                    scale="1.0" iconrotation="ON" />
8          </icons>
9          <help>
10             <text>Help Text</text>
11             <picture>helpPicture.svg</picture>
12             <link>externalHelpDocumentation.pdf</link>
13         </help>
14         <ports>
15             <port x="1,0" y="0.5" a="0" name="P1" visible="true"//>
16             <port x="0,0" y="0.5" a="180" name="P2" visible="true"//>
17             <port x="0.5" y="0,0" a="270" name="Kc" visible="false"//>
18         </ports>
19     </modelobject>
20 </hopsanobjectappearance>
```

The first line contains basic information about the XML code, and should always look the same. A description of the remaining tags follows:

- **hopsanobjectappearance** - Main tag for appearance file
  - **version** - Should always be 0.3 with this XML layout
- **modelobject** - Main tag for the component
  - **typename** - Unique type name of the component
  - **subtypename** - Specific version of a type name component (optional)
  - **displayname** - Name for the component shown in the graphical interface
- **icons** - Contains information about icons. At least one type, user or iso is required

  - **type** - The icon type, user or iso (for ISO 1219 graphics)
  - **path** - Relative path from the `.xml` file to the `.svg` icon
  - **scale** - Lets you adjust the scale of the `.svg` icon (default = 1.0)
  - **iconrotation** - Tells whether or not the icon rotates when the component is rotated
- **help** - Allows you to specify help information about the component (optional)
  - **text** - The help text (optional)
  - **picture** - The path to the `.svg` help picture (optional)
  - **link** - Link to external document relative this `.xml` file (optional)
- **ports** - Defines the positions and orientations for each port
  - **name** - Name of the port as defined in the code
  - **x** - X-position as fraction of component icon width (0.0 = left, 1.0 = right)
  - **y** - Y-position as fraction of component icon height (0.0 = top, 1.0 = bottom)
  - **a** - Angle of port, 0 = right, 90 = down, 180 = left, 270 = up
  - **visible** - Default visibility state

Component Icon Files

Icons for the graphical interface are stored in the `.svg` (Scalable Vector Graphics) format. A good and free tool for creating and editing such files is *Inkscape*. If you want to use bitmaps graphics, `.jpg`, `.png` or similar formats, such graphics can be embedded in a `.svg` file.

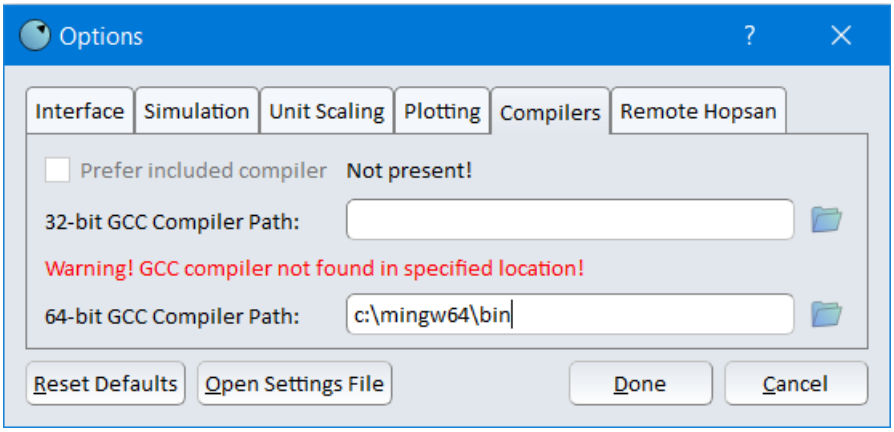
Example

We shall now demonstrate how to build a simple library using the built-in editing capabilities in Hopsan. The library will contain only one simple arithmetic component.

1. **Open the Hopsan Graphical User Interface**
   Start Hopsan by running hopsangui.exe, located in the `bin` folder under the installation directory.
2. **Configure compiler path**
   First, we must configure the compiler. Click on he options icon:

   Options

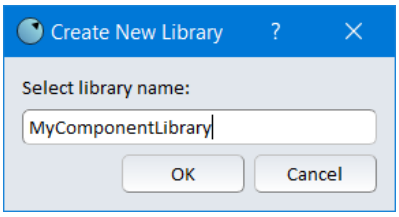
   Hopsan needs access to either a 32-bit or a 64-bit compiler, depending on the Hopsan architecture. Click on the "Compilers" tab. Then make sure a required compiler, or the included compiler, is specified. It should look something like this:



3. **Create a new library**
   Now we will create a new empty library. Click on `Create external library` in the library widget to the left in the program:

   Create external library

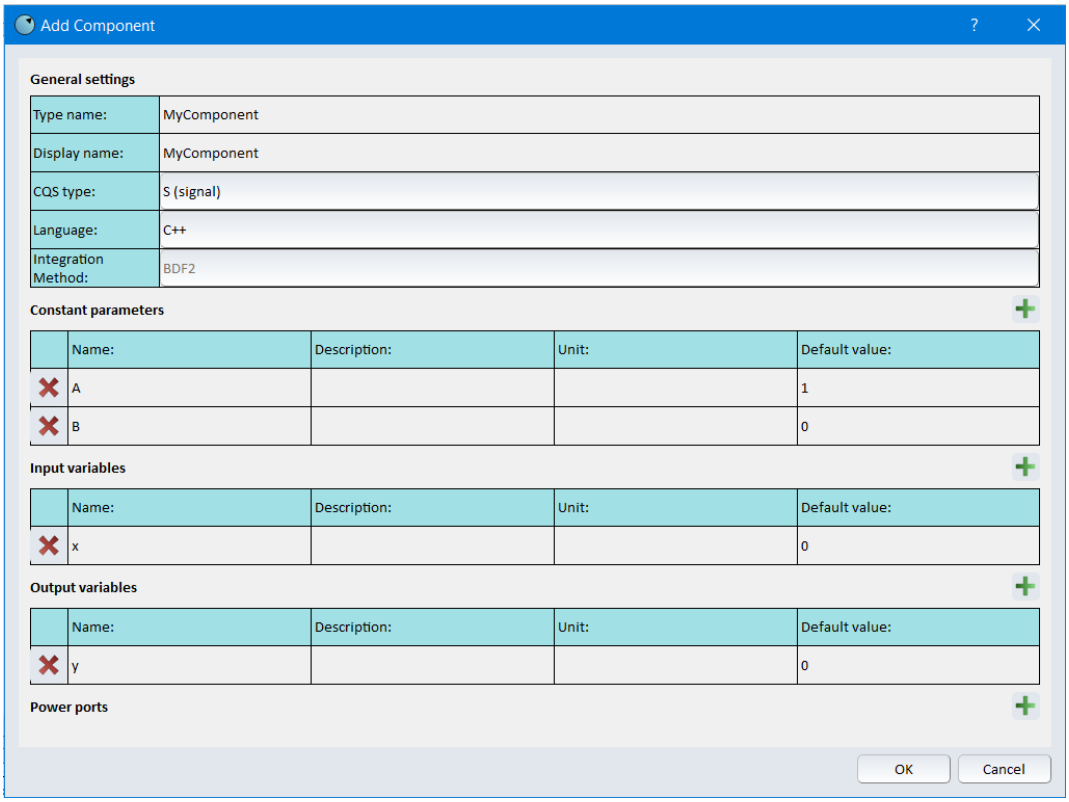
   A dialog appears that asks you to name the library. Give the library a name without spaces, for exampe "MyComponentLibrary":



   When clicking "OK", the program asks you to choose a folder for the library. Make sure the folder you choose is empty. Two files have now been created in the folder, one `.xml` file and one .cpp file.

4. **Add a component**
   A component library is not very useful without any components. We now want to create a component that solves the equation $y = A * x + b$. Right-click on the component library and choose "Add New Component".

   A component generator dialog appears. We need to choose a name. We also need 2 constants (A and B), 1 input variable (x) and 1 output variable (y). Variables are added by clicking on the green plus icons to the right. Give A and B the default values 1 and 0. This means that with default values, $y = x$. The dialog should then look like this:



   Ignore units and descriptions for now and click "OK". Two new files will be created, one `.hpp` and one `.xml`.

5. **Write component code**
   Now it is time to implement the equation in the component. The source code file `MyComponent.hpp` should have opened automatically. If not, right-click on the component and choose "Edit Source Code".

   As you can see, all functions mentioned in the previous chapter has been generated. In this case we only care about the `simulateOneTimeStep()` function. Add the following under the line "//Simulation code":

   ```
   y = mA*x+mB;
   ```

   Note that in the code, the constants A and B are renamed to `mA` and `mB`. This is to indicate that they are *member* variables of the class. Remember the semicolon! C++ requires each line to end with a semicolon.

   The component is now ready to be compiled. It is possible to modify the port positions in the `.xml` file, and to create a graphical icon. Let's ignore that for now.

6. **Compile library**
   Right-click on the library in the list to the left and select "Recompile". If compiler path is correct and there are no mistakes in the code, the library should now be successfully compiled. Otherwise, look for errors in the compilation dialog.

7. **Test your new component**
   Build a new model and verify that your own first component works!