RobotSDK 4.0 : A Complete Top-down Modular Software Development Framework

[Quick Manual]

HMW-Alexander

2015-04-18

Contents

Chapter 1

Introduction

After two weeks of hard programming with many accumulated new ideas and important feedbacks, a powerful RobotSDK 4.0 is coming. It has indeed realized a complete top-down software development framework compared with its predecessors, but still follows their principle of abstract graph model and modular software development.

1.1 New Features of RobotSDK 4.0

1.1.1 Robot-X

For RobotSDK 2.1, 2.2 and 3.0, users need to program a high level application to realize a conceptual graph-model, in which functional modules could embed. Although, RobotSDK 3.0 provides a set of simplified functions to draw such graph-model, users are still trapped in the problems like misspelling of node name, frustrating complex network and confusion of node types.

Now, in RobotSDK 4.0, instead of programming annoying high level applications, users only need to draw it on Robot-X without considering different node types (Fig.1.1). And then, users can freely embed required functional modules via shared library and fully control all nodes in it. Therefore, in RobotSDK 4.0, for common development, additional high level programming is not needed.

Meanwhile, Robot-X provides a convenient Config Panel UI (Fig.1.2) to modify configuration values in XML file for each node. With this tool, users need not to manually modify confused XML config file.

1.1.2 Simple and Free Module Development

From RobotSDK 2.2, assistant tools were created to automatically generate interface source codes for module development. However, the auto-generated source codes are so long and complex that developers could not easily handle them. About 40% error feedbacks are related to such sick source codes. Another bad thing is that the auto-generated source codes limited developers to their constant frame and even for experienced developers, it is not free to program modules as they want.

For RobotSDK 4.0, thanks to C++11 techniques, the module development becomes simple and free. The auto-generated interface source codes become very short and for each module, the number of generated files shrinks from 6 to 2 (1 header and 1 source). Moreover, a set of intuitive syntax is introduced to simplify programming. For functions development, there is no constant framework. Just like C programming, it only needs a entry function, which is also named as ”main”, therefore, developers can freely program module in C style.

1.1.3 Kernel Update

Up to now, there are about 50% bug feedbacks are related to Kernel and there are 4 critical bugs related to design defects of RobotSDK 3.0:

1. Data stream is not stable: caused by unbalanced producer-consumer model.

2. Simulator is not accurate: caused by mixture of intra-thread and inter-thread communication.

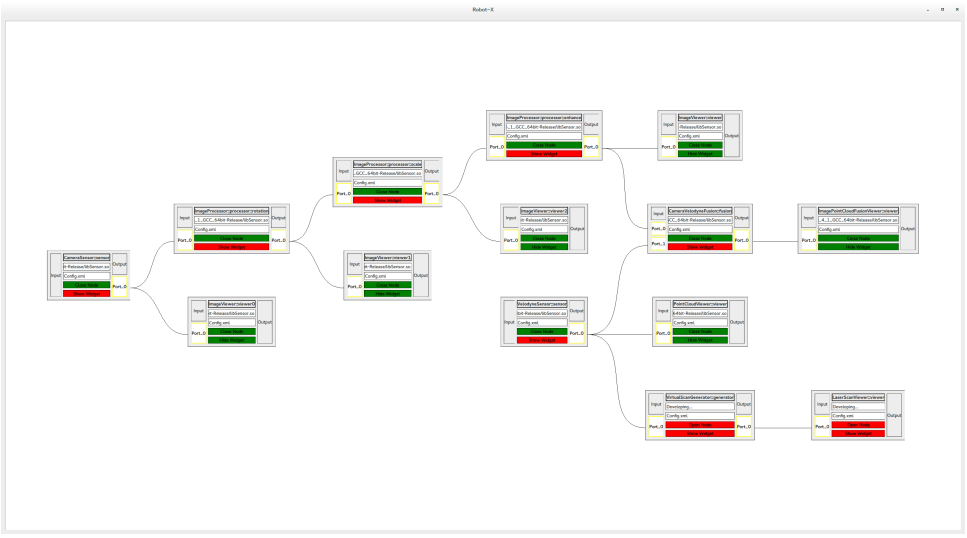

Figure 1.1: Robot-X and the Conceptual Graph-Model

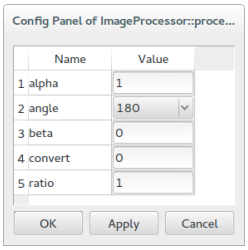

Figure 1.2: Config Panel UI to modify configurations values of ImageProcessor node

3. Timestamp of received data is not accurate: same with 2.

4. High-speed data stream overwhelms others with low-speed for node while using Edge Monitor: same with 2.

In RobotSDK 4.0, the construction, maintenance and destruction of data stream are redesigned after learning related Qt source codes.

Each node, if it has input, output and some intrinsic triggers, it will have 4 independent threads: Main thread, InputPorts thread, OutputPorts thread and QObjectPool thread. Unlike RobotSDK 3.0, these threads will not be closed until the node is destroyed (not closed, see abstract node state machine in Fig.1.3). Therefore, even the node is closed, the independent input ports could still receive data in its InputPorts thread and also send the packed data (if and only if the port buffer fulfills user's requirement) to the Main thread. Then, for the node thread, it uses a EventFilter technique provided by Qt to filter out packed data when the node is closed. This guarantees the balance of producer-consumer model, as well as no deadlock while closing node.

The mixture of intra-thread and inter-thread communication is the most significant design defect of RobotSDK 3.0. The default intra-thread communication between signal and slot is direct call and the

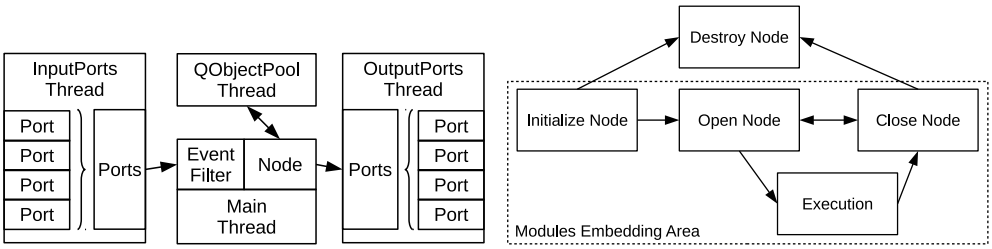

Figure 1.3: Threads of a node and abstract node state machine

inter-thread communication between them is based on event-loop. The direct call would block event-loop and event-loop itself means lag existence for high frequency communication. I will explain the problems with some examples below.

Take Simulator in RobotSDK 3.0 for example, it takes a QTimer in the abstract node to trigger itself according the data record. However, the QTimer is not in the node's Main thread, therefore, the signal timeout" emitted by QTimer is queued in the event-loop of Main thread. Then high frequent data record will lead to signal block in the Main thread, which will cause simulator not accurate. In RobotSDK 4.0, developers could freely set QTimer in module and link it to the Main thread without moving it to the QObjectPool thread. Then the link between QTimer and the Main thread is intra-thread communication and there will be no lag except that the execution time costs exceeds time interval.

Take EncoderIMU node developed by Chen for example, it is a duplex communication node for COM port. It uses inter-thread communication for data receiving: after serial port receiving a byte data, it will emit a signal to Main thread to ask node to check serial port buffer. However, the serial port object is in another thread, therefore, the signal is queued in the event-loop. If the COM frequency is very high, then the event-loop will be blocked. Meanwhile, the timestamp is determined in the Main thread after checking serial port buffer. Therefore, the timestamp is not accurate. In RobotSDK 4.0, we recommend developers to redesign serial port and make it check its own buffer and determine the timestamp of data before emit signal to Main thread. With such serial port, it is safe to move it to the QObjectPool thread and use inter-thread communication.

Take Edge Monitor in RobotSDK 3.0 for example, to make sure it can monitor all nodes behavior in real-time, the communication between node and edge is forcefully set as "DirectConnection" and thus they are actually in the same thread with intra-thread communication. The surprising thing is that it is not nodes that move into the edge's thread, but is on the contrary, meanwhile, all nodes are still in their own threads. That is the problem, for a high frequent node, it communicates with edge using intra-thread mechanism (direct call), which will indirectly block other nodes' direct call and event-loop. Therefore, in RobotSDK 4.0, we removed the Edge Monitor and developers could freely customize monitor tools by implementing extended node (see Chapter 4).

1.1.4 Integrate Mature Tools

RobotSDK 4.0 integrates 3 mature tools for module development.

- Sync: synchronize multi-input data
- GLViewer: 3D visualization tools (needs Eigen library)
- ROSInterface: Communication interface to ROS (needs Linux and ROS)

These tools have been tested for a long time and are reliable for developers.

1.2 How to Install RobotSDK 4.0

1.2.1 Linux

For Linux users, to install RobotSDK 4.0 is very easy.

1. **Open TermianI:** Ctrl + Alt + t
2. **Get Source Code:** git clone https://github.com/RobotSDK/RobotSDK.git
3. **Go to RobotSDK:** cd RobotSDK
4. **Install RobotSDK:** sh install.sh

Then RobotSDK 4.0 will be installed under $(HOME)/SDK/RobotSDK_4.0, there are four folders.

- Build: intermediate products of compiling
- Doc: documentation
- Kernel: RobotSDK kernel
- Robot-X: The super high level application

1.2.2 Windows

For Windows users, to install RobotSDK 4.0 needs additional work.

1. **Prerequisite:**
   - Visual Studio 2013 or later, C++11 support:
     `https://msdn.microsoft.com/en-us/library/hh567368.aspx`
   - Qt (includes QtCreator and Qt Visual Studio Add-in):
     `https://www.qt.io/download-open-source/`
   - GraphViz (for Robot-X), only 32bit installation, compile 64bit version from source code:
     `http://www.graphviz.org/Download_windows.php`
     `https://github.com/RobotSDK/graphviz`
   - Eigen (for GLViewer [optional]):
     `http://eigen.tuxfamily.org/index.php?title=Main_Page`
2. **Configure PATH:** You need to add Qt's and GraphViz's bin folders to PATH and restart system.
3. **Get Source Code:** Download from GitHub:
   `https://github.com/RobotSDK/RobotSDK.git`
4. **Open Visual Studio Console and cd to source code:**
   Only Visual Studio Console has nmake
5. **Set Environment Variables:** input commands in console
   set GRAPHVIZ_PATH=(Graphviz Installation path)
   set EIGEN_PATH=(Eigen Installation Path [optional])
   (It is better to add them to system environment variables. If you have added them, you need not to input these in console for installation and this will be convenient for functional module development.)
6. **Install RobotSDK:** input commands in console
   install.bat

Then RobotSDK 4.0 will be installed under c:/SDK/RobotSDK_4.0, there are also four folders like Linux and the VS project file is in the Build folder.

1.3 About the Quick Manual

In this Quick Manual, I only show how to develop software system via RobotSDK 4.0 without too much details. If you have any questions, please leave issues on my GitHub [`https://github.com/RobotSDK/RobotSDK/issues`]. I will respond it as soon as possible.

All the examples in this manual are conducted in Linux, the general operation steps in Windows are same with that in Linux, and we will inform readers the differences in Windows just after each example.

The images in this manual are in .eps format, therefore, you could zoom in for clear view of details.

Chapter 2

Robot-X: A Super High Level Application

2.1 Abstract Graph Model Construction

Many softwares could be described as an abstract graph model, whose node represents functional module and edge represents communication link. The abstract graph model is a high level concept which is the start of developing software in top-down way. RobotSDK is designed to help developers to easily conduct top-down software development.

After starting Robot-X, an empty canvas is shown as Fig.2.1. You can add nodes and connect them with edges to form an abstract graph model, and also modify it as you want. After construction of graph model, you can use Robot-X to generate source code for modules development. Finally, after module development, you can use Robot-X to launch the software. In this chapter, we will introduce how to use Robot-X.

2.1.1 Node

2.1.1.1 Add Node

To add a node, right click on the empty part of the canvas and a menu pop up as Fig.2.2 left. There are two kinds of node you can add to the graph: virtual node and real node. The virtual node is an abstract node, whose functional module has not been developed. But it can still be included in the abstract graph model and this is the key for top-down development. The real node's functional module has been compiled in some shared libraries, which should be loaded after creating the node as Fig. 2.2 right.

As shown in the middle of Fig.2.2, in order to create a node, you need to give it a *"Node Full Name"*. The name rule is:

NodeClass::NodeName[::ExName]

The "NodeClass" of a node is one part of query ID for RobotSDK to locate its functional module in shared library. For each module, it should have a unique "NodeClass" in one shared library. The

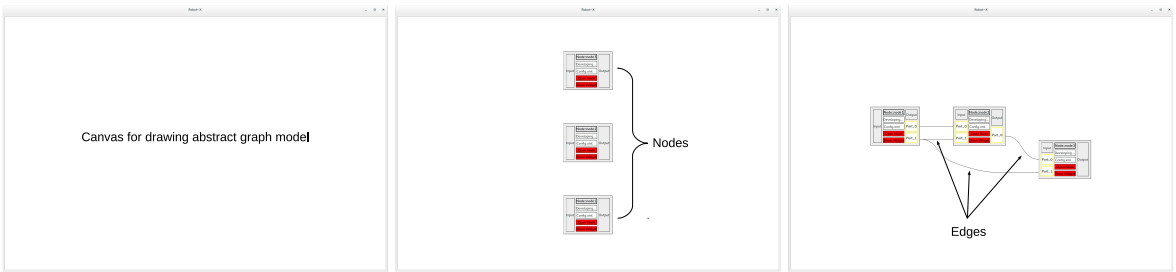

Figure 2.1: Robot-X UI

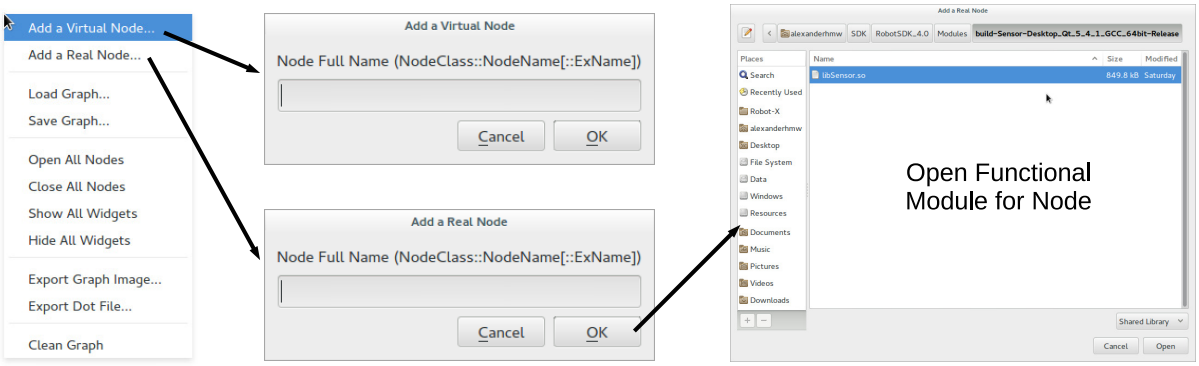

Figure 2.2: Add Virtual Node and Real Node Operations

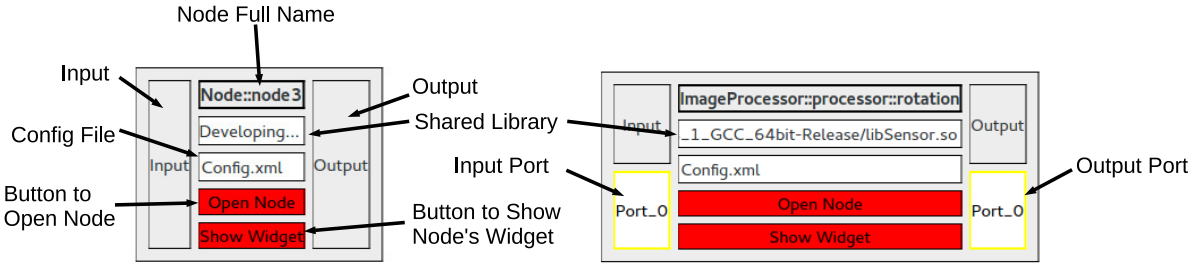

Figure 2.3: New Virtual Node (Left) and Real Node (Right) in Abstract Graph Model

"NodeName" could be any name you want to instantiate a "NodeClass". The "ExName" is optional and is another part of the query ID for RobotSDK to locate the module's extended functionality in the shared library. In a graph model, users only need to guarantee the uniqueness of "Node Full Name".

2.1.1.2 Virtual Node and Real Node

After adding either virtual or real node, it will be included in the graph as shown in the middle of Fig.2.1. From Fig.2.3, we could find that the only differences between new created virtual and real nodes are 1) input or output ports and 2) shared library. The virtual node does not have any input or output ports and the shared library is still in "Developing...". The real node has input or output ports which is defined in the functional module from a certain shared library.

2.1.1.3 Set Port Number

After adding virtual node, it does not have both input and output ports as shown in Fig.2.3 left, and users need to to set the input and output port number. To set port number, right click on the "Input" or "Output" label and a menu pop up as Fig.2.4 left. Then choose "Change Port Num" and a dialog for setting port number is shown. After setting the port number and click "OK", the ports will be added to the virtual node as Fig.2.3 right. After change port number, you will find a "Input" or "Output" label with black frame on the top and several "Port_x" labels with yellow frame below.

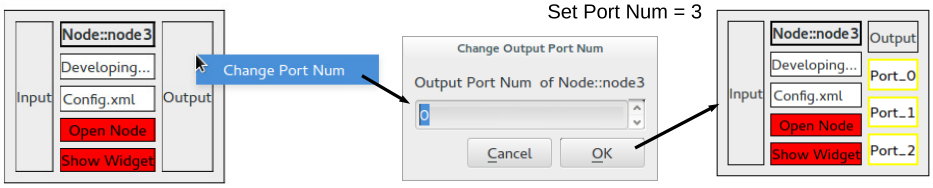

Figure 2.4: Set Port Number

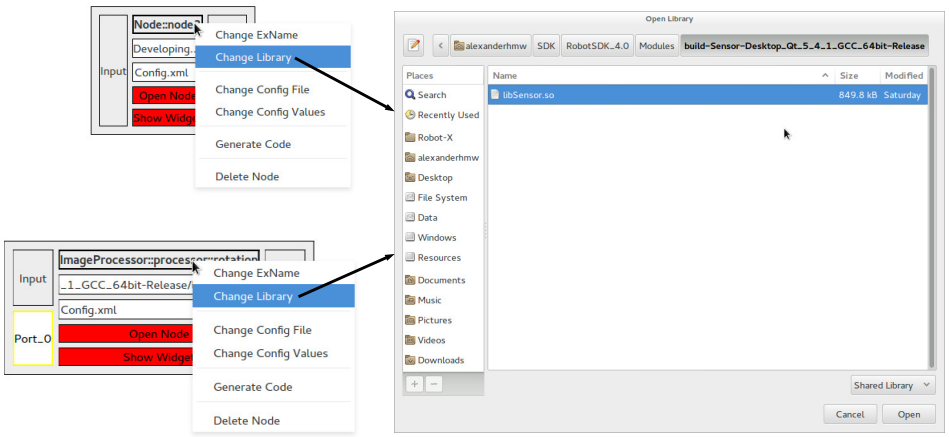

Figure 2.5: Change Shared Library of Virtual and Real Nodes

For real node, it is compatible to change its port number and there will not be any problems while running the software. However, if the new port number is larger than defined one, the exceeded ports will only act as an abstract ports, which convert the real node to be a semi-real or semi-virtual node. If the new port number is smaller than defined one, the lost ports will not appear in the abstract graph model and thus are not available for the software, which only means no input or output for the lost ports.

2.1.1.4 Change Shared Library

According to the principle of top-down software development, virtual node does not have functional module and corresponding shared library at the beginning. Then, after abstract graph model construction and functional module development, the virtual node should turn to a real node, which is to change the shared library from ”Developing...” to real one. To change shared library, right click on the ”Node Full Name” label and a menu pop up as Fig.2.5 top-left. Then choose ”Change Library” and an open file dialog is shown to load shared library just like adding real node operation. The only thing you must guarantee is that the loaded shared library contains the module with desired ”NodeClass” (and ”ExName”).

According to the principle of modular software development, a real node is an interface in which various modules with similar functionalities could embed. Therefore, Robot-X also allows users to change shared library for real node. The operation is same to that for virtual node as Fig.2.5 bottom-left. RobotSDK supports hot-plug of shared library, which means you need not to restart application for changing shared library. However, there is a very important thing, to which the users must pay attention:

> RobotSDK uses QLibrary to load shared library. Once a shared library is loaded, it will not be destroyed until the application exists and thus the shared library with same name will not be loaded again. Therefore, if you modify a loaded shared library, the modification will not be applied unless you restart the application.

2.1.1.5 Change ExName

The ”ExName” is a part of query ID for RobotSDK to locate its module’s extended functionalities. Therefore, once a node is created, you could and only could change the ”ExName” of the ”Node Full Name” to use different extended functionalities. For example, as shown in Fig.2.6, the ”ImageProcessor” base module is to consecutively rotate, scale and enhance an input image and output the processed image. However, these three steps are not always needed. Therefore, we extend it as ”ImageProcessor::–::rotation”, ”ImageProcessor::–::scale” and ”ImageProcessor::–::enhance”. Here, we stress that, these three extensions are not new modules, they are still in base module ”ImageProcessor” and their input and output are same with base module ”ImageProcessor”. The only difference is the ”main function” are different (see Chapter 3).

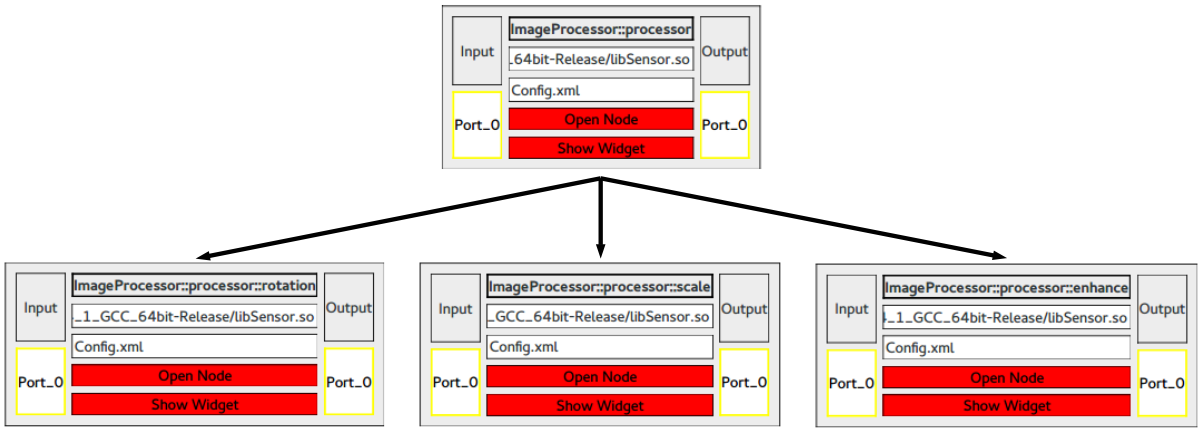

Figure 2.6: Extension of Module ”ImageProcessor”

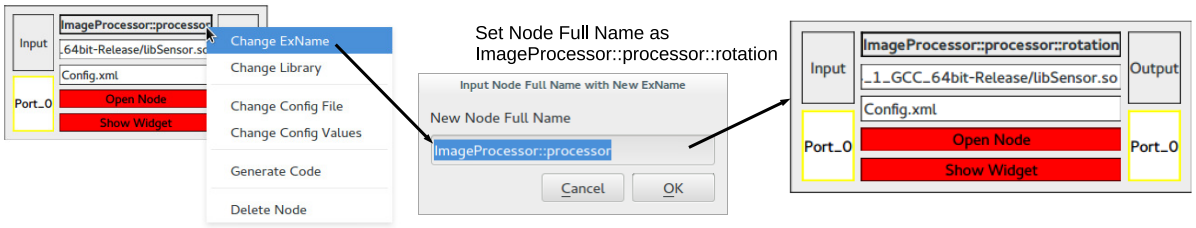

Figure 2.7: Change ExName of Module ”ImageProcessor”

To change the ”ExName”, right click on the ”Node Full Name” label and a menu pop up as shown in Fig.2.7 left. Then choose ”Change ExName” and a dialog to change ”ExName” is shown. Click ”OK” after changing, the node changes its ”ExName”. In the example, the changing is from real node’s base module ”ImageProcessor” to extension ”ImageProcessor::–::rotation”. The first thing to be clarified is that the changing from extension to base module or among module’s extensions is also available. The second thing is that for virtual node, the changing of ”ExName” is also available and it is just the modification of abstract graph model. The third thing is that for real node, the changing of ”ExName” will automatically load desired modules.

2.1.1.6 Change Config File and Values

Just as RobotSDK 3.0, each node is related to a configuration file in XML format. In RobotSDK 4.0, Robot-X provides a way to change configuration file. To change the ”Config File”, right click on the ”Node Full Name” label and a menu pop up as shown in Fig.2.8 left. Then choose ”Change Config File” and an open file dialog is shown to change ”Config File”. Click ”Open” will change node’s ”Config File”. In order to apply the new configuration file, you need to reopen the node, because as shown in Fig.1.3, the configuration file will be loaded in ”Initialize Node” and ”Open Node” states.

In RobotSDK 3.0, users need to manually change XML file contents to realize modification of parameters. In RobotSDK 4.0, Robot-X provides a UI to modify parameters stored in XML file as shown in Fig.2.9 middle. To change the ”Config Values”, right click on the ”Node Full Name” label and a menu pop up as shown in Fig.2.9 left. Then choose ”Change Config Values” and a table dialog is shown to modify parameters. After modification, click ”OK” for save & close, click ”Apply” only for save and click ”Cancel” for abort & close. There are two types of parameters modification: 1) Change value via ”QLineEdit” and 2) Choose value via ”QComboBox”. The corresponding contents in XML file are shown in Fig.2.9 right. To register such parameters, please see Chapter 3.

2.1.1.7 Generate Code

Just like RobotSDK 3.0’s ConfigModule tool, Robot-X could generate interface code for module development after setting up node. To generate code, right click on the ”Node Full Name” label and a

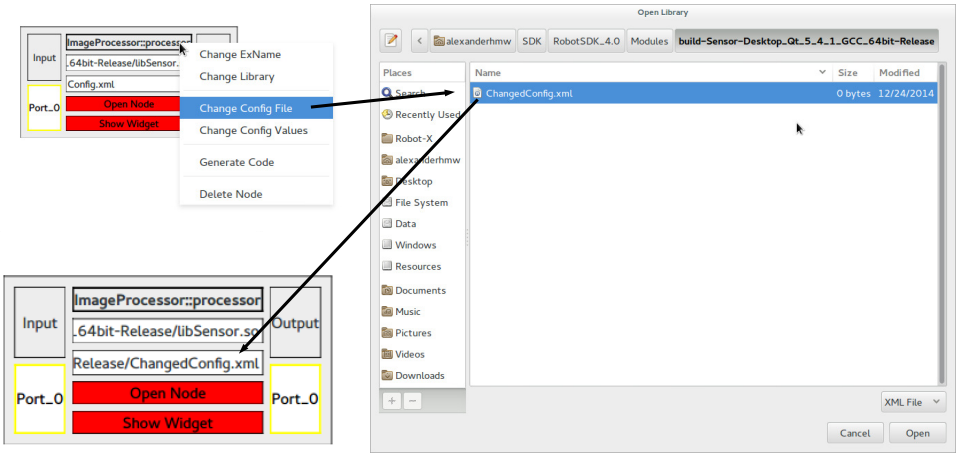

Figure 2.8: Change Configuration File

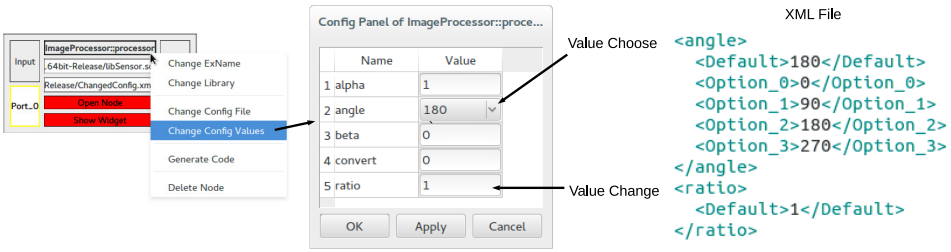

Figure 2.9: Change Configuration Values

menu pop up as shown in Fig.2.10 top-left. Then choose ”Generate Code” and a path open dialog is shown to decide where to store code files. After choosing desired path and clicking open, two code files named as ”NodeClass” will be generated as Fig.2.10 bottom-left and their contents are shown in Fig.2.11. In the source file, there are four predefined functions: ”initializeNode”, ”openNode”, ”closeNode” and ”main”. Their correspondences to Node State in Fig.2.10 bottom-right are: ”Initialize Node”, ”Open Node”, ”Close Node” and ”Execution”. How to program the interface code please see Chapter 3.

2.1.1.8 Delete Node

To delete a node, right click on the ”Node Full Name” label and a menu pop up as shown in Fig.2.12 top. After choosing ”Delete Node”, the selected node will be deleted from the abstract graph mode, as well as its connected edges and communications with other nodes.

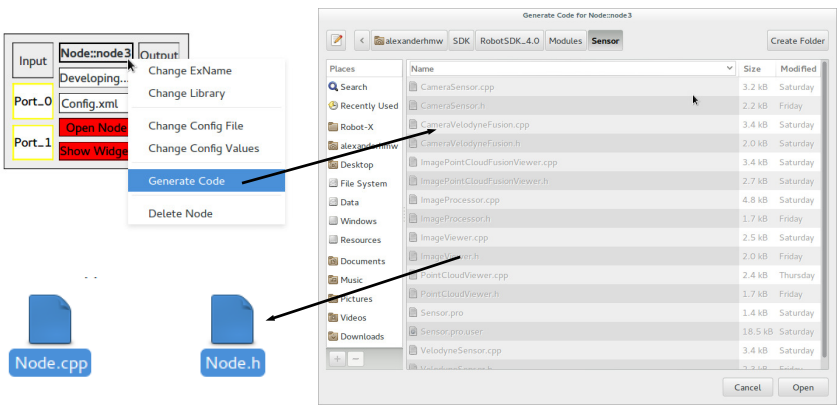

Figure 2.10: Generate Interface Code for Module Development

Node.h

```
#ifndef NODE
#define NODE

//============================================================
//Please add headers here:


//============================================================
#include<RobotSDK.h>
//============================================================
//Port configuration

#undef NODE_CLASS
#define NODE_CLASS Node

#undef INPUT_PORT_NUM
#define INPUT_PORT_NUM 2

#undef OUTPUT_PORT_NUM
#define OUTPUT_PORT_NUM 3

//Uncomment below PORT_DECL and set input node class name
//PORT_DECL(0, InputNodeClassName)
//PORT_DECL(1, InputNodeClassName)

//============================================================
//Params types configuration

//If you need refer params type of other node class, please uncomment below and comment its own params type.
//NODE_PARAMS_TYPE_REF(RefNodeClassName)
class NODE_PARAMS_TYPE : public NODE_PARAMS_BASE_TYPE
{

};

//============================================================
//Vars types configuration

//If you need refer vars type of other node class, please uncomment below and comment its own vars type.
//NODE_VARS_TYPE_REF(RefNodeClassName)
class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{

};

//============================================================
//Data types configuration

//If you need refer data type of other node class, please uncomment below and comment its own data type.
//NODE_DATA_TYPE_REF(RefNodeClassName)
class NODE_DATA_TYPE : public NODE_DATA_BASE_TYPE
{

};

//============================================================
//You can declare functions here


//============================================================

#endif
```

Node.cpp

```
#include"Node.h"

//If you need use extended node, please uncomment below and comment the using of default node
//USE_EXTENDED_NODE(ExtendedNodeClass[,...])
USE_DEFAULT_NODE

//================================================
//Original node functions

//If you don't need initialize node, you can delete this code segment
NODE_FUNC_DEF_EXPORT(bool, initializeNode)
{
    return 1;
}

//If you don't need manually open node, you can delete this code segment
NODE_FUNC_DEF_EXPORT(bool, openNode)
{
    return 1;
}

//If you don't need manually close node, you can delete this code segment
NODE_FUNC_DEF_EXPORT(bool, closeNode)
{
    return 1;
}

//This is original main function, you must keep it
NODE_FUNC_DEF_EXPORT(bool, main)
{
    return 1;
}
```

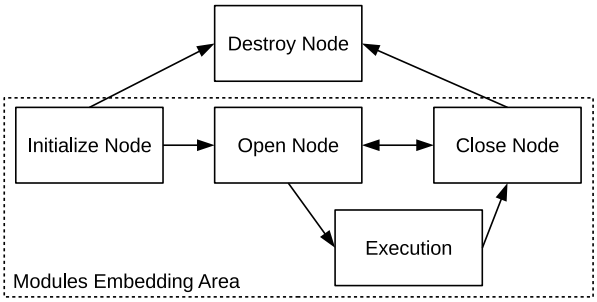

Figure 2.11: Contents of Interface Code and Their Correspondences to Node State

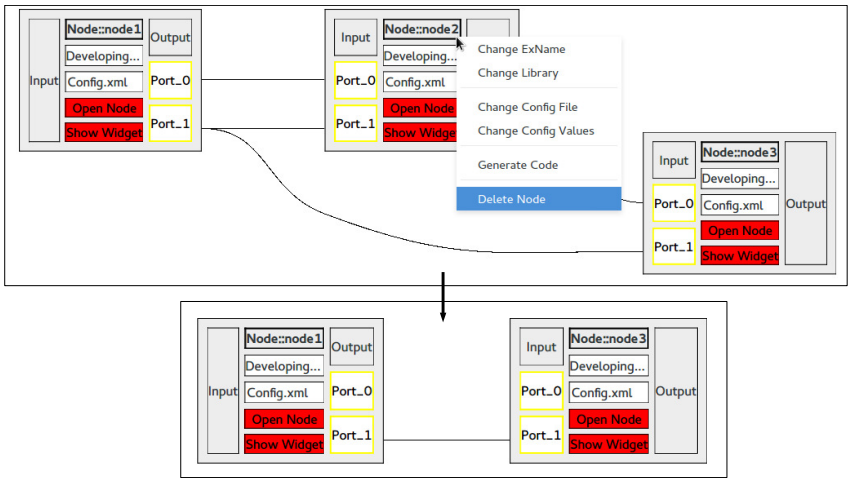

Figure 2.12: Delete Node from Abstract Graph Model

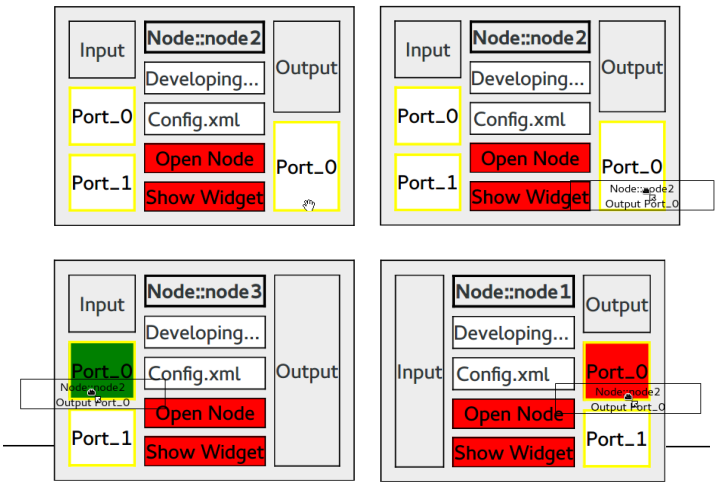

Figure 2.13: Drag-n-Drop Operation

2.1.2 Edge

2.1.2.1 Add an Edge

Edge in abstract graph model is to connect a pair of output port and input port in different nodes or in same node (loop, feedback). Robot-X provides a drag-n-drop operation for users to interactively add edge as shown in Fig.2.13. When mouse moves into a port, its cursor turns to open-hand shape (Fig.2.13 top-left). When press mouse left button, a port drag instance will be created as a transparent text box with port information (Fig.2.13 top-right). When move the port drag instance into a correct port (input $\leftrightarrow$ output), the latter one will turn to green, which means ready to be linked, and then an edge will be added if drop the former one (Fig.2.13 bottom-left). When move the port drag instance into a wrong port, the latter one will turn to red and no edge will be added (Fig.2.13 bottom-right). Fig.2.14 illustrates complete operations to add an edge between corresponding ports.

If an edge connects two real nodes, then the communication (signal-slot inter-thread connection) between these two ports will be automatically built. Otherwise (include semi-virtual node), the edge only appears in abstract graph model. Once a virtual or semi-virtual node turns to a real node by changing shared library, Robot-X will automatically check its related edges to build communications with other real nodes. Both edges in abstract graph model and communications will be kept after changing shared library or ”ExName” of real node.

2.1.2.2 Check an Edge

The organization of graph model is conducted by GraphViz, which tries its best to optimize edge-crossing. However, complex graph will indeed generate a mass of edges. Therefore, Robot-X provides a way to check edge’s information and path as shown in Fig.2.15. When mouse moves on an edge, its cursor turns to open-hand shape and a tooltip will appear to show edge’s information (Fig.2.15 top). When press mouse left button, the edge’s path will turn to red for clear identification (Fig.2.15 bottom).

2.1.2.3 Delete an Edge

To delete an edge, right click on the checked edge and a menu pop up as shown in Fig.2.16 top. After selecting ”Delete Edge”, the edge will be removed from abstract graph model as Fig.2.16 bottom. If the edge corresponds to a communication, the communication will be disconnected automatically.

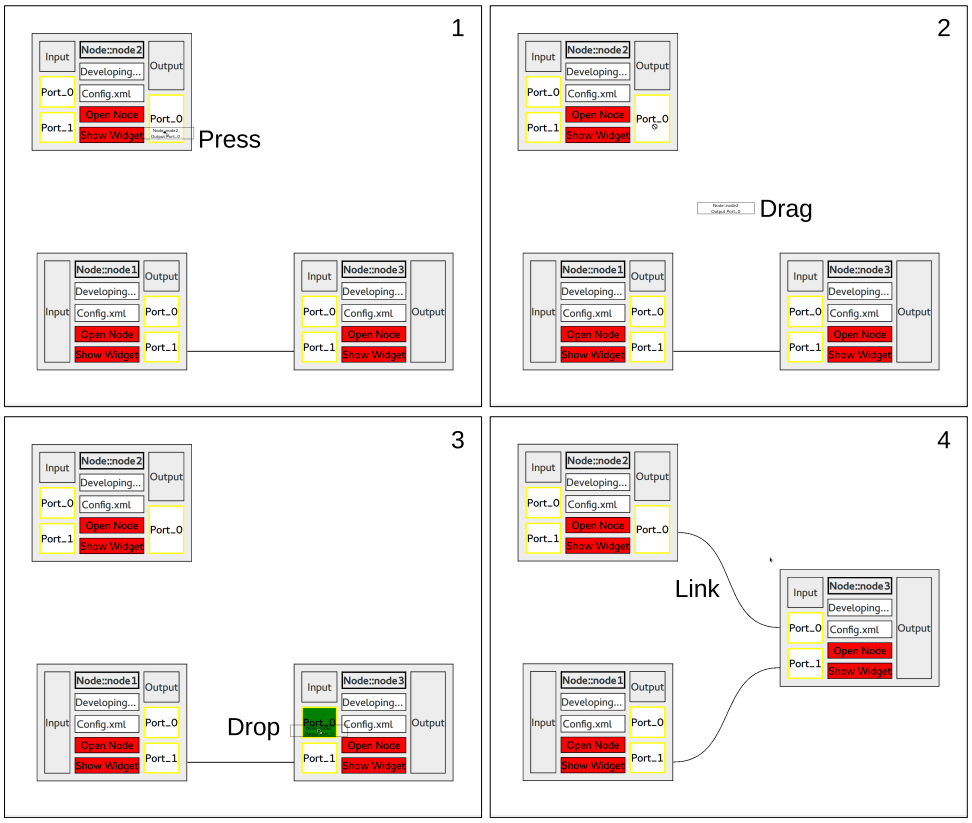

Figure 2.14: Complete Operations to Add an Edge

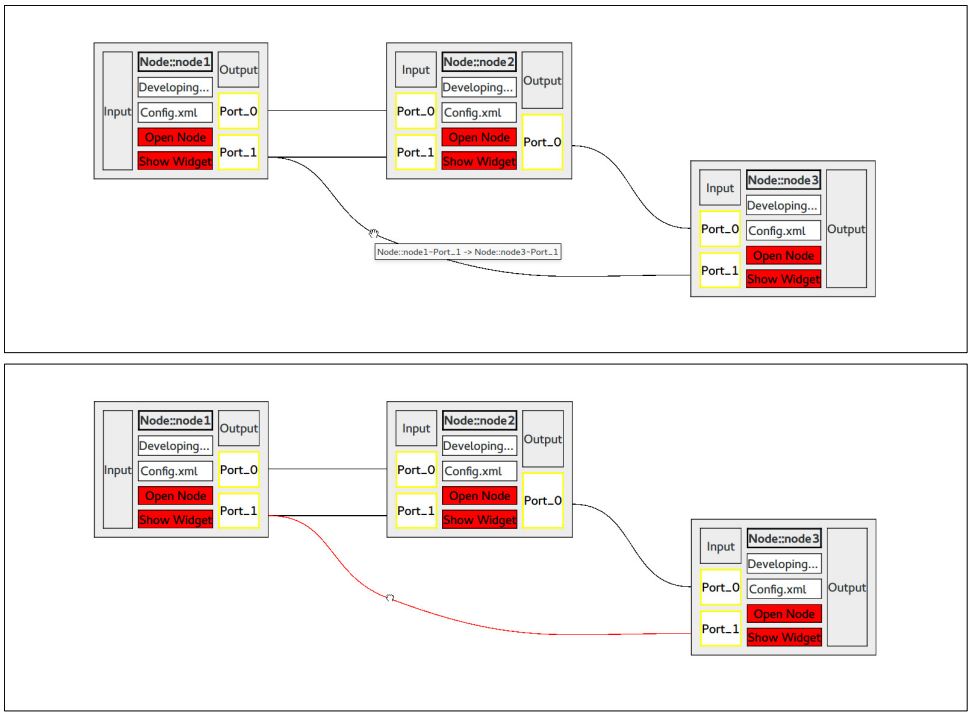

Figure 2.15: Check Edge's Information and Path

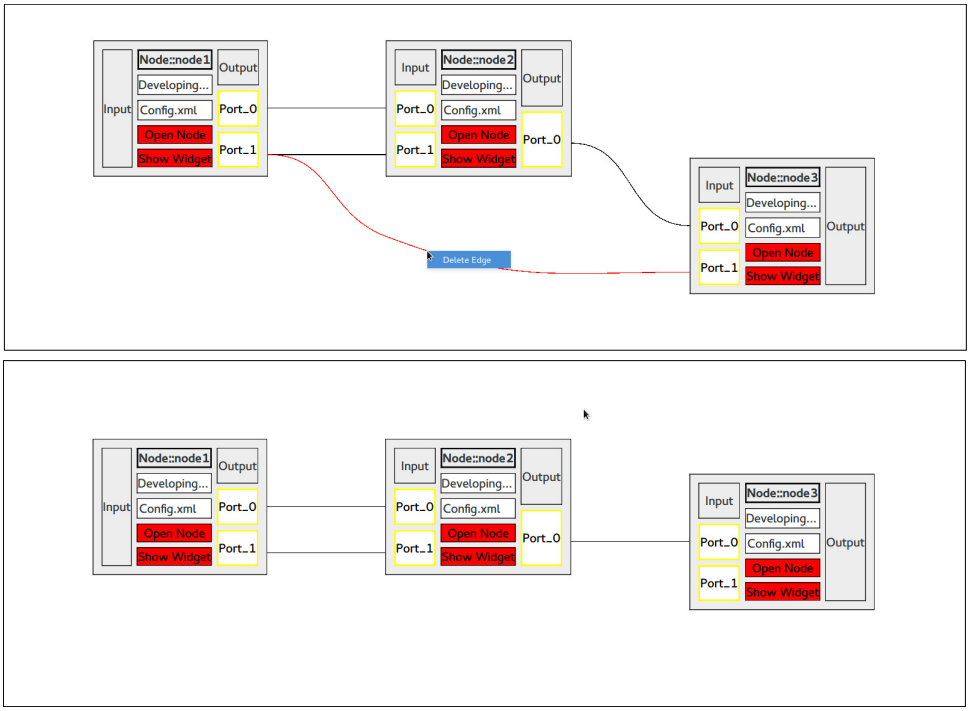

Figure 2.16: Delete Edge

2.1.3 Graph

2.1.3.1 Save and Load a Graph

When you have finished an abstract graph model, you hope to store it as a file for further development, sharing with collaborators or launching software quickly. Robot-X provides functions for saving and loading abstract graph model.

Right click on the empty part of Robot-X canvas and a menu pop up as shown in Fig.2.17 top-left. To save a graph, choose ”Save Graph” and then a file save dialog appears. The graph file’s suffix is .x and the contents are shown in Fig.2.17 bottom-left. The graph file contains two parts: 1) nodes description and 2) edges description. For node description, there are 5 values except for the first identifier ’N’: 1) Node Full Name, 2) Shared library (empty for virtual node), 3) Configuration file, 4) Input ports number and 5) Output ports number. For edge description, there are 4 values except for the first identifier ’E’: 1) Output Node Full Name, 2) Output port ID, 3) Input Node Full Name and 4) Input port ID.

To load a graph, choose ”Load Graph” and then a file load dialog appears. You can load any numbers of graph files into Robot-X and a graph union operation will be performed after loading as shown in Fig.2.18. This could be used to merge many sub-systems into one system. RobotSDK 4.0 uses constant shared_ptr to protect output data, therefore, nodes with same Node Full Name in different sub-systems could be safely merged as one node.

2.1.3.2 Export Graph Image and Dot File

Sometimes, it is desirable to output standard image of abstract graph model for presentation of publishing. Robot-X uses GraphViz to render .png image file as shown in Fig.2.19. To export graph image file, right click on empty part of Robot-X canvas and a menu pop up as shown in Fig.2.20 left. Choose ”Export Graph Image...” and a save file dialog appears. After saving, a image file is stored as Fig.2.20 bottom-right.

Dot file is a popular graph file type and most graph-related applications could read it. For example, here is a website [`http://graphs.grevian.org/graph`] which reads dot file contents and draw the graph as shown in Fig.2.21. To export dot file, right click on empty part of Robot-X canvas and a menu pup up as shown in Fig.2.22 top-left. Choose ”Export Dot File...” and a save file dialog appears. After saving, a dot file is stored and its contents are shown in Fig.2.22 right.

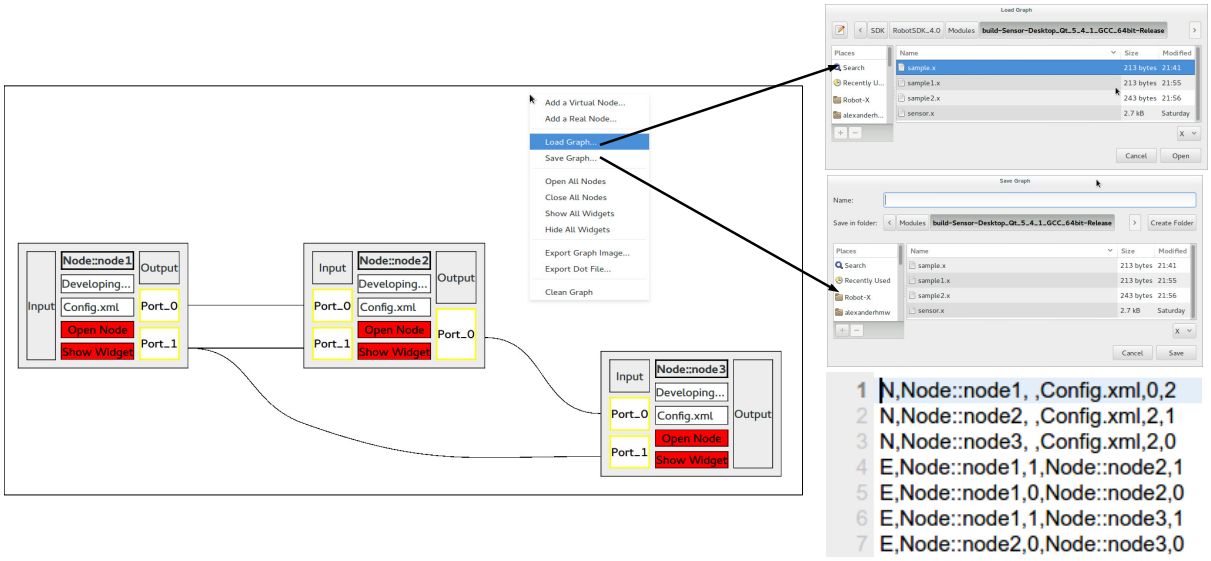

Figure 2.17: Save and Load a Graph. The Contents of Graph File

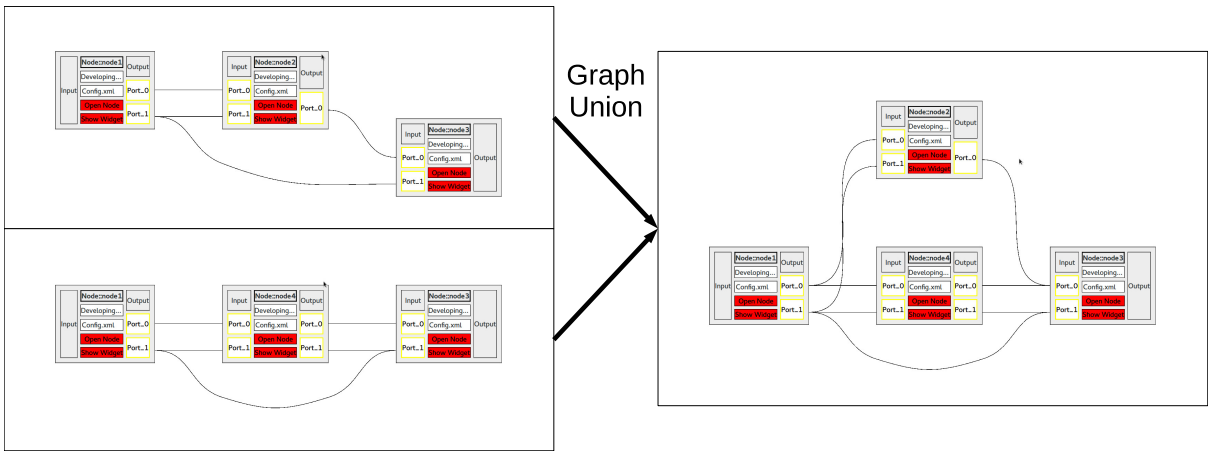

Figure 2.18: Graph Union by Loading Two Graph Files into Same Graph Model

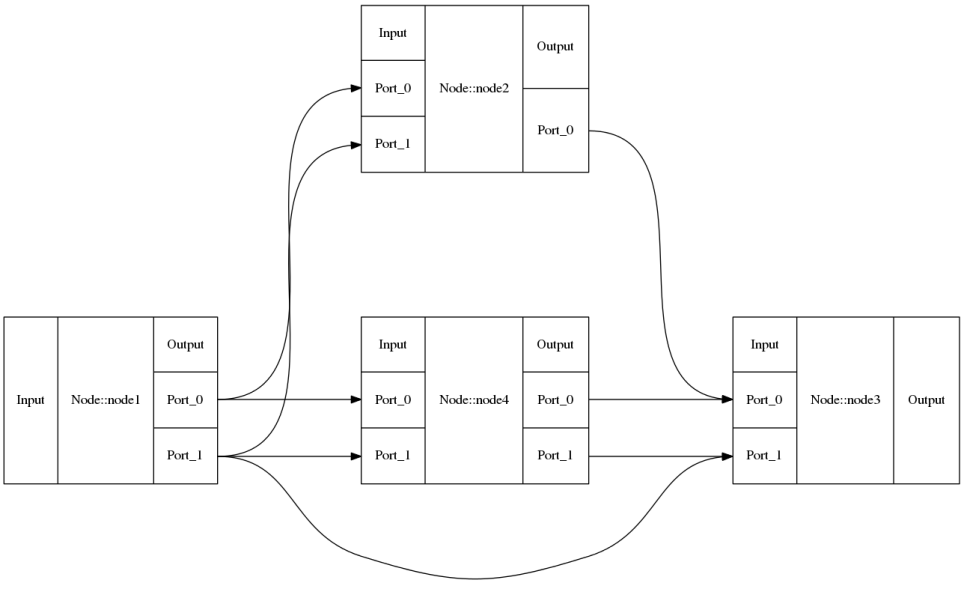

Figure 2.19: Image File of Graph Union Example

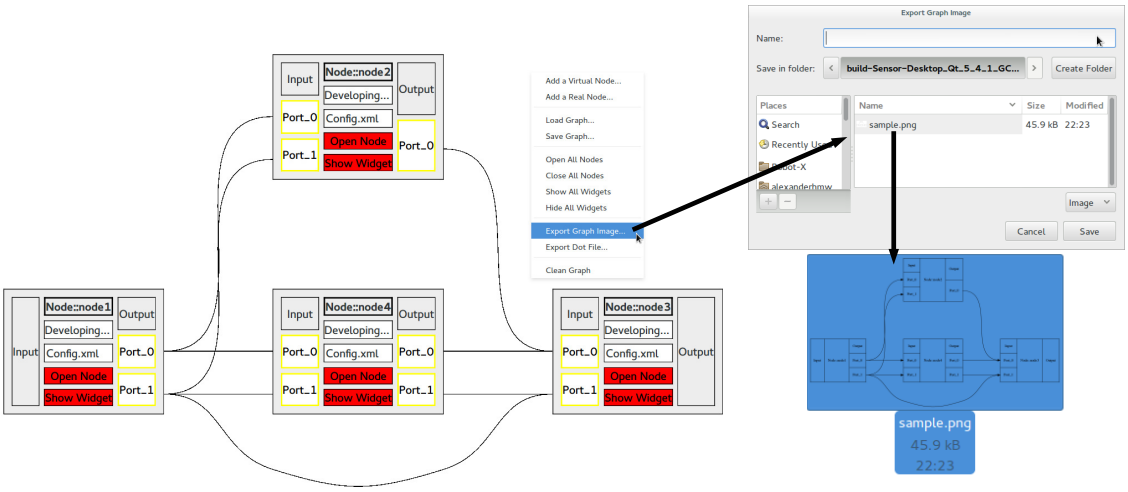

Figure 2.20: Export Graph Image of Abstract Graph Model

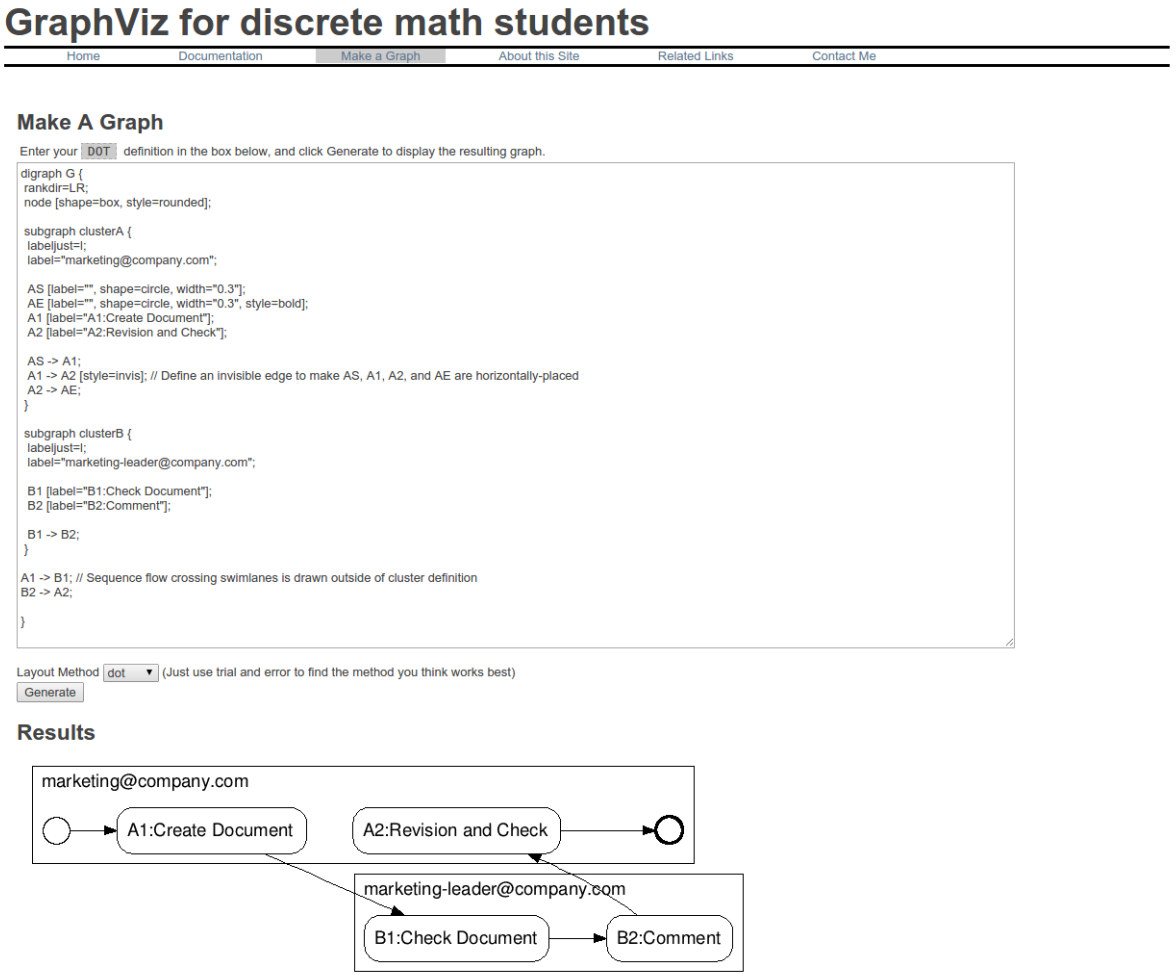

Figure 2.21: Application Uses Dot File for Graph Generation

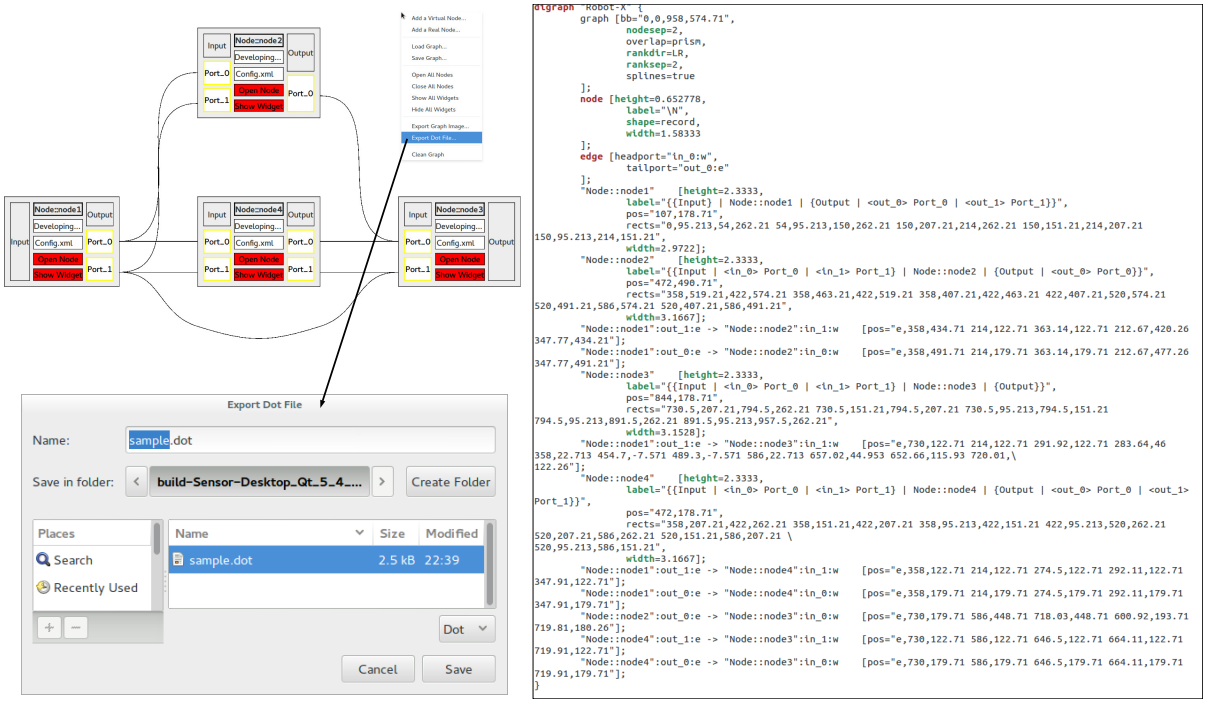

Figure 2.22: Export Dot File of Abstract Graph Model and Its Contents

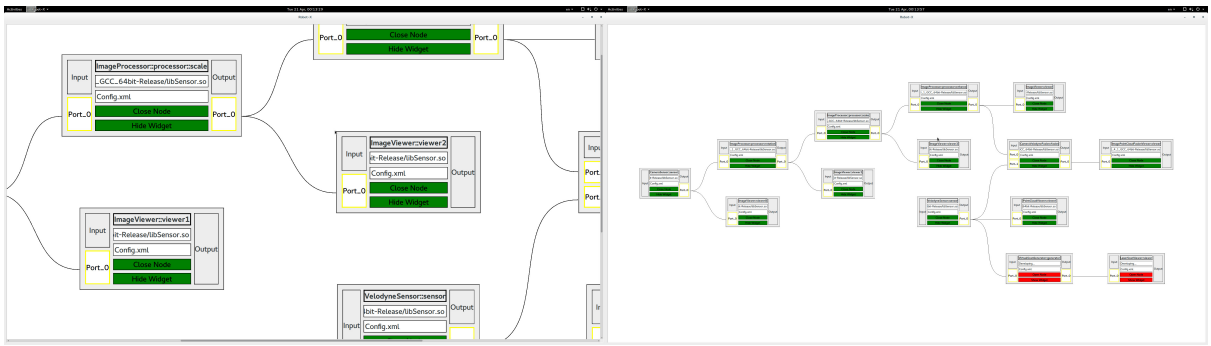

Figure 2.23: Zoom in and out of a Graph Model

2.1.3.3 Zoom in and out

If a graph is too large, the view port of Robot-X canvas will not have enough space to show entire graph. Robot-X provides a zoom in/out operation by using Ctrl + mouse wheel as shown in Fig.2.23.

2.1.3.4 Clear Graph

To clear graph, right click on empty part of Robot-X canvas and a menu pup up as shown in Fig.2.24 left. After choosing ”Clear Graph” all nodes and edges will be removed from graph model as well as functional modules and their communications as Fig.2.24 right.

2.2 Use Robot-X to Launch and Control Software

After top abstract graph model construction and down functional modules development. Robot-X can launch software according to the concreted graph model (virtual nodes turn to real nodes), and fully control the software. Normally, the abstract graph model has been saved as graph file and it is not necessary to convert all virtual nodes to real nodes (sub-system of concreted sub-graph model could be launched as well).

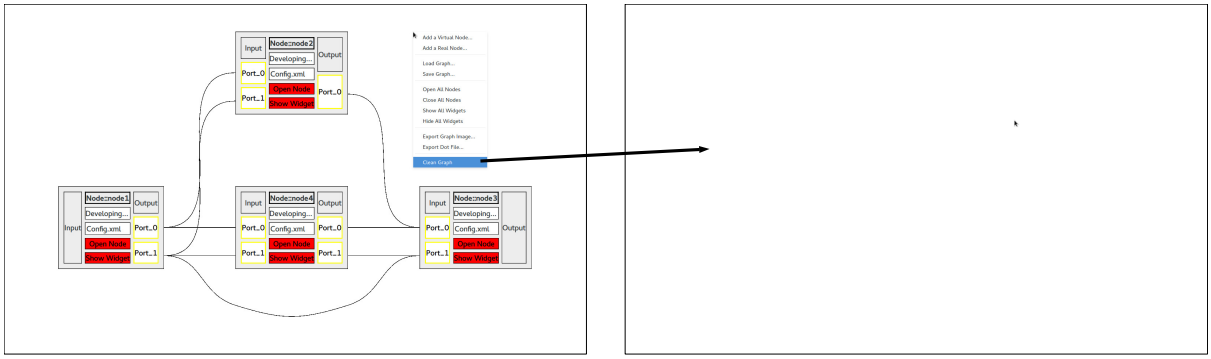

Figure 2.24: Clear an Abstract Graph Model

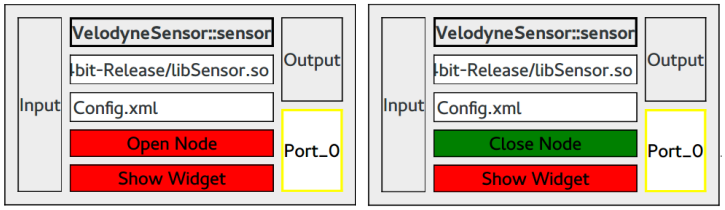

Figure 2.25: Open and Close a Node

2.2.1 Open and Close Nodes

You can use node's open/close button to control it as shown in Fig.2.25. The red color means the node is closed and you can open node by pushing this button. The green color means the node is open and you can close node by pushing this button. If a node is failed to open, the color will still be red and vise versa. Therefore, this could be regarded as an open/close status indicator of node.

Another convenient way is to use graph menu as shown in Fig.2.26 top. Right click on empty part of Robot-X canvas, choose "Open All Nodes", then Robot-X will try to open all nodes. If successfully open a node, its open/close button will turn to green and if failed, its open/close button will keep in red as shown in Fig.2.26 bottom. There are two virtual nodes in this example, therefore, after opening all nodes operation, they are still in red color. For closing all nodes operation, it is same with that of opening all nodes.

2.2.2 Show and Hide Widgets

Widget is a GUI to show something on the screen. RobotSDK 4.0 provides a uniform way to program such widget (see Chapter 3). Similar to open/close nodes, to show/hide widgets could be conducted by pushing each node's show/hide widget button as shown in Fig.2.27 and also by using graph menu as shown in Fig.2.28. The widgets of this example is shown as Fig.2.29.

There is one thing need to be noticed. If a node has widget and the node also wants to control this widget, then it must be in the GUI thread, which is also the thread of application (For example, Robot-X). For a non-GUI node without showing widget, if you push show widget button, it will also turn to green, however, no widget will appear. Because unlike open/close operation, show/hide operation does not check widget status but only sends a command. Therefore, if you manually close a widget by pressing its top-right cross button, the show/hide button will not turn to red. Fortunately, this will not cause crash and you can still use show widget button to show the closed widget again.

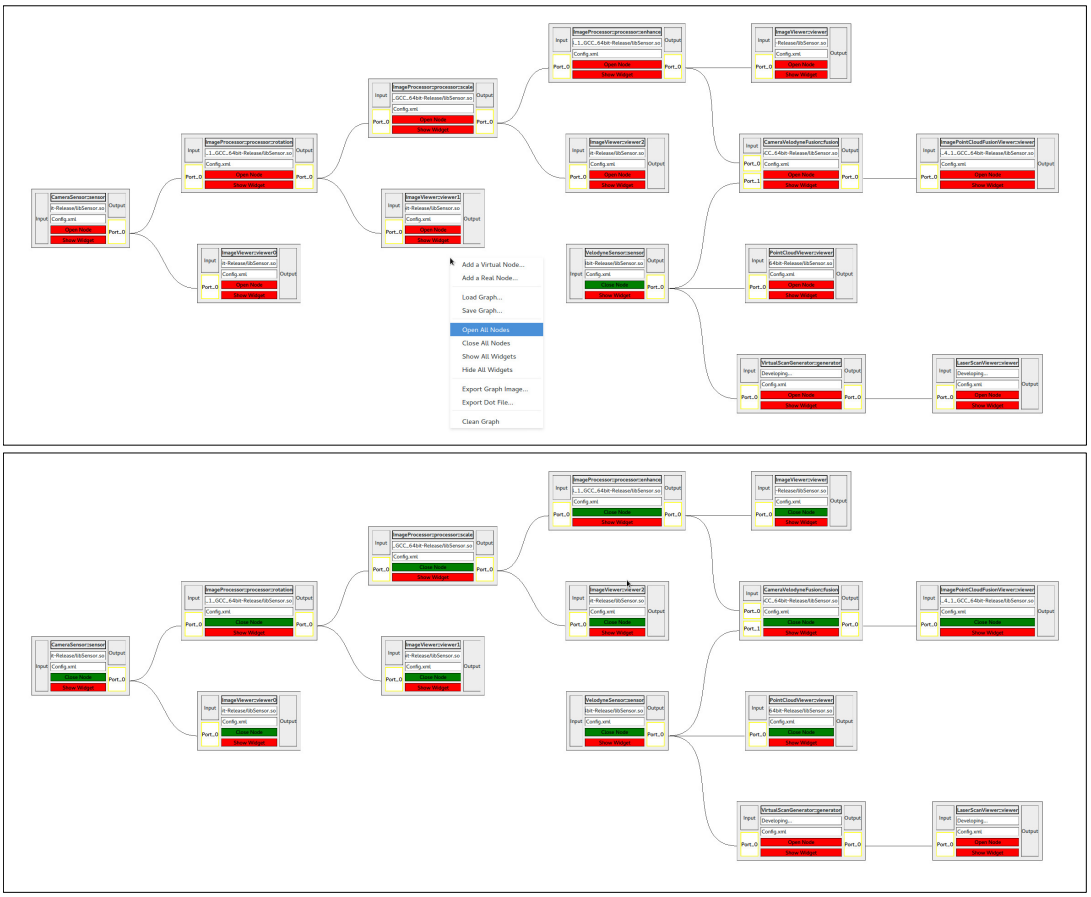

Figure 2.26: Open All Nodes

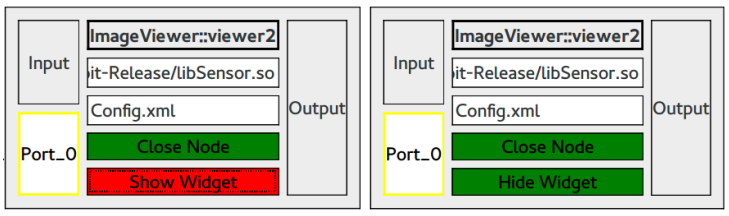

Figure 2.27: Show and Hide Widget

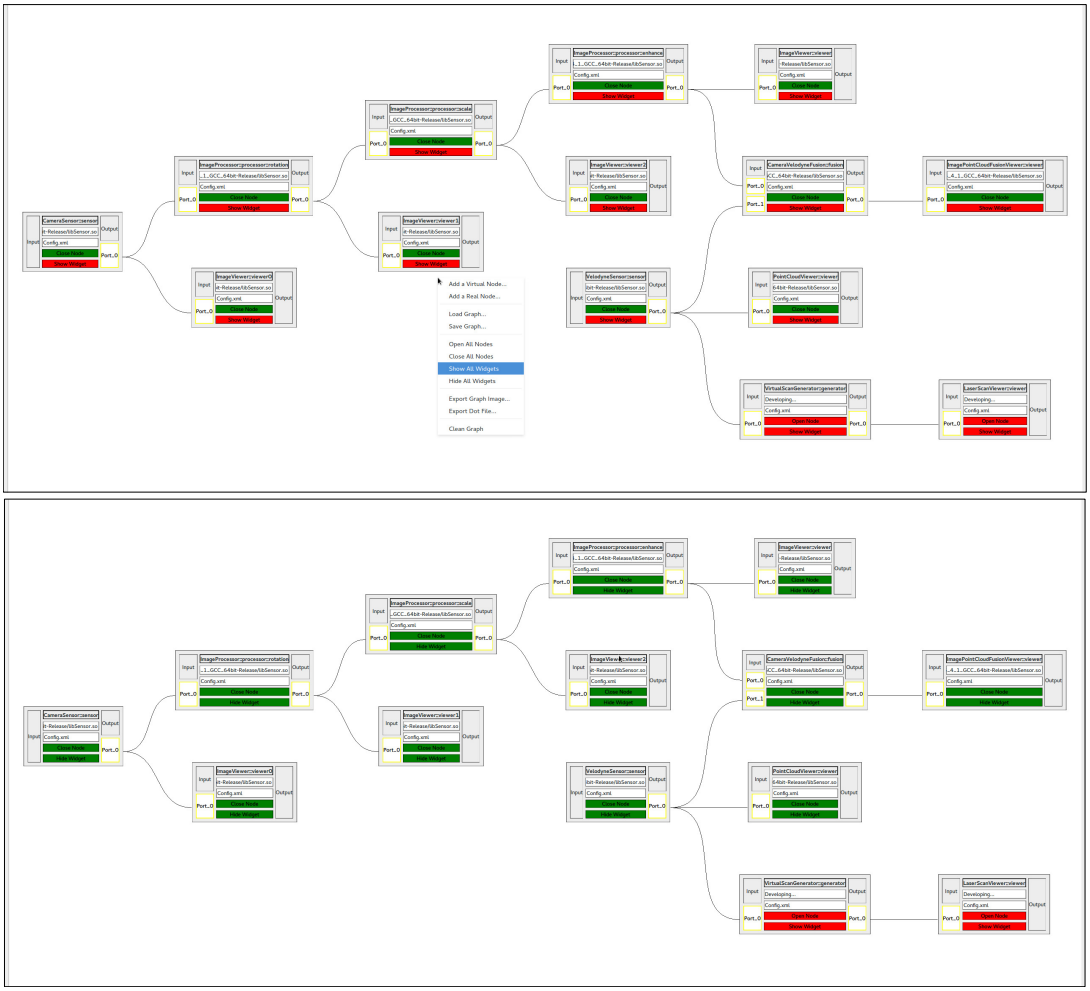

Figure 2.28: Show All Widgets

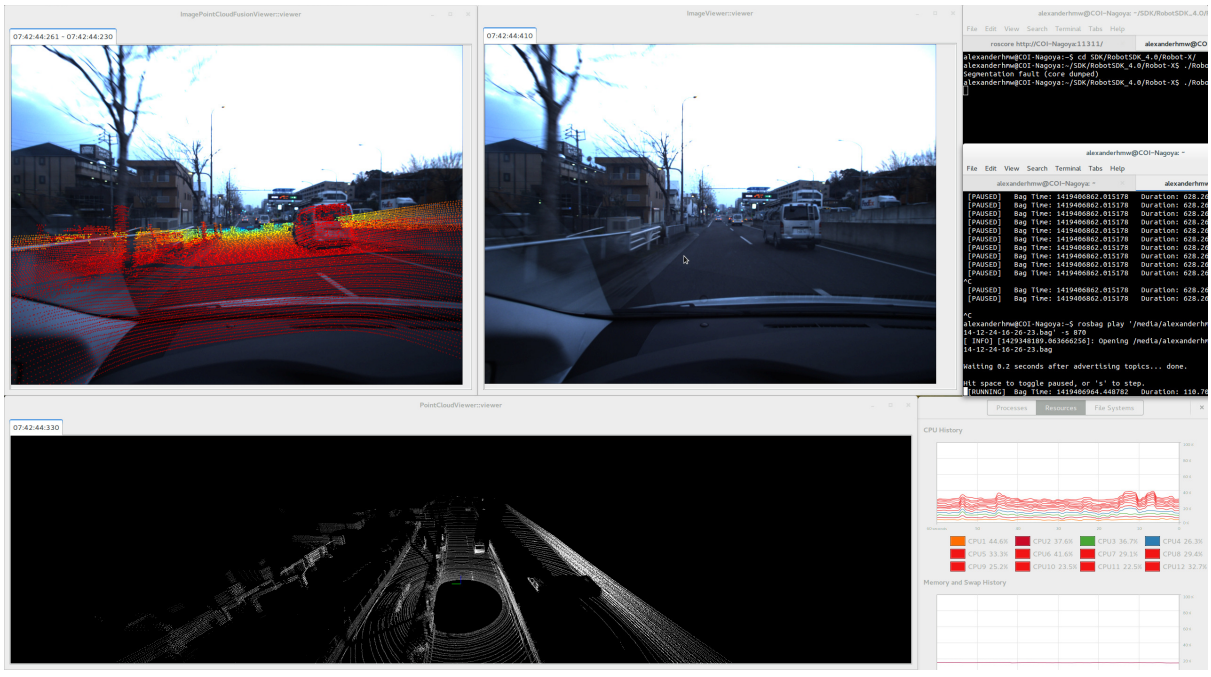

Figure 2.29: Widgets of the Example: Fusion of Camera and Velodyne

Chapter 3

Functional Module Development: Free C Style Programming

Robot-X is a super high level application, therefore, from RobotSDK 4.0, users need not to develop high level applications and could focus on functional modules development. In this chapter, we will present basic but enough instructions for common functional module development. We strongly recommend developers to use QtCreator as IDE for functional module development, however, we will still briefly introduce how to use Visual Studio for such development in Windows just after presenting that with QtCreator.

3.1 Project Configuration

3.1.1 Create Shared Library Project

The functional modules are packed in shared library, therefore, we should first build a shared library project. After starting QtCreator, click ”New Project”, and then choose ”Library” and ”C++ Library” as Fig.3.1. Then a series of configuration dialog will appear as Fig.3.2 and you only need to to make sure that the ”Library Type” should be ”Shard Library” in the first dialog. Then you will get a ”shared library project” as Fig.3.2 bottom-right. You can freely delete the auto-generated .h and .cpp files as you wish, because they are useless and will not affect functional modules.

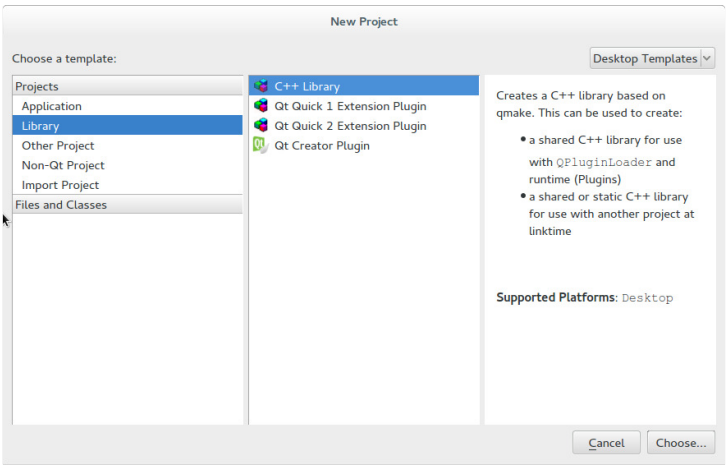

Figure 3.1: Create a C++ Library Project

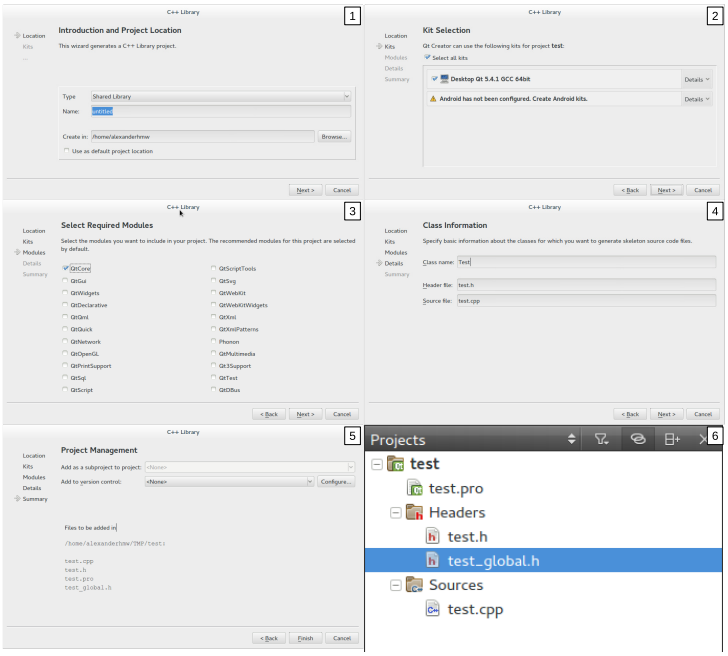

Figure 3.2: Configure a C++ Library Project

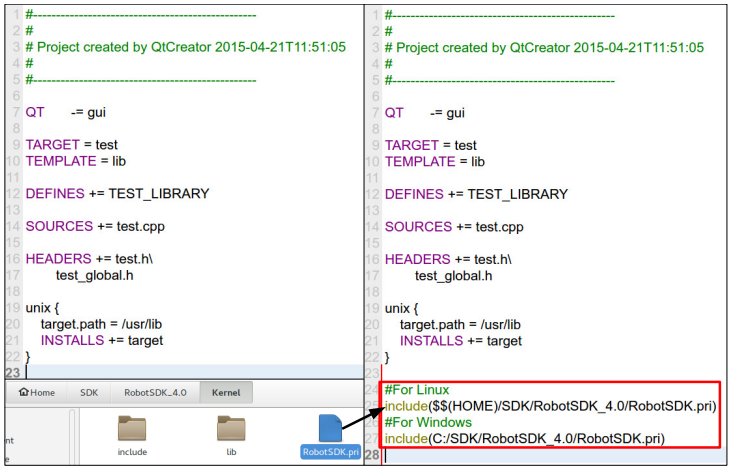

Figure 3.3: Configure a C++ Library Project

3.1.2 Configure Project with RobotSDK

After creating shared library project, we need to configure this project with RobotSDK. Open the .pro file in the project panel as Fig.3.2 bottom-right and we will see its contents as Fig.3.3 top-left. The only thing you need to do is to include ”RobotSDK.pri” file as Fig.3.3 right (you need to choose one according to your OS). The configuration is done.

3.1.3 For Visual Studio

For Visual Studio, you need to create a dynamic link library (dll) project via Qt Visual Studio Add-in. Then you can regard RobotSDK as a library and add its include path and static libraries ”Kernel_Debug.lib” or ”Kernel_Release.lib” (Fig.3.3 bottom-left) to the VS project. You also need to add Eigen include path if you want to use GLViewer or Kernel is already compiled with GLViewer.

Figure 3.4: Add Required Headers

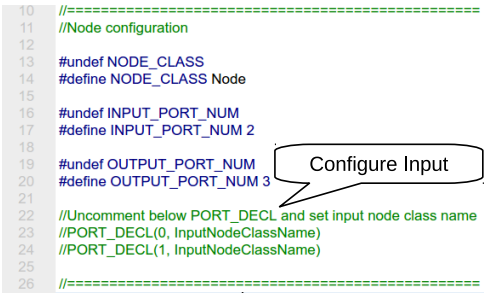

Figure 3.5: Configure Node

3.2 Basic Steps to Program Functional Module

From Chapter 2, we know that the Robot-X will generate two files for each functional module. Then you need to add them to the created shared library project. In this section we will separately introduce how to program these files.

3.2.1 Header File

3.2.1.1 Step 1: Add Required Headers [C Style]

Just like C programming, you need to refer other headers for programming as Fig.3.4. Normally, you need to add other module's header as required input node and library headers, like "OpenCV", "PCL", etc. Here we stress the position of headers as below.

> Included headers must be in front of Node Configuration (see following section)

3.2.1.2 Step 2: Configure Node

As shown in Fig.3.5, The "Node Class", "Input Port Number" and "Output Port Number" have already been automatically set. If the node has input ports, a segment of commented codes will appear as Fig.3.5 bottom. You need to uncomment these codes and fill in the "Input Node Class Name". If you forget this, you cannot access input "Params" and "Data" via RobotSDK.

3.2.1.3 Step 3: Add Elements to "Params", "Vars" and "Data"

Similar to RobotSDK 3.0, a node contains a public "Params" and a private "Vars", and outputs "Data" frame by frame. You still need to add elements for them. However, thanks to C++11 techniques listed below, a set of syntax is created for convenient adding and managing elements. We will present how to add elements in section 3.3.

- Non-static data members initializers
  `http://en.cppreference.com/w/cpp/language/data_members`
- Lambda function
  `http://en.cppreference.com/w/cpp/language/lambda`

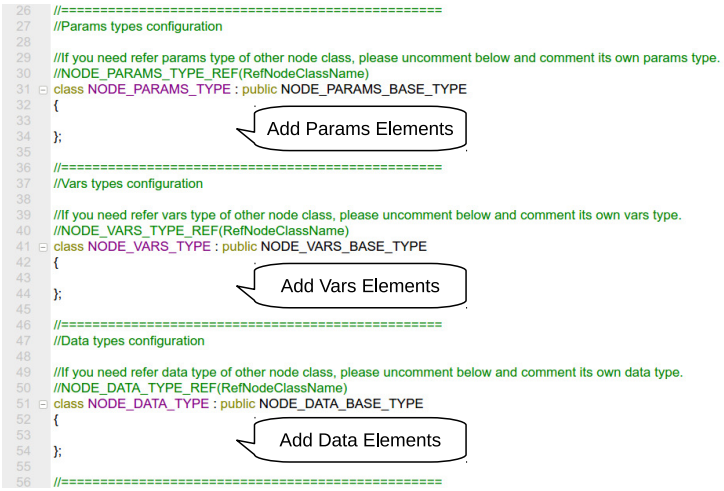

Figure 3.6: Add Elements to ”Params”, ”Vars” and ”Data”

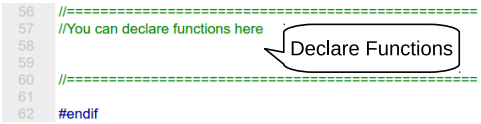

Figure 3.7: Declare Functions

- Variadic Macro
  `http://en.wikipedia.org/wiki/Variadic_macro`

3.2.1.4 Step 4: Declare functions [C Style]

Just like C programming, sometimes, you need to declare functions first in the header file as in Fig.3.7. There are two types of declaration: 1) original C style function declaration and 2) wrapped function declaration by RobotSDK syntax. The only difference between them is the former one cannot directly access node's ”Params”, ”Vars”, ”Data” and input nodes' ”Params”, ”Data” (**Ordinary Function**), and the latter one can directly access them via RobotSDK syntax (**Node Function**). We will present how to declare such function in section 3.3.

3.2.2 Source File

3.2.2.1 Step 1: Choose Node

RobotSDK 4.0 uniformed all node types used in RobotSDK 3.0 and this enables Robot-X to be a super high level application. However, it is still free for developers to extend the default node and use extended node in module programming. As shown in Fig.3.8, the default node is used automatically. If you want to use extended node, you just need to replace the ”USE_DEFAULT_NODE” with ”USE_EXTENDED_NODE(ExtendedNodeClass[,...])”. We will discuss the extended node in Chapter 4.

> Here, the ”NodeClass” is different with the module's ”Node Class”. The former one is a real class inherited from ”Node” class in Kernel and the latter one is only an identifier for the module.

3.2.2.2 Step 2: Functions Programming [C Style]

Now, you are free to program module's functions. From section 3.2.1.4, we know there are two types of function: 1) Ordinary Function and 2) Node Function. Here we stress them again and show details of Node Function in section 3.3.

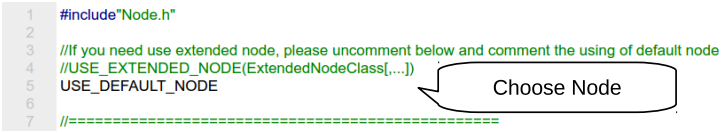

Figure 3.8: Choose Node

Figure 3.9: Original Functions

- **Ordinary Function**: original C style function which cannot directly access node's "Params", "Vars", "Data" and input nodes' "Params", "Data"
- **Node Function**: wrapped function by RobotSDK syntax which can directly access them

As shown in Fig.3.9, there are 4 default Node Functions: "initializeNode", "openNode", "closeNode" and "main" and their return value type are boolean. From the comments, only the "main" Node Function is mandatory and similar to C programming, it is the main entry of the module (you should not replace it with Ordinary Function). However, it is not the only main entry for the module, because, you can freely add main entries by extending default node (see Chapter 4). The correspondences between these four default functions and node's state is shown as Fig.2.11.

You can define your own Ordinary Functions and Node Functions as you want. But here we must stress the function call rule for them below and we will discuss the conversion between Ordinary Function and Node Function in section 3.3.

- Ordinary Function can call Ordinary Function via original C way.
- Ordinary Function cannot call Node Function.
- Node Function can call Ordinary Function via original C way.
- Node Function can call Node Function via RobotSDK syntax.

Another thing is about the extension of module. As Fig.3.10, we create a node with extension and generate its code as Fig.3.11. You can find that Robot-X appends the extension code to the source file. Therefore, we say that module extension shares the node configuration, value types and some functions with the base module. If you only generate code for module extension, Robot-X will first generate code of base module and then append code of extension. From the comments, all the 4 default functions are

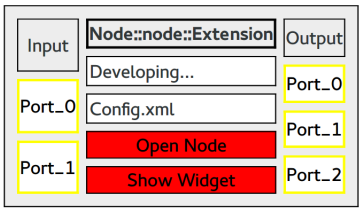

Figure 3.10: Node with Extension

```
33  //================================================
34  //Extended node functions ( Extension )
35
36  //If you don't need initialize node, you can delete this code segment
37  NODE_EXFUNC_DEF_EXPORT(bool, initializeNode, Extension)
38  {
39      return 1;
40  }
41
42  //If you don't need manually open node, you can delete this code segment
43  NODE_EXFUNC_DEF_EXPORT(bool, openNode, Extension)
44  {
45      return 1;
46  }
47
48  //If you don't need manually close node, you can delete this code segment
49  NODE_EXFUNC_DEF_EXPORT(bool, closeNode, Extension)
50  {
51      return 1;
52  }
53
54  //As an extended main function, if you delete this code segment, original main function will be used
55  NODE_EXFUNC_DEF_EXPORT(bool, main, Extension)
56  {
57      return 1;
58  }
```

Extended Functions Programming

Figure 3.11: Extended Functions

not mandatory. Here we need to inform users a controversial rule as below.

During node construction, if the node requires an extended Node Function, the Kernel will first locates it with ”Node Class” and ”ExName”. If not found, the Kernel will try to locates its original Node Function with ”Node Class” for instead. If not found again, the node construction failed.

To develop a functional module, you only need to:

- remember the basic steps above
- use RobotSDK syntax below for programming

Now, we will show the most important part of this manual: RobotSDK Syntax. For convenience, we will start a new page here. We encourage users to print section 3.3 out as handbook for functional module development.

3.3 RobotSDK Syntax

3.3.1 Configure Node

```
PORT_DECL(portID, inputNodeClass)
```

- Declare an input node with *inputNodeClass* for *#portID* input port
- Arguments:
  - portID: port id, must be a constant number
  - inputNodeClass: ”Node Class” of input node
- Example:

```
//================================================
//Please add headers here:
#include"CameraSensor.h"
#include"VelodyneSensor.h"
#include<opencv2/opencv.hpp>
//================================================
#include<RobotSDK.h>
//================================================
//Node configuration

#undef NODE_CLASS
#define NODE_CLASS CameraVelodyneFusion

#undef INPUT_PORT_NUM
#define INPUT_PORT_NUM 2

#undef OUTPUT_PORT_NUM
#define OUTPUT_PORT_NUM 1

//Uncomment below PORT_DECL and set input node class name
PORT_DECL(0, CameraSensor)
PORT_DECL(1, VelodyneSensor)
```

3.3.2 Configure ”Params”, ”Vars” and ”Data”

3.3.2.1 ”Params”

```
NODE_PARAMS_TYPE
NODE_PARAMS_BASE_TYPE
```

- ”Params” class name and ”Params” base class name.
- Base class must be publicly inherited and ”Params” can multi-inherit other class.
- Example:

```
class NODE_PARAMS_TYPE : public NODE_PARAMS_BASE_TYPE
{

};
```

```
NODE_PARAMS_TYPE_REF(RefNodeClassName)
```

- Refer to *RefNodeClassName*'s "Params" as its own "Params"
- Arguments:
  - RefNodeClassName: "Node Class" of referred node.
- Example:

```
NODE_PARAMS_TYPE_REF(CameraSensor)
```

```
ADD_PARAM(valueType, valueName, valueDefault)
ADD_ENUM_PARAM(valueType, valueName, valueDefault)
ADD_UENUM_PARAM(valueType, valueName, valueDefault)
```

- Add and register variable *valueName* in type *valueType* with default value *valueDefault*.
- The registered variable will be automatically managed by RobotSDK
- You can *change* its value via Robot-X Config Panel.
- Arguments:
  - valueType: type of the variable
  - valueName: name of the variable
  - valueDefault: default value of the variable
- Example:

```
class NODE_PARAMS_TYPE : public NODE_PARAMS_BASE_TYPE
{
public:
    ADD_PARAM(QString, calibfilename, "#(CameraCalibFileName)")
};
```

- Note:
  It is not recommended to add an uninitialized variable in normal C++ way, because you cannot assign value to it in Node Function ("Params" is protected by const shared_ptr).

```
ADD_PARAM_WITH_OPTIONS(valueType, valueName, valueDefault, valueOptions)
ADD_ENUM_PARAM_WITH_OPTIONS(valueType, valueName, valueDefault, valueOptions)
ADD_UENUM_PARAM_WITH_OPTIONS(valueType, valueName, valueDefault, valueOptions)
```

- Add and register variable *valueName* in type *valueType* with default value *valueDefault* and options *valueOptions*.
- The registered variable will be automatically managed by RobotSDK
- You can *choose* its value via Robot-X Config Panel.
- Arguments:
  - valueType: type of the variable
  - valueName: name of the variable
  - valueDefault: default value of the variable
  - valueOptions: options of the variable

- Example:

```
class NODE_PARAMS_TYPE : public NODE_PARAMS_BASE_TYPE
{
public:
    ADD_PARAM_WITH_OPTIONS(double, angle, 0
                           , QList<double>()<<0<<90<<180<<270)
};
```

3.3.2.2 "Vars"

```
NODE_VARS_TYPE
NODE_VARS_BASE_TYPE
```

- "Vars" class name and "Vars" base class name.
- Base class must be publicly inherited and "Vars" can multi-inherit other class.
- Example:

```
class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{

};
```

```
NODE_VARS_TYPE_REF(RefNodeClassName)
```

- Refer to *RefNodeClassName*'s "Vars" as its own "Vars"
- Arguments:
  - RefNodeClassName: "Node Class" of referred node.
- Example:

```
NODE_VARS_TYPE_REF(ImageViewer)
```

```
ADD_VAR(valueType, valueName, valueDefault)
ADD_ENUM_VAR(valueType, valueName, valueDefault)
ADD_UENUM_VAR(valueType, valueName, valueDefault)
```

- Add and register variable *valueName* in type *valueType* with default value *valueDefault*
- Arguments:
  - valueType: type of the variable
  - valueName: name of the variable
  - valueDefault: default value of the variable
- Example:

```
class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{
public:
    cv::Mat extrinsicmat;
    cv::Mat cameramat;
```

```
    cv::Mat distcoeff;
public:
    ADD_VAR(QString, topic, "/image_raw")
    ADD_VAR(u_int32_t, queuesize, 1000)
    ADD_VAR(int, queryinterval, 10)
public:
    typedef ROSSub<sensor_msgs::ImageConstPtr> rossub;
    ADD_INTERNAL_QOBJECT_TRIGGER(rossub, camerasub, 1
                                 , topic, queuesize, queryinterval)
    ADD_INTERNAL_DEFAULT_CONNECTION(camerasub, receiveMessageSignal)
};
```

- Note:
  It is recommended to add variable in non-standard type (no conversion between QString) in normal C++ way, because XML value manager does not know how to convert it. You can assign value to it in Node Function, because unlike ”Params”, ”Vars” is not protected.

```
ADD_VAR_WITH_OPTIONS(valueType, valueName, valueDefault, valueOptions)
ADD_ENUM_VAR_WITH_OPTIONS(valueType, valueName, valueDefault, valueOptions)
ADD_UENUM_VAR_WITH_OPTIONS(valueType, valueName, valueDefault, valueOptions)
```

- Add and register variable *valueName* in type *valueType* with default value *valueDefault* and options *valueOptions*.
- The registered variable will be automatically managed by RobotSDK
- You can *choose* its value via Robot-X Config Panel.
- Arguments:
  - valueType: type of the variable
  - valueName: name of the variable
  - valueDefault: default value of the variable
  - valueOptions: options of the variable
- Example:

```
class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{
public:
    ADD_VAR_WITH_OPTIONS(QString, topic, "/image_raw"
      , QList<QString>()<<"/image_raw"<<"/image_color"<<"/image_gray")
};
```

```
ADD_INTERNAL_QOBJECT_TRIGGER(triggerType, triggerName, poolThreadFlag, ...)
```

- Add an internal QObject trigger *triggerName* in type *triggerType*.
- Arguments:
  - triggerType: type of the internal trigger, must be derived from QObject
  - triggerName: name of the internal trigger
  - poolThreadFlag:
    - =1, trigger will goto QObjectPoolThread and use inter-thread communication
    - =0, trigger will stay with node and use intra-thread communication

  - ...: [Variadic arguments], used for arguments of trigger's construction function.

- Example:

```
class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{
public:
    ADD_INTERNAL_QOBJECT_TRIGGER(QTimer, timer, 0)
    ADD_INTERNAL_DEFAULT_CONNECTION(timer, timeout)
};

class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{
public:
    cv::Mat extrinsicmat;
    cv::Mat cameramat;
    cv::Mat distcoeff;
public:
    ADD_VAR(QString, topic, "/image_raw")
    ADD_VAR(u_int32_t, queuesize, 1000)
    ADD_VAR(int, queryinterval, 10)
public:
    typedef ROSSub<sensor_msgs::ImageConstPtr> rossub;
    ADD_INTERNAL_QOBJECT_TRIGGER(rossub, camerasub, 1
                                 , topic, queuesize, queryinterval)
    ADD_INTERNAL_DEFAULT_CONNECTION(camerasub, receiveMessageSignal)
};
```

`ADD_INTERNAL_QWIDGET_TRIGGER(triggerType, triggerName, ...)`

- Add an internal QWidget trigger *triggerName* in type *triggerType*.
- QWidget trigger must stay in GUI thread (application thread)
- Arguments:
  - triggerType: type of the internal trigger, must be derived from QObject
  - triggerName: name of the internal trigger
  - ...: [Variadic arguments], used for arguments of trigger's construction function.
- Example:

```
class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{
public:
    ADD_INTERNAL_QWIDGET_TRIGGER(QPushButton, trigger, "Trigger")
    ADD_INTERNAL_DEFAULT_CONNECTION(trigger, clicked)
};
```

`ADD_INTERNAL_DEFAULT_CONNECTION(triggerName,signalName)`

- Add an internal connection from *triggerName*'s signal *signalName* to node's slot "slotDefault-Trigger"
- "slotDefaultTrigger" of default node will call main Node Function and output "Data" if return value is true.

- ”slotDefaultTrigger” is a virtual slot, user can re-implement it to change its behavior (node extension).
- Arguments:
  - triggerName: name of the internal trigger
  - signalName: name of the signal function
- Example:

```
class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{
public:
    cv::Mat extrinsicmat;
    cv::Mat cameramat;
    cv::Mat distcoeff;
public:
    ADD_VAR(QString, topic, "/image_raw")
    ADD_VAR(u_int32_t, queuesize, 1000)
    ADD_VAR(int, queryinterval, 10)
public:
    typedef ROSSub<sensor_msgs::ImageConstPtr> rossub;
    ADD_INTERNAL_QOBJECT_TRIGGER(rossub, camerasub, 1
                                 , topic, queuesize, queryinterval)
    ADD_INTERNAL_DEFAULT_CONNECTION(camerasub, receiveMessageSignal)
};

class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{
public:
    ADD_INTERNAL_QWIDGET_TRIGGER(QPushButton, trigger, "Trigger")
    ADD_INTERNAL_DEFAULT_CONNECTION(trigger, clicked)
};
```

`ADD_INTERNAL_USER_CONNECTION(triggerName,signalName,slotName,...)`

- Add an internal connection from *triggerName*’s signal *signalName* to node’s slot *slotName*
- You can use this to connect extended node’s slot function.
- Arguments:
  - triggerName: name of the internal trigger
  - signalName: name of the signal function
  - slotName: name of the slot function
  - ...: [Variadic arguments], used for arguments’ type of signal and slot functions.
- Example:

```
class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{
public:
    ADD_INTERNAL_QWIDGET_TRIGGER(QCheckBox, checker, "Checker")
    ADD_INTERNAL_USER_CONNECTION(checker, stateChanged, slotExtended, int)
};
```

```
ADD_QWIDGET(widgetType, widgetName, ...)
ADD_QLAYOUT(layoutType, layoutName, ...)
```

- Add QWidget or QLayout. They are not used as trigger.
- Arguments:
  - widgetType: type of widget
  - widgetName: name of widget
  - layoutType: type of layout
  - layoutName: name of layout
  - ...: [Variadic arguments], used for arguments of widget's or layout's construction function.
- Example:

```
class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{
public:
    QVector<QRgb> colortable;
public:
    ADD_QLAYOUT(QHBoxLayout, layout)
    ADD_QWIDGET(QTabWidget, tabwidget)
    ADD_QWIDGET(QScrollArea, scrollarea)
    ADD_QWIDGET(QLabel, viewer, "Image Viewer")
};
```

```
ADD_CONNECTION(emitterName,signalName,receiverName,slotName,...)
```

- Add connection between *emitterName*'s signal *signalName* to *receiverName*'s slot *slotName*
- Arguments:
  - emitterName: name of emitter, must be QObject or QWidget
  - signalName: name of signal function
  - receiverName: name of receiver, must be QObject or QWidget
  - slotName: name of slot function
  - ...: [Variadic arguments], used for arguments' type of signal and slot functions.
- Example:

```
class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{
public:
    ADD_INTERNAL_QOBJECT_TRIGGER(QTimer, timer, 0)
    ADD_INTERNAL_DEFAULT_CONNECTION(timer,timeout)
    ADD_QWIDGET(QPushButton, start, "Start")
    ADD_CONNECTION(start, clicked, timer, start)
    ADD_QWIDGET(QSpinBox, interval)
    ADD_CONNECTION(interval, valueChanged, timer, start, int)
};
```

3.3.2.3 "Data"

```
NODE_DATA_TYPE
NODE_DATA_BASE_TYPE
```

- ”Data” class name and ”Data” base class name.
- Base class must be publicly inherited and ”Data” can multi-inherit other class.
- Example:

```
class NODE_DATA_TYPE : public NODE_DATA_BASE_TYPE, public cv::Mat
{

};
```

```
NODE_DATA_TYPE_REF(RefNodeClassName)
```

- Refer to *RefNodeClassName*’s ”Data” as its own ”Data”
- Arguments:
  - RefNodeClassName: ”Node Class” of referred node.
- Example:

```
NODE_DATA_TYPE_REF(VelodyneSensor)
```

3.3.3 Choose Node

```
USE_DEFAULT_NODE
```

- Use default node
- Example:

```
#include"CameraSensor.h"

//If you need to use extended node,
//please uncomment below and comment the using of default node
//USE_EXTENDED_NODE(ExtendedNodeClass[,...])
USE_DEFAULT_NODE
```

```
USE_EXTENDED_NODE(nodeType, ...)
```

- Use extended node as class *nodeType*
- Arguments:
  - nodeType: class name of the extended node.
  - ...: [Variadic arguments], used for arguments of extended node’s construction function.
- Example:

```
#include"UDP.h"

//If you need to use extended node,
//please uncomment below and comment the using of default node
USE_EXTENDED_NODE(SourceDrain)
//USE_DEFAULT_NODE
```

3.3.4 Node Function

3.3.4.1 Node Function Declaration

```
NODE_FUNC_DECL(returnType, funcName, ...)
NODE_EXFUNC_DECL(returnType, funcName, exName, ...)
```

- Declaration of Node Function and Extended Node Function
- Arguments:
  - returnType: return type of Node Function
  - funcName: name of Node Function
  - exName: extended name of Node Function
  - ...: [Variadic arguments], arguments of Node Function.
- Example:

```
NODE_FUNC_DECL(int, sum, int a, int b) // return a+b
NODE_EXFUNC_DECL(int, sum, squred, int a, int b) // return a^2+b^2
```

- Note:
  The arguments of original Node Function and Extended ones should be same.

3.3.4.2 Node Function Definition

```
NODE_FUNC_DEF_EXPORT(returnType, funcName, ...)
NODE_EXFUNC_DEF_EXPORT(returnType, funcName, exName, ...)
```

- Definition of Node Function and Extended Node Function
- Export with symbol, which can be resolved by QLibrary. Usually for interface functions.
- Arguments:
  - returnType: return type of Node Function
  - funcName: name of Node Function
  - exName: extended name of Node Function
  - ...: [Variadic arguments], arguments of Node Function.
- Example:

```
//This is original main function, you must keep it
NODE_FUNC_DEF_EXPORT(bool, main)
{
    return 1;
}
//If you don't need to initialize node, you can delete this code segment
NODE_EXFUNC_DEF_EXPORT(bool, initializeNode, Extension)
{
    return 1;
}
```

- Note:
  The arguments of original Node Function and Extended ones should be same.
  It does not need declaration in header.

```
NODE_FUNC_DEF(returnType, funcName, ...)
NODE_EXFUNC_DEF(returnType, funcName, exName, ...)
```

- Definition of Node Function and Extended Node Function
- No symbol. QLibrary cannot resolve it. Usually for custom functions.
- Arguments:
  - returnType: return type of Node Function
  - funcName: name of Node Function
  - exName: extended name of Node Function
  - ...: [Variadic arguments], arguments of Node Function.
- Example:

```
NODE_FUNC_DEF(int, sum, int a, int b)
{
    return a+b;
}
NODE_EXFUNC_DEF(int, sum, squred, int a, int b)
{
    return a*a+b*b;
}
```

- Note:
  The arguments of original Node Function and Extended ones should be same.

3.3.4.3 Access Node's Values

```
NODE_PARAMS
NODE_VARS
NODE_DATA
```

- Access node's "Params", "Vars" and "Data" in Node Function
- Example:

```
NODE_FUNC_DEF_EXPORT(bool, main)
{
    auto params=NODE_PARAMS;
    auto vars=NODE_VARS;
    auto data=NODE_DATA;
    data->result=params->value+vars->LineEditWidget->text().toInt();
    return 1;
}
NODE_FUNC_DEF(int, sum, int a, int b)
{
    NODE_DATA->result=a+b+NODE_PARAMS->offset;
    return data->result;
}
```

- Note:
  auto specifier is C++11 syntax (`http://en.cppreference.com/w/cpp/language/auto`).
  "params", "vars" and "data" are variable name, you can use any others for instead.

3.3.4.4 Access Obtained Port's Values

```
PORT_PARAMS_SIZE(portID)
PORT_DATA_SIZE(portID)
```

- The number of input "Params" and "Data" obtained from #*portID* input port
- Arguments:
  - portID: port id, must be a constant number.
- Example:

```
NODE_FUNC_DEF_EXPORT(bool, main)
{
    auto vars=NODE_VARS;
    int num=PORT_PARAMS_SIZE(0);
    vars->LabelWidget->setText(QString("Obtain %1 data").arg(num));
    return 1;
}
NODE_FUNC_DEF(int, sum)
{
    int i,num=PORT_DATA_SIZE(0);
    for(i=0;i<num;i++)
    {
        NODE_DATA->result+=PORT_DATA(0,i)->value;
    }
    return data->result;
}
```

```
PORT_PARAMS(portID, paramsID)
PORT_DATA(portID, dataID)
```

- The *paramsID*-th input "Params" or *dataID*-th "Data" obtained from #*portID* input port
- Small dataID $\Rightarrow$ new data; Large dataID $\Rightarrow$ old data.
- Arguments:
  - portID: port id, must be a constant number.
  - paramsID: params id in obtained params list.
  - dataID: data id in obtained data list.
- Example:

```
NODE_FUNC_DEF_EXPORT(bool, main)
{
    int i,num=PORT_PARAMS_SIZE(0);
    for(i=0;i<num;i++)
    {
        auto inputparams=PORT_PARAMS(0,i);
        qDebug()<<QString("%1 from node %2::%3::%4")
            .arg(inputparams->_nodeclass)
            .arg(inputparams->_nodename)
            .arg(inputparams->_exname);
    }
    return 1;
}
```

```
NODE_FUNC_DEF(void, copyLatestTimeStamp)
{
    int num=PORT_DATA_SIZE(0);
    if(num>0)
    {
        NODE_DATA->timestamp=PORT_DATA(0,0)->timestamp;
    }
}
```

- Note:
  If there is only one input node for one port, input "Params" are same copy.
  If there are many input nodes for one port, copies of different nodes' "Params" are mixed.
  You can use its "_nodeclass", "_nodename" and "_exname" to identify the source.
  NODE_DATA_TYPE inherits member "QTime timestamp" from NODE_DATA_BASE_TYPE.
  You may get NULL pointer, therefore, it is better to check what you get.

```
IS_INTERNAL_TRIGGER
```

- Judge the call is from data stream (input ports) or internal trigger.
- data stream call will pass valid port values.
- internal trigger call will pass invalid (empty) port values.
- Example:

```
//UDP Communication
NODE_FUNC_DEF_EXPORT(bool, main)
{
    auto vars=NODE_VARS;
    if(IS_INTERNAL_TRIGGER)
    {
        auto data=NODE_DATA;
        vars->udp->receiveData(data->message);
        return 1;
    }
    else
    {
        auto data=PORT_DATA(0,0);
        vars->udp->sendData(data->message);
        return 0;
    }
}
```

3.3.4.5 Call Node Function

```
NODE_FUNC(funcName, ...)
NODE_EXFUNC(funcName, exName, ...)
```

- Call Node Function from Node Function.
- Arguments:
  - funcName: name of Node Function
  - exName: extended name of Node Function
  - ...: [Variadic arguments], arguments of Node Function.

- Example:

```
NODE_FUNC_DEF(int, sum, int a, int b)
{
    auto params=NODE_PARAMS;
    return a+b+params->offset;
}
NODE_FUNC_DEF_EXPORT(bool, main)
{
    auto vars=NODE_VARS;
    auto inputdata=PORT_DATA(0,0);
    auto data=NODE_DATA;
    data->result=NODE_FUNC(sum, inputdata->value, vars->value);
    return 1;
}
```

3.3.5 Obtain Values from Input Ports

3.3.5.1 Obtain Behavior

```
enum ObtainBehavior
{
    CopyOldest=NUM_0,
    GrabOldest=NUM_1,
    CopyLatest=NUM_2,
    GrabLatest=NUM_3,
    CopyOldestStrictly=NUM_4,
    GrabOldestStrictly=NUM_5,
    CopyLatestStrictly=NUM_6,
    GrabLatestStrictly=NUM_7
};
```

- Explanation:
  - CopyOldest: Copy the oldest ”Params” and ”Data” without removing.
  - GrabOldest: Grab the oldest ”Params” and ”Data” with removing.
  - CopyLatest: Copy the latest ”Params” and ”Data” without removing.
  - GrabLatest: Grab the latest ”Params” and ”Data” with clearing buffer.
  - CopyOldestStrictly: If the all input buffer count fulfill requirement, CopyOldest.
  - GrabOldestStrictly: If the all input buffer count fulfill requirement, GrabOldest.
  - CopyLatestStrictly: If the all input buffer count fulfill requirement, CopyLatest.
  - GrabLatestStrictly: If the all input buffer count fulfill requirement, GrabLatest.
- Set it via NODE_VARS's functions in any Node Functions any time.

```
void setInputPortObtainDataBehavior(uint portID
                                    , ObtainBehavior obtainDataBehavior);
void setInputPortObtainDataBehavior(QList<ObtainBehavior> obtainDataBehavior);
```

- Example:

```
NODE_FUNC_DEF_EXPORT(bool, initializeNode)
{
    auto vars=NODE_VARS;
    vars->setInputPortObtainDataBehavior(0, CopyOldest);
```

```
    vars->setInputPortObtainDataBehavior(QList< ObtainBehavior >()
              <<CopyOldest<<GrabOldest);
    return 1;
}
```

- Note:
  Default value is ”GrabLatest”
  It is safe when portID is out of range or list size does not equal input port size

3.3.5.2 Obtain Parameters

- Parameters List (all in NODE_VARS):
  - buffersize: maximum input buffer size
    - 0: without size limitation.
    - >0: with size limitation.
    - Default: 0
    - Functions:

      ```
      void setInputPortBufferSize(uint portID, uint bufferSize);
      void setInputPortBufferSize(QList< uint > bufferSize);
      ```

  - obtaindatasize: size of obtain values each time
    - 0: obtain all values in the buffer.
    - >0: obtain certain size of values. Threshold for ”Strictly” obtain behavior.
    - Default: 1
    - Functions:

      ```
      void setInputPortObtainDataSize(uint portID, uint obtainDataSize);
      void setInputPortObtainDataSize(QList< uint > obtainDataSize);
      ```

  - triggerflag: flag to allow port to trigger node
    - false: not allowed.
    - true: allowed.
    - Default: true
    - Functions:

      ```
      void setInputPortTriggerFlag(uint portID, bool triggerFlag);
      void setInputPortTriggerFlag(QList< bool > triggerFlag);
      ```

  - Example:

    ```
    NODE_FUNC_DEF_EXPORT(bool, initializeNode)
    {
        auto vars=NODE_VARS;

        vars->setInputPortObtainDataBehavior(0, GrabOldestStrictly);
        vars->setInputPortBufferSize(0, 10);
        vars->setInputPortObtainDataSize(0, 10);

        vars->setInputPortBufferSize(1, 0);
        vars->setInputPortObtainDataSize(1, 0);

        vars->setInputPortTriggerFlag(QList<bool>()<<1<<0);
        return 1;
    }
    ```

- Note:
  You can set them in any Node Functions any time.
  It is safe when portID is out of range or list size does not equal input port size

3.3.6 Emit Values from Output Ports

- Parameters List (in NODE_DATA):
  - filterflag: flag to filter output ports
    - false: Values will not be emitted from this port.
    - true or NULL: Values will be emitted from this port.
    - Default: NULL
    - Functions:
      ```
      void setOutputPortFilterFlag(QList< bool > filterFlag);
      ```
  - Example:
    ```
    NODE_FUNC_DEF_EXPORT(bool, main)
    {
        auto data=NODE_DATA;
        uint out=qrand()%2;
        if(out==0)
        {
            data->setOutputPortFilterFlag(QList<bool>()<<0<<1);
        }
        else
        {
            data->setOutputPortFilterFlag(QList<bool>()<<1<<0);
        }
        return 1;
    }
    ```
- Note:
  You can set it in any Node Functions except for ”initializeNode”, ”openNode” and ”closeNode” any time.
  It is safe when list size does not equal input port size
  If not set, all ports are allowed to emit values.

3.3.7 Node’s Flags

3.3.7.1 GUI Thread Flag

- Parameters List (in NODE_VARS):
  - guithreadflag: flag to move node into GUI thread.
    - false: Node will go to its own thread.
    - true: Node will stay in GUI thread.
    - Default: false
    - Functions:
      ```
      void setNodeGUIThreadFlag(bool guiThreadFlag);
      ```
  - Example:
    ```
    NODE_FUNC_DEF_EXPORT(bool, initializeNode)
    {
        auto vars=NODE_VARS;
    ```

```
    vars->setNodeGUIThreadFlag(1);
    return 1;
}
```

- Note:
  You can only set it in ”initializeNode” Node Function.
  If node wants to control (write operation) its widgets, this must be set to false.

3.3.7.2 Show Widget Flag

- Parameters List (in NODE_VARS):
  - showwidgetflag: flag to allow non-GUI node to show widget.
    - false: Non-GUI node is not allowed to show widget.
    - true: Non-GUI node is allowed to show widget.
    - Default: false
    - Functions:
      `void setNodeShowWidgetFlag(bool showWidgetFlag);`
  - Example:

```
NODE_FUNC_DEF_EXPORT(bool, initializeNode)
{
    auto vars=NODE_VARS;
    vars->setNodeShowWidgetFlag(1);
    return 1;
}
```

- Note:
  You can set it in any Node Functions any time.
  It only affects non-GUI node.
  If non-GUI node only reads its widgets or is triggered by its widgets, this could be set to true.

3.3.8 Node Central Widget

- Every node has its own central widget stored in NODE_VARS and named as ”widget”.
- All widgets in NODE_VARS should embed into ”widget”, which could be shown by Robot-X.
- Its default state is invisible and is controlled by guithreadflag and showwidgetflag.
- Example:

```
NODE_FUNC_DEF_EXPORT(bool, initializeNode)
{
    auto vars=NODE_VARS;
    vars->viewer->setAlignment(Qt::AlignCenter); //QLabel
    vars->scrollarea->setWidget(vars->viewer); //QScrollArea
    vars->tabwidget->addTab(vars->scrollarea,"TimeStamp"); //QTabWidget
    vars->layout->addWidget(vars->tabwidget); //QHBoxLayout
    vars->widget->setLayout(vars->layout); //central widget
    vars->setNodeGUIThreadFlag(1);
    return 1;
}
```

3.3.9 Sync

`ADD_SYNC(syncName, basePortID)`

- This should be used in ”Vars” and it will add a Sync to ”Vars”
- Arguments:
  - syncName: name of Sync
  - basePortID: indicates which port is the base for Sync (extract one by one)
- Example:

```
//================================================
//Node configuration

#undef NODE_CLASS
#define NODE_CLASS DPMModifier

#undef INPUT_PORT_NUM
#define INPUT_PORT_NUM 2

#undef OUTPUT_PORT_NUM
#define OUTPUT_PORT_NUM 1

//Uncomment below PORT_DECL and set input node class name
PORT_DECL(0, CameraSensor)
PORT_DECL(1, DPMDetector)

//================================================
/*...*/
class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{
public:
      ADD_SYNC(dpmsync, 1)
};
```

- Node:
  You can add as many as Syncs simutaneously.

`SYNC_START(sync)`

- If you have a Sync, you need to call SYNC_START at least for one time during ”Excecution” loop.
- Argument:
  - sync: Sync instance
- Example:

```
NODE_FUNC_DEF_EXPORT(bool, main)
{
    auto vars=NODE_VARS;
    bool flag=SYNC_START(vars->dpmsync);
    if(flag)
    {
        auto imagedata=SYNC_DATA(vars->dpmsync,0);
        auto dpmdata=SYNC_DATA(vars->dpmsync,1);
        /*...*/
    }
```

```
    return 1;
}
```

- Node:
  This call will return a boolean flag to indicate whether the Sync has available synchronized data.
  If the Sync has synchronized data, then the synchronized data will be moved to ready-to-read array from synchronizing buffer.

```
SYNC_PARAMS(sync, portID)
SYNC_DATA(sync, portID)
```

- Get Synchronized input ”Params” and ”Data” from Sync’s ready-to-read array
- Example:
  - sync: Sync instance
  - portID: port id of desired input ”Params” and ”Data”, must be constant number.
- Example:

```
NODE_FUNC_DEF_EXPORT(bool, main)
{
    auto vars=NODE_VARS;
    while(SYNC_START(vars->dpmsync))
    {
        auto imagedata=SYNC_DATA(vars->dpmsync,0);
        auto dpmdata=SYNC_DATA(vars->dpmsync,1);
        /*...*/
    }
    return 1;
}
```

- Note:
  It will get same value unless you call SYNC_START again.

Chapter 4

Node Extension: Customize Node Behavior

Node extension is an advanced topic and it is not neccessary for common top-down modular software development. Therefore, in this chapter, we will briefly introduce how to extend default node.

4.1 Extend Default Node

4.1.1 Default Node

As shown in Fig.4.1, the default node behavior contains two routes to access ”Main Entry” (main Node Function in module) and then emit its ”Params” and new ”Data” according to the boolean return value of ”Main Entry”. The first route starts from ”Input Ports” and node’s virtual slot function ”slotObtainParams-Data” is called (inter-thread call). This route passes input ”Params” and ”Data” to the ”Main Entry”. The second route starts from ”Internal Triggers” and node’s virtual slot function ”slotDefaultTrigger” is called (inter-thread call if the trigger is in ”QObjectPoolThread” or is a QWidget, otherwise intra-thread call). This route does not pass input ”Params” and ”Data” to the ”Main Entry”.

4.1.2 5 Kinds of Extension

Developers can extend default node’s behavior by inheriting default ”Node” class and there are 5 things you can do for extension.

1. Re-implement virtual slot function ”slotObtainParamsData” to modify ”1.2” and ”1.3” in Fig.4.1.

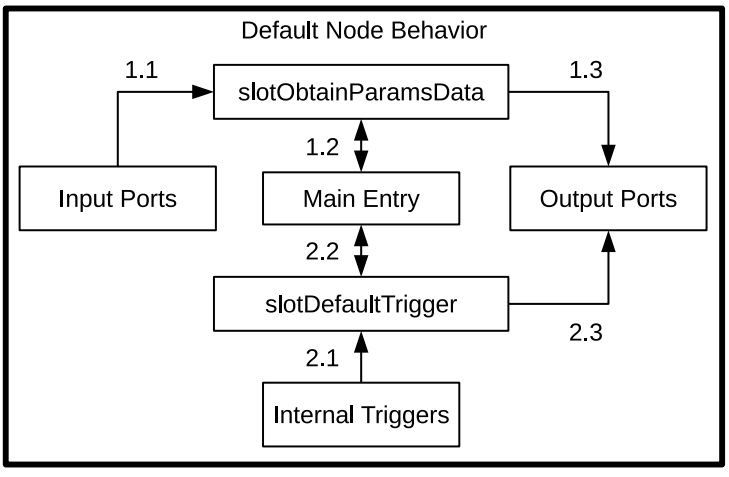

Figure 4.1: Default Node Behavior

2. Re-implement virtual slot function ”slotDefaultTrigger” to modify ”2.2” and ”2.3” in Fig.4.1.
3. Add new slot functions to enrich ”2.1” ends in Fig.4.1.
4. Add new interface functions for new entries.
5. Add new signal functions for high-level application development
   (original Robot-X couldn't work with new signal functions, but you can extend it with slot functions).

The source code files of extended node class could be put anywhere, for example, in the module project, in the Kernel or standalone as compiled static library. Below we will show some examples about these 5 kinds of extension without too much details.

4.1.2.1 1st Extension

- Original ”slotObtainParamsData”

```
void Node::slotObtainParamsData(PORT_PARAMS_CAPSULE inputParams
                                , PORT_DATA_CAPSULE inputData)
{
    INPUT_PARAMS_ARG=inputParams;
    INPUT_DATA_ARG=inputData;
    NODE_DATA_ARG=generateNodeData();
    if(NODE_FUNC_PTR(main))
    {
        emit signalSendParamsData(NODE_PARAMS_ARG,NODE_DATA_ARG);
    }
}
```

- Extension Example:
  Assume we have defined another entry as ”ExMain” with ”int” type return value and additional argument QTime (see ”4th Extension”), and we also add a signal function ”signalExecution-Result(int returnValue)” (see ”5th Extension”). We can extend ”slotObtainParamsData” as:

```
//header
#include<node.h>
namespace RobotSDK
{
class ExNode : public Node
{
public slots:
    void slotObtainParamsData(PORT_PARAMS_CAPSULE inputParams
                              , PORT_DATA_CAPSULE inputData);
    /*...*/
}
}

//source
using namespace RobotSDK;
void ExNode::slotObtainParamsData(PORT_PARAMS_CAPSULE inputParams
                                  , PORT_DATA_CAPSULE inputData)
{
    INPUT_PARAMS_ARG=inputParams;
    INPUT_DATA_ARG=inputData;
    NODE_DATA_ARG=generateNodeData();
    QTime time=QTime::currentTime();
    int result=NODE_FUNC_PTR(ExMain, time);
    emit signalExecutionResult(result);
```

```
}
}

//module source
USE_EXTENDED_NODE(ExNode)
```

4.1.2.2 2nd Extension

- Original ”slotDefaultTrigger”

```
void Node::slotDefaultTrigger()
{
    INPUT_PARAMS_ARG.clear();
    INPUT_DATA_ARG.clear();
    NODE_DATA_ARG=generateNodeData();
    if(NODE_FUNC_PTR(main))
    {
        emit signalSendParamsData(NODE_PARAMS_ARG,NODE_DATA_ARG);
    }
}
```

Extension Example:
With same assumption of ”1st Extension”

```
//header
#include<node.h>
namespace RobotSDK
{
class ExNode : public Node
{
public slots:
    void slotDefaultTrigger();
    /*...*/
}
}

//source
using namespace RobotSDK;
void ExNode::slotDefaultTrigger()
{
    INPUT_PARAMS_ARG.clear();
    INPUT_DATA_ARG.clear();
    NODE_DATA_ARG=generateNodeData();
    QTime time=QTime::currentTime();
    int result=NODE_FUNC_PTR(ExMain, time);
    emit signalExecutionResult(result);
    if(result>0)
    {
        emit signalSendParamsData(NODE_PARAMS_ARG,NODE_DATA_ARG);
    }
}

//module source
USE_EXTENDED_NODE(ExNode)
```

4.1.2.3 3rd Extension

- Extension Example:
  With same assumption of ”1st Extension”

```
//header
#include<node.h>
namespace RobotSDK
{
class ExNode : public Node
{
public slots:
    void slotExtendedTrigger(int msecs);
    /*...*/
}
}

//source
using namespace RobotSDK;
void ExNode::slotExtendedTrigger(int msecs)
{
    INPUT_PARAMS_ARG.clear();
    INPUT_DATA_ARG.clear();
    NODE_DATA_ARG=generateNodeData();
    QTime time=QTime::fromMSecsSinceStartOfDay(msecs);
    int result=NODE_FUNC_PTR(ExMain, time);
    emit signalExecutionResult(result);
}

//module header
class NODE_VARS_TYPE : public NODE_VARS_BASE_TYPE
{
public:
    ADD_INTERNAL_QWIDGET_TRIGGER(QSpinBox, timeMsec, 0)
    ADD_QWIDGET(QSpinBox, interval)
    ADD_INTERNAL_USER_CONNECTION(QSpinBox, valueChanged
                                 , slotExtendedTrigger, int)
};

//module source
USE_EXTENDED_NODE(ExNode)
```

4.1.2.4 4th Extension

- Extension Example:
  Realize assumptions used above

```
//header
#include<node.h>
namespace RobotSDK
{
class ExNode : public Node
{
public:
    ADD_NODE_FUNC_PTR(int, ExMain, 1, QTime time)
    /*...*/
}
}
```

```
//module source
USE_EXTENDED_NODE(ExNode)

NODE_FUNC_DEF_EXPORT(int, ExMain, QTime time)
{
    auto data=NODE_DATA;
    data->second=time.second();
    return data->second;
}
```

4.1.2.5 5th Extension

- Extension Example:
  Realize assumptions used above

```
//header
#include<node.h>
namespace RobotSDK
{
class ExNode : public Node
{
signals:
    void signalExecutionResult(int result);
    /*...*/
}
}

//Extension of Robot-X (code segments)
#include<exnode.h> //[Extension]
/*...*/
RobotSDK::Graph * graph=new RobotSDK::Graph; //[Robot-X]
/*...*/
QString nodeFullName, libraryFileName, configFileName; //[Robot-X]
/*...*/
graph->addNode(nodeFullName, libraryFileName, configFileName); //[Robot-X]
/*...*/
//[Extension]
RobotSDK::Node * node=graph->getNode(nodeFullName);
connect(node, SIGNAL(signalExecutionResult(int))
      , this, SLOT(slotExecutionResult(int)));
/*...*/
```

4.2 RobotSDK Syntax for Node Extension

```
ADD_NODE_FUNC_PTR(returnType, funcName, mandatoryFlag, ...)
```

- Add an interface function to link module's Node Function.
- It will be automatically managed by RobotSDK.
- Arguments:
  - returnType: return type of interface function.
  - funcName: name of interface function
  - mandatoryFlag: flag to show whether it is a mandatory interface function
    - false: RobotSDK will try to link it to module, if failed, node will still be constructed.

        * true: RobotSDK will try to link it to module, if failed, node will not be constructed.
    - ...: [Variadic arguments], used for arguments of interface function.
- Example:
  see ”4th Extension”

`NODE_FUNC_PTR(funcName, ...)`

- Load an interface function.
- Arguments:
  - funcName: name of interface function
  - ...: [Variadic arguments], used for arguments of interface function.
- Example:
  see ”1st, 2nd and 3rd Extenstions”

`CHECK_NODE_FUNC_PTR(funcName)`

- Check whether an interface function is available if it is not mandatory.
- Arguments:
  - funcName: name of interface function
- Example:

```
//in Kernel header
class Node : public QObject
{
    /*...*/
    ADD_NODE_FUNC_PTR(bool, openNode, 0)
    /*...*/
}

//in Kernel source (code segment)
if(CHECK_NODE_FUNC_PTR(openNode))
{
    _openflag=NODE_FUNC_PTR(openNode);
}
```