Developing Plug-ins for NMM

Motama GmbH, Saarbruecken, Germany
(http://www.motama.com)

April 2010

This document describes the development of plug-ins (called nodes) for NMM.

1. Introduction

By now, you should be familiar with the core concepts of NMM, namely nodes, jacks, messages, and formats. You should know what a flow graph of NMM nodes is. If not, please read the corresponding documentation first.

Developing a plug-in or node in NMM consists of several steps. Each of these steps will be described with some examples.

2. Step A: Thinking

First, make up some functionality that you want or need and that is not yet supported by another NMM node or a combination of nodes. This can be anything from a simple audio-filter to high-quality chroma-keying with overlay. You will also have to think about what kind of interaction you want to provide. For example, a wav file reading node will need a filename and could provide seeking.

3. Step B: What kind of node? One node or many nodes?

You have to decide into how many nodes you want to split up the desired functionality. At the same time, you have to decide what kind of node you want to use for each part.

In general, one could say the more nodes you can create the better. A node represents the smallest entity of processing. In order to be able to extensively reuse specific nodes in different application scenarios, nodes should represent very fine-grained processing units.

For each of these processing units, you then have to choose the appropriate super class. Each of these classes provides a framework that simplifies the development of new nodes. Roughly speaking, only the processBuffer()-method has to be implemented and some other thing set up. Following generic classes for are provided.

- GenericSourceNode provides an output jack called "default". It also provides an input jack called "default" that should only be used by an application for in-stream events to be sent downstream, and that is not to be connected. An example for a source would be a file reader or a camera.
- GenericProcessorNode is the most widely used super class, and provides one input jack and one output jack, both named "default". This class can be used for most converters.
- GenericConverterNode is derived from GenericProcessorNode and can be used for converter nodes that support several input and output formats. This class offers an comfortable way to register methods for handling different input and output formats. For example, a video converter supporting different YUV to RGB operations should be derived from this class.
- GenericFilterNode is derived from GenericProcessorNode and additionally offers the interface IFilterNode, which allows to enable or disable the processing of buffers. Per definition filters do not change the format of the stream. An example for a filter would be an audio effect that uses the same input and output format.
- GenericSinkNode provides an input jack called "default". It also provides an output jack called "default" that should only be used by an application for in-stream events to be sent upstream, and that is not to be connected. An example for a sink would be a node that displays a video image.
- GenericMultiplexerNode already provides a single output jack called "default". All wanted input jacks have to be added by calling addInputStream("some_name"). This can be done in the constructor of the node or in the doInit()-method. The second option is needed, if the number of supported input streams depends on some settings to be done in the CONSTRUCTED state. An example for a multiplexer would be a video processing node that takes two video streams and merges them together.

- GenericDemultiplexerNode already provides a single input jack called "default". All wanted output jacks have to be added by calling addOutputStream("some_name"). Again, this can be done in the constructor of the node or in the doInit()-method. The second option is needed, if the number of supported output streams depends on some settings to be done in the CONSTRUCTED state. Some nodes also need to analyze some data buffers before they can add their output streams. Then the doInitOutput()-method should be used for this reason. An example for a demultiplexer would be an demultiplexer that splits the stream in its components (e.g. video and audio).
- GenericMuxDemuxNode is a combination of GenericMultiplexerNode and GenericDemultiplexerNode. All wanted input and output jacks have to be added.

Sink nodes that need to present their data synchronized with other sink nodes (or with a constant rate, e.g. 25 fps) should be derived from special super classes. These classes will be explained only briefly, for further information read the NMM synchronization manual.

- GenericBSyncSinkNode and GenericUSyncSinkNode both provide an input jack called "default". The 'B' stands for buffered, the 'U' for unbuffered. Bufferd sink nodes are nodes that store some data internally (like an audio device). Therefore the incoming data cannot be presented right away. Unbuffered sink nodes are nodes that can present the incoming data with little or no delay. For both nodes, the processBuffer()-method is replaced by the prepareBuffer()- and the presentBuffer()-method. In prepareBuffer() all time-independent pre-processing should be done (e.g. copying an video frame to a non-visible frame buffer), in presentBuffer() all time-dependent processing should be done (e.g. making the currently non-visible frame buffer visible). Both classes also provide an output jack called "default" that should only be used by an application for in-stream events to be sent upstream, and that is not to be connected. An example for a synchronized sink would be a node that displays a video image.

Deriving from a wanted super class is done as follows. Create an hpp-file (AudioDecodeNode.hpp in this example) with following:

```
#ifndef NMM_AUDIODECODENODE_HPP
#define NMM_AUDIODECODENODE_HPP NMM_AUDIODECODENODE_HPP


namespace NMM {
  class AudioDecodeNode: public GenericProcessorNode {
  public:
    AudioDecodeNode(const char* = "AudioDecodeNode");

    virtual ~AudioDecodeNode();
  };
}
```

The cpp-file (AudioDecodeNode.cpp) looks as follows:

```
#include "AudioDecodeNode.hpp"

namespace NMM {
```

```
  AudioDecodeNode::AudioDecodeNode(const char *name)
    :GenericProcessorNode(name,
                          StreamQueue::MODE_SUSPEND,
                          StreamQueue::DEFAULT_SIZE)
  {
    // your code goes here
  }

  AudioDecodeNode::~AudioDecodeNode()
  {
    // your code goes here
  }
}
```

The are several noticable things in these few lines. First, this plug-in follows naming convention for NMM nodes. The class name is also used as name of the node. This is important because nodes are registered with their name in the registry. Therefore, try to chose a meaningful name and avoid duplicates. Furthermore, the stream queues of this node are always in MODE_SUSPEND which means that they will not discard incoming messages. The size of the queues are set to the DEFAULT_SIZE.

If you want to allow the user of your node to change the behavior of the stream queues you have to write following in the hpp-file (and change the cpp-file accordingly):

```
#ifndef NMM_AUDIODECODENODE_HPP
#define NMM_AUDIODECODENODE_HPP


namespace NMM {
  class AudioDecodeNode: public GenericProcessorNode {
  public:
    AudioDecodeNode(const char* = "AudioDecodeNode",
                        StreamQueue::Mode = StreamQueue::MODE_SUSPEND,
                        int = StreamQueue::DEFAULT_SIZE);

    virtual ~AudioDecodeNode();
  };
}
```

The default queue mode DEFAULT_QUEUE_MODE is currently MODE_SUSPEND and the DEFAULT_QUEUE_SIZE is currently set to 16. If you do not want to rely on the default, feel free to specify your own values.

Notice, that these file will not necessarily compile. Remember, we are still stuck at step 1 of creating a new plug-in.

4. Step C: What interfaces should be supported?

Now you have to consider what kind of interaction your node should support. This can either mean out-of-band interaction or instream interaction. In this context, out-of-band means interaction between the application and a node; instream means between two or more nodes. An interface can be seen as a logical grouping of methods to interact with the node and a number of events a node should handle when received instream. An example would be an interface for cameras, which provides out-of-band interaction for setting camera parameters like brightness and focus. Another example would be an interface for a file handler sink node, which can handle the instream event called "endOfFile".

If you are not sure what interfaces your node should support, do not worry. All generic nodes provide the Node-interface. You can go on and read the following sections and then start developing a new node and add interfaces lateron. Or you might not need any additional interfaces at all.

Out-of-band interaction can be performed in two ways: either by sending an event from the application to a node or by calling the corresponding interface method. We recommand that you use the second option as in the following example:

```
INode_var camera( /* get a camera, e.g. from the registry */ );

// try to get the ICamera interface from camera node
ICamera_var icamera(camera->getInterface< ICamera >());

// if camera supports ICamera interface, call methods from ICamera
if(icamera.get()) {
  if(icamera->hasBrightness()) {
    // set brightness to some useful value
    pair< int, int > range = icamera->getBrightnessRange();
    icamera->setBrightness((range.first + range.second)/2);
  }
}
```

The following example uses events instead of method calls. Although the code will essential do the same, there are several drawbacks: it is not guaranteed to be working in future NMM versions, it might even produce syntax errors. It is not type safe. It does not care about exceptions. It is a lot harder to read and understand. And finally it is a lot more characters to write down. To sum up, use the IDL compilers and do not worry about it.

```
// NOTE: the following code is only for demonstration purposes!

INode_var camera( /* get a camera, e.g. from the registry */ );

// ignore interfaces and send event to node
TOutValue< pair< int, int > > value;
// take static member from ICamera as event string
Event event(ICamera::getBrightnessRange_event);
event.setReturnValue(&value);
```

```
if(camera->getParentObject().sendEvent(event) == FAILURE) {
  cerr << "Could not set brightness! "
       << "Maybe node does not support ICamera?" << endl;
}
else {
  pair< int, int >& range = value.getRValue();
  Event event2(ICamera::setBrightness_event,
               new TInValue< int >((range.first + range.second)/2));
  if(camera->getParentObject().sendEvent(event2) == FAILURE) {
    cerr << "Could not set brightness! "
         << "Maybe node does not support ICamera?" << endl;
  }
}
// NOTE: the preceding code is only for demonstration purposes!
```

On the other side, instream interaction is done by creating an event and returning it in the processBuffer()-method. By setting the direction, the event can either be sent upstream or downstream. This will be further explained in step 6. For now, it is only important to decide what instream events your node should be able to handle.

Both kinds of interaction should be specified by using interfaces. This is done in several steps. First, find the wanted interface or interfaces your node should support. This is done by examining the idl-Files in the interfaces subdirectory. If you cannot find an interface for your purpose, you have to write an IDL description. This step is described in detail in the corresponding manual page. Then, generate the hpp- and cpp-files for the interface and implementation part from this IDL description - or even better, let them be generated automatically. Again, this step is explained in detail in the IDL manual page. Your node then has to be derived from the wanted implementation class.

```
#include "nmm/plugins/video/camera/ICameraImpl.hpp"

namespace NMM{

  class MyCamera : public GenericSourceNode, public ICameraImpl
  {
  public:

    MyCamera(const char* name = "MyCamera");

    virtual ~MyCamera();
  }
}
```

Then you have to implement the interface. This is done by providing an implementation for all pure virtual methods inherited by the interface implementation class. Note, each method can represent out-of-band interaction, instream interaction, or both.

```
#include "nmm/plugins/video/camera/ICameraImpl.hpp"
```

```
namespace NMM{

  class MyCamera : public GenericSourceNode, public ICameraImpl
  {
  public:

    MyCamera(const char* name = "MyCamera");

    virtual ~MyCamera();

  protected: // interface methods

    virtual void setBrightness(const int& value);
    virtual int getBrightness();
    virtual pair< int, int > getBrightnessRange();
    virtual bool hasBrightness();

    // some more ...
  }
}
```

The cpp-Files looks like this.

```
#include "MyCamera.hpp"

namespace NMM {

  MyCamera::MyCamera(const char* name)
  : GenericFireWireNode(name)
  {
    // your code goes here
  }

  void MyCamera::setBrightness(const int& value)
  {
    // your code goes here
  }

  // some more ...
}
```

5. Step D: What formats will be supported?

5.1. General approach

Since nodes are processing multimedia data, it is important to specify the input and output formats of a node. There are two places where you have to setup the formats of your node. First, the generally supported formats should be specifies in doInit() (use the methods getOwnInputProperty() or getOwnOutputProperty()). Secondly, formats specific to this instance of the node should be specified in doInitOutput() (use the methods getOwnWorkingInputProperty() or getOwnWorkingOutputProperty()).

```
Result JPEGDecodeNode::doInit() {
  // set static in and out formats in own property
  Format *myInputFormat = getOwnInputProperty()->addNewFormat("video/jpeg");
  myInputFormat->addWildcard(Format::x_resolution_param);
  myInputFormat->addWildcard(Format::y_resolution_param);
  myInputFormat->addStringParamValue(Format::colorspace_param,"rgb24");
  myInputFormat->addStringParamValue(Format::channelorder_param,"bgr");
  myInputFormat->addIntParamValue(Format::bitperpixel_param,24);
  myInputFormat->addStringParamValue(Format::endian_param, "little");
  myInputFormat->addWildcard(Format::framerate_param);
  getOwnInputProperty()->setDefaultFormat(myInputFormat);

  Format *myOutputFormat = getOwnOutputProperty()->addNewFormat("video/raw");
  myOutputFormat->addWildcard(Format::x_resolution_param);
  myOutputFormat->addWildcard(Format::y_resolution_param);
  myOutputFormat->addStringParamValue(Format::colorspace_param,"rgb24");
  myOutputFormat->addStringParamValue(Format::channelorder_param,"rgb");
  myOutputFormat->addIntParamValue(Format::bitperpixel_param,24);
  myOutputFormat->addStringParamValue(Format::endian_param, "little");
  myOutputFormat->addWildcard(Format::framerate_param);
  myOutputFormat->addStringParamValue(Format::format_param, "packet");
  getOwnOutputProperty()->setDefaultFormat(myOutputFormat);

  myInputFormat->setIOPartner(myOutputFormat);
  myOutputFormat->setIOPartner(myInputFormat);

  return SUCCESS;
}

Result JPEGDecodeNode::doInitOutput() {
  // set in and out formats in working property

  const Format* in_format = getInputFormat();
  if (in_format == NULL) { return FAILURE; }

  int x_resolution = in_format->getIntValue(Format::x_resolution_param);
  int y_resolution = in_format->getIntValue(Format::y_resolution_param);
  float framerate = in_format->getFloatValue(Format::framerate_param);

  Format* myInputFormat = new Format( *in_format );
```

```
    getOwnWorkingInputProperty()->addNewFormat(myInputFormat);
    getOwnWorkingInputProperty()->setDefaultFormat(myInputFormat);

    Format *myOutputFormat = getOwnWorkingOutputProperty()->addNewFormat("video/raw");
    myOutputFormat->addIntParamValue(Format::x_resolution_param, x_resolution);
    myOutputFormat->addIntParamValue(Format::y_resolution_param, y_resolution);
    myOutputFormat->addStringParamValue(Format::colorspace_param,"rgb24");
    myOutputFormat->addStringParamValue(Format::channelorder_param,"rgb");
    myOutputFormat->addIntParamValue(Format::bitperpixel_param,24);
    myOutputFormat->addStringParamValue(Format::endian_param, "little");
    myOutputFormat->addFloatParamValue(Format::framerate_param, framerate);
    myOutputFormat->addStringParamValue(Format::format_param, "packet");
    getOwnWorkingOutputProperty()->setDefaultFormat(myOutputFormat);

    myInputFormat->setIOPartner(myOutputFormat);
    myOutputFormat->setIOPartner(myInputFormat);

    return SUCCESS;
  }
```

The do-Methods will be called by the corresponding methods for state transitions. Of course, doInit() and doInitOutput() have to be declared in the corresponding hpp-file.

```
namespace NMM {

  class JPEGDecodeNode : public GenericProcessorNode {
    // some more code ...

  protected:
    virtual Result doInit();

    virtual Result doInitOutput();

    // some more code ...

  }
}
```

Possible parameter values are as follows.

```
 // single value
OFormat1->addFloatParamValue("framerate", 7.5);

// set with two entries
OFormat2->addFloatParamValue("framerate", 7.5);
OFormat2->addFloatParamValue("framerate", 15.0);

// range of values
OFormat3->addRFloatParamValue("framerate", 7.5, 30.0);
```

```
// wildcard
OFormat4->addWildcard("framerate");
```

5.2. Analyzing Data

Nodes, such as decoders, often need to analyze the data (i.e. NMM buffers) received from the predecessors to determine their current output formats. Often, a possible implementation for this step often requires to provide the same functionality as during normal operation of the node. So, when implementing the doInitOutput-method (in which the working output properties need to be set), NMM allows to analyze data within the processBuffer-method. For fully understanding the processBuffer-method, please refer to the following sections.

```
Result AudioDecodeNode::doInitOutput()
{

  // first we need to analze the stream of incoming buffers to
  // determine the current output format by starting to analyze data:
  // this will block and trigger the processBuffer method as usual
  // until the node determines that this phase has completed.

  if (analyseData() == false) {
      ERROR_STREAM("analyseData failed in doInitOutput" << endl);
      return FAILURE;
  }

  // now, set the current output format(s) in the working
  // output property ..

}
```

Within the processBuffer-method, analysing data and thereby returning to the flow of execution within the doInitOutput-method is done as follows.

```
Message* AudioDecodeNode::processBuffer(Buffer *in_buffer)
{

  // normal operation of processBuffer ..

  // if output format can be determined,
  // stop the analysis-phase
  // (which will bring you back to doInitOutput()
  //  when returning from processBuffer)
  if(getAnalysingFlag() == true) {
    setAnalysingFlag(false);
  }

  // continue as usual ..
```

```
}
```

6. Step E: What needs to be done during state transitions?

We have already seen that some nodes will specify their formats in doInit(). In general, for all other state transitions, there exits a corresponding do-method, which can be implemented. See step D for details.

7. Step F: What functionality has to be performed? Handling instream messages

7.1. The processBuffer()-method

For a plug-in programmer, the processBuffer()-method is the central part of a node. Within this method, multimedia data can be generated, processed, or consumed. Furthermore, new events are generated here. Remember, events are not handled within this method, but within registered handling methods of interfaces. The signature of the processBuffer()-method is as follows.

```
Message* processBuffer(Buffer *);
```

7.2. General strategy - processor, converter, and filter nodes

The overall processing is the same for all node types: the processBuffer()-method is called and the node can return a message.

We will now further define the conditions under which the processBuffer()-method is called, what is passed as parameter, what can be returned, and what differences exists between the different node types (namely sources, processors, sinks, multiplexers, and demultiplexers). Do not worry if you do not understand all details, since the generic layer will provide all described functionality for you. Nevertheless it is necessary to roughly understand the described mechanims.

Let us first consider a GenericProcessor node. This node has a single input and a single output. It will try to get a message from its input if a message can be passed to its output. A message can only be dequeued

from an input if the predecessor of the node already provided one. A node cannot pass a message to its output if its successor does not allow so. This is the case if the input of the successor is already congested.

Since nodes not only support downstream messages but also upstream messages, the same test will be performed with roles switched for input and output. Furthermore, since upstream messages are considered to be relatively rare and thus more important, this test is performed first. All following explanations will focus on downstream messages, upstream messages are treated analogous.

If the above criterion is satisfied, a message will be dequeued. If the message is an event (or more precisely a composite event), it will be handled by calling the registered methods. The message will then be passed on, either downstream (if it was dequeued from an input) or upstream (if it was dequeued from an output). Additionally, you can delete events from a composite event, insert new events, or manipulate the parameters associated with an event.

```
Result MyNode::handleEventAndDelete()
{
  // return DELETE to delete current instream event
  return DELETE;
}

Result MyNode::handleEventAndInsertNewEvent()
{
  //create the new event.
  Event* new_event = ISomeInterface::create_Foo("Add a new Event");

  //insert the event
  insertEvent(new_event);

  return SUCCESS;
}

Result MyNode::handleEventAndReplace()
{
  //create the new event.
  Event* new_event = ISomeInterface::create_Foo("Add a new Event");

  //insert the event
  insertEvent(new_event);

  // return DELETE to delete current instream event
  return DELETE;
}

Result MyNode::handleEventAndManipulateParameters(int &i)
{
  // change value
  ++i;

  return SUCCESS;
}
```

The used create_-methods are automatically generated for instream events by the IDL compilers (see NMM IDL documentation).

If the message is a buffer, the processBuffer()-method will be called with the buffer passed as parameter. The node can modify the data in the buffer and return the same buffer. This processor node would be a filter then. Notice, that the node then needs to request a writable instance of the current buffer, since every buffer (or every message in general) might be in different parts of a flow graph.

```
Message* MyNode::processBuffer(Buffer *in_buffer)
{
  // check reference count of buffer
  in_buffer = in_buffer->getWriteableInstance();

  // get data of buffer
  char* p = in_buffer->getData();

  // modify data ...

  return in_buffer;
}
```

The method getWritableInstance() checks the reference count and returns a copy of the buffer if necessary.

The node could also request a new buffer (perhaps with a different size), fill that buffer with data and finally return it. The incoming buffer then has to be released.

```
Message* MyNode::processBuffer(Buffer *in_buffer)
{
  // get new buffer with other size
  Buffer* out_buffer = getNewBuffer( in_buffer->getSize() * 2 );

  // get data of buffer
  char* p = in_buffer->getData();

  // some code ...

  // release in_buffer
  in_buffer->release();

  return out_buffer;
}
```

The node could also release the buffer and return null (or only release every second buffer). Of course, if null is returned, nothing will be forwarded to the successor of the node.

```
Message* MyNode::processBuffer(Buffer *in_buffer)
{
```

```
  in_buffer->release();

  return 0;
}
```

The node could also release the buffer and create an instream message.

```
Message* MyNode::processBuffer(Buffer *in_buffer)
{
  in_buffer->release();

  // return new event
  return
    new CEvent( ISomeInterface::create_Foo("new Event") );
}
```

7.3. The working-flag

But what if a node wants to create two or more buffers out of one? Then the working-flag has to be set. If this flag is true, processBuffer() is called with a null-pointer even if a message (i.e. a buffer or an event) could be dequeued. This flag should be set to true in two cases.

First, the node receives one buffer as input and produces more than one buffer or event as output.

```
Message* MyNode::processBuffer(Buffer *in_buffer)
{
  Buffer* out_buffer = 0;

  if(!getWorkingFlag()) {
    // if the working-flag is false, processBuffer() was
    // triggered with new in_buffer -> start processing
    setWorkingFlag(true);

    // save buffer for next processBuffer()
    current_in_buffer = in_buffer;

    // get new out_buffer
    out_buffer = getNewBuffer(some_size);

    // some code ...

  }
  else {
    // working-flag is true
    // -> processBuffer() was triggered with 0-pointer
    // -> continue work on old in_buffer
```

```
    // get new out_buffer
    out_buffer = getNewBuffer(some_size);

    // some code ...

    if(finished_current_in_buffer) {
      // release current in_buffer
      current_in_buffer->release();
  }

  // both cases -> return out_buffer
  return out_buffer;
}
```

Secondly, the node receives an event (either instream or out-of-band) and wants to produce one or more buffers or events in response.

```
// some registered event handler
Result MyNode::handleSomeEvent()
{
  // set working-flag to indicate that buffers or events
  // can be produced without incoming buffers
  setWorkingFlag(true);

  // set flag to indicate that new buffers should be generated
  produce_new_out_buffer = true;

  return SUCCESS;
}

Result MyNode::handleSomeOtherEvent()
{
  // set working-flag to indicate that buffers or events
  // can be produced without incoming buffers
  setWorkingFlag(true);

  // set flag to indicate that new event should be generated
  produce_new_event = true;

  return SUCCESS;
}

Message* MyNode::processBuffer(Buffer *in_buffer)
{
  if(!getWorkingFlag()) {
    // if the working-flag is false, processBuffer()
    // was triggered with new in_buffer
    // -> start processing in_buffer

    // get new out_buffer
```

```
    Buffer* out_buffer = getNewBuffer(some_size);

    // some code ...

    // do not forget to release in_buffer
    in_buffer->release();

    return out_buffer;
  }
  else {
    // working-flag was set to true in
    // MyNode::handleSomeEvent() or MyNode::handleSomeOtherEvent()
    // -> processBuffer() was triggered with 0-pointer
    // -> generate new buffers or events

    // flag was set to true in MyNode::handleSomeEvent()
    if(produce_new_out_buffer) {
      // get new out_buffer
      Buffer* out_buffer = getNewBuffer(some_size);

      // some code ...

      return out_buffer;
    }
    // flag was set to true in MyNode::handleSomeOtherEvent()
    else if(produce_new_event) {
      // some code ...

      // return new event
      return
        new CEvent( ISomeInterface::create_Foo("new Event") );
    }
    else {
      return 0;
    }
  }
}
```

Notice: the access to the working-flag is not guarded by a mutex. If you need this feature, you will have to implement a wrapper method.

7.4. Upstream and downstream messages

Messages created within processBuffer() can be sent either upstream or downstream by setting the direction of the message. A typical processBuffer()-method might look similar to this.

```
Message* MyNode::processBuffer(Buffer* in_buffer)
{
  // some other code ...
```

```
  // switch between different states
  switch(state) {
  case CREATE_SOME_DOWNSTREAM_EVENT:
    // now node is really finished and will return end track

    // default direction for all messages is downstream,
    // so simply create new event and return it
    return
      new CEvent( ISomeInterface::create_Foo("some downstream event") );
    break;
  case CREATE_SOME_UPSTREAM_EVENT:
    // default direction for all messages is downstream,
    // so set direction to upstream and return it
    CEvent *e =
      new CEvent( ISomeInterface::create_Foo("some upstream event") );
    e->setDirection(UPSTREAM);
    return e;
    break;
  case DATA:
    // get out buffer
    // default direction for all messages is downstream,
    // so simply create new event and return it
    out_buffer = getNewBuffer(some_size);

    // do real work

    // some code ...

    // release in_buffer
    in_buffer->release();

    // return buffer
    return out_buffer;
    break;
  default:
    // error, return null
    ERROR_STREAM ("No internal state found!" << endl);
    in_buffer->release();
    return 0;
  }
}
```

7.5. Sink nodes

As a sink node is not connected to a successor, it should always return a null-pointer from its processBuffer()-method. Do not forget to release incoming buffers.

7.6. Source nodes and the producing-flag

As a source node is not connected to a predecessor, its processBuffer()-method is called with a null-pointer. But remember that a source might also receive upstream messages. If the working-flag would always be set to true for sources, these messages would never be dequeued. Reread the explanation about the working-flag if this is not clear to you. Therefore, another flag is used, the producing-flag. If this flag is true, processBuffer() is called with a null-pointer unless there is a message (i.e. a buffer or an event) that can be dequeued. This flag should mainly be set to true in one case. The node is a source node and wants to produce messages all the time while still being able to process new messages received. GenericSourceNode has set this flag to true by default, all other generic nodes to false. Some other node might set this flag to true or false depending on a timer (e.g. to achieve certain buffer output).

Notice: the access to the producing-flag is not guarded by a mutex. If you need this feature, you will have to implement a wrapper method.

7.7. Multiplexer nodes

A multiplexer node has one output and several inputs. Per default all its inputs are enabled. When trying to dequeue a message from the inputs of a multiplexer, all enabled inputs are tested for messages. If a message was dequeued from an input, another input will be preferred next time. More precesily, the next enabled input in the list of all inputs will be queried first. This strategy is called round robin. If an input is not enabled, it will not be queried, even if messages are available. The input from which the current message was dequeued can be queried by calling getCurrentRecvInputStream(). The following example shows a multiplexer with two inputs. Buffers will be dequeued in turn from these inputs.

```
Result MyNode::doInit()
{
  // some code ...

  // add two inputs, both are enabled per default
  addInputStream("input1");
  addInputStream("input2");

  // disable input2
  setRecvInputStreamEnabled("input2", false);

  return SUCCESS;
}

Message* MyNode::processBuffer(Buffer* in_buffer)
{
  if(!strcmp(getCurrentRecvInputStream(), "input1")) {
    // disable input1, enable input2
    setRecvInputStreamEnabled("input1", false);
    setRecvInputStreamEnabled("input2", true);

    // some code ...
```

```
  }

  if(!strcmp(getCurrentRecvInputStream(), "input2")) {
    // disable input2, enable input1
    setRecvInputStreamEnabled("input2", false);
    setRecvInputStreamEnabled("input1", true);

    // some code ...
  }
}
```

The next example shows an multiplexer with both of its inputs enabled. The input from which the current buffer was dequeued is queried by calling the getCurrentRecvInputStream()-method. None of the inputs will be disabled at any time. The round robin dequeueing strategy will then decide which input to chose.

```
Result MyNode::doInit()
{
  // some code ...

  // add two inputs, both are enabled per default
  addInputStream("input1");
  addInputStream("input2");

  return SUCCESS;
}

Message* MyNode::processBuffer(Buffer* in_buffer)
{
  if(!strcmp(getCurrentRecvInputStream(), "input1")) {
    // some code ...
  }

  if(!strcmp(getCurrentRecvInputStream(), "input2")) {
    // some code ...
  }
}
```

7.8. Demultiplexer nodes

A demultiplexer node has one input and several outputs. To specify to which output a message is to be sent when returning from processBuffer(), the setCurrentSendOutputStream()-method is used.

```
Result MyNode::doInit()
{
  // some code ...

  // add two inputs, both are enabled per default
```

```
  addOutputStream("output1");
  addOutputStream("output2");

  return SUCCESS;
}

Message* MyNode::processBuffer(Buffer* in_buffer)
{
  if( /* sent generated buffer or event to output1 ? */ ) {
    setCurrentSendOutputStream("output1");

    // some code ...

    return my_output_message;
  }

  if( /* sent generated buffer or event to output2 ? */ ) {
    setCurrentSendOutputStream("output2");

    // some code ...

    return my_output_message;
  }
}
```

7.9. Multiplexer-demultiplexer nodes

The GenericMuxDemuxNode simply combines a multiplexer and a demultiplexer. This means that this node supports more than one input and more than one output.

7.10. Developing a processBuffer()-method for a new node

That is all you need to know. And that is all we can tell you. Unfortunately, we cannot tell you how to program plug-in X. As always, there are several ways to realize a wanted functionality within a given programming model. Maybe the best way is to check existing nodes.

7.11. Summary

Maybe you might now think that all different types of nodes can be represented by a GenericMuxDemuxNode. That is right. In fact, the class GenericNode, which is the super class for all other generic nodes, is roughly speaking a node with n inputs and m outputs. The sub classes only set the correct numbers of inputs and outputs.

Well, now it is time to open your editor and do some coding. If everything is running and you have already created a small example application with your new node, you can go to the next step to see how to register your node with the NMM registry.

8. Step G: How can a node be registered?

The NMM registry stores information about nodes. It can be queried for nodes with certain features and it can make reservations for nodes. Every node is stored in the registry with name (e.g. "CameraNode"), type (e.g. SOURCE), its interfaces (e.g. ICamera) and its supported input and output formats stored in the property of the node.

Registering a node can be very simple. If a node does not pose any restrictions on the number of instantiations, you simply have to add one line of code. In general, nodes that do not rely on any special hardware (like a soundcard or a camera) can be instantiated as often as wanted. Or more precisely, as often as your host machine allows. For these nodes you have to add the following code in MyNode.cpp

```
#include "nmm/plugins/NMMRegisterPlugin.hpp"
#include "MyNode.hpp"

NMM_REGISTER_PLUGIN(MyNode, NMM, "MyNode")

namespace NMM {
...
```

Other nodes can only be instantiated a certain number of times. For example, a node representing a TV-grabber can only be instantiated once. In these cases, you have to write code to check how many nodes can be instantiated (e.g. how many devices are available). You then have to implement a new sub class of TPlugin where the nmmRegistryInit()- and the nmmFactoryInit()-method have to be re-implemented. This step is not described here in detail. See PlaybackNode or the Plugin1394 for an example.

Is is also important to assure that a node can be initialized (by calling init()) and deinitialized (be calling deinit()) without crashing since these two methods are called by the registry. In order to find a plugin, the registry has to be able to locate the plug-in. For this, if NMM_DEV_DIR is set, then the registry will search NMM_DEV_DIR/dev-lib/ for plug-ins. If NMM_DEV_DIR is not set, the registry will search $prefix/lib for plug-ins, where $prefix is the path where NMM was installed by calling 'make install'.

If a node depends on a device (such as /dev/dsp or /dev/video0), and there may be multiple devices usable by the node (for example /dev/video0, /dev/video1, ... for a TV grabber card), then the node should be registered once for each usable device. The number of instances for each registered node must be the number of times, each device can be opened simultaneously. This is typically 1, but may be higher (for example for sound devices). The TDevicePlugin class can simplify this task. You can register your node using the following code in MyNode.cpp

```
#include "nmm/multimedia/TDevicePlugin.hpp"
#include "MyNode.hpp"

static NMM::TDevicePlugin<NMM::MyNode> plugin("/dev/video", false, "MyNode");
NMM_REGISTER_SPECIAL_PLUGIN(MyNode, NMM, plugin)
```

This causes the plugin to check the devices /dev/device0, /dev/device1, ... and register a node for each device that can be opened. Once a device can't be opened, it stops checking for devices. For details about the constructor arguments of TDevicePlugin, see the doxygen documentation of this class.

Your node determines whether a device is useable or not. To be able to use TDevicePlugin, your node must implement the IDevice interface correctly. That is, it must be derived from IDeviceImpl, and it must provide a setDevice method which checks whether the device is useable and closes the device again. If the device can not be opened, then it must throw a DeviceNotFoundException. This exception is handled by the TDevicePlugin class. The setDevice method is only called in state CONSTRUCTED. If the device is useable, then the node must store the device name and use it in doInit to open the device. Note that usually it is enough to just check whether the device can be opened and to make sure that doInit fails if the device is not useable. Then TDevicePlugin will usually do the right thing. Otherwise you can still write a subclass of TDevicePlugin to implement extended behaviour. For an example of this, see PlaybackNode.