

NMM Interface Definition Language (NMM-IDL)

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

April 2010

Copyright (C) 2005-2010
Motama GmbH, Saarbruecken, Germany
<http://www.motama.com>

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being all sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found in the file COPYING.FDL.

THE DOCUMENT IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE DOCUMENT BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE DOCUMENT OR THE USE OR OTHER DEALINGS IN THE DOCUMENT.

This document describes the usage of the Interface Definition Language (IDL) and compilers used in the NMM project.

1. Motivation

Within the NMM environment, IDL files are used to describe two things: the interfaces an object supports and the events and object should be able to process instream. From such an IDL description a set of IDL compilers generates source code. Four files are generated. An hpp- and cpp-file for the client side, and

an `hpp`- and `cpp`-file for the server side. The client side is also called interface class; the server side is also called implementation class. An interface object will be used in an application program to control the server side object. The server class will be used as superclass for a class that wants to implement the interface.

2. IDL Files

2.1. Modules and Interfaces

IDL files are similar to C++ header files. In an IDL file, one or more interfaces are defined as part of a module. Each interface may inherit from other interfaces.

2.2. Methods

Each interface may also contain one or more methods. Each method has a return type, a name, and zero or more parameters.

2.3. Parameters

For return types and parameter types, NMM-IDL allows arbitrary types (only `&`-types are currently not supported). Additionally each parameter has to be specified as `in`, `out`, or `inout` parameter. `in` parameters are only sent to the method; `out` parameters are only received from the method; and `inout` parameters are sent to and received from the method. Additionally, a method can be declared to throw certain exceptions.

2.4. Modifier `instream`

Each method may be further specified by several modifiers. The `instream` modifier is used for `instream` events received by a NMM node that supports the interface. In this case, the method can be called from the application or an `instream` event can be generated by using the corresponding `create`-method. This method is named `create_<method name>` and will take the same additional parameters as the original method as defined in the IDL. An `instream` method should return a `Result`.

2.5. Modifier `limited`

The `limited` modifier can be used to limit the possible execution of a method to a certain state. If the object is not in this state, an exception is thrown (`IllegalStateException`).

2.6. Modifier required

The required modifier can be used to specify that a certain method has to be called with success before the current state can be left and another state can be reached. In this context, success means means two things. A method that returns a Result has to return SUCCESS. A method that does not return a Result has to be completed without throwing an exception. If a required method was not called with success, and the state of the object is changed, an exception is thrown (UnsuccessfulRequiredCallException). The limited and the restricted modifiers are extensively used to control the behavior of NMM nodes. To specify more than one state, the required modifier can be used several times.

2.7. Includes

IDL files can also be included in other idl files. This is done by adding an include statement to the beginning of the IDL file. This is needed if an interface inherits an interface defined in another IDL file.

2.8. .in Files

For adding specific C++ source code (such as already existing type definitions) to the source code generated by the IDL compilers from a given IDL file, so called .in files can be used. Such an in-file can include arbitrary valid C++ source code. Depending on the name of the in-file, this source code will be included in the beginning of the generated .hpp or .cpp file of the interface class, or the Impl.hpp or Impl.cpp file of the implementation class.

3. Invocation of IDL Compilers

As mentioned above four files are generated from one IDL file. The ihpp and icpp commands generate the corresponding interface files, the ihppimpl and icppimpl generate the files for the implementation side. Usually, the invocation of the IDL compilers is performed in a Makefile, see for example the interfaces-subdirectories.

4. Requirements for Cross-platform Development

As mentioned above, the NMM-IDL allows for using arbitrary parameters. If you want to define interfaces for NMM applications or NMM components (e.g. NMM nodes) that should operate on different platforms, such as little/big-endian or 32/64 bit, you are only allowed to use portable and well-defined types in interface definitions written in NMM-IDL.

In the include file called "nmm/nmm_types.hpp", NMM defines following basic data types (or, POD types, as an acronym for "plain old data").

```
nmm_char
nmm_uchar

nmm_int16
nmm_int32
nmm_int64

nmm_uint16
nmm_uint32
nmm_uint64

nmm_float
nmm_double

nmm_bool
```

All interfaces originally provided with NMM make use of these types. In addition, complex types that are composed out of basic data types should also use the nmm_ types.

5. Examples

The first example is an IDL file for a simple interface for a camera.

```
module NMM {
  interface ICamera {
    void setDevice(in string device) raises(DeviceNotFoundException)
      required(INode::CONSTRUCTED, INode::INITIALIZED) limited(INode::CONSTRUCTED);
    void setBrightness(in nmm_int32 value)
      limited(INode::ACTIVATED);
    void getBrightness(out nmm_int32 value)
      limited(INode::ACTIVATED);
    pair < nmm_int32, nmm_int32 > getBrightnessRange()
      limited(INode::ACTIVATED);
    void trySetBrightness(inout nmm_int32 value);
      limited(INode::ACTIVATED);
  }
}
```

Here, an interface ICamera is declared as part of the module NMM. A camera needs to be configured with a device, which has to be set in the CONSTRUCTED state of a node to be able to reach the INITIALIZED state. This method is also limited to this state. If the given device is invalid, an exception is thrown. The brightness of the camera might be set and queried in the ACTIVATED state. Notice, that the getBrightness method does not return the current brightness. Instead the corresponding out parameter will be set by this

method. The range of possible values for the brightness can be queried with another method. In this case, a return value is specified. The trySetBrightness method will try to set the brightness to the given value. It will then set the value to the actual value which depends on the possible range of values.

The second example is an IDL file for two simple file handler interfaces.

```
module NMM {
  interface IFileHandler {
    void setFilename(in string filename)
      raises(FileNotFoundException, OtherException)
      required(INode::CONSTRUCTED, INode::INITIALIZED) limited(INode::CONSTRUCTED);
  }
  interface IFileHandlerWithEOS : IFileHandler {
    Result endOfStream()
      instream limited(INode::STARTED) limited(INode::ACTIVATED);
  }
}
```

The setFilename method is similar to the setDevice method described above. The only difference is that two different exceptions are specified. The IFileHandlerWithEOS interface inherits from IFileHandler and supports one instream event. Upon receiving an endOfStream event, an object implementing this interface can take certain actions, for instance close the current file. This method is limited to the STARTED and the ACTIVATED state. Since it is only specified as instream, an application or another node can use the IFileHandlerWithEOS::create_endOfStream method to create such an event. This method is generated automatically by the IDL compilers.

The third example is an in-file called "IMyInterface.hpp.in" which contains a C++ statement for including the serialization operators of the time utility classes. When generating the source files from the IDL file called "IMyInterface.idl", this statement will automatically be inserted in the beginning of the file "IMyInterface.hpp".

```
#include "nmm/comm/serialize/SerializeUtilsTime.hpp"
```