

# XML Configurations for the NMM Multimedia-Box

**Marc Klein**

**mklein@graphics.cs.uni-sb.de**

**June, 14th 2002**

Copyright (c) 2002-2005  
NMM work group,  
Computer Graphics Lab,  
Saarland University, Germany,  
<http://www.networkmultimedia.org>

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 no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found in the file COPYING.FDL.

## 1. Introduction

The MMBox Menu structure consists of two parts. The first one is the XML configuration file that describes the general menu structure like main menu, submenus, look of the menu entries. The second part is the implementation part. That's the code that is executed if a menu is activated.

## 2. The XML configuration file

Like every XML document the configuration file consists of different tags. The root of a MMBox document is the `<configuration>` tag. In this configuration part the `<menu>` tags can be placed. This can look like this:

```
<?xml version="1.0"?>
```

```

<configuration>

  <menu id="MainMenu" columns="2">
    <entry
      index = "1"
      on    = "../resources/mmbox/icon/dvd.2.png"
      off   = "../resources/mmbox/icon/dvd.1.png"
      x     = "86"
      y     = "50">DVDPlayer</entry>

    <entry
      index = "2"
      on    = "../resources/mmbox/icon/tv.2.png"
      off   = "../resources/mmbox/icon/tv.1.png"
      x     = "446"
      y     = "50">TV</entry>

    <entry
      index = "3"
      on    = "../resources/mmbox/icon/audiocd.2.png"
      off   = "../resources/mmbox/icon/audiocd.1.png"
      x     = "86"
      y     = "338">AudioCD</entry>

    <entry
      index = "4"
      on    = "../resources/mmbox/icon/mp3.2.png"
      off   = "../resources/mmbox/icon/mp3.1.png"
      x     = "446"
      y     = "338">MP3</entry>
  </menu>

  <menu id="DVDPlayer">
    <entry
      index = "1"
      on    = "../resources/mmbox/icon/audiocd.2.png"
      off   = "../resources/mmbox/icon/audiocd.1.png"
      x     = "86"
      y     = "50">menuid_01</entry>

    <entry
      index = "2"
      on    = "../resources/mmbox/icon/mp3.2.png"
      off   = "../resources/mmbox/icon/mp3.1.png"
      x     = "246"
      y     = "338">menuid_02</entry>

  </menu>

</configuration>

```

You can see that two menus (MainMenu and DVDPlayer) are defined with various submenus. Each menu has got an identifier (id attribute) that is used later in the implementation part to reference the menus,

but also to reference the menus within the XML file. The menus have <entry> tags that describes the submenus and the entry data contains the menu id of the menu that should be activated if the entry is activated. With these ids the complete menu tree can be build. Additonal information about the menu entries can be given by attributes that describe which bitmap (png file) should used for the entry and the position of the bitmap (x and y attributes). The off attribute contains the bitmap that is used for unselected menus, the on attribute is used for the selected menus. The index attribute is a help to navigate within the menu. The automatic generate instances of the menu implementation use this to select the correct item if a navigation key (up, down, left, right) is pressed. To handle a menu which submenu icons are placed in lines and columns, a special "column" attribute in the menu tag can be set. For example we have a menu called MainMenu that looks like this:

```

*****      *****
* DVD *      * TV *
*****      *****

*****      *****
* CD *       * MP3 *
*****      *****

```

Then we must set column=2 to be sure that we can navigate correctly within the menu.

### 3. Using XML menus in an own application

First we need a MMBBoxApplication Object.

```

// Create new MMBBoxApplication Object
app = new MMBBoxApplication();

```

This object has got the "control" methods like Up, Down, Left, Right, Back, etc. You can look at the include file to have an overview of all methods, but it's easy to add own methods to this object if needed. Now we must have a way to call this methods in our own application. This can be done directly by calling the method ( app->Up, app->Down,...), but a smart way to start this methods is to let they called by an event dispatcher:

```

// Create event dispatcher
dispatcher = new EventDispatcher();
dispatcher->registerEvent( "KEY_Up",      new TEDObject0<MMBoxApplication>(
    app, &MMBoxApplication::Up ) );
dispatcher->registerEvent( "KEY_Down",    new TEDObject0<MMBoxApplication>(
    app, &MMBoxApplication::Down ) );
dispatcher->registerEvent( "KEY_Left",    new TEDObject0<MMBoxApplication>(
    app, &MMBoxApplication::Left ) );
dispatcher->registerEvent( "KEY_Right",   new TEDObject0<MMBoxApplication>(
    app, &MMBoxApplication::Right ) );
dispatcher->registerEvent( "KEY_Return",  new TEDObject0<MMBoxApplication>(
    app, &MMBoxApplication::Return ) );
dispatcher->registerEvent( "KEY_q",      new TEDObject0<MMBoxApplication>(
    app, &MMBoxApplication::Back ) );

```

This means that if the event dispatcher received events (simply strings) such as "KEY\_UP...", it starts the corresponding method that was registered before. The next question is:

*Where the events come from?*

Objects that generate such events we call producers. Producers can be for example: keyboard, remote control, mouse, etc. So far there are two producers, one that sends keyboard events from a X display and one that sends events from a remote control:

```
/* X Producer */
Xevents* Xproducer = new Xevents();
Xproducer->registerDisplay( display->getID() );
Xproducer->readConfig( "configuration.xml" );
Xproducer->connectTo( dispatcher );
Xproducer->start();

/* Remote Control Producer */
LircEvents* LircProducer = new LircEvents();
LircProducer->readConfig( "remote.conf" );
LircProducer->connectTo( dispatcher );
LircProducer->start();
```

First we must read a configuration file ( producer->readConfig(...) ) where the mapping from keys to events is described. Normally this should be a xml file, but for the remote control we use the configuration file that is part of the lirc package.

Example for the producer configuration file:

```
<keyboard>
  <event key="a">KEY_a</event>
  <event key="b">KEY_b</event>
  ...
  <event key="UP">KEY_Up</event>
  <event key="DOWN">KEY_Down</event>
  ...
</keyboard>
```

After we read the config file, we have to connect the producer with the dispatcher and start it.

So far, the MMBoxApplication can handle up, down, left, right, Return(select menu), Back commands. You can navigate with the arrow keys and select a menu by pressing Return. With "Q" you come back to the parent menu.

After that we must load a XML configuration file and handle some errors:

```
if ( app->ParseXML( "configuration.xml", osd ) == FAILURE ) {
    cerr << "ParseXML Error" << endl; }
```

The ParseXML method needs the filename of the XML file and a pointer to a Multiple OSD Node because the MMBoxApplication uses this to draw the menu items on it. After successfully loading the XML file, we must initialize the app and tell which menu is the main menu by giving the menu id to the init method:

```
app->init( "MainMenu" );
```

After this call the menu is completely generated with the information of the XML file. You can also navigate within the menus. To make a real application, we must implement the functions "behind" these menu icons yet.

## 4. Implementation Part

To add functionality to our application we must do some actions (cd, dvd player) after selecting a menu. We can do this by adding our own "states" to the application. In our case, "states" are the implementation of a functionality. The MMBoxApplication Class always knows which state is active and so it always knows where it should send the key event so that only one state has the input to any time. How do we create a new state? - This can be done by writing a new class that is derived from MMBoxState, this looks like that:

```
class MyState : public MMBoxState {
public:
    Result Up( MMBoxApplication* );
    Result Down( MMBoxApplication* );
    Result Left( MMBoxApplication* );
    Result Right( MMBoxApplication* );
    Result Return( MMBoxApplication* );
    Result Back( MMBoxApplication* );
    Result initialize( MMBoxApplication* );
    Result uninitialize( MMBoxApplication* );
};
```

Here we must implement what should happen if we get a key up, down, etc. message by overwriting the corresponding methods. Each method has got a pointer to the MMBoxApplication, so that we can save global data there if needed or access the Multiple OSD Node to draw text or icons. There are two special methods. The first one is called "initialize", it is called at the beginning always when we switch to the given state, it could be compared with the constructor of a class. The second one is called "uninitialize" and it is called before we switch to another state. It could be compared with the destructor of a class. At last we must tell the MMBoxApplication that it should use our new state with the "registerState" method:

```
app->registerState( "mystateID", new MyState );
```

This method needs a string to identify the class and a pointer to a MMBoxState. Note: all registered states will be deleted automatically, that means no delete MyState is needed at the end of the program. How we see above, each state has got a string identifier. So we can simply switch between states by calling the changeState method with the new id:

```
changeState( app, newID )
```

