# Bellagio OpenMAX Component writer's guide

by
Giulio Urlini

Ver. 0.1

28 July 2006

## REVISION HISTORY

| VERSION | DATE | AUTHORS | COMMENTS |
|---|---|---|---|
| 0.1 | 28th July 2006 | G. Urlini | First draft |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

OpenMAX is a registered trademark of the Khronos Group. All references to OpenMAX components in this whitepaper are referenced from the publicly available OpenMAX IL specification on the Khronos web-site at:

http://khronos.org/openmax

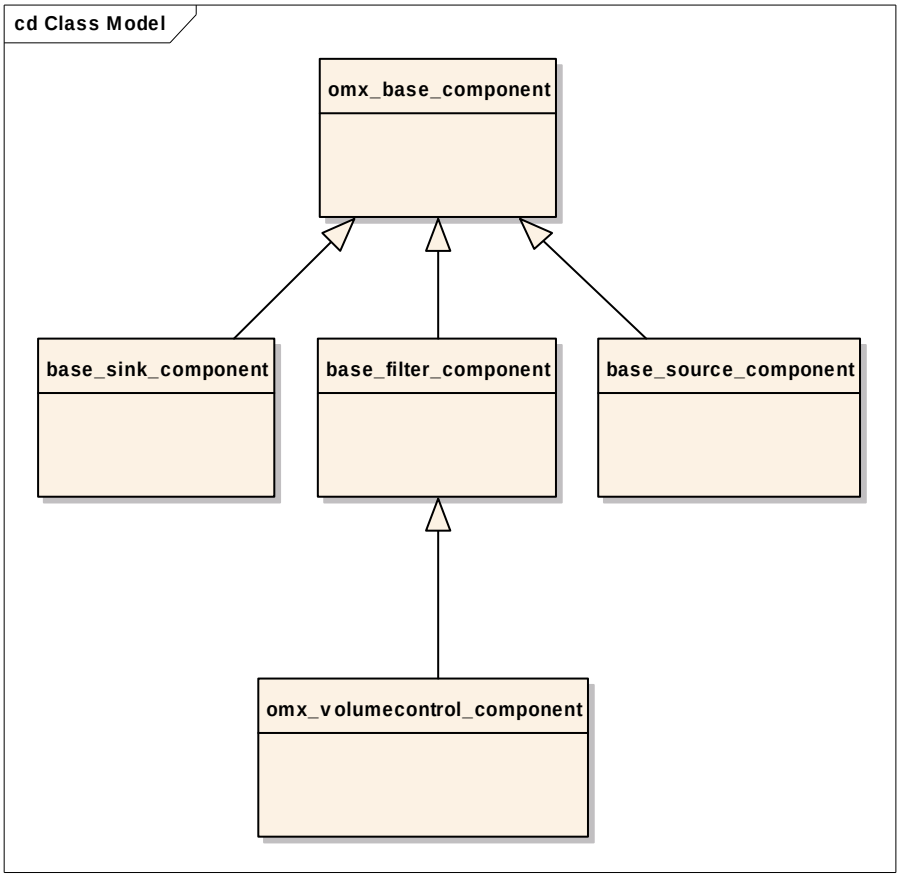
# Introduction

This guide aims at explaining how OpenMAX components can be built based on the Bellagio opensource distribution available at http://sourceforge.net/projects/omxil.
The text is based on Bellagio 0.2, but it is anticipated that some details may change in the future with new project releases.
OpenMAX components are written in C but an object-oriented approach has been used to avoid code duplication, so that common OpenMAX functions are implemented in a so-called “base component” and can be overridden by derived components.

This guide is divided in two main parts. In the first chapter the base components are presented, with the description of the main functionalities and the relations between base and first-derived components (sink, source and filter). The typical final component is also presented.
In the second part a real example is presented, based on the volume control component that is included in the Bellagio distribution.

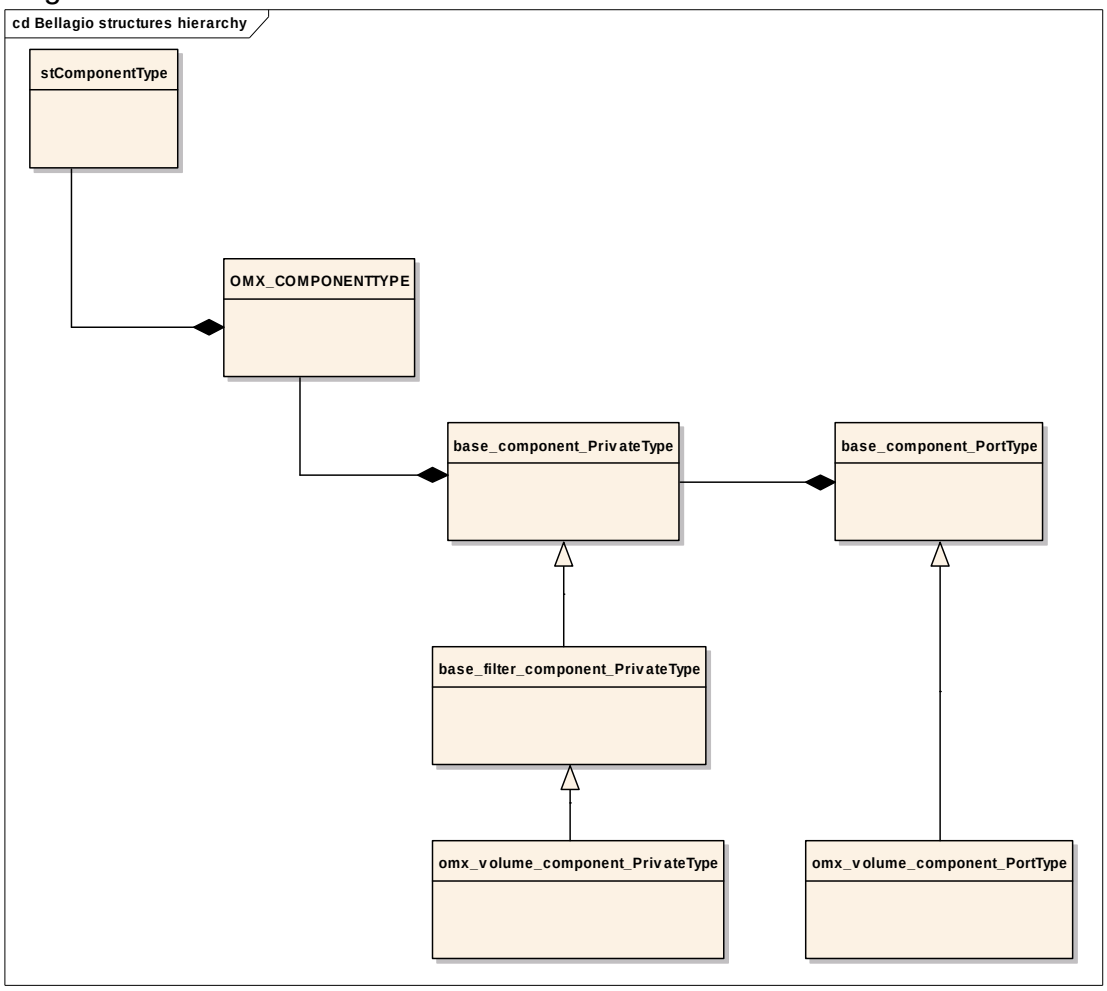
# 1. OpenMAX hierarchy

The Bellagio OpenMAX component hierarchy can be described as in the following class diagram:



## *OpenMAX Bellagio data structures*

Each component in the Bellagio implementation is described as a set of parameters and methods contained in a structure named stComponentType. This structure contains the functions pointers to be filled by each component that can be executed by the core. This structure contains also the OpenMAX standard structure used to describe the component, named OMX_COMPONENTTYPE.
Inside the OMX_COMPONENTTYPE structure it is present a pointer to a private structure. This pointer is used by the Bellagio components to store private information needed by the component and this structure is extended for derived components, so that specific fields can be added. In the following diagram the relations between structures are shown.




In this diagram is presented only the hierarchy of a two ports component, from the base component through the filter to the final component (in this example a volume control component).

### Override mechanism

The structures presented in the preceding paragraph contain several function pointers, which refer to specific set of functionalities.
The stComponentType contains the functionalities that the core must use directly, the constructor, destructor and the message handler. All these

functions are used by the Bellagio core and are implementation specific, not OpenMAX defined.
The base_component_PrivateType contains a set of function pointers that contain functionalities to be used only inside the component.
Finally the OpenMAX standard structure contains a set of pointers to the standard OpenMAX functions.
The override mechanism consists in replacing a function pointed by a member of this structure with a different (derived) function. This mechanism is more flexible than a direct override, because the derived function can execute specific code AND call the parent function, if needed. In any case all the calls to these functions are made through function pointers.

## *OpenMAX base component*

The base component implements four sets of functionalities, based respectively on:

- the allocation and default value assignment of the base `stComponentType` structure
- The functions pointed by the `stComponentType` structure. These functions are grouped here because they are the functions used directly by the core. They are the constructor, the destructor and the message handler entry points.
- The function pointed by the base private structure, that are initialization functions, buffer allocation and de-allocation in case of tunneling, state transition core function.
- The OpenMAX standard functions.

A complete list of these functions, divided by groups, with a short description, is presented here.

### Main structure allocation

The function that allocates the main structure, the `stComponentType` structure, is `base_component_CreateComponentStruct`. This function MUST be called by the final component in the register template mandatory function.

### Main structure function pointers

These functions are called directly by the core. They are:

- `base_component_Constructor`: the constructor fills the stComponent structure with default values. It does not fill any field that belongs to derived classes.
- base_component_Destructor: the destructor function de-allocates everything allocated in the base component.
- `base_component_MessageHandler`: this function is the message handler for each component. It reads any request from the user through the `SendCommand` OpenMAX API, and handles it. This function does not need any override, basically, but if some special message handling is needed, the developer can implement a derived message handler function, and call this base function after its special

handling. In the components included in the Bellagio distribution, no special message handling is implemented

### Base component private structure function pointers

The following functions are used by the base component to init and de-init, to change the state of the component and to allocate and de-allocate the buffers in case of tunneling.

- `base_component_Init`: is usually overridden by the final component. The final function should call this base function at the beginning of its execution, and after this call it implements the custom initialization related to the specific component.
- `base_component_Deinit`: this function is the base disposal function. It is overridden by the final component implementation, as the Init function. The final function executes its specific code, and at the end it should call the base Deinit function.
- `base_component_DoStateSet`: this function handles the state transition requested by the user. It does not need any override, except for special handling possibly needed by the final component. In the components included in the Bellagio distribution, no override of this function is provided.
- `base_component_AllocateTunnelBuffers:` This function provides the allocation of needed buffers for the port that is tunneled. It does not need any override, except for special handling possibly needed by the final component. In the components included in the Bellagio distribution, no override of this function is provided.
- `base_component_FreeTunnelBuffers:` This function provides the de-allocation of buffers for the port that is tunneled. It does not need any override, except for special handling possibly needed by the final component. In the components included in the Bellagio distribution, no override of this function is provided.

### OpenMAX standard functions

The OpenMAX standard functions implemented in the base components are the following. The full description of these functions is out of the scope of this document. It is only documented when these functions are used as they are, and when they need an override by the derived components.

- `base_component_GetComponentVersion`: no override needed.
- `base_component_GetParameter`: this function is overridden by the final component for each parameter specific to the component. For the basic parameters this function is called.
- `base_component_SetParameter`: this function is overridden by the final component for each parameter specific to the component. For the basic parameters this function is called.
- `base_component_GetConfig`: this function is overridden by the final component for each configure value specific to the component. For the basic configure values this function is called.

- `base_component_SetConfig`: this function is overridden by the final component for each configure value specific to the component. For the basic configure values this function is called.
- `base_component_GetExtensionIndex`: no override needed.
- `base_component_GetState`: no override needed.
- `base_component_UseBuffer`: no override needed, for standard ways to allocate memory.
- `base_component_AllocateBuffer`: no override needed, for standard ways to allocate memory.
- `base_component_FreeBuffer`: no override needed, for standard ways to allocate memory.
- `base_component_SetCallbacks`: no override needed.
- `base_component_SendCommand`: no override needed.
- `base_component_ComponentDeInit`: no override needed.
- `base_component_EmptyThisBuffer`: no override needed.
- `base_component_FillThisBuffer`: no override needed.
- `base_component_ComponentTunnelRequest`: no override needed.

## *The derived classes*

There are three first level derived classes: the filter component, the sink component and the source component. In the future the mixer and splitter components will be added.
For Filter, sink and source:

### Filter component

Common overrides:

- `base_filter_component_Constructor`: the base constructor is overridden, in order to fill all the port parameters, specific for sink, source and filter, and to add the function pointer for the central buffer handling functionality (`BufferMgmtFunction`) and the port flush functionality(`FlushPort`)

The new functionalities added are:

- `base_filter_component_BufferMgmtFunction`: this function is executed in a separate thread, and is responsible for receiving the buffers from a queue, filled by the `EmptyThisBuffer` function, and put the filtered data in the output buffers that are in another queue filled by the `FillThisBuffer` function. The main filter functionality is executed by the **`BufferMgmtCallback`** that is implemented in the final filter class.
- `base_filter_component_FlushPort`: this function allows the flushing of buffers for input and output port in case of port disabled, or component switched from idle to loaded state. This function can work only with the default buffer management function.

### Source and sink components

The same functions presented for the filter are implemented in the sink and source base components. The only difference is related to the number of ports. For instance the `BufferMgmtCallback` prototype in case of sink component has only an input buffer parameter, and the source component function has only the output buffer parameter.

## *Final class (from filter) volume control*

The final component must implement three more functions in order to work with the default components hierarchy. The new functions are:

- `omx_volume_component_register_template`: this function allows the component in the Bellagio framework to be loaded by the core. This mechanism is specific for this implementation, but in any case the library loading mechanism is not covered by the OpenMAX spec, and is let free to each implementation.
- `omx_volume_component_DomainCheck`: this function is used to check the domain for tunneling of two components. Since the tunneling functions are implemented in the base classes, the final component must check if its domain is compatible with the one given by the function parameter, and return an error if there is some domain incompatibility.
- `omx_volume_component_BufferMgmtCallback`: this function is the central filtering functionality. It receives a buffer in input and produced a buffer in output. This function can be used only with the default buffer management function. It implements a specific filtering mechanism, used by the ffmpeg library for instance. If a different mechanism is needed, the basic buffer management function, this function and the flush functions must be overridden.

### Overrides

The final component must override the following functions, in order to be OpenMAX compliant, and usable inside the Bellagio hierarchy framework.
The needed function to comply with the hierarchy is:
`omx_volume_component_Constructor`: this function must fill the function pointers for any needed standard function, like the set/get parameter config specific functions, any other override not specified in this guide but needed by the final component. Finally it must fill any specific filed related to this component. For instance the volume control component must specify a gain value that is used as default value by the component, and changed dynamically by the SetConfig specific function.

The functions to override for the OpenMAX compliance are:
omx_volume_component_GetConfig
omx_volume_component_SetConfig
omx_volume_component_GetParameter
omx_volume_component_SetParameter
In these functions the OpenMAX parameters specific to the final component must be added.

# 2. Use case: volume control example

In this chapter is presented an example of final component construction based on Bellagio hierarchy. The final component realized is the volume control component.
The following steps are necessary for insert the files in the current Bellagio make framework.
Step 1: create a new directory in the `src/components` directory of Bellagio distribution.
Step 2: create in the new directory three files, `volumecontrol.c`, `volumecontrol.h` and `Makefile.am`.
Step 3: add to the `configure.in` file, in the root directory, the line `src/component/volumecontrol/Makefile` between the other makefile lines in the bottom part of the file.
Step 4: add in the `src/components/Makefile.am` file the line `SUBDIRS+=volumecontrol`
The content of the new files is described in the following paragraphs.

## *Makefile.am*

The `Makefile.am` file should be written as follows:

```
omxvolcontroldir = $(libdir)/omxilcomponents

omxvolcontrol_LTLIBRARIES=libomxvolcontrol.la

libomxvolcontrol_la_SOURCES = omx_volume_component.c
noinst_HEADERS = omx_volume_component.h

INCLUDES = -I../../../include/ -I../../ -I./ -
I../../base/
```

With this makefile the volume control component will be compiled in a separate library. This library will be put in a subdirectory of the installation directory named omxcoponents. This is the default behavior for components compilation, and also for the dynamic allocation done with the omxregister command.

## *volumecontrol.c*

This file should include the volumecontrol.h file and the omxcore.h file.
The functions to be implemented are presented here in detail.

### omx_volume_component_register_template

this function is called at the beginning, when the library is loaded. It happens for example when an application that links the library is launched. In the Bellagio distribution the library is not linked by the application, but loaded dynamically when the application needs it, through the core loading mechanism.

```
void __attribute__ ((constructor))
omx_volume_component_register_template() {
```

the base component structure, handled by the core, is the following. Is allocated by a base function, the component create struct functions. See the base component section for details.

```
stComponentType *component;
component = base_component_CreateComponentStruct();
```

When the stComponent structure has been allocate and filled with the default values, the specific fields are filled. The first one is the standard OpenMAX name:

```
component->name = "OMX.volume.component";
```

The other mandatory functions are the specific contructor (explained later in this section), and the set/get config parameters functions.

```
component->constructor = omx_volume_component_Constructor;
component->omx_component.SetConfig = omx_volume_component_SetConfig;
component->omx_component.GetConfig = omx_volume_component_GetConfig;
component->omx_component.SetParameter =
omx_volume_component_SetParameter;
component->omx_component.GetParameter =
omx_volume_component_GetParameter;
```

The final component MUST fill also the field that contains the number of ports.

```
component->nports = 2;
```

the final step is to call the register function of the core, that adds this component to the possible open max components available. This list is provided by the core through the standard OpenMAX API calls.

```
register_template(component);
}
```

## omx_volume_component_DomainCheck

This function is called by the base setup tunnel functionality, in order to check if two components are compatible in relation to the domain. In this function a parameter that represents the remote port is passed to this function.

```
OMX_ERRORTYPE omx_volume_component_DomainCheck(
OMX_PARAM_PORTDEFINITIONTYPE pDef){
```

The implementation of this function is left to the developer. It must return an error if for some reason the developer thinks that the compatibility between the current port to be tunneled and the remote port parameter is not reached. A possible implementation is the following.

The domain is first checked. In the example the domain is audio, and if the remote port is not an audio port, the compatibility is not satisfied.

```
if(pDef.eDomain!=OMX_PortDomainAudio)
   return OMX_ErrorPortsNotCompatible;
```

The type of coding is checked. In this case no check is performed, since that this volume control applies to uncompressed data. In other cases, such as an mp3 decoder, a check for encoding format should be implemented.

```
else if(pDef.format.audio.eEncoding == OMX_AUDIO_CodingMax)
   return OMX_ErrorPortsNotCompatible;
```

The domain check is satisfied, and the return can be `OMX_ErrorNone`.

```
return OMX_ErrorNone;
}
```

## omx_volume_component_BufferMgmtCallback

This function is called in the middle of the `base_filter_component_BufferMgmtFunction` function. In the case of

a sink component, the buffer management callback will be called in the `base_sink_component_BufferMgmtFunction` function.
This function represents the central elaboration of the filter. The parameters used are the stComponent that describes the component, the input buffer and the output buffer. The implementation is left to the developer, but here a useful example is presented. This case is simple because the size of the data remains unchanged during the processing, since that this component is not an encoder or decoder, but simply a filter.

```
void         omx_volume_component_BufferMgmtCallback(stComponentType*
stComponent, OMX_BUFFERHEADERTYPE* inputbuffer, OMX_BUFFERHEADERTYPE*
outputbuffer) {

int i;
```

In the decoded stream that this component can handle, each sample is contained in two buffers.

```
int sampleCount = inputbuffer->nFilledLen / 2;
omx_volume_component_PrivateType* omx_volume_component_Private =
stComponent->omx_component.pComponentPrivate;
```

For every sample in input, the value of the sample is modified with the gain value of this component.

```
for (i = 0; i < sampleCount; i++) {
 ((OMX_S16*) outputbuffer->pBuffer)[i] = (OMX_S16)
 (((OMX_S16*) inputbuffer->pBuffer)[i] *
(omx_volume_component_Private->gain / 100.0f));
}
```

When the entire input buffer has been processed, the output size is assigned, as the OpenMAX rule specifies, and the execution of the function can be concluded.

```
outputbuffer->nFilledLen = inputbuffer->nFilledLen;
inputbuffer->nFilledLen=0;
}
```

Any synchronization issue related to receive or send buffer is handled in the `MgmtFunction` of the parent component, in this example the base filter.

## omx_volume_component_Constructor

This function overrides the same base function. Each level of this hierarchy implements a constructor. Each constructor must call at the end the constructor of the parent. The parameter stComponent has been instantiated in the register function.

```
OMX_ERRORTYPE omx_volume_component_Constructor(stComponentType*
stComponent) {
OMX_ERRORTYPE err = OMX_ErrorNone;
omx_volume_component_PrivateType* omx_volume_component_Private;
omx_volume_component_PortType *inPort,*outPort;
OMX_S32 i;
```

The constructor must allocate the component private structure, and check eventually if the memory is available or not.

```
stComponent->omx_component.pComponentPrivate = calloc(1,
sizeof(omx_volume_component_PrivateType));
if(stComponent->omx_component.pComponentPrivate==NULL)
return OMX_ErrorInsufficientResources;
omx_volume_component_Private = stComponent-
>omx_component.pComponentPrivate;
```

If the component needs specific port description structure, it must allocate it here. In other cases, when the default structure is enough, the allocation can be left to the base components.

```
if (stComponent->nports && !omx_volume_component_Private->ports) {
  omx_volume_component_Private->ports = calloc(stComponent->nports,
    sizeof (base_component_PortType *));
  if (!omx_volume_component_Private->ports) return
    OMX_ErrorInsufficientResources;
  for (i=0; i < stComponent->nports; i++) {
    omx_volume_component_Private->ports[i] = calloc(1,
    sizeof(omx_volume_component_PortType));
    if (!omx_volume_component_Private->ports[i]) return
      OMX_ErrorInsufficientResources;
  }
}
```

The parent constructor can be called.

```
err = base_filter_component_Constructor(stComponent);
```

This instruction should be replicated here after the parent constructor.

```
omx_volume_component_Private = stComponent-
>omx_component.pComponentPrivate;
```

The volume control component should define the following parameters in order to be openMAX compliant. The same list of parameter must be added in the set/get parameter section.

```
omx_volume_component_Private->ports[OMX_BASE_FILTER_INPUTPORT_INDEX]-
>sPortParam.eDomain = OMX_PortDomainAudio;
omx_volume_component_Private->ports[OMX_BASE_FILTER_INPUTPORT_INDEX]-
>sPortParam.format.audio.cMIMEType = "raw";
omx_volume_component_Private->ports[OMX_BASE_FILTER_INPUTPORT_INDEX]-
>sPortParam.format.audio.bFlagErrorConcealment = OMX_FALSE;
omx_volume_component_Private-
>ports[OMX_BASE_FILTER_OUTPUTPORT_INDEX]->sPortParam.eDomain =
OMX_PortDomainAudio;
omx_volume_component_Private-
>ports[OMX_BASE_FILTER_OUTPUTPORT_INDEX]-
>sPortParam.format.audio.cMIMEType = "raw";
omx_volume_component_Private-
>ports[OMX_BASE_FILTER_OUTPUTPORT_INDEX]-
>sPortParam.format.audio.bFlagErrorConcealment = OMX_FALSE;

inPort = (omx_volume_component_PortType *)
omx_volume_component_Private->ports[OMX_BASE_FILTER_INPUTPORT_INDEX];
outPort = (omx_volume_component_PortType *)
omx_volume_component_Private-
>ports[OMX_BASE_FILTER_OUTPUTPORT_INDEX];

setHeader(&inPort->sAudioParam,
sizeof(OMX_AUDIO_PARAM_PORTFORMATTYPE));
inPort->sAudioParam.nPortIndex = 0;
inPort->sAudioParam.nIndex = 0;
inPort->sAudioParam.eEncoding = 0;

setHeader(&outPort->sAudioParam,
sizeof(OMX_AUDIO_PARAM_PORTFORMATTYPE));
outPort->sAudioParam.nPortIndex = 1;
outPort->sAudioParam.nIndex = 0;
outPort->sAudioParam.eEncoding = 0;
```

The specific internal parameters for the final component should be filled here. In this example the only needed value is the gain of the volume control.

```
omx_volume_component_Private->gain = 100.0f;
```

The critical function pointers must be assigned. These functions have been already described above.

```
omx_volume_component_Private->BufferMgmtCallback =
omx_volume_component_BufferMgmtCallback;
omx_volume_component_Private->DomainCheck  =
&omx_volume_component_DomainCheck;
```

No other constructing operations are needed at this point. The function can return.

```
return err;
}
```

### omx_volume_component_SetParameter and omx_volume_component_GetParameter

These two functions must provide the support for the OpenMAX parameters specifc of the final component. In the volume control example the parameters supported by both these functions are:

```
OMX_IndexParamAudioInit
OMX_IndexParamAudioPortFormat
```

The parameter that is supported only by the `GetParameter` function is:

```
OMX_IndexParamAudioPcm
```

The other parameters are supported by the parent components.

### omx_volume_component_SetConfig and omx_volume_component_GetConfig

These functions support only the config value:

```
OMX_IndexConfigAudioVolume
```

That is the gain of the volume control.

The other config values are supported by the parent components.