

# OpenCOM API Specification

Michael Clarke, Nikos Parlavantzas, Geoff Coulson, Gordon Blair

Distributed Multimedia Research Group,  
Department of Computing,  
Lancaster University,  
Bailrigg, Lancaster,  
LA1 4YR, U.K.

E-mail: [mwc, geoff, gordon]@comp.lancs.ac.uk,  
N.Parlavantzas@lancaster.ac.uk

## 1. Introduction

This document reflects the current state of the OpenCOM API describing the configuration and meta interfaces available for developers to use and also the interfaces they must implement themselves in order to produce an OpenCOM compliant component.

## 2. Definitions

An *OpenCOM (OCM) component* is a COM component enhanced with receptacles, lifecycle and meta-interfaces thus implementing reflective (re)configuration capabilities.

An *OpenORB (OPO) component* is an OpenCOM component that exposes additional interfaces that implement middleware.

*OpenORB* is a general term meaning a **sample** collection of OpenORB components collectively implementing componentised, reflective and (re)configurable middleware in a single OS process. Our current OpenORB implementation is *GOPI-based*, i.e. made up of OpenORB components that define and implement the middleware interfaces defined by GOPI using original GOPI code.

*OpenCOM* is a general term meaning **the** collection of OpenCOM components and interfaces that collectively implement the mechanisms needed to extend COM with reflective (re)configuration capabilities. These are the OpenCOM component itself, MetaEnvironment and MetaEncapsulation components, the Delegator component and the interfaces `ILifeCycle` and `IReceptacles`.

## 3. The OpenCOM Component

The OpenCOM (OCM) component is created using the normal `::CoCreateInstance()` COM operator. This allows an OpenORB configuration to be bootstrapped; all OpenCOM compliant OpenORB components are then created from the OpenCOM component. Each process (OpenORB configuration) houses only one unique OpenCOM component (it is implemented as a singleton). The OpenCOM component currently implements three interfaces, `IOCM_OCM` (the OpenCOM API), `IMetaEncapsulation2` (a helper interface for each `IMetaEncapsulation` interface implemented by an OpenCOM compliant component) and `OCM_IDebug` for dumping the contents of the system graph it maintains.

## 4. OpenCOM Compliant Components

All OCM compliant components currently contain (but in the future will aggregate) standard implementations of the `IMetaEnvironment` and `IMetaEncapsulation` interfaces (in the form of the `MetaEnvironment` and `MetaEncapsulation` components). In addition, they implement the `ILifeCycle` and `IReceptacles` interfaces themselves.

## 5. Interfaces

### 5.1 Pre Implemented Interfaces

#### 5.1.1 IOCM\_OCM

```
interface OCM_IOCM : IUnknown {
    HRESULT createInstance([in] REFCLSID rclsid, [out] IUnknown **ppIUnknown,
                          [in, string] const unsigned char *name);
    HRESULT deleteInstance([in] IUnknown *pIUnknown);
    HRESULT connect([in] IUnknown *pIUnkSource, [in] IUnknown *pIUnkSink,
                  [in] REFIID iid, [in] OCM_RecpID_t recpID,
                  [out] OCM_ConnID_t *pConnID);
    HRESULT disconnect([in] OCM_ConnID_t connID);
    HRESULT getConnectionInfo([in] OCM_ConnID_t connID,
                              [out] OCM_ConnInfo_t **ppConnInfo);
    HRESULT freeConnectionInfo([in] OCM_ConnInfo_t *pConnInfo);
    HRESULT enumComponents([out] IUnknown* *ppComps[], [out] int *pcElems);
    HRESULT getComponentName([in] IUnknown *pIUnknown, [out] unsigned char **ppName);
    HRESULT getComponentPIUnknown([in, string] const unsigned char *name,
                                   [out] IUnknown **ppIUnknown);
    HRESULT getComponentCLSID([in] IUnknown *pIUnknown, [out] CLSID *pclsid);
};
```

**createInstance** creates an OpenCOM compliant component instance based on the specified CLSID. The instance can be named with a string, e.g. for compile time access to the instance, but can be unnamed if NULL is passed as the name parameter.

**deleteInstance** deletes an instance based on its pointer to IUnknown. This method also invokes the instances `ILifeCycle::shutdown` method during the process.

**connect** connects the source's receptacle to the sink's interface. A unique connection identifier is returned to manipulate the connection at a later stage. This method invokes the source instances `IReceptacles::connect` method so that connection specific actions can take place and the interface pointer can be stored in the appropriate receptacle. `AddRef()` is automatically called on the interface pointer.

**disconnect** disconnects a previously established connection using the identifier returned when the connection was established. This method invokes the instances `IReceptacles::disconnect` method so that connection specific actions can take place and the interface pointer can be deleted from the appropriate receptacle. `Release()` is automatically called on the interface pointer.

**getConnectionInfo** returns a block of information describing a previously established connection.

**freeConnectionInfo** releases the memory internally allocated to hold the information block returned by `getConnectionInfo`.

**enumComponents** returns a `cElems` sized array of IUnknown pointers to all the component instances currently in the system graph. The array should be freed using `CoTaskMemFree`.

**GetComponentName** returns an instances name based on its process unique pointer to IUnknown. An error is returned if the component has no name.

**GetComponentPIUnknown** returns a named instance's pointer to IUnknown

**GetComponentCLSID** returns an instances CLSID (component type) based on its pointer to IUnknown.

### 5.1.2 IMetaEncapsulation

```
interface IMetaEncapsulation : IUnknown
{
    HRESULT enumConnsToIntf([in] REFIID riid,
                           [out] OCM_ConnID_t *ppConnsToIntf[],
                           [out] int *pcElems);
    HRESULT enumConnsFromRecp([in] REFIID riid,
                              [out] OCM_ConnID_t *ppConnsFromRecp[],
                              [out] int *pcElems);
    HRESULT enumRecps([out] OCM_RecpMetaInfo_t *ppRecpMetaInfo[],
                     [out] int *pcElems);
    HRESULT enumIntfs([out] IID *ppIntf[],
                     [out] int *pcElems);
};
```

**enumConnsToIntf** returns a cElems sized array of connection identifiers detailing all the connections established on the specified interface of the current component instance.

**enumConnsFromRecp** returns a cElems sized array of connection identifiers detailing all the connections established by the receptacle of the specified interface type on the current component instance.

**enumRecps** returns a cElems sized array of OCM\_RecpMetaInfo\_t structures defining the type of all the receptacles hosted by the current component instance in terms of their IID and pointer storage class (single, multiple, multiple with context). The array should be freed using CoTaskMemFree.

**enumIntfs** returns a cElems sized array of interface IIDs defining the type of all the interfaces hosted by the current component instance. The array should be freed using CoTaskMemFree.

### 5.1.3 IMetaEnvironment

```
interface IMetaEnvironment : IUnknown
{
    HRESULT createDelegator([in] OCM_DelegatorType_t delType,
                           [in] REFIID riid, [out] OCM_IDelegator **ppOCM_IDelegator);
    HRESULT deleteDelegator([in] OCM_DelegatorType_t delType,
                            [in] OCM_IDelegator *pOCM_IDelegator);
};
```

**createDelegator** creates a delegator of the given type on the specified interface of the current component instance and returns a pointer to that delegators control interface. The component is typically located by its name, though it can be located dynamically by its CLSID and piUnknown (followed by a reverse lookup to establish its static name). The delegator intercepts on a per component interface basis (i.e. all methods within the interface are intercepted) by overwriting the interfaces vtable pointer (lpvtbl), in the components object layout, with its own pointer. The original vtable pointer is stored by the delegator to allow the originally intended method to be invoked after delegation. Once interception has been performed on a particular component interface, all users of that interface are subject to delegated calls; even those that bound to the interface before interception.

**deleteDelegator** deletes a delegator of the specified type from its resident interface. Delegators of different types have different deletion requirements.

### 5.1.3 IDelegator

```
interface IDelegator
{
    HRESULT addPreMethod([in] const unsigned char *DLLName,
                        [in] const unsigned char *methodName);
    HRESULT delPreMethod([in] const unsigned char *methodName);
    HRESULT addPostMethod([in] const unsigned char *DLLName,
                        [in] const unsigned char *methodName);
    HRESULT delPostMethod([in] const unsigned char *methodName);
    HRESULT viewPreMethods([out] unsigned char *methodNames[]);
    HRESULT viewPostMethods([out] unsigned char *methodNames[]);
    HRESULT deleteDelegator(void);
};
```

**addPreMethod** dynamically adds a named pre-processing method (from the named DLL) onto the list of pre-processing routines attached to the specified delegator component. The method is attached at back of the list and will be invoked in order behind any other list members.

**delPreMethod** deletes the specified pre-processing routine from the list of pre-processing routines.

**addPostMethod** adds a post-processing routine to a delegator component similarly to **addPreMethod()**.

**delPostMethod** deletes the specified post-processing routine from the list of post-processing routines.

**viewPreMethods** returns a pointer to an array of strings representing the current list of pre-processing routines attached to the specified delegator component.

**viewPostMethods** returns a pointer to an array of strings representing the current list of post-processing routines attached to the specified delegator component.

**deleteDelegator** deletes the current (this) delegator. This method is called indirectly from `IMetaEnvironment::deleteDelegator`.

## 5.2 Developer Implemented Interfaces

### 5.2.1 ILifeCycle

```
interface OCM_ILifeCycle : IUnknown
{
    HRESULT startup([in] OCM_IOCM *pOCM_IOCM);
    HRESULT shutdown(void);
};
```

**startup** allows a component instance to take lifecycle actions after it has been created. Like component connection, it is the responsibility of client code (not the OCM component) to invoke this method. Separating creation from lifecycle in this way allows the instance to have its receptacles connected up in the interim period and thus startup can call on its receptacle based interfaces for service as necessary. A pointer to the OCM component instance is passed in to allow the instance to store it as a member variable thus allowing convenient access from member methods.

A component's **shutdown** method allows an instance to take lifecycle actions while it is being deleted. It is called from the OpenCOM component's `deleteInstance` method when the

instance is to be deleted.

## 5.2.2 IReceptacles

```
interface OCM_IReceptacles: IUnknown
{
    HRESULT connect([in] void *pSinkIntf, [in] REFIID riid,
                   [in] OCM_ConnID_t provConnID, [in] OCM_RecpID_t recpID);
    HRESULT disconnect([in] REFIID riid, [in] OCM_ConnID_t connID,
                      [in] OCM_RecpID_t recpID);
};
```

**connect** allows a component instance to take connection specific actions when a connection between a receptacle and an interface is being established. The connection identifier is passed down from `IOCM_OCM::connect`.

**disconnect** allows a component instance to take connection specific actions when a connection between a receptacle and an interface is being torn down.