CoolMon 2 PlugIn SDK

Written by Christian Aaangel
Last Revised on December 18th, 2004
Written for Core Version: 0.6.1.0

DISCLAIMER
The information in this document and CoolMon 2 may be subject to change, we believe we have found a rather great way for the plugins and coolmon to interact. But we also have to think about the fact that we are still very early in the development of coolmon and we might have missed something. In order to allow CoolMon 2, to run older plugin, as the development progresses, we have included the RequiredCoreVersion function; this must return the core version number which is required to run this plugin.

CoolMon 2 No longer supports plugins designed for Core version 0.2.9.9 or Lower.

IMPORTANT INFORMATION
Developers that use a **different development environment than Delphi** to develop plugins **must use core version 0.4.7.0 or higher to run those**, since prior version are incompatible due to a flaw in the plugin engine.

Plugins (COM) created in a .NET development language currently won't work in coolmon 2, the install methods, for a COM object, between win32 and .NET is different, but we are working on a solution. Currently we don't know how long it will take before they will work.

Introduction

First let me start by thanking your for taking the time to download the software development kit (SDK) for CoolMon 2. The most innovative step from CMOne to CoolMon 2 is the ability to do plugins and with your help creating them, CoolMon 2 should easily become the most expandable system monitor available.

The CoolMon 2 plugin system uses COM technology which lets us tie in just about any form of program code, from any development language, while still keeping a certain object feel. This has made it easier for me to develop the plugin engine because a COM object truly is something you plug in at one end as oppose to raw procedure code which you normally have in DLL's.

Understanding the principals

In my humble opinion it would probably be easier for you to develop the most efficient plugin if you know how CoolMon 2 ticks.

When CM2 starts up it will load all registered plugins (general as well as visual) into memory. Then it will read through the CML configuration file. Please consider this example.

```
<form name="MyForm">
 <parameters>
  <color>clskyblue</color>
  <left>10</left>
  <top>100</top>
  <width>100</width>
  <height>25</height>
  <transparent>no</transparent>
 </parameters>
 <content>
  <visual name="Text">
   <parameters>
    <bold>yes</bold>
    <top>0</top>
    <left>0</left>
    <width>100</width>
    <height>25</height>
    <background>clskyblue</background>
    <font>255</font>
    <color>clwhite</color>
    <size>12</size>
   </parameters>
   <content>
    <sensor name="Test.Random Numbers" update="1">
     <parameters>
      <range>1000</range>
     </parameters>
    </sensor>
   </content>
  </visual>
 </content>
</form>
```

This is not a complete CML, just a part of it, more specifically a form and it's contents. We will not go into depths on how the CML syntax is or how it works here because I could probably write an entire paper just on that alone. We will just run through what CM2 does when it meet certain nodes. This example should produce an output looking like this. **256**

When CM2 hits the form node it will create the form with the parameters CM2 finds in the parameters block, it will create a pointer to this form and thus linking the form to all sub nodes.

Then when it hits the visual tag it will, using the pointer to the parent form, create a visual object (IE. an instance of the visual plugin desired, in this case the text plugin). When doing this CM2 will pass an hDC to the form on which the plugin will have to draw output. Then it will send the parameters to the plugin.

When the sensor tag is reached, CM2 will create an item in the variable manager which serves as a host for all dynamic values. It will then make sure that the plugin is polled for the desired value when required. When it is polled the variable manager will trigger an event that will cause the visual plugin to update its current value and then redraw accordingly.

Getting started

Before you actually can code a plugin, you will have to sit down and decide what type of plugin you wish to develop. Read below and then determine what you want to do is best archived.

General Plugin

A general plugin will serve up information for CM2 by exporting one or more sensors to CM2, when writing this type of plugin you must give some thought to what you want to make available for CM2 to poll. It would also be wise to think about how many polling sensors you want. What parameter each of these will require in order to function. In theory you can export just one sensor sending different information back depending entirely on parameters. But it would be smart (and easier for the user to use) if you export a sensor for every major thing and then use parameters for minor thing about it, say altering data, this could for example be rounding a number.

Visual plugin

Visual plugins are what they sound like; they are the plugin that will actually serve up the information to the user. As oppose to general plugins everything in visual plugins are parameter driven. Colour, layout, style etc. everything must be driven by parameters. This also means that there is a lot of parameter interpreting work when writing a visual plugin. Furthermore visual plugins can handle events, CoolMon 2 will pass on any events triggered within a visual plugins rectangle. In order to write event code in your plugins you must use the interface ICM2VisPlug2.

cmlX plugin

The cmlX plugin will enable you to do some altering already at the CML level. This plugin type will enable you to handle a certain CML tag in the configuration file. When CM2 hits this tag it automatically is passed on to your plugin. With this plugin type you can actually return CML to extend the configuration. Creating these plugin will require mucho work since the parameters send to this plugin type is CML, this in turns means that you must incorporate an XML parser if you want to make something really high powered.

Versioning in CoolMon 2

All version numbers in coolmon 2 must be built as following

```
XXXX
| | |
| |  2 digit Build version
|  Minor Version
Major version
```

So if you want to give a version number of 1.0 you will actually need to set the version to 1000 and opposite a 1.5.2.0 would be 1520.

The Basic Plugin Structures

CoolMon 2 supports three types of plugins, three plugin design that will do three very different things but nevertheless basically look somewhat alike. They all use a basic plugin set that will enable coolmon to do things that must be done in all three plugin types. The following functions must be present in all types of plugins.

Function Name(OUT Name: Pchar)…

Provide the name for the plugin.

Function Version(OUT Version: word)…

Provide the version number for the plugin. (Please refer to the version chapter earlier in this document).

Function RequiredCoreVersion(OUT RCV: Word)…

You must provide the CM2 core version your plugin requires in order to run properly. (Please refer to the version chapter earlier in this document).

Function GetLastErrorMsg(OUT Msg: PChar)…

Whenever a function call to your plugin fails, this function will be called by CM2 in order to get some sort of error description of the error.

Function Init…

Ran when the plugin is being loaded, in the visual plugin this function will also have a parameter called PaintDC, it will hold the device content that your plugin must draw on.

Function Terminating…

Ran when the plugin is about to be shutdown

Function Setup…

Ran when the user click “setup” on your plugin, if you do not have anything to set up you might wish to display a dialog saying that there is nothing to set up, so the user doesn’t click repeatedly on setup in CM2. This can be done in 2 ways, one you can code it manually or you can save time and **simply return “E_NOTIMPL”** instead of “S_OK” as the function result. This will cause CM2 to display a standard nothing to setup dialog.

The Info List

During development of the framework it became apparent that we needed to be able to attach some information to all plugins. This information is not handled by CoolMon 2; it's merely a way to communicate basic information to the user of the plugin. This information can be anything from credits to plugin requirements to copyright information. There is not set standard or requirement for what information you must provide (however we do recommend always providing the 4 mentioned below).

Information Name	***Information Value***
Programmed by	<your name>
Created	<date of plugin creations>
Updated	<date of last update>
Copyright	<your copyright information>

You can of course add as many field as your like to your list, these are just what I feel all plugin should have as minimum.

Function GetInfoCount(OUT InfoCount: Integer)…

Provide the length of your info list in "InfoCount", if the above list it would be 4.

Function GetInfo(Index: Integer; OUT InfoName: PChar; OUT InfoVar: PChar)…

Provide the info for the record at index, in the above list if CM2 was asking for 1 (provided in index) it you should set InfoName to "Created" and InfoVar to "<date of plugin creations>" since the list starts at zero.

Making a General Plugin

When making a general plugin you must, along with the basic plugin structure, export the following functions.

Function SetParentHandle(Handle: THandle)…

Some plugin might need a handle to the parent form in order to push message boxes etc. This you can get in the function, it will be ran when your plugin is loaded. Just remember to save the value.

Function GetSensorCount(OUT SensorCount: Integer)…

Provide the sensor count your plugin exports.

Function GetSensorNameOnly(Index: Integer; OUT SensorName: PChar; OUT SensorStyle: Byte)…

A quick informative function ran at start up, you must provide the "SensorName" and "SensorStyle" for the sensor at "index". If you sensor returns a Numeric value set sensor style to "0", if it returns a Non-Numeric value set it to "1" or if it's unknown return "2". The last one should only be used for sensors which output type is determined by parameters.

Function GetSensor(Index: Integer; Parameters: PChar; OUT SensorName: PChar; OUT SensorStyle: byte; OUT SensorString: PChar; OUT SensorValue: Double)…

The actual function used when CM2 is polling a sensor.
Index is the sensor CM2 is polling
parameters are of course the parameters passed. Since CM2 has to send multiple parameters using only one string it will be send like this "aName=aValue|Min=0|Max=100", then you will have to cut it down into relevant bits.
Now to the "OUT" properties
SensorName = The sensor name
SensorStyle = 0 if the result is a Number, 1 if it's Non-numeric – Unknown (2) is not allowed in the actual polling of the sensor.
SensorString = The result if it's Non-numeric
SensorValue = The result if it's numeric

The Default and Optional Lists

The GetDefault, GetDefaultCount, GetOptional and GetOptionalCount are used to pass information about the various parameters.

You will have to provide it for a certain sensor, which will be passed in “Index”. The Default list are the parameters that the sensor MUST have in order to function properly and the optional list is what can be used but aren’t crucial to the sensor

Other than that the lists works just like the info list, except they also have a description item which of course should be used to inform the user of what exactly that parameter will do.

Function GetDefaultCount(SensorIndex: Integer; OUT DefaultCount: Integer)…

Provide the Default parameter count for the sensor at “sensorindex”.

Function GetDefault(SensorIndex: Integer; Index: Integer; OUT Name: Pchar; OUT Value: PChar; OUT Description: PChar)…

Provide the list item at “Index” for the sensor at “SensorIndex”. “Name” is the name of the parameter, “Value” is the default value for the parameter and “Description” is a small description of the parameter.

Function GetOptionalCount(SensorIndex: Integer; OUT OptionalCount: Integer)…

Provide the Optional parameter count for the sensor at “sensorindex”.

Function GetOptional(SensorIndex: Integer; Index: Integer; OUT Name: Pchar; OUT Value: PChar; OUT Description: PChar)…

Provide the list item at “Index” for the sensor at “SensorIndex”. “Name” is the name of the parameter, “Value” is the default value for the parameter and “Description” is a small description of the parameter.

Using the Editor Support Interface

You can extend your plugin to take advantage of the features in the WYSIWYG editor. To do that you must implement the Editor Support interface (for General Plugins) into your plugin. In order to do that you must export the following functions.

Function ShowSensor(Index: Integer; OUT ShowAt: Byte)…

This function will allow you to hide a sensor from the editor. For example sensors you want to use for debugging your plugin. Index will refer to the sensor number, ShowAt will either be

saDesignTime = 0	The sensor will show in the editor
saRunTimeOnly = 1	The sensor will not show in the editor

Function ObjIns(SensorIndex: Integer; ParameterName: PChar; OUT ValueType: Byte; OUT ValueRange: PChar)…

The function will allow you to take advantage of the object inspector (OI) in the WYSIWYG editor.

SensorIndex	Then Index of sensor in question
ParameterName	The name of the parameter to which the valuetype and Valuerange must be supplied
ValueType	0 = Unknown. 1 = Fixed, causes the OI to present a dropdown list. 2 = Range, will cause the OI to only accept a number between the range you define. 3 = Color, OI will only accept a color value. 4 = Font, OI will present a dropdown list with fonts 5 = Files, OI will present a file dialog
ValueRange	This depends on what the Valuetype is 0 = set it to an empty string 1 = send your fixed values, separated with a line break eg. “Item1|Item2|…|…|…” 2 = like above but first minimum value then line break followed by the maximum value eg. “0|255” 5 = can be used to make a file filter - “Bmp|jpg|jpeg|png”

Making a Visual Plugin

When you write a visual plugin you must, along with the basic plugin structure, export the following functions.

Function SetDC(CONST NewDC: THandle)…

Should the plugin for some reason be required to draw on a different form/surface then this function will be called to give your plugin a new hDC.

Function SetDimensions(Left, Top, Width, Height : Integer)…

This will give you the four values that give your plugin the rectangle within it must draw.

Function SetConfigString(NewConfigString: PChar)…

Called upon creating the visual object and again if one of the parameter values has changed. The string will contain multiple parameters formed like this.
"aName=aValue|Min=0|Max=100"

Function Redraw…

Called when, CM2 needs your plugin to redraw. It's called right after an update or when the entire parent form needs to redraw, for what ever reason.

Function Update…

Called when, the value is updated. If Sensorstyle is "0" then the value is numeric and you updated value is in "NewValue" if the sensorstyle is "1" you're value is non-numeric and should be extracted from the "NewString". If Sensorstyle is anything else your plugin would be wise to trigger an error. Do this by returning "E_FAIL" as the function result.

The Default and Optional Lists

The GetDefault, GetDefaultCount, GetOptional and GetOptionalCount are used to pass information about the various parameters.

The Default list are the parameters that the plugin MUST have in order to function properly and the optional list is what can be used but aren't crucial to the plugin.

Other than that the lists works just like the info list, except they also have a description item which of course should be used to inform the user of what exactly that parameter will do.

Function GetDefaultCount(OUT DefaultCount: Integer)…

Provide the Default parameter count.

Function GetDefault(Index: Integer; OUT Name: Pchar; OUT Value: PChar; OUT Description: PChar)…

Provide the list item at "Index". "Name" is the name of the parameter, "Value" is the default value for the parameter and "Description" is a small description of the parameter.

Function GetOptionalCount(OUT OptionalCount: Integer)…
Provide the Optional parameter count.

Function GetOptional(Index: Integer; OUT Name: Pchar; OUT Value: PChar; OUT Description: PChar)…
Provide the list item at "Index". "Name" is the name of the parameter, "Value" is the default value for the parameter and "Description" is a small description of the parameter.

Using the Editor Support Interface
You can extend your plugin to take advantage of the features in the WYSIWYG editor. To do that you must implement the Editor Support interface (for Visual Plugins) into your plugin. In order to do that you must export the following functions.

Function ObjIns(ParameterName: PChar; OUT ValueType: Byte; OUT ValueRange: PChar)…
The function will allow you to take advantage of the object inspector (OI) in the WYSIWYG editor.

ParameterName	The name of the parameter to which the valuetype and Valuerange must be supplied
ValueType	0 = Unknown. 1 = Fixed, causes the OI to present a dropdown list. 2 = Range, will cause the OI to only accept a number between the range you define. 3 = Color, OI will only accept a color value. 4 = Font, OI will present a dropdown list with fonts 5 = Files, OI will present a file dialog
ValueRange	This depends on what the Valuetype is 0 = set it to an empty string 1 = send your fixed values, separated with a line break eg. "Item1|Item2|…|…|…" 2 = like above but first minimum value then line break followed by the maximum value eg. "0|255" 5 = can be used to make a file filter - "Bmp|jpg|jpeg|png"

Writing in events

One of the most anticipated things in CM2 is the usage of events, this will really be where CM2 will set it self apart from other similar programs. **Events like executing a standard windows command can be done by using a tag in the CML so you don't need to write in code to execute a windows command**. The event code you should write in would be plugin extending events. Such as a where a leftclick would expand or collapse a dropdown panel. It's all controlled by the following function.

Function OnEvent(Sender: Pchar; EventType: Byte; ModKeys: Byte; X, Y: Integer; Var CallBack: Pchar)...

This function is only supported in the ICM2VisPlug2 interface so you will need to use the ICM2Visplug2 interface instead of the ICM2VisPlug in your plugin. Now for the explanation

Sender	Currently this one isn't used. But the plan is to use this as a plugin sender when you send an event to other plugin than the one clicked
EventType	This is the type of event that has been triggered **etUnknown** – Unknown event **etDblClick** – Doubleclick (currently doesn't work) **etMouseDownLeft** – Left mouse button is pressed **etMouseDownRight** – Right mouse button is pressed **etMouseDownMiddle** – Middle mouse button is pressed **etMouseUpLeft** – Left mouse button is released **etMouseUpRight** – Right mouse button is released **etMouseUpMiddle** – Middle mouse button is released **etMouseWheelAny** – Mouse scrollwheel is pushed in either direction **etMouseWheelUp** – Mouse scrollwheel is pushed forward **etMouseWheelDown** – Mouse scrollwheel is pushed backwards **etMouseMove** – Mouse is moving within plugin area.
ModKeys	Used to track modifier keys when the event is triggered. **mksUnknown** – Unknown modifier state **mksNone** – No modifier keys where pressed **mksShift** – Shift pressed **mksAlt** – Alt pressed **mksCtrl** – Control pressed **mksAltShift** – Alt and Shift are pressed **mksAltCtrl** – Alt and Control are pressed **mksCtrlShift** – Control and shift are pressed **mksAltCtrlShift** – Alt, Control and Shift are pressed **mksAny** – It doesn't matter what combo is pressed. This one is in theory only used in the CML when writing event code.

X	The X coordinate, relative to the form, the event occurred
Y	The Y coordinate, relative to the form, the event occurred
CallBack	This will allow you to send a command back to CM2 for further processing. The following CallBack commands are supported **Repaint** – Will cause CM2 to redraw the visual object.

Making a cmlX Plugin

When you write a cmlX plugin you must, along with the basic plugin structure, export the following functions.

Function TagName(OUT Tag: PChar)…

This function will tell coolmon what CML Tag that will be passed onto your plugin.

Function Perform(Attribute: PChar; Parameters: PChar; OUT ResultString: PChar; OUT ResultType: Byte)…

This function is called when a tag has been encountered in the plugin and needs to be resolved. The attribute line is passed like this

"aName=aValue|anotherName=anotherValue|…|…"

The parameters are the underlying CML from the cmlX node. So if you want to use these parameters you will need an XML parser to resolve it. The result of your function must be passed back in "ResultString". The "ResultType" determines how CM2 should handle the result.

rtNone = 1	CoolMon 2 will not handle the return value
rtSensor = 2	CoolMon 2 will handle the return value as normal value (will automatically be converted to numeric if possible)
rtCML = 3	CoolMon 2 will treat the return value as CML, thus extending the CML tree.

Implementing Auto Update

You can add your plugin to CM2 auto updater. This is a program that will update all installed CM2 components (Programs & Plugins) that is installed on a computer. But in order to do this you must implement the ICM2AutoUpdate Interface, then you must store an UPD (update) file and the plugin itself, the plugin itself (and support files) must be compressed in RAR to allow the auto updater to unpack it, if you don't have the RAR system, I recommend – [www.winrar.com or www.7-zip.org]. The server that you store the UPD file and the RAR file must support direct downloading.

If you don't want to support Auto Update, you can just leave out the interface.

The ICM2AutoUpdate interface can be implemented in all plugin types. If you don't define this interface the auto updater will list your plugin update status as "Auto Update Not Supported"

Function UpdateFileURL(OUT URL: Pchar)…

This function is called when the plugin is installed, it will then save the value extracted to the registry. The URL must be the path to the UPD file that contains the update information for your plugin, should you return an empty string here the auto updater will list your plugin update status as "Auto Update Disabled".

The layout of the UPD file

The UPD file must contain at least 2 lines… The first must be the version number (please read "Versioning in CM2" above, for a detailed description on versioning in CoolMon 2) of the plugin available online. The second line must be a link to the plugin available online (please remember that this file must be in rar format). The rest of the file will make up the "what's new" section, displayed in the updater.

Example

```
0500
http://www.coolmon.org/cm2/core.rar
What's new
* One Item
* Another Item
```

Credits

In order to get this SDK out to as many developers as possible, we want this SDK (the test plugins) translated into as many development languages as possible. Below is the list of the people that has translated a plugin into another language. Should your development language not appear here and you want to help translate it. Please mail us at contact@coolmon.org

Translation for Delphi (Original)
General: Christian Aaangel
Visual: Christian Aaangel

Translation for C++
General: Olle Westman
Visual: Olle Westman

Translation for C# (C Sharp)
General: Olle Westman
Visual: Olle Westman

Appendix A – The General Plugin Interface

```
ICM2GenPlug = Interface
  ['{4B1160AE-8CD7-4070-B8B5-5CD46A31E965}']
   Function Name(OUT Name: Pchar): HResult; stdcall;
   Function Version(OUT Version: word): HResult; stdcall;
   Function RequiredCoreVersion(OUT RCV: Word): HResult; stdcall;
   Function SetParentHandle(Handle: THandle): HResult; stdcall;
   Function GetLastErrorMsg(OUT Msg: PChar): HResult; stdcall;
   Function GetInfoCount(OUT InfoCount: Integer): HResult; stdcall;
   Function GetInfo(Index: Integer; OUT InfoName: PChar; OUT InfoVar: PChar):
HResult; stdcall;
   Function GetSensorCount(OUT SensorCount: Integer): HResult; stdcall;
   Function GetSensorNameOnly(Index: Integer; OUT SensorName: PChar; OUT
SensorStyle: Byte): HResult; stdcall;
   Function GetSensor(Index: Integer; Parameters: PChar; OUT SensorName: PChar;
OUT SensorStyle: byte; OUT SensorString: PChar; OUT SensorValue: Double):
HResult; stdcall;
   Function Init: HResult; stdcall;
   Function Terminating: HResult; stdcall;
   Function Setup: HResult; stdcall;
   Function GetDefaultCount(SensorIndex: Integer; OUT DefaultCount: Integer):
HResult; stdcall;
   Function GetDefault(SensorIndex: Integer; Index: Integer; OUT Name: Pchar;
OUT Value: PChar; OUT Description: PChar): HResult; stdcall;
   Function GetOptionalCount(SensorIndex: Integer; OUT OptionalCount: Integer):
HResult; stdcall;
   Function GetOptional(SensorIndex: Integer; Index: Integer; OUT Name: Pchar;
OUT Value: PChar; OUT Description: PChar): HResult; stdcall;
 End;
```

The editor support interface for the general plugin

```
ICM2EditorSupport10 = Interface
  ['{A8C505F5-6812-4EB8-9319-71A3B9DC6E2F}']
  Function ShowSensor(Index: Integer; OUT ShowAt: Byte): HResult; stdcall;
  Function ObjIns(SensorIndex: Integer; ParameterName: PChar; OUT ValueType:
                  Byte; OUT ValueRange: PChar): HResult; stdcall;
End;
```

Appendix B – The Visual Plugin Interfaces

```
ICM2VisPlug = Interface
  ['{E68EB965-D026-4646-B527-B0B3B11C057A}']
   Function Name(OUT Name: Pchar): HResult; stdcall;
   Function Version(OUT Version: word): HResult; stdcall;
   Function RequiredCoreVersion(OUT RCV: Word): HResult; stdcall;
   Function GetLastErrorMsg(OUT Msg: PChar): HResult; stdcall;
   Function GetInfoCount(OUT InfoCount: Integer): HResult; stdcall;
   Function GetInfo(Index: Integer; OUT InfoName: PChar; OUT InfoVar: PChar):
    HResult; stdcall;
   Function SetDC(CONST NewDC: THandle): HResult; stdcall;
   Function SetDimensions(Left, Top, Width, Height : Integer): HResult; stdcall;
   Function SetConfigString(NewConfigString: PChar): HResult; stdcall;
   Function Redraw: HResult; stdcall;
   Function Update(SensorStyle: Byte; NewString: PChar; NewValue: Double):
    HResult; stdcall;
   Function Init(PaintDC: THandle): HResult; stdcall;
   Function Terminating: HResult; stdcall;
   Function Setup: HResult; stdcall;
   Function GetDefaultCount(OUT DefaultCount: Integer): HResult; stdcall;
   Function GetDefault(Index: Integer; OUT Name: Pchar; OUT Value: PChar; OUT
    Description: PChar): HResult; stdcall;
   Function GetOptionalCount(OUT OptionalCount: Integer): HResult; stdcall;
   Function GetOptional(Index: Integer; OUT Name: Pchar; OUT Value: PChar; OUT
    Description: PChar): HResult; stdcall;
 End;

ICM2VisPlug2 = Interface
  ['{D337B641-BA21-4E6B-9B2E-363155E84041}']
   Function Name(OUT Name: Pchar): HResult; stdcall;
   Function Version(OUT Version: word): HResult; stdcall;
   Function RequiredCoreVersion(OUT RCV: Word): HResult; stdcall;
   Function GetLastErrorMsg(OUT Msg: PChar): HResult; stdcall;
   Function GetInfoCount(OUT InfoCount: Integer): HResult; stdcall;
   Function GetInfo(Index: Integer; OUT InfoName: PChar; OUT InfoVar: PChar):
    HResult; stdcall;
   Function SetDC(CONST NewDC: THandle): HResult; stdcall;
   Function SetDimensions(Left, Top, Width, Height : Integer): HResult; stdcall;
   Function SetConfigString(NewConfigString: PChar): HResult; stdcall;
   Function Redraw: HResult; stdcall;
   Function Update(SensorStyle: Byte; NewString: PChar; NewValue: Double):
    HResult; stdcall;
   Function Init(PaintDC: THandle): HResult; stdcall;
   Function Terminating: HResult; stdcall;
   Function Setup: HResult; stdcall;
   Function GetDefaultCount(OUT DefaultCount: Integer): HResult; stdcall;
   Function GetDefault(Index: Integer; OUT Name: Pchar; OUT Value: PChar; OUT
    Description: PChar): HResult; stdcall;
   Function GetOptionalCount(OUT OptionalCount: Integer): HResult; stdcall;
   Function GetOptional(Index: Integer; OUT Name: Pchar; OUT Value: PChar; OUT
    Description: PChar): HResult; stdcall;
   Function OnEvent(Sender: Pchar; EventType: Byte; ModKeys: Byte; X, Y:
    Integer; Var CallBack: PChar): HResult; stdcall;
 End;
```

The editor support interface for the visual plugin

```
ICM2EditorSupportVis10 = Interface
  ['{7A17F3E7-64F5-43AB-BF77-B4278D74653B}']
  Function ObjIns(ParameterName: PChar; OUT ValueType: Byte; OUT ValueRange:
   PChar): HResult; stdcall;
 End;
```

Appendix C – The cmlX Plugin Interface

```
ICM2CMLXPlug = Interface
  ['{2D6FBF95-EC3F-4138-BBBC-321463AD940E}']
   // Declare Basics
   Function Name(OUT Name: Pchar): HResult; stdcall;
   Function Version(OUT Version: word): HResult; stdcall; // Uses MMRB Notation
   Function RequiredCoreVersion(OUT RCV: Word): HResult; stdcall; // Uses MMRB
Notation - Please refer to CM2Plugtypes for details
   Function SetParentHandle(Handle: THandle): HResult; stdcall;
   // Declare Error commands
   Function GetLastErrorMsg(OUT Msg: PChar): HResult; stdcall;
   // Declare Info system
   Function GetInfoCount(OUT InfoCount: Integer): HResult; stdcall;
   Function GetInfo(Index: Integer; OUT InfoName: PChar; OUT InfoVar: PChar):
HResult; stdcall;
  // Declare "Event" types
   Function Init: HResult; stdcall;
   Function Terminating: HResult; stdcall;
   Function Setup: HResult; stdcall;
   // Plugin Speficic
   Function TagName(OUT Tag: PChar): HResult; stdcall;
   Function Perform(Attribute: PChar; Parameters: PChar; OUT ResultString:
PChar; OUT ResultType: Byte): HResult; stdcall;
  End;
```

Appendix D – The Auto Update Interface

```
ICM2AutoUpdate = Interface
 ['{164E31E4-CD60-4E78-A0A4-CD28178B1899}']
 Function UpdateFileURL(OUT URL: Pchar): HResult; stdcall;
End;
```