Eovia

Carrara Studio SDK™

Overview

Extension Architecture

©2001-2002 Eovia
All rights reserved

Version 2.0
July 2002

Copyright ©2001 Eovia , Corp. All rights reserved.

The software described in this manual is furnished under a licensing agreement contained in the license.txt file in the SDK folder. It may only be used or copied in accordance with the terms of this license. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical or otherwise, without the prior express written permission of Eovia, Corp.

The information in this user guide is provided for informational use only, is subject to change without notice, and should not be construed as a commitment by Eovia, Corp. Eovia, Corp. assumes no responsibility or liability for any errors or inaccuracies that may appear in this user guide.

Licensee acknowledges that the CarraraSDK Development Toolkit may contain bugs, errors and other problems that could cause system failures. Consequently, the CarraraSDK is provided to Licensee “AS IS,” and Eovia disclaims any warranty or liability obligations to Licensee of any kind. Accordingly, Licensee acknowledges that any research or development that it performs regarding the CarraraSDK or any product associated with the CarraraSDK is done entirely at Licensee’s own risk.

LICENSEE ACKNOWLEDGES THAT Eovia MAKES NO EXPRESS, IMPLIED, OR STATUTORY WARRANTY OF ANY KIND FOR THE PRODUCT INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY WITH REGARD TO PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE.

Eovia SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF REVENUE, LOSS OF PROFITS, BUSINESS INTERRUPTION, LOSS OF INFORMATION OR DATA AND THE LIKE) ARISING OUT OF THE USE OF OR INABILITY TO USE THE PROTOTYPE EVEN IF Eovia HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Table of Content

Chapter 1 -Introduction 7

Roadmap: How to use this documentation 7
What is in the public SDK? 7
An Open Architecture for 3D Illustration and Animation 7
What this 3D Open Architecture is 7
What this 3D Open Architecture is not 8
Supported Platforms and Compilers 8
3D Shell and 3D Extensions 8
Introducing COM dynamic linking 8
How the whole thing works 9
Identifying Components at startup: Auto Plug-And-Play 9
Family ID, Class ID... 9
...And Instances 10
Communicating between the 3D Shell and the 3D Component 10
How the Component User Interface and Public Data is managed by the 3D Shell 10

Chapter 2 -Overview 13

API Levels (Managing versions) 13
List of Family IDs 13

Chapter 3 -Using Eovia's COM Dynamic Linking 17

About COM 17
Component Registration Process 17
How the Component Server Works 17
How the Application Calls your Extension 17
Extension Entry Points 18
Minimal resources 18
How your Extension Calls into the Application 19
Using Shell objects received as parameters 19
Using preset Shell objects stored in the library global variables 19
Instanciating a Shell object yourself 19
The ÒPMapÓ resource 20
IMCUnknown class 21
What is a COM object? 21
QueryInterface 21
AddRef 21
Release 22
More on QueryInterface, AddRef and Release 22
Registering your Class ID with MetaCreations 22
How to register 22
Contacting MetaCreations - Developers Mailing list 23

©2001-2002 Eovia. All rights reserved.

Chapter 4 -General Information About Resources 25

Component registration resources 25
COMP resources 25
GUID resources 26
How to create a GUID resource in the .r file 26
PMap resources 26
List of types used in the ÒPMapÓ 27
Shaders and PMaps 28
How to build a resource file (the .dta file) 28
On Windows 28
On MacOS 29
Using Res2Data and MPW 29
Using the Res2Data extension directly in Code Warrior 30

Chapter 5 -Platforms and Compilers issues 31

Compiling for Windows 31
Compiling for MacOS 31
Debugging on MacOS 31

Chapter 6 -User Interface 33

User Interface Options 33
No User Interface 33
Auto PMap 33
PMap and a Part 34
Samples using a PMap and .rsr file 35
Part Extensions 35
To create a part extension: 35
Manually Controlling Your Own User Interface 35
Creating your User Interface Containers 36
Examples of Specific user interface elements 38
Registration Dialog box 38
Tab Part 39
Creating a Component Chooser 40
MCSketch 40
Extra Tokens for Each Part 40
List of all UI elements (parts) 41
TMFEditTextPart - Edit Text part 41
TMFIconButtonPart 41
TMFTextButtonPart 42
TMFDialPart 42
TMFHierarchicalListPart 42
TMFImagePart 43
TMFCheckboxPart 43
TMFColorPart 43
TMFOverlayImageControlPart 44

©2001-2002 Eovia. All rights reserved.

TMFRadioPart 44
TMFToolbarPart 44
TMFNodePart 45
TMFAutoCenteringPart 45
TMFBalloonPart 46
TMFWindoidPart 46
TMFDialogoidPart 46
TMFCollapsiblePart, TMFSelectableCollapsiblePart 47
TSingleComponentChooser 47
TMultipleComponentChooser 47
TMFFramePart 48
TMFGradientPart 48
TMFNodeReferencePart 48
TMFRadioCluster 49
TMFScrollPart 49
TMFSlider, TMFCircularSliderPart, TMFLinearSliderPart 49
TMFSolidColorPart 50
TMFSplit3Part 50
TMFCollapsibleListPart 50
TMFVectorPart 51
TMFListPart, TMFRuledListPart, TMFStringListPart 51
TMFOffscreenPart 51
TMFPoppedPart 52
TMFRectPart 52
TMFStaticTextPart 52
TMFTextPopupPart 53
Interfaces supported by each UI element (part) 53

Chapter 7 -Counted Pointers 55

Defining Classes that May Need to be Counted 55
TMCCountedObject/TMCSMPCountedObject 55
Using AddRef() and Release() 56
Summary: How to Convert ÒOldÓ Files to ÒNewÓ Files 57
Implementing Create 58
Accessing Counted Objects 58
Implementing a Counted Pointer Access Method 58
Implementing QueryInterface() 58
Helper Classes for Counted Pointers 58
Example of TMCCountedGetHelper 59
Example of TMCCountedCreateHelper 60
CountedPtr vs Interface Pointer 60
Two functions returning InterfacePtr 61
Never return a CountedPtr 62
Creating objects 62
Aggregation 62

©2001-2002 Eovia. All rights reserved.

Chapter 8 -Utilities 63
Various Shell Utilities (IShUtilities) 63
Thread Utilities (IShTreadUtilities) 63
Component Utilities (IShComponentUtilities) 63
SMP Utilities (IShSMPUtilities) 63
Action Utilities (IShActionManager) 63
Menu Utilities (IShMenuUtilities) 64
Part Utilities (IShPartUtilities) 64
Resource Utilities (IShResourceUtilities) 64
File Handling (IShFileUtilities) 64
Creating Files 64
Opening Files 65
Working with Streams 65
Personality Utilities (IShPersonalityUtilities) 65
Drag & Drop Utilities (IShDragAndDropUtilities) 65
Change Management (IChangeManager) 66
Clipboard Utilities (IMFClipboard) 66
3D Utilities (I3DShUtilities) 66

Chapter 9 -MCCommon References 67
MCCore Interfaces 67
MCF Interfaces 68
User Interface Flexibility 68
MCImage Interfaces 68

Chapter 10 -Failure Handling 69
Using the Failure Handling 69
General rules 69
A first simple example 69
Initializing your local variables properly 70
C++ Objects allocated in the stack 71
The Failure Handling Chain 71
Triggering a Failure yourself by using throw, ThrowIfNil(), ThrowIfNoMem(), and ThrowIfError() .. 71
Error codes 72
Message parameter coding 73
Failing Silently 73
Recovering from a Failure 74

Chapter 11 -SMP 75

©2001-2002 Eovia. All rights reserved.

Chapter 1 - Introduction

This chapter introduces the main concepts of the Carrara SDK. It also gives important tips on how to use this documentation and will help you in making the main technical choices so you can have a smooth and exciting experience developing your 3D Components.

1.0.1 Roadmap: How to use this documentation

You will find different types of chapters in this documentation, each type answering a specific need of the 3D Components developer:

- The **Overview**: This book provides an overview to the SDK and explains the major concepts you need to understand to extend the functionality of Carrara.
- The **Cookbook**: This book provides step-by-step / how-to descriptions that guide you through the main points of each subject. It often make references to the Toolkit examples. You will find these chapters most helpful to quickly understand the main concepts and tasks behind each 3D Extension, and they will be especially appreciated by newcomers.
- **Reference**: “3D Components API Reference”, “3D Shell API Reference”, “Data Structures Reference”, and the “Database Overview” chapters are found. The “Database Overview”, gives you the mathematical background and the definition of the different 3D terms used throughout the documentation. If you find anywhere an unknown 3D concept or term, chances are that it is explained in the “Database Overview” chapter. etc. This is where you will find all the little picky details on each extension function, shell procedure or data structure.

Cookbook and Reference books are designed to be complementary: the Cookbook is used to get the general concepts and identify the steps to follow, and the Reference is used to get to the bottom of each call parameter. You’ll find both of them useful resources.

Also, the examples and the .h files in the Carrara SDK are invaluable sources of information.

What is in the public SDK?

- Interface headers (Iex*.h, Ish*.h, I*.h)
- .h and .cpp files (for types or basic implementations of some classes)
- Libraries
- Sample code
- Documentation (Overview, Cookbook, and Reference)
- Sample executables
- Tools

1.1.An Open Architecture for 3D Illustration and Animation

1.1.1 What this 3D Open Architecture is

The architecture described in this document was created with the following design requirements in mind:

- Enable an extension mechanism for 3D Illustration and Animation applications to share major components such as shaders, renderers, light sources, atmospheric effects, animation effects, 3D primitives, and more between applications
- Provide a seamless integration of plug-ins into the application user interface

©2001-2002 Eovia. All rights reserved.

- Provide a Plug-and-Play component installation
- Provide a cross-platform solution (Windows and Macintosh)
- Remove the burden out of writing 3D Extensions. When a choice had to be made, the burden was put on the host application to facilitate the development of 3D Extensions

These technical choices were made because 3D is a complex field. Anyone having any specific 3D needs has to invest in enormous programming efforts to achieve his 3D project.

For example, a researcher needing to make a 3D data visualization and simulation project will have to implement a 3D renderer, a 3D Database system, 3D manipulation tools and user interface, etc. Another example is an architect who needs to render 3D buildings, but cannot find any on-the-shelves 3D program that can support his or her exotic 3D architecture file format.

Often the problem to solve is very simple to code, but the 3D tools around it are vast and complex.

This is why an open architecture is especially welcomed in the 3D field. Applications such as Carrara built on this architecture can be extended very easily this way. For example, adding a Spherical camera takes only a few hundreds lines of code and is about a one day project.

The other obvious advantage of programming 3D Components is to provide solutions that run in different client applications, thus expanding the potential market of your 3D Component to many users without being forced to choose a specific 3D application. This advantage is typical in open architectures, and OLE programmers are familiar with it.

1.1.2 What this 3D Open Architecture is not

- A rendering library
- A CAD oriented API

What we explain in this document is a way for 3D Applications and 3D Components to communicate together. As such, it describes both the client and server sides, and how they interact.

1.1.3 Supported Platforms and Compilers

The supported platforms are MacOS PPC, Windows 95, Windows 98, Windows NT. The supported compilers are Code Warrior Pro 5.0 and later on the Macintosh, and Microsoft Visual C++ on Windows.

On the Macintosh, the use of Steve Jasik's The Debugger or Metrowerks MetroNub debugger is highly recommended for debugging.

1.1.4 3D Shell and 3D Extensions

The 3D Application is the client of the different services provided by the **3D Components**. Because the 3D Components add features, they are often called **3D Extensions**, or **plug-ins**.

The 3D Application is in charge of organizing the data flow between the Components, and it also offers a set of services to the 3D Components to allow then to interact with its own internal data (like the 3D Database). The 3D Application is called the **3D Shell**, which is a better and more generic term.

1.2.Introducing COM dynamic linking

The communication between the 3D Shell and the 3D Components is done by using the Component Object Model (**COM**).

©2001-2002 Eovia. All rights reserved.

The choice was made to use the COM because COM is a very widely spread industry standard (COM is the low-level layer on which OLE is built), and is actively promoted by major industry players. So if you are already familiar with OLE, you will find it easier to get started with this documentation. COM offers a nice and clean C++-like interfacing, and it is highly recommended that you read the excellent book “Inside OLE 2” by Kraig Brockschmidt from Microsoft Press to learn everything about it.

COM users should read Chapter 3 -, “Using Eovia’s COM Dynamic Linking” chapter to learn all details about each technique and how to implement them.

1.3.How the whole thing works

1.3.1 Identifying Components at startup: Auto Plug-And-Play

When the 3D Shell is launched, it first identifies which 3D Components are available. To do this, it looks in its Extension directory and in all sub-directories of the Extension directory. On Windows, this will be the “ext” directory, on MacOS the “Carrara Extensions” folder.

On Windows:

All files with the “.mcx” extension are considered as Components.

On MacOS:

All files with the ‘MCEX’ file type are considered as Components.

Please note that there is nothing to register anywhere in the operating system to allow the identification of the Component. It is completely automatic. **Even if you develop a Windows extension, it is not necessary to register your Component in the Windows Registry Database.** This allows a true Plug-And-Play installation of Components. No problem for uninstalling, no full path names issues, no conflicts between different versions or languages, etc.

For each Component file found, the 3D Shell tries to find the corresponding resources. The resources are located in a “.dta” file next to the Component file on Windows, and in a “data” suffix on MacOS.

Now that the resources are found, the 3D Shell looks for all ‘**COMP’** resources in the file (you can put several Components in the same file). The ‘COMP’ resource is the key to each Component. It identifies the name of the Component, the API version number it is based on, his own version number, and most important of all, its **Family ID** and its **Class ID**.

1.3.2 Family ID, Class ID...

Each Component belongs to a **Family**. The Component Family defines the kind of Component the 3D Shell is dealing with: a Shader, a Camera, an Export Filter, etc. All the items in bold in the “3D Pipeline” chart in the Cookbook are the available Families.

Each Family has a 4 letter code, its **Family ID**. For example, the Family ID for Shaders is ‘shdr’. Your Component must belong to an existing Family, otherwise the 3D Shell will not know what to do with it. Family IDs are described for each Component in the Cookbook and Reference chapters, and there is a general table in “List of Family IDs” on page 13.

In the COM terminology, a Family is strictly equivalent to an **Interface**. Interfaces are pure virtual classes inheriting from IMCUnknown (basically a COM definition). The Interface files are the I*.h headers.

©2001-2002 Eovia. All rights reserved.

Then Each Component has a **Class ID**. Its Class ID describes uniquely your Component. For example, in the Shader Family, there are different classes of shaders: the Checker Shader, the Marble Shader, the Wood Shader, etc. Each of this shader has a unique Class ID. Its Class ID is the Component's key to its data and instanciation process. This is how your Component will be able to store its private data in a Carrara file and retrieve it later.

For this reason, if you intend to distribute your Component outside, even as a Freeware or Shareware, **it is vital that you register your Class ID from MetaCreations to ensure that it will be unique**. This Class ID will then be yours forever. This registration process is done by e-mail or fax, and it is free. See "Registering your Class ID with Eovia" on page 22 to learn how to register Class IDs.

In the COM terminology, a Class is strictly equivalent to a **Class**.

1.3.3 ...And Instances

So far, this is all the 3D Shell does at startup. It simply identifies all available Components for later use. No Component code has been loaded or executed.

Then comes the time of instanciation. For example, a file is read, and it contains a 3D Object with a Marble Shader. The 3D Shell does not know anything about Marble Shaders. It just gets the Family ID and the Class ID. Looking up in its Component Class directory, it **instanciates** a Marble shader by asking the Component implementing it to create one. The Shell just keeps an anonymous pointer on the Component **Instance**, and this will be good enough to communicate with it: reading/writing its data, handling the user interface, executing it, etc. This will be explained in detail in the sections that follow.

Note that each instance has its own data values. For example, if another object in the file has another Marble texture on it, then a new instance will be created in order to store the parameters of this second Marble texture. This is how you can have several Marble textures with different vein spacing on different objects: they are different instances of the same Marble Class.

In the COM terminology, an Instance is strictly equivalent to an **Instance** (sorry COM-savvy readers, you got the point already...).

1.3.4 Communicating between the 3D Shell and the 3D Component

Once the Component has been instanciated, the 3D Shell will need to access the different Component services (i.e. routines). Think of the 3D Shell as the orchestra director. The 3D Shell will call your Component routines when needed. "Do not call us, you will be called" could be the 3D Shell motto.

However, there are times when your Component needs additional services from the 3D Shell. For example, a 3D Import Filter will have to be able to create and manipulate data structures in the 3D Shell Database. Therefore, the 3D Shell offers a complete API to do all kind of things. When a routine of your Extension is called, you can then call back the Shell to complete your job.

1.3.5 How the Component User Interface and Public Data is managed by the 3D Shell

Often, the 3D Shell will want to show to the user the Component's User Interface so its parameters can be set before executing. The 3D Shell has unique way to do this so the user interface of your Component is seamlessly integrated in the Shell user interface, providing a consistent and integrated experience to the user.

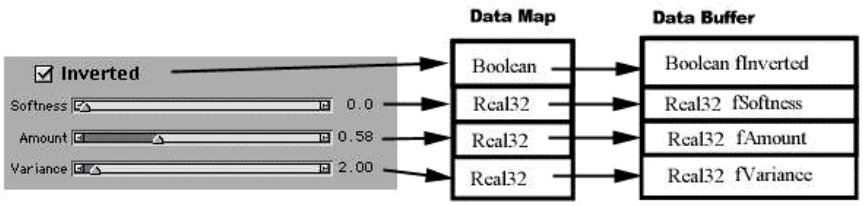
The key to this UI integration is the '**PMap**' resource. This resource is also the key to saving and reloading the Component's data.

©2001-2002 Eovia. All rights reserved.

The ‘PMap’ resource describes a table called the **Data Extension Map**. During the initialization process, the Component told the Shell where its “public data” was stored: this is called the **Extension Data Buffer**, and it is located in the extension’s own RAM space. It is a record of the Component parameters (like the vein spacing values and the veins colors of the Marble Shader). The 3D Shell merely has an anonymous pointer on the Extension Data Buffer. The Data Extension Map will help the 3D Shell identifying the types and addresses of each data in the Extension Data Buffer, and build the relationship with the user interface.

Put simply, the Data Extension Map has two values for each entry: an ID to identify the user interface element (button, slider, etc.), and a type to identify the kind of data store in the Extension Data Buffer (Boolean, fixed, long, etc.).

This way, when the user changes something like the state of a check box, the Shell is able to compute the address of the corresponding item in the Data Buffer, and change it directly. Then it calls the Component and tells it that its data have been changed, so that the Component can react and update any internal pre-processed data.



Note that you do not have to write any code to handle your user interface. The Shell reads the resources describing your user interface from your Component file, and takes care of everything: it reacts when the user clicks on buttons, sliders, etc. because it knows about all these user interface elements.

The other advantage of this mapping system of the 3D Component public data is that it provides the 3D Shell with a simple way to save the Component data in a file and retrieve it later without any knowledge of the Component’s purpose in life. To store the Component data in the file, the Shell just writes the data ID and the value, and that’s it. Later, after the Component is instanciated, the 3D Shell will write back this data in the Extension Data Buffer.

You will find details on how to build the user interface itself (using the MCSketch, the user interface building tool) and how to compile a .dta file in the chapters that follow.

©2001-2002 Eovia. All rights reserved.

©2001-2002 Eovia. All rights reserved.

Chapter 2 - Overview

2.1.API Levels (Managing versions)

Carrara provides a version scheme to control compatibilities issues between the Shell (the application) and the extensions. The system is very simple, and is based on the idea of “API Level”. Every time the Carrara API changes (which happens at each major or minor release), the API Level is incremented. For example, in Carrara 1.0, the API level is 1.

When you develop an Extension, you tell the Shell which minimum API level you are compatible with by putting this number in your COMP resource. For example, let’s say you make a Shader for Carrara 1.0, you would put 1 in your COMP resource.

Now let’s imagine that Eovia comes up with a 1.01 release (therefore API level 2), and that in this new version, the Shader interface has not changed. Carrara 1.01 will know that it is backward compatible for this type of extension, so it will accept running it.

Then let’s imagine the a 1.2 comes up (API Level 3), and that the Shader interface has changed so dramatically that backward compatibility could not be maintained. Carrara 1.2 will then **require** API level 3 for this type of extension, so the user will see at startup an alert saying that this extension has become “too old” and that he or she needs to upgrade it.

On the other hand, if another user tries to use your updated extension under Carrara 1.0 (API Level 1), then Carrara will refuse to use it and tell the user that this extension is “too new.” (He or she obviously needs to upgrade Carrara).

To find the API level of your copy of Carrara, convert the DTA file of the application to a resource file, and open the APIV (1000) resource. An easier way is to look at the table below:

Table 1: API Levels and Carrara versions

API Level	Carrara Version number
1	1.0
1	1.1
2	2.0

The table in the next section gives the minimum API level for each type of Interface.

2.2.List of Family IDs

Here is the list of Extension Family ID and the corresponding COM IID for each type of Extension. It also gives the minimum API level supported for each type (see previous section for a discussion on API levels).

©2001-2002 Eovia. All rights reserved.

Table 2: Family ID and Interfaces of Carrara (API Level 2)

Family	Family ID	COM IID	API Version
Ambient Light	ambi	I3DExAmbientLight	2
Animation Method	amet	I3DExAnimationMethod	2
Atmospheric shader	atmo	I3DExAtmosphericShader	2
Attached generic data	data	I3DExDataComponent	2
Backdrop	drop	I3DExBackdrop	2
Background shader	back	I3DExBackground	2
Camera transformation	came	I3dExCamera	2
ColorPickers	pick	I2DExColorPicker	2
Deformation Zones	dfzn	I3DExDeformationZone	2
DropAreas	drpA	IMFExDropArea	2
DropCandidates	drpC	IMFExDropCandidates	2
Export	3Dou	í3DExExportFilter	2
External Module	modu	I3DExModule	2
External View (External Part)	part	IMFExPart	2
External List View	part	IMFExListPart	2
Final Renderer	frnd	I3DExFinalRenderer	2
Gel	'gel '	I3DExLightsourceGel	2
ICC Processor	iccp	I2DExICCProcessor	2
Import	3Din	í3DExImportFilter	2
Interactive Renderer	RndB	I3DExRendererBox	2
Light	lite	I3DExLightsource	2
Modifiers	modi	I3DExModifier	2
Motion Link	link	I3DExMotionLink	2
Post Render Filter	post	I3DExPostRenderer	2
Primitive	prim	I3DExGeometricPrimitive	2
Reflection Render Features	fleF	I3DExReflectionFeature	2
Refraction/Transparency Render Features	fraF	I3DExRefractionFeature	2
Registerers	regi	IExRegisterer	2
Scene Modifiers	smod	I3DExModifier	2
Scene Commands (Scene Operations)	scmd	I3DExSceneCommand	2

©2001-2002 Eovia. All rights reserved.

Table 2: Family ID and Interfaces of Carrara (API Level 2)

Family	Family ID	COM IID	API Version
Shader	shdr	I3DExShader	2
Shadow Render Features	shdF	I3DExShadowFeature	2
Tweener	twee	I3DExTweener	2

©2001-2002 Eovia. All rights reserved.

©2001-2002 Eovia. All rights reserved.

Chapter 3 - Using Eovia's COM Dynamic Linking

This chapter describes in more detail COM Dynamic Linking supported in Carrara.

Before reading this chapter, it is highly recommended to read the Introduction chapter, especially the section "How the whole thing works" on page 9.

3.1.About COM

COM (Component Objects Model) is the low-level dynamic linking system on which OLE is built. It is a very widely spread industry standard, and is actively promoted by major industry players. So if you are already familiar with OLE, you will find it easy to be started with this documentation. COM offers a nice and clean C++ like interfacing.

The best reference for COM is "Inside COM" by Dale Rogerson from Microsoft Press (ISBN 1-57231-349-8). The first chapters of "Inside OLE 2" by Kraig Brockschmidt from Microsoft Press are also a good introduction (ISBN 1-55615-618-9). You can also find all the necessary information in any edition of the MSDN (Microsoft Developers Network) CDs. On the Web, look at http://www.microsoft.com/com/.

3.2.Component Registration Process

Component registration provides true auto-plug and play installation of components. Simply drop component files in the correct folder and run. You don't need to use the Windows registry or full-path names. There is no conflict between versions or languages (due to being based on COM).

Each component has one COMP and one GUID resource associated with it. Each extension file (.mcx file) can have more than one component by defining multiple COMP and GUID resources. To learn more about the resources refer to "Component registration resources" on page 25.

3.3.How the Component Server Works

At start-up the Component Server identifies components in the application directory and below. It looks for .mcx extension (or for the 'MCEX' file type on MacOS). For each extension file found, it looks for a resource file with the .dta suffix (or the 'Data' suffix on MacOS). For each of these files, it looks for more components; it identifies components by a 'COMP' resource and adds the component to a list. There is one list of components for each family.

In additional to the general component registration resources, each type of component may have its own specific resources. For example, a module has a 'Modu' resource that adds on functionality that modules can do. To learn more, refer to the chapter on family-specific resources in the Cookbook.

3.4.How the Application Calls your Extension

"Simple" Components are extensions that do not call back into the Shell or application to perform their duties. Such extensions are typically Primitives, Shaders, Light sources, Ambient lights, Gels, Atmospheric Shaders, Background and Backdrop Shaders, and Cameras.

©2001-2002 Eovia. All rights reserved.

3.4.1 Extension Entry Points

Your COM Extension has these entry points:

- **MCDllGetClassObject()**: standard COM entry, used by the Shell to ask your code to instanciate your Extension through a Class Factory
- **MCDllCanUnloadNow()**: standard COM entry, not used in Carrara
- **MCDllInit()**: low-level intialization entry, used for memory allocation inits, and other utilities intialization of the libraries.
- **MCDllCleanUp()**: counterpart of MCDllInit(), used for deallocation when unloading the DLL
- **MCDllInitExceptionTranslator()**: Failure Handling initialization
- **MCDllCleanUpExceptionTranslator()**: Failure Handling clean-up

The library provided in the Carrara SDK already implements these calls, so you won't have to worry too much about them (just make sure that they are exported properly by your DLL or Shared Library). The library also already implements the Class Factory scheme and everything, so the actual routines that you will have to implement are these:

```
// Initialization routine:
void Extension3DInit(IMCUnknown* utilities);

// Clean-up routine:
void Extension3DCleanup();

// Instanciation routine:
TBasicUnknown* MakeCOMObject(const MCCLSID& classID);
```

That's it. The library takes care of the rest for you.

Initialize your global data in `Extension3DInit`. Get rid of it in `Extension3DCleanup`. Instanciate your extension in `MakeCOMObject`, using a simple `new` command:

```
TBasicUnknown* MakeCOMObject(const MCCLSID& classID)
{
   if (classID == CLSID_MyExtension)
      return new TMyExtension;
   else
      return NULL;
}
```

Your extension will always inherit more or less indirectly from TBasicUnknown (see the discussion about the TBasic* types later on).

3.4.2 Minimal resources

Make sure you choose a CLSID and that you copy its value in the GUID resource (see "GUID resources" on page 26) for more details. Make sure that all your COMP, PMap, GUID, Node, etc. resources have the same ID number and that this number is the one returned by **GetResID**. You still need to choose a 4-letters Class ID and put it in your COMP resource.

To learn more about the Node resource, refer to <Section>Chapter Chapter 4 -, "General Information About Resources.<Section>"

And, once again, Windows programmers do not need to enter their IID and CLSID in the Windows Registry

©2001-2002 Eovia. All rights reserved.

Database. Simply give your Component the .MCX suffix, and put it in the EXT directory (or in any sub-directory of the EXT directory). Carrara will find it automatically.

Use the samples of the Carrara SDK as a framework to start your own extensions.

3.4.3 How your Extension Calls into the Application

Some extensions need to call back the Shell. For example, an Import Filter may need to create 3D objects and put them in the scene.

There can be three types of situations:

- A Shell object pointer is passed to you as a parameter of one of your Extension procedures.
- A Shell object pointer is provided to you as a global variable.
- You need to instanciate a new Shell object and work on it.

Using Shell objects received as parameters

This is a very simple case. The Shell already allocated an object and gives you its pointer as the parameter of one of your Component procedures.

Example:

```
IMFExResponder::Receive(int32 message, IMFResponder* source, void* data);
```

The *source* parameter is a Shell object of the **IMFResponder** Interface. You simply call its methods as you wish. There is no need to call **AddRef**() on it. Do not call **Release**() either, the Shell will take care of its own children.

Example:

```
int32 instanceID;
instanceID = source->GetInstanceID();
```

Using preset Shell objects stored in the library global variables

Many useful methods can be accessed via global pointers pointing to objects of the Shell. The global utilities pointers are grouped by functionality into different COM interfaces (gShellUtilities, gResourceUtilities, ...). These interfaces can all be found in the header files called *something*Utilities.h. All of the global variables are initialized by the extension's entry points MCDllInit and MCDllCleanUp, so you do not need to worry about that.

Example:

```
uint32* myArray; // array of 640*480 unsigned long
myArray = gMemoryUtilities->Calloc(sizeof(uint32), 640*480);
...
MCCOMErr result = gMemoryUtilities->Free(myArray);
```

Instanciating a Shell object yourself

There are times when you need to create an object, like creating a 3D Primitive to put it in the Scene.

The procedure is a little bit unusual for COM programmers. Instead of calling the global procedure

©2001-2002 Eovia. All rights reserved.

CoCreateInstance(), you will call **IShComponentUtilities::CoCreateInstance**() on the *gComponentUtilities* or *gSh3DComponentUtilities* global variable (depending on your needs).

Example:

We create a Group and put it in the Scene:

```
HRESULT TDXFImporter::DoImport(char* fullPathName, I3DShScene* scene,
       I3DShTreeElement* fatherTree)
{

 I3DShGroup*       group = NULL;
 I3DShTreeElement* groupTree = NULL; // The same as group, but as a Tree Element
.......

 gSh3DComponentUtilities->CoCreateInstance(CLSID_StandardGroup, NULL,
       CLSCTX_INPROC_SERVER, IID_I3DShGroup, (void**) &group);

 // Do whatever we want on the group
 ......

 // Use it as a Tree Element to put it in the scene
  group->QueryInterface(I3DShTreeElement, (void**) &groupTree);
  groupTree->SetScene(scene);
  fatherTree->InsertLast(groupTree);
  groupTree->Release(); // We do not need this interface anymore
  group->Release();     // We do not need this interface anymore

}
```

3.4.4 The "PMap" resource

A resource "*PMap*" describes the different parameters that the Shell can access. The PMap is key to UI integration as all components with UI have a PMap resource.

The PMap describes the UI data for the shell. There are 4 values in the PMap for each UI element:

- a four letters ID (to identify the element)
- a four letter type code (boolean, long, etc.)
- a flags field (animate this element, etc.)
- the name of the element (appears in the Time Line)
- an extra string containing optional additional data

Whether you create your resource on the Mac or PC, your "PMap" should look like this:

```
resource 'PMap' (128)
{ /* First Array Element */
  'SIZH',       /* ID of the UI element in the view */
  'in32',       /* type of the value */
  interpolate,  /* value can be animated */
  "Hori. Size", /* name */
  "",           /* Optional data */
  /* Second Array Element */
  'SIZV',       /* ID of the UI element in the view */
  'in32',       /* type of the value */
  interpolate,  /* value can be animated */
  "Vert. Size", /* name */
```

©2001-2002 Eovia. All rights reserved.

```
  "",           /* Optional data */
}
```

To learn more about PMaps and PMap types, refer to "PMap resources" on page 26, "Auto PMap" on page 33, and "PMap and a Part" on page 34.

3.5.IMCUnknown class

What is a COM object?

A COM object is any object derived from IMCUnknown (simple inheritance only).

```
class IMCUnknown
{
public:
   virtual MCErr MCCOMAPI QueryInterface(const MCIID& riid, void** ppvObj)=0;
   virtual uin32 MCCOMAPI AddRef()=0;
   virtual uin32 MCCOMAPI Release()=0;
   };
```

It is a key element of the Carrara COM implementation as it provides a way for a client to communicate with a component. These functions are explained in greater details in the sections that follow.

When you are creating components to be used with Carrara, you always inherit more or less indirectly from IMCUnknown.

3.5.1 QueryInterface

QueryInterface is the cornerstone of COM. In COM, a client always communicates with a component through an interface. QueryInterface defines the component; the interfaces that a component supports are the interfaces for which QueryInterface returns an interface pointer. You use QueryInterface to discover whether a component supports a particular interface. If the component supports the interface, QueryInterface returns a pointer to that interface.

```
   virtual MCErr MCCOMAPI QueryInterface(const MCIID& riid, void** ppvObj)=0;
```

The first parameter is the Interface identifier structure or the **IDD**. The second parameter is the address where QueryInterface places the requested interface pointer.

If you use the following notation:

if "ObjB = A->QI('B')" means: ObjA->QueryInterface(REFIID_B, &ObjB)

QueryInterface should follow the following rules:

```
(1) A->QI('A') == A
(2) (A->QI('B'))->QI('A') == A
(3) ((A->QI('B'))->QI('C'))->QI('A') == A
```

3.5.2 AddRef

AddRef and Release use reference-counting to manage memory. Reference counting enables components to delete themselves (once they are no longer being used). AddRef increases the reference count and Release

©2001-2002 Eovia. All rights reserved.

decrements the reference count.

When do you use AddRef?

- If you are asked for an object by someone, call AddRef before returning a pointer to it.
- If you plan to keep a reference on an object passed as a parameter in a function, the function needs to call AddRef and then Release when done.

To learn methods that allow you to optimize reference counting and avoid using AddRef, see the MCCountedPtr template. Refer to Chapter 7 -, "Counted Pointers" to learn more.

3.5.3 Release

As mentioned earlier, Release decrements the reference count. When you are finished with an interface, you should call Release on that interface.

When do you use Release?

- QueryInterface increments the count; when done, call Release.
- A function you call to get an object should have incremented the count, so you only need to call Release.
- If you plan to keep a reference on an object passed as a parameter in a function, the function needs to call AddRef and then Release when done.

To learn methods that allow you to optimize reference counting and avoid using Release, see the MCCountedPtr template in <Section>Chapter Chapter 7 -, "Counted Pointers.<Section>"

3.5.4 More on QueryInterface, AddRef and Release

It is highly recommended to read carefully <Section>Chapter Chapter 7 -, "Counted Pointers.<Section>" for more details about using QueryInterface, AddRef and Release.

3.6.Registering your Class ID with Eovia

As explained in the section "How the whole thing works" on page 9 in the Introduction chapter, it is vital that you make sure that your Class ID is unique before distributing your software outside, even as a Freeware or Shareware. The reason is that if someone else uses the same Class ID in his own extension, the 3D Shell will be confused when trying to instanciate and initialize your 3D Component, causing a failure or a crash.

Registering your Class ID is free. There are no obligations. After acceptation, your Class ID will be yours forever and will never be assigned to someone else.

- A Class ID is a set of 4 ASCII characters.
- Ranges of Class ID will **not** be registered by MetaCreations (like "all IDs from 'AAAA' to 'AAAZ'").
- Class IDs are case-sensitive.
- Class IDs with all lowercase letters are reserved to MetaCreations (like 'abcd'). Combos are fine, though (like 'Abcd').
- Try avoiding special characters (ASCII code > 128), because these characters are different between MacOS and Windows.

©2001-2002 Eovia. All rights reserved.

3.6.1 How to register

Simply email the following information to Eovia. Make sure you include the word "registration" in your e-mail title to allow faster processing.

Name:
Company:
Address: (if sending via mail)
Phone:
Fax: (if sending via Fax)
E-Mail:

Component Name	Family ID	Class ID

Please include a brief description of your Component(s):

Send this information to:

E-Mail: sdk@eovia.com

You will then receive a confirmation message from Eovia by email or Fax. If a conflict occurs, Eovia will offer an alternate ID.

3.6.2 Contacting Eovia - Developers Mailing list

We will set up a developer mailing list. You will get more information later.

©2001-2002 Eovia. All rights reserved.

©2001-2002 Eovia. All rights reserved.

Chapter 4 - General Information About Resources

For each extension, you need some resources with the same ID as the ones you give in the function **GetResID()** of the API.

In all cases you need a resource "*COMP*" that describes your extension. To learn about the COMP resource, refer to"COMP resources" on page 25.

In most of them, you need a resource "*PMap*" that describes the different parameters that the Shell can access. To learn about the PMap, refer to "The "PMap" resource" on page 20 for an introduction and to "PMap resources" on page 26 for details.

You also need to add a "*GUID*" resource to specify the COM IID and CLSID. To learn more, refer to "GUID resources" on page 26.

You have to provide a user interface. This is done by building a "*Node*" resource (formerly known as *View* resource in the Ray Dream Studio toolkit). This resource is created using a UI builder tool called **MCSketch**, that you can find in the SDK. Refer to "MCSketch" on page 40 to learn more.

Some extensions require some additional resources like a "Modu" resource for Modules. To learn more about family-specific resources, refer to the Cookbook.

4.1.Component registration resources

When you create an extension, you have to tell the Shell what kind of extension it is. You have to give a name to put in the right menu, palette or window. And you have to tell where it can find the different methods of the extension.

You have to create a "COMP" resource that contains the type of extension (Family) and the name. In addition to the COMP resource, you have to create the "GUID" resource to tell the Shell that the extension is COM and to give the interface ID (IID) of the object and its class ID (CLSID).

4.2.COMP resources

The resource "COMP" describes the type of extension and some additional information.

The fields in a COMP resource:

- Family ID (every component belongs to one family)
- Class ID (every component has a unique class ID)
- Component name
- SubFamily name (used to further categorize components)
- Version information for the application and component
- Comments
- API version

It will look like this:

```
resource 'COMP' (128) /* resource ID = 128*/
{ 'shdr', /* type of extension, the family name */
  'COch', /* Class ID, to recognize your extension in the family */
```

©2001-2002 Eovia. All rights reserved.

```
  “Checker”, /* Name of your extension, it will appears in the Shell to describe
      and select your object */
  “COM Examples”, /* Sub Family Name, for the shaders, it will create a new sub-
      menu call “COM Examples” in the menu “Component” of the Shader Editor */
  0x00030103, /* Version */
  “3.1.3”, /* Version String */
  “Examples”, /* Comments (it will appear in the About extension window) */
  1       /* API Version the Extension is based on */
}
```

4.3.GUID resources

GUID stands for Globally Unique IDentifier, and is a 128-bit (16-byte) structure that uniquely identifies an interface.

When you create a COM object, you need to also create an unique GUID. To tell the 3D Shell that your extension can be called with the COM mechanism, you have to create a GUID resource that contains the IID (Interface ID) and the CLSID (Class ID) of the COM object.

4.3.1 How to create a GUID resource in the .r file

Microsoft provides two utilities: UUIDGEN.EXE (command-line) and GUIDGEN.EXE (a UI-based version of the same application), that generate unique GUIDs. MacOS users can use "Create GUID.ppc", a similar Microsoft utility.

You create the GUID declaration resource in the .r file associated with your component.

Here is an example of a GUID declaration:

```
resource 'GUID' (128)
{
    {
        R_IID_I3DExModifier,
        R_CLSID_ExBarycenter
    }
};
```

The definitions of `R_IID_I3DExModifier` and `R_CLSID_ExBarycenter` are usually stored in a .h file that is shared by the .r file and the .cpp files (it is just a convenience to avoid duplicating definitions). See the toolkit samples for an illustration of this little trick.

4.4.PMap resources

A resource “*PMap*” is used to define all parameters your extension needs to save and restore.

Usually, they are user interface parameters, but they can be purely internal data which your extension needs to have saved so that it can restore its state from a file. For more information on PMaps and how they relate to your user interface, refer to “Auto PMap” on page 33.

You’ll find an introduction and an example of a PMap in “The “PMap” resource” on page 20.

©2001-2002 Eovia. All rights reserved.

4.4.1 List of types used in the “PMap”

Table 3: Types used in PMap

C++ Type	Keyword	Comments
Boolean	bool	= unsigned char.
short, uint16, int16	in16	
long, uint32, int32	in32	
float, real, real32	re32	Single precision floating point
double, real64	re64	Double precision floating point
TMCRGBColor	RGBc	color
	colF	Fixed color 2.14
TMCRect	rect	2D Rectangle
TMCPoint	poit	2D Point
TMCString255	s255	255 characters string
TMCDynamicString	Dstr	Dynamic string (unlimited length)
TVector2	vec2	2D vector
TVector3	vec3	3D vector
TVector3	dire	Direction parameter
TMatrix33	mx33	3 x 3 matrix
TSingleCompo-nentChooser	comp	Component Chooser (allows the selection/creation of a component of one or more Families)
TMultipleCompo-nentChooser	cmp#	Multiple Component Chooser (allows the selection/creation of one or more Families)
ActionNumber	actn	Same as an int32
IMCUnknown	iunk	Data is an anonymous IMCUnknown pointer

For a list of general PMap types, go to MCCommon/PMapTypes.h.

For developers porting RDS code, use the conversion table below (note that Carrara still supports these types):

Table 4: RDS vs Carrara PMap codes

RDS code	Carrara code to use
colo	RGBc
shrt	in16

©2001-2002 Eovia. All rights reserved.

Table 4: RDS vs Carrara PMap codes

RDS code	Carrara code to use
long	in32
qf16	re32
qfix	re32
qfv2	vec2
qfv3	vec3
ROTA	mx33

Shaders and PMaps

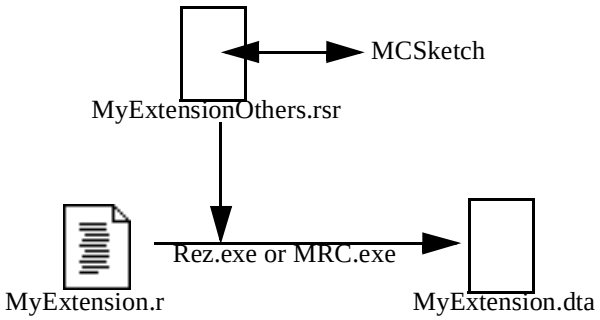
PMaps for shaders have a specific feature that allows reusing the previous values of a shader when switching to another one. If some entries of the new PMap have the same IDs as some entries of the previous PMap, then the values are copied. This is just a convenience for the user, and is implemented only in the Shader Editor.

4.5.How to build a resource file (the .dta file)

4.5.1 On Windows

Resource files (also called data files or .dta files) are built by compiling a .r file that contains the list of resources described in a text format. This format is the same as the Rez format defined by Apple Computer and used on the Macintosh. The result is a binary file that Carrara can parse and use (the format of this binary file is simply a Macintosh resource fork). This binary file is called a resource file.

Text .r files can also include other (already compiled) resource files to build a more complex resource file. Some tools, like MCSketch, the UI builder provided with Carrara, builds a resource file that can be used with a resource compiler.



Depending on which platform you are using, the resource compiler can be the Windows version of Rez from Apple Computer, or MRC.EXE from Microsoft.

MRC is part of the Microsoft cross-platform version of Visual C++. You can find more about it by searching for “MRC” on the Microsoft Web site.

©2001-2002 Eovia. All rights reserved.

The Rez compiler can be found in the QuickTime toolkit on Apple Web site.

Steps to set-up the Rez.exe Compiler on Windows:

1. We suggest you use Rez, an Apple version of the Macintosh resource compiler, distributed as part of AppleÕs QuickTime SDK.
2. So go to http://apple.com/quicktime/developers/qtsdk.html and click ÒdownloadÓ in QuickTime 4 Windows section.
   (After downloading Rez you should get a folder (~8Meg) which has 5 sub-folders: CIncludes, ComponentIncludes, Libraries, RIncludes, and Tools.) We suggest you make this folder SDK/Tools/Rez, then you do not need to modify the BuildDTA.bat file.
3. Create an empty SysTypes.r file in QT RIncludes directory that was part of the Rez download.
4. Go into the SDK/Build/BuildDTA.bat file and make sure it uses the correct path to rez for your machine. If you put Rez where we suggest, this step is unnecessary.

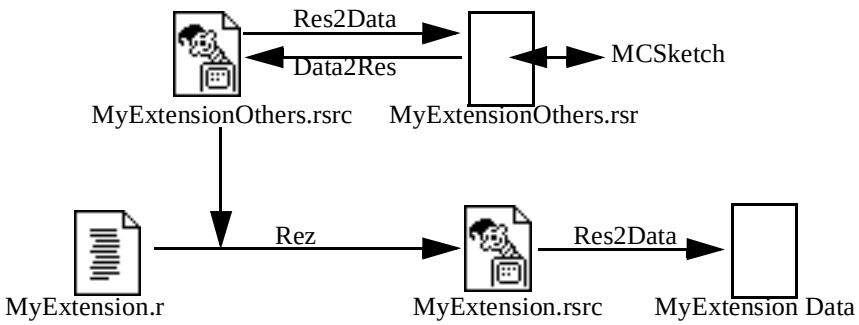
An alternative solution is to compile the *.dta files on a Macintosh and copy them on your PC.

If you want to manually set up your resource compilation using Rez, and you are used to using MRC from Microsoft, here are some minor syntax differences to be aware of:

- You need space between options like -i pathname, etc. (mrc takes space or no space)
- “-D” is “-d” in Rez
- Rez sometimes output a .qtr file (even with -o name.dta provided), so you may need to change the output name.

4.5.5 On MacOS

The process on MacOS is very similar, but there is an additional step at the end: the Macintosh development tools know how to treat resource files directly, so resources are kept in the “resource fork” of the intermediate files. However, Carrara still expect to find its resources in the “data fork”. The same goes for the Mac version of MCSketch (which is based on the same framework as Carrara), so there is some conversion involved there too.



The .rsrc files are standard MacOS resource files that can be edited with the usual resource tools: ResEdit, Resorcerer™, etc. There are also Resorcerer™ templates provided in the SDK tools folder to edit all Carrara special resources.

4.5.6 Using Res2Data and MPW

Res2Data and Data2Res are two MPW tools that allow you to convert files back and forth between the

©2001-2002 Eovia. All rights reserved.

resource fork and the data fork.

The MPW syntax is simply:

```
Res2Data extension.rsrc extension.dta
```

and:

```
Data2Res extension.dta extension.rsrc
```

4.5.7 Using the Res2Data extension directly in Code Warrior

A Res2Data extension for Code Warrior is also provided in the toolkit. It allows you to build the Data file directly within Code Warrior. Just create a new “target” in your Code Warrior project., and use “Res2Data post linker” in your compiler’s settings. Look at the samples of the Cookbook for more details.

©2001-2002 Eovia. All rights reserved.

Chapter 5 - Platforms and Compilers issues

This chapter describes details about specific issues related to compilers and platforms.

5.1.Compiling for Windows

The supported compiler is Visual C++ 6.0 or better from Microsoft. You should install the Service Pack 3.

Another option is to refer to http://www.ultradrive.com/rds_exten.htm, where Evelio Perez-Albuerne explains how to write extensions for RDS using Inprise C++ Builder. You can apply this information to Carrara as well.

Note that COM extensions are simply standard 32-bit DLLs with a .MCX suffix. Take a look at the sample make files in the Carrara SDK and you'll see that there are no special tricks there. Data files (files containing resources) have the .DTA suffix.

5.2.Compiling for MacOS

The supported compiler is Code Warrior Pro 7.0 and later from Metrowerks.

COM extensions are standard **Shared Libraries**. They just need to have the 'MCEX' file type and the 'CRRA' creator.

Data files (files containing resources) have the "Data" suffix in their name. The file type is not important, but you can use 'Waxed' and the 'CRRA' creator to use the Carrara icons.

5.2.1 Debugging on MacOS

You can debug your extension using the Code Warrior build-in debugger:

1. Make sure you generate a .xSym file and that this file is copied next to the extension in the Carrara Extensions folder.
2. Make sure the MetroNub init is properly installed.
3. Double-click on your .xSym file to open the debugger. Set up your break points, etc.
4. Launch Carrara.

Also you can use the excellent TheDebugger™ from Steve Jasik. This debugger is worth the learning curve to do any serious Macintosh programming. See http://www.jasik.com for more information.

©2001-2002 Eovia. All rights reserved.

©2001-2002 Eovia. All rights reserved.

Chapter 6 - User Interface

MCSketch is a resource editor provided with the SDK. The resources used in Carrara are fully cross-platform, and can be created on either a Macintosh or a PC, since MCSketch exists on both platforms.

In this part, we will describe all the needed resources on both platform (Macintosh or Windows).

Make sure you are familiar with the main concepts of Components before reading this chapter. Specifically, be sure you read "How the whole thing works" on page 9.

6.1.User Interface Options

When you are creating an extension, you need to decide which strategy to use for creating the user interface. For example, if you are creating an extension such as an importer or exporter, you may not even need to create a user interface. Refer to "No User Interface" on page 33 to learn more. Once you've determined that your extension needs a mechanism for user interaction, you still have several different options for creating the user interface. If you only need simple controls, such as a slider or radio button, you can use an automatically generated PMap. Refer to "Auto PMap" on page 33 to learn more.

However, a PMap combined with a **Part** definition allows you the greatest flexibility in creating a user interface for your extension. Refer to "PMap and a Part" on page 34 to learn more.

To summarize, your extension can provide:

- No user interface
- An auto-generated user interface (Auto-PMap)
- A user interface specified with a static resource. In this case:
    - the data buffer is automatically maintained
    - many user interface elements are available: check box, edit text, color picker, component chooser, etc.
- A user interface generated from dta file, using standard MCFrame user interface elements
- A user interface generated from dta file, using extended MCFrame user interface elements (**Part Extensions**)

6.1.1 No User Interface

If you are creating an extension such as an importer or exporter, or if you have created a renderer, such as a postrenderer or final renderer, you may not need to create any user interface elements for your extension; it simply functions without user interaction.

Refer to the Cookbook and the Samples folder to see several examples of extensions created without a user interface.

6.1.2 Auto PMap

A resource "*PMap*" describes the different parameters that the Shell can access. All components with user interface have a PMap resource. In some cases, a PMap is all that you need to create the user interface for your component.

Here is an example of a PMap:

©2001-2002 Eovia. All rights reserved.

```
resource 'PMap' (129)
{
    {   // TODO: Add your parameters by providing:
        //   four letters ID,
        //   four letters type ID,
        //   interpolate or zero,
        //   name
        //   extra token string (usually "")
         'WIEG','re32', interpolate, "Weight!","", // this parameter is animated
        //'WIEG','re32',noFlags,"Weight","", // this parameter is NOT animated
    }
};
```

6.1.3 PMap and a Part

If your extension needs a user interface (UI), and what is provided by the Auto-PMap is not enough, you can use a "part".

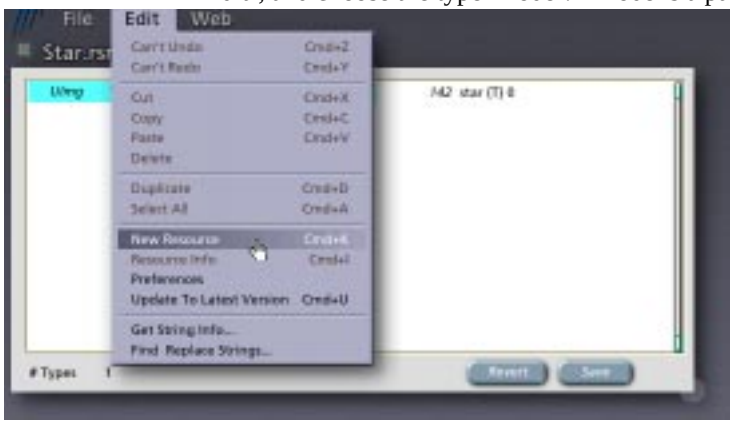
A part is a piece of user interface. You create and edit a part using MCSketch. The main benefit of using a part instead of an Auto-PMap is that you can personalize the UI of your extension with your icons, your names and copyrights, your URL, and so on. The ideal is to let the Auto-PMap do the job while you debug and test the algorithms of your extension, and in the end finalize the extension with the UI.

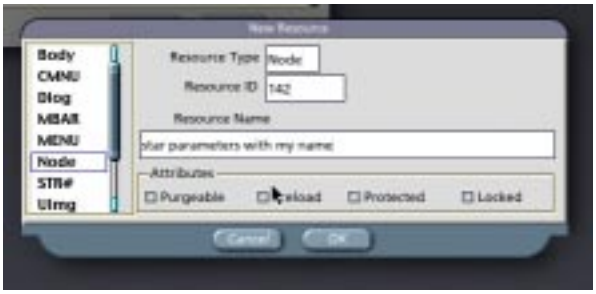
You can combine a PMap with a part definition contained in a .rsr file. If the PMap and the part have the same number, they automatically hook together. Note that the PMap is defined in a .r file, which must include the .rsr file.

For example, the Star sample has no part in its .rsr file, Star.rsr. The PMap defined in Star.r is:

```
resource 'PMap' (142)
{
    {
        'NBBR','in16',interpolate,"# Branche","",
    }
};
```

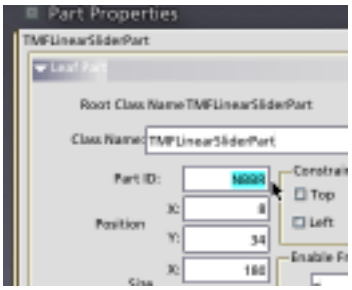
Since there is no part of ID 142, the Star sample uses the Auto-PMap mechanism (which is enough for testing). If you want to add your own UI, open Star.rsr in MCSketch, create a ’New Resource’ via the menu ’Edit’, and choose the type ’Node’. A node is a part that can contain other parts.





Insert in that Node part all the UI elements that you need. For example insert a ’Static Text part’ to show your copyright, and a ’Linear Slider part’ to display and edit the parameter of the PMap. Edit the ID of the slider part to make it fit the ID of the parameter in the PMap: NBBR. This allows the Shell to match the PMap fields and the UI parts as shown.

©2001-2002 Eovia. All rights reserved.



Samples using a PMap and .rsr file

The introduction to the cookbook provides a list of samples that demonstrate creating different user interface elements. To see examples that use a PMap and an rsr file (created using MCSketch), to define the following user interface elements, refer to the samples.

- Checkbox
- Color chooser
- Icon
- Radio button group
- Scrollable edit text
- Sliders
- Dialog boxes
- Windows
- Menus
- Properties (set up and update)
- Sequencer hierarchy

6.1.4 Part Extensions

Another way to create a user interface, is to create **Part Extensions**. Once you create a part extension, you need to tie it to a part of your dialog or window to customize its behavior.

To create a part extension:

1. Create the resource with MCSketch. For example, you could create a node part. Then, add user interface elements such as a slider.
2. Make sure the ID of the part is the same as the resource ID of the PMap. (This is the number in the resource list in MCSketch, and the resource number in the .r file.)
3. Verify that the first field of the PMap in the .r file is set to the part ID (for the parts within the node).
4. Make sure that the object parameters (in the .h file) have a field for each element and that the size of the field matches the PMap elementÕs second field (which is size).
5. Include the resource file in the .r file. (#include new.rsr).

For an example, refer to “Tab Part” on page 39.

6.2.Manually Controlling Your Own User Interface

In some cases, you’ll need to manually control the user interface for your component. The following

©2001-2002 Eovia. All rights reserved.

samples show you how to create and control specific user interface elements.

6.2.1 Creating your User Interface Containers

A dialog box and a window are both examples of user interface containers. The sections below, show how to create them.

6.3.2Using a Dialog box

To manage dialogs, you must get used to the following interfaces: IShResourceUtilities, IShPartUtilities, and IMFDialogPart.

Here is some pseudocode that shows how to create a dialog box.

```
TMCCountedPtr<IMFPart> dialogPart;
TMCCountedPtr<IMFDialogPart> dialog;
void* oldResources = NULL;
boolean result = false;

//-- Load the resources from the DTA file
gResourceUtilities->SetupComponentResources(yourFamilyID,
yourClassID, &oldResources);
gPartUtilities->CreatePartByResource(&dialogPart,
kMFDialogResourceType, yourDialogID);
gResourceUtilities->RestoreComponentResources(oldResources);

//-- Now open the dialog
dialogPart->QueryInterface(IID_IMFDialogPart, (void**)&dialog);
result = dialog->Go();

//-- Get the value if we exited thru OK
if (result)
{
   ... do something ...
}

//-- We are done with the dialog
dialog->Finished();

return result;
```

To learn more about TMCCountedPtr, refer to “Implementing a Counted Pointer Access Method” on page 58.

6.4.3Using a Window

A window can also be created from resources. That is mostly useful when you create a modeler and therefore need to have a window of your own. Here is sample code of the creation of a window. This piece of code comes from the Modeler sample.

```
TMCCountedPtr<IMFPart> window;
TMCCountedPtr<IMFPart> mainPart;

QueryInterface(IID_I3DShModule, (void**) &shModule);
ThrowIfNil(shModule);
```

©2001-2002 Eovia. All rights reserved.

```
void* oldResources = NULL;
gResourceUtilities->SetupComponentResources(kRID_ModuleFamilyID, kMode-
    lerID, &oldResources);

shModule->CreateWindowByResource(&window, kModelerView, true);
ThrowIfNil(window);

// Retrieve a pointer on the main part, which ID is 'MODv' (see in MCSketch)
window->FindChildPartByID(&mainPart, 'MODv');
ThrowIfNil(mainPart);

// Restore the resources chain.
gResourceUtilities->RestoreComponentResources(oldResources);
```

Once your window exists, you can show it or hide it, activate or deactivate it. To do so, use the IMFWindow interface, and call Show() and Activate(). You should only need to do that in the I3DExModule's methods Hydrate/Dehydrate, and Activate/Deactivate.

Check the Modeler sample for implementation.

6.5.4Setting a Child PartÕs Values

Before opening an dialog, you must set up the dialog to the current values, the default values, or the last values that the user entered. To do so, you must first find a pointer on the part's interface. That is done with IMFPart::FindChildPartByID. Use this method on a part at the top of you parts hierarchy to find the item you want to modify, and then change the value of that item either by using its own interface or via the regular IMFPart interface. If we rework the example of the dialog box given above, here is what we can have:

```
TMCCountedPtr<IMFPart> dialogPart;
TMCCountedPtr<IMFDialogPart> dialog;
void* oldResources = NULL;
boolean result = false;

gResourceUtilities->SetupComponentResources(yourFamilyID,
yourClassID, &oldResources);
gPartUtilities->CreatePartByResource(&dialogPart,
kMFDialogResourceType, yourDialogID);
gResourceUtilities->RestoreComponentResources(oldResources);

//-- Now let's say you want to set a item value before opening the dialog:

TMCCountedPtr<IMFPart> editText;
real32 value;
dialogPart->FindChildPartByID(&editText, yourItemID);
if (!editText) return result;
value = 1.0;
editText->SetValue( (void*)&value, kReal32ValueType, true, false);


//-- Now open the dialog
dialogPart->QueryInterface(IID_IMFDialogPart, (void**)&dialog);
result = dialog->Go();

//-- Get the value if we exited thru OK
if (result)
{
   editText->GetValue( (void*)&value, kReal32ValueType);
```

©2001-2002 Eovia. All rights reserved.

```
    ... do something with it...
}

//-- We are done with the dialog
dialog->Finished();

return result;
```

Most of the parts can be accessed using the IMFPart interface. For example, the TMFStaticTextPart, the TMFEditTextPart, the TMFLinearSliderPart, and the TMFRadioClusterPart are handled by the IMFPart interface. Then of course, various IMF* child part examples tie into different IMF* interfaces, like tab parts tie into IMFTabPart.

6.6.Examples of Specific user interface elements

The following examples show how to implement specific user interface elements.

6.6.1 Registration Dialog box

Here is a code snipet that shows a registration dialog that collects a name and number. This illustrated how to use text boxes (EditTextPart parts). Before getting to the code, you need to create the appropriate items in MCSketch:

- Create a ’Dlog’ resource and choose an ID for it (say 1000)
- Put 2 TMFEditTextPart parts inside your dialog. If you drag and drop the Edit Text element of the Part Palette on the left, it comes inside a TMFScrollPart, which will allow your text to scroll. Be careful to select the TMFEditTextPart and not the TMFScrollPart (look at the Part Properties palette on the right)
- Assign a 4 letters "part ID" to each of your TMFEditTextPart parts (say ’Name’ and ’SeNb’).
- Add some Static text parts to have a nice looking dialog. Give a title to your dialog.

```
boolean RegistrationDialog(TMCString& name, TMCString& serialNumber)
{
TMCCountedPtr<IMFPart> dialogPart;
TMCCountedPtr<IMFDialogPart> dialog;
void* oldResources = NULL;
boolean result = false;

name = kNullString;
serialNumber = kNullString;

//-- Load dialog resources
gResourceUtilities->SetupComponentResources(yourFamilyID,
yourClassID, &oldResources);
gPartUtilities->CreatePartByResource(&dialogPart,
kMFDialogResourceType, 1000);
gResourceUtilities->RestoreComponentResources(oldResources);

//-- Find dialog items
TMCCountedPtr<IMFPart> nameTextPart;
TMCCountedPtr<IMFPart> serialTextPart;

dialogPart->FindChildPartByID(&nameTextPart, ’Name’);
if (!nameTextPart) return result;
dialogPart->FindChildPartByID(&serialTextPart, ’SeNb’);
if (!serialTextPart) return result;
```

©2001-2002 Eovia. All rights reserved.

```
//-- Now open the dialog
dialogPart->QueryInterface(IID_IMFDialogPart, (void**)&dialog);
result = dialog->Go();

//-- Get the value if we exited thru OK
if (result)
{
   nameTextPart->GetValue( (void*)&name, kStringValueType);
   serialTextPart->GetValue( (void*)&serialNumber, kStringValueType);
}

//-- We are done with the dialog
dialog->Finished();

return result;

}
```

6.6.2 Tab Part

To create a tab part you'll need to set up the .r and .rsr files as described below.

In the .r file:

- Add a new part in your .r file, that lists all the tabs you want to include in your user interface.
- Create a new resource, called TABS, and select an ID number for this new part. The ID number must be within the range of 127 and 10,000.
- Create a list of all your pages, respectively giving another ID for your page, its name and the ID of the icon with which you want to associate it.

You will obtain something like this:

```
resource 'TABS' ( id_of_tab )
{
      {
           id_of_first_page, name_of_the_page, id_of_the_assiociated_icon,
           ...
       }
}
```

In the .rsr file:
In this file, you first have to create a TabPart. To do so, create a simple node and change the class name into TMFTabPart. This part will contain two other nodes which will be the hosts for the icons and pages areas respectively. Then, put on some parameters in the Custom Tokens area. There actually are four possible tokens:

- TABS: it's the only required parameter. Enter it next to the id you have previously chosen for your tab (id_of_tab in the .r file).
- icoT: it's an optional integer. Set it equal to 1 to associate icons with your tab pages.
- alNI: it's optional boolean. By default, Carrara will put the name of your page above your icons. If you set this boolean to 1, the name of your pages will be then displayed beside your icons.
- Page: it's another optional integer. By default, Carrara won't open any tab page. If you set this integer to a value (between 1 and the number of your pages), Carrara will open the indexed page. This index is based on the order of the pages in the .r file (within the resource 'TABS' you previously created).

To learn more about custom tokens, see "Extra Tokens for Each Part" on page 40.

Creating icons

©2001-2002 Eovia. All rights reserved.

Let's deal now with the icons area. Create a node inside the TMFTabPart, and change the class name into TMFTabArea. Also change the Part ID to TabA. This is the default name for the shell to recognize this node as the host for icons. To finish, put on the 'top', 'bottom' and 'left' constraints.

Then, for the pages area, create a simple node within the TMFTabPart. Change the Part ID into Host. This is also the default name for the shell to recognize this part as the host for user interface tab pages. To finish, set all the constraints to on.

6.6.3 Creating a Component Chooser

Quite often in the interface you need the user to choose an item. And quite often, the possible item is one component plugged into the Shell. For example, the Properties Palette of Carrara for any object allows to apply a Constraint to that object by selecting in the list of all the constraints currentyl plugged into Carrara's Shell. Once chosen, the UI's component shows up, and the user can set the parameters of the component. You may also notice that most of the time, the component chooser is a collapsable part.

That kind of UI item is always done the same way in Carrara. The whole operation is done by what we call the component chooser.

A component chooser is a very powerfull tool, with only a few limitations. For example, although most of the time choosing the component is done for a given family, you are not restricted to only one family. Or you may want the part to be collapsable or not.

There are two major types of component choosers :

- the single component chooser; to be used when you need just ONE component at the time.
- the multiple component chooser; to be used when you want a dynamic list of components.

In this example, let's focus on the most common : the single component chooser. To create a single component chooser you'll need to set up the .r and .rsr files as described below.

In the .rsr file:
In this file, you first have to create a single component chooser. To do so, create a simple node and change the class name into TSingleComponentChooser. Then, put on some parameters in the Custom Tokens area.

To learn more about custom tokens, see "Extra Tokens for Each Part" on page 40.

6.7.MCSketch

zot this is a placeholder. This section needs help from Ian.

MCSketch provides a cross-platform tool for user interface data creation. It adds the ability to edit your own resource template.

6.8.Extra Tokens for Each Part

Many parts have features that do not show up in MCSketch as buttons in the Properties palette. You set the values for these extra features in the "Custom Tokens" box in the Part Properties palette of MCSketch. Warning : there is no control from MCSketch about the content of this text box. Therefore, you need to follow strictly the Token Manager syntax rules (which are the same as the syntax rules of the file format : pairs of token and value).

©2001-2002 Eovia. All rights reserved.

For an example of how to create tokens, see the SDK\Samples\LightSources\Light\Light.r and Light.rsr files.

6.9.List of all UI elements (parts)

When you want to create a UI element in MCSketch (for example, a button in a dialog), you can either drag and drop a preset one from the Part Palette on the left, or select the right menu item from the MFWindow/ Plugin Group#xxx sub menus.

The Part Properties palette then shows several important pieces of information about the part you just created:

- The **Root Class Name** : this is the name of the C++ object behind the element.
- The **Class Name** : often, you want in fact to instanciate a more sophisticated version of this C++ object. For example, you can create a TMFEditTextPart (a text box), but in fact you want a TMF3DUnitEditTextPart (to have units in your text box). Simply type "TMF3DUnitEditTextPart" in the Class Name field.
- The **Custom Tokens** field : allow you to set up additionnal parameters to your part if they cannot be set through an editor (see "Extra Tokens for Each Part" on page 40).
- The Editor specific to the part : for example, a TMFEditTextPart editor allows you to change the default text, set the text type, justification, etc.

zot : what follows is a template of the information that MUST be provided for each part. There is a lot of things, but this is still the bare mininum. The hardest part are the Custom Tokens. Also,if there is an Editor for the part, then we should list only the Tokens that are NOT tied to the editor (this is what I did for the Edit Text part sample). - Yann

6.9.1 TMFEditTextPart - Edit Text part

Purpose : Free text entry or numerical values entry
Root Class : TMFEditTextPart
Default Class Name : TMFEditTextPart
Other possible Class Names : TMF3DUnitEditTextPart
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types : kInt16ValueType, kInt32ValueType, kReal32ValueType, kInt32MinMaxOnlyValueType, kReal32MinMaxOnlyValueType, kStringValueType
Editor in MCSketch : yes
Custom Tokens :

Token	Type	Default value	Purpose
NtyA	boolean	false	Notify on all changes (not just the end of the editing)
SpnB	boolean	true	Display a "spin box"
Ntfy	boolean	true	Notify if value is out of range

Comments : None

6.9.2 TMFIconButtonPart

Purpose : **zot**
Root Class : TMFIconButtonPart
Default Class Name : TMFIconButtonPart
Other possible Class Names : none
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder

©2001-2002 Eovia. All rights reserved.

Supported value types :
Editor in MCSketch : yes
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.3 TMFTextButtonPart

Purpose :
Root Class : TMFButtonPart
Default Class Name : TMFButtonPart
Other possible Class Names : TMFIconButtonPart,
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch : yes
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.4 TMFDialPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.5 TMFHierarchicalListPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder

©2001-2002 Eovia. All rights reserved.

Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.6 TMFImagePart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.7 TMFCheckboxPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.8 TMFColorPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder

©2001-2002 Eovia. All rights reserved.

Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.9 TMFOverlayImageControlPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.10 TMFRadioPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.11 TMFToolbarPart

Purpose : Selection of tools
Root Class : TMFNodePart
Default Class Name : TMFToolbarPart
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder,

©2001-2002 Eovia. All rights reserved.

IMFToolbarPart
Supported value types : kInt16ValueType, kInt32ValueType
Editor in MCSketch : no
Custom Tokens :

Token	Type	Default value	Purpose
HorL	boolean	true	Defines whether the toolbar is in a horizontal (true) or vertical (false) orientation
TBAR	int16	0	Reference to a TBAR resource whose tools will be loaded into the toolbar
Glob	boolean	true	Defines whether the toolbar is global (true) or local (false). Global toolbars enforce mutual-exclusivity of tool selection across all global toolbars, while local toolbars only enforce it within itself.
TlRc	IDType		Defines the part ID of the part that should receive tool selections (SelfMenuAction or SelfToolAction). If this token is not defined, tool selections will be passed to the first responder.
SpSz	int32	5	Defines the space between tools
RAct	boolean	false	If true, tools will respect the activeness of their enclosing window (i.e., tools will be disabled if the window is disabled). If false, tools will ignore the window activeness.

Comments :

6.9.12 TMFNodePart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.13 TMFAutoCenteringPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

©2001-2002 Eovia. All rights reserved.

Comments :

6.9.14 TMFBalloonPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.15 TMFWindoidPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.16 TMFDialogoidPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

©2001-2002 Eovia. All rights reserved.

Comments :

6.9.17 TMFCollapsiblePart, TMFSelectableCollapsiblePart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, IMFResponder, and IMFCollapsiblePart
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.18 TSingleComponentChooser

Purpose : **Let the user choose a component from a pop-up menu. Many features included automagicaly, like selection/drag'n'drop, mini-part display, etc.**
Root Class : TMFLeafPart
Default Class Name : TSingleComponentChooser
Other possible Class Names : TSingleCompChooserWithWireFrame, TSingleCompChooserWithPreview
Supported interfaces : IMCUnknown, IMFPart, IShComponent, IMFResponder, IMFCollapsiblePart, and ISingleComponentChooser
Supported value types : 'comp' (AKA kComponentValueType
Editor in MCSketch : no
Custom Tokens :

Token	Type	Default value	Purpose
fmly	IDType		Family ID of the components to fit into the menu.
bnon	boolean	true	Is ÕnoneÕ a valid option ?
shof	boolean	true	Show the family name.
MskI	uint32	0	Include Mask for the pop-up menu
MskE	uint32	0	Exclude Mask for the pop-up menu
Sort	boolean	false	Sort the pop-up menu by family name.
Subm	boolean	false	Sub-menu mode : true will create sub-menus of the items by sub-family. The title of the sub-menu is the sub-family string (in the COMP resource).
Mini	boolean	false	Use a mini part ? If true, the collapsible state displays the part that is specified in the COMP resource of the component, and linked to the PMap.
Name	string		NAme of the component chooser.
shpp	boolean	true	Show the pop-up.
SHos	IDType		ID of the selectable host.; that must be a parent of the chooser.

Comments :

6.9.19 TMultipleComponentChooser

Purpose : **zot**
Root Class :

©2001-2002 Eovia. All rights reserved.

Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.20 TMFFramePart

Purpose : Draws a frame in the bounds of the part
Root Class : TMFNodePart
Default Class Name : TMFFramePart
Other possible Class Names : none
Supported interfaces : IMCUnknown, IMFPart, IShComponent, IMFResponder, and IMFFramePart
Supported value types : none
Editor in MCSketch : no
Custom Tokens : none
Comments :

6.9.21 TMFGradientPart

Purpose : Draws a horizontal or vertical gradient between arbitrary colors and transparencies
Root Class : TMFNodePart
Default Class Name : TMFGradientPart
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types : none
Editor in MCSketch : no
Custom Tokens :

Token	Type	Default value	Purpose
SrtF	Fill Attr.	black	Start Fill
EndF	Fill Attr.	white	End Fill
SrtT	real32	0	Start Transparency (0.0 to 1.0)
EndT	real32	0	End Transparency (0.0 to 1.0)
Vert	boolean	false	Defines whether the gradient runs left to right (false) or top to bottom (true)

Comments :

6.9.22 TMFNodeReferencePart

Purpose : **zot**
Root Class :
Default Class Name :

©2001-2002 Eovia. All rights reserved.

Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.23 TMFRadioCluster

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.24 TMFScrollPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.25 TMFSlider, TMFCircularSliderPart, TMFLinearSliderPart

Purpose : **zot**
Root Class :
Default Class Name :

©2001-2002 Eovia. All rights reserved.

Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.26 TMFSolidColorPart

Purpose : **Fills the part bounds with a color**
Root Class : TMFNodePart
Default Class Name : TMFSolidColorPart
Other possible Class Names : none
Supported interfaces : IMCUnknown, IMFPart, IShComponent, IMFResponder, and IMFSolidColorPart
Supported value types : none
Editor in MCSketch : no
Custom Tokens :

Token	Type	Default value	Purpose
Fill	Fill Attr.	white	Fill (color)
Trns	real32	0	Transparency (0.0 - 1.0)

Comments :

6.9.27 TMFSplit3Part

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.28 TMFCollapsibleListPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :

©2001-2002 Eovia. All rights reserved.

Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.29 TMFVectorPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.30 TMFListPart, TMFRuledListPart, TMFStringListPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.31 TMFOffscreenPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :

©2001-2002 Eovia. All rights reserved.

Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.32 TMFPoppedPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.33 TMFRectPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.34 TMFStaticTextPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :

©2001-2002 Eovia. All rights reserved.

Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.9.35 TMFTextPopupPart

Purpose : **zot**
Root Class :
Default Class Name :
Other possible Class Names :
Supported interfaces : IMCUnknown, IMFPart, IShComponent, and IMFResponder
Supported value types :
Editor in MCSketch :
Custom Tokens :

Token	Type	Default value	Purpose

Comments :

6.10.Interfaces supported by each UI element (part)

The following table summarizes the list of implemented interfaces for each User Interface element that you can create in MCSketch. This information can also be found in the invidual decription of each part (see "List of all UI elements (parts)" on page 41). Very often, you need to talk to a part using several interfaces, using QueryInterface to get pointers on these interface. In the dialog example above, we talk to the dialog as a IMFPart and as a IMFDialogPart.

All parts implement :

IMCUnknown, IMFPart, IShComponent, and IMFResponder.

Most of the time, using IMFPart::SetValue, IMFPart::GetValue, and other IMFPart calls is good enough to set up your UI elements. However, some parts implement additionnal interfaces for special purposes.

©2001-2002 Eovia. All rights reserved.

Table 5: Additionnal interfaces supported by some UI elements

Class Name	Additionnal supported interfaces
TMFCheckboxPart TMFColorPart TMFOverlayImageControlPart TMFRadioPart	IMFImageControlPart
TMFToolbarPart	IMFImageControlPart, IMFToolbarPart
TMFWindoidPart	IMFWindow
TMFDialogoidPart	IMFWindow, IMFDialogPart
TMFFramePart	IMFFramePart
TMFCircularSliderPart TCircularConstraintSliderPart TMFLinearSliderPart TLinearContraintSliderPart	IMFSliderPart
TBitmapSliderPart	IMFSliderPart, IMFBitmapSliderPart
TMFSolidColorPart TColorPickerSwatch	IMFSolidColorPart
TMFTabPart	IMFTabPart
TComponentUIPart	IMFParameterComponentPart
TParameterPart	IMFParameterPart
TTextureMapPart	ITextureMapPart
TMFListPart TMFRuledListPart	IMFListPart
TMFTextPopupPart TMF3DUnitsPopupPart TMFFontListPopupPart TMFFontStylePopupPart TMFIconPopupPart	IMFTextPopupPart

©2001-2002 Eovia. All rights reserved.

Chapter 7 - Counted Pointers

Using counted pointers allows you to optimize performance by avoiding calling AddRef() or Release() directly.

Note: After construction of objects, reference count should be 1.

Before you begin, it is important to outline why reference counting is important:

1. It is much safer. You can much more easily avoid double-deletes, leaks and other memory problems using reference counting. Sometimes you need to keep references to things that normally would be deleted. For instance, in TMFAction and subclasses, it is convenient to keep references to items that are removed, reordered, deleted, etc. Without reference counting, it may be impossible to keep the pointer valid without leaking.
2. It is actually easier in some cases to keep reference counts. Keeping reference counts alleviates the need for ownership of pointers. To use a metaphor: last one out turns off the lights.
3. You have no choice. In COM-based applications, reference counting is required. For instance, in order to pass an ISceneDocument interface to an external component, you need to implement a reference count for that object. If you simply reference count TSceneDocument, and not its parent classes (including TMFResponder), then half of the object is reference counted and half is not. And once the reference count becomes zero, the object is destructed, even though the pointers that were not reference counted still think that the object is valid.

7.1.Defining Classes that May Need to be Counted

Here are some rules for defining classes that may need to be counted:

- Derive your class from TMCCountedObject.
- Make the destructor protected; it can be private if there is no sub-class.
- Make the constructor protected; it can be private if there is no sub-class.
- Provide a static “Create::(T**)” method. (“Create::(T** P1, ...)” if other parameters are needed for contructor.)

TMCCountedObject/TMCSMPCountedObject

These objects implement simple reference counting (the SMP variation uses an atomic counter for things that need one, like components). The reference count starts at zero.

Note: AddRef() and Release() need to be virtual MCCOMAPI's so that real COM objects can subclass from TMCCountedObject and override the functions. However, this should not add any size overhead, since subclasses of TMCCountedObject should already have virtual destructors.

Destructors

Destructors for all subclasses of TMCCountedObject should be protected and virtual. You should never call delete on any such object — use Release() instead.

Be careful: C++ lets you make members more public without warning (so check your class to make sure what access level the destructor should be).

Also, be sure you do not rely on destructors to remove their objects from global lists.

©2001-2002 Eovia. All rights reserved.

This clearly cannot work, since if an object is on a global list (that is presumably reference counted), the destructor will never be called, since there will always be a reference to the object.

Using AddRef() and Release()

Basically, there are only two rules for proper reference counting:

1. Call AddRef() whenever you assign a reference counted pointer to a variable.
2. Call Release() whenever you are done.

Experienced COM programmers out there might realize that we are one rule shy of listing all of the COM rules for reference counting. The rule that we are missing is: Call AddRef() before returning. That's true, it's gone, but because of the way TMCCountedPtr works, we need to adjust for it. TMCCountedPtr AddRef()s on assignment, which includes function return values. So technically, the AddRef() occurs after the function return, so rule (1) applies.

However, if you don't use a TMCCountedPtr, you need to explicitly call AddRef() on function returns when they return a reference counted value. This impacts third-party developers, depending on whether or not we give them TMCCountedPtr.

Note: We're still OK on QueryInterface() though, since the assignment happens inside the function, so you won't need to call AddRef() after returning from QueryInterface().

TMCCountedPtr

The above rules imply a lot of calls to AddRef() and Release(), which can clutter the code to unreadability. TMCCountedPtr's make things much easier since they call AddRef and Release to implement the rules above.

TMCCountedPtr is a behavior-wrapper for a pointer to any class that implements AddRef() and Release() (including TMCCountedObject). TMCCountedPtr overrides many operators (including *, ->, &, =, and cast) in order to call AddRef() and Release() at appropriate times and check pointers when necessary. See TMCCountedPtr.h for more information.

The only data in TMCCountedPtr is the pointer itself and there are no virtueless, so the class has no size overhead.

It is a template class and should be used thus:

```
TMFResponder*            fMyResponder;  //old becomes
TMCCountedPtr<TMFResponder> fMyResponder;  //new
```

Since the destructor calls Release(), TMCCountedPtrs are best used as stack-based objects or class data members only. Do not use TMCCountedPtr* or use TMCPtrArray<TMCCountedPtr<T>*>, since you want their destructor to be called automatically.

Do not pass TMCCountedPtr's as arguments to functions. It is not necessary, since passing parameters does not keep a reference (so no AddRef() needs to be called).

Remember, only if you assign to a variable do you need to call AddRef().

TMCCountedArray

TMCCountedArray is to TMCPtrArray what TMCCountedPtr is to T*.

TMCCountedArray<T> is a template class that is a subclass of TMCClassArray< TMCCountedPtr< T > >.

©2001-2002 Eovia. All rights reserved.

It should be used thus:

```
TMCPtrArray<TMFResponder*>  fMyResponders;  //old becomes
TMCCountedArray<TMFResponder>fMyResponders;  //new
```

These arrays are a cross between TMCPtrArray and TMCClassArray, since they have properties of both. For instance, TMCCountedArray<T>::iterator::First() returns a TMCCountedPtr<T> (like a TMCPtrArray), but it is a class array, so destructors are automatically called (and thus Release()) on removal from the list.

Miscellaneous

Note: You also cannot make TMCCountedObjects on the stack, since that involves an implicit destructor at the end of the scope. For instance, you cannot do this:

```
{
        TMFWindow myWindow(...);
}
```

Instead, use a TMCCountedPtr and a new operator. The release will be called by the implicit destructor on the TMCCountedPtr.

```
{
        TMCCountedPtr<TMFWindow> myWindow = new TMFWindow(...);
}
```

Note: You need to be careful to handle cycles correctly. A cycle occurs when object A has a reference to B and B has a reference to A. This happens with TMFNodePart, which has references to its children, and each child has a reference to its parent.

Reference Count Optimizations

TMCCountedPtr's are indiscriminate. They reference count everything, even when you don't really need to. For instance,

```
{
        TMCCountedPtr<TMFResponder> r = GetFirstResponder();

     r->DoSomethingThatWillNotRelease();
        r->DoSomethingElse();
}
```

You do not really need a counted pointer here, since you are not keeping a reference beyond the scope of the function, and the function DoSomethingThatWillNotRelease() will not decrement the reference count on r, so it won't go away out from under you.

7.1.3 Summary: How to Convert “Old” Files to “New” Files

1. Make destructors or subclasses of TMCCountedObject protected.
2. Make data members and local variables of type TMCCountedObject* (or subclasses) into TMCCountedPtr<TMCCountedObject> (or subclass).
3. Make data members and local variables of type TMCPtrArray<TMCCountedObject*> into TMCCountedArray<TMCCountedObject> (or subclass).
4. Change TMCPtrArray::GetEqualItemNo() into TMCCountedArray::FindElem().
5. Change TMCPtrArray::Remove() into TMCCountedArray::RemoveElem().
6. Replace delete fMyObject by fMyObject.Release() (fMyObject should now be of type TMCCountedPtr).
7. When passing a TMCCountedPtr<T> to a function as a parameter that expects a T*, use TMCCount-

©2001-2002 Eovia. All rights reserved.

edPtr::GetAddress() rather than the & operator, since that returns a T**.

7.1.8 Implementing Create

Implement Create as follows:

```
void TMyClass::Create(TMyClass** a)
{
   ReleaseIfNotNull<TMyClass>(a);
   *a = new TMyClass;
   ThrowIfNil(*a);  // can also use error code
}
```

Accessing Counted Objects

Don't return counted pointers. Use Get(TMyClass**) to access them.

Only in exceptional cases, should you use TObjectClass* GetNoAddRef().

Implementing a Counted Pointer Access Method

```
void TMyClass::Get(TObjectClass** a)
{
   if (MCVerify(a)
   {
      ReleaseIfNotNull<TMyClass>(a);
      *a = fObject;  // fObject is a counted pointer
      if (*a)
         (*a)->AddRef();
   }
}
```

OR (in exceptional cases)

```
TObjectClass* GetNoAddRef() {return fObject}
```

7.1.9 Implementing QueryInterface()

Always return something for IUnknown.

Another example:

```
void TRasterLayer::CreateChannel(IShChannel** ch)
{
   ReleaseIfNotNull<IShChannel>(ch);
   TChannel* newC = NULL;
   TChannel::Create(&newC);  // don't use counted ptr
   *ch = newC;
}
```

7.1.10 Helper Classes for Counted Pointers

Carrara provides two helper classes for Counted Pointers: TMCCountedGetHelper and TMCCountedCreateHelper.

©2001-2002 Eovia. All rights reserved.

From now on, you should use these classes instead of calling AddRef and Release directly. Furthermore, you shouldn't have to use ReleaseIfNotNull anymore. Either of these two classes should do.

TRefAddingPtr is now obsolete (but mapped to TMCountedGetHelper for backward compatibility).

You use TMCCountedGetHelper to implement a “Get” operation, or a similar interface. This will Release the previous object if there was any, clear the handle, and help you affect a new value, and do the AddRef. It will check as many things are OK as possible.

You use TMCCountedCreateHelper in a “Create” method (Typically when you create your counted object with the new operator (which initializes the RefCount to 1)). It also does plenty of validity checks.

It is *imperative* to use these new classes if you want to be able to track AddRefs/Release calls. Without that, the debugging code can’t match AddRef and Release. It is *imperative* to *not* call AddRef and Release explicitly, as these calls can't be tracked either.

All COM objects should implement the proper QueryInterface for IID_IMCUnknown.

Here are a few examples of uses for these two new classes (please study them carefully.)

Example of TMCCountedGetHelper

```
//Use with void**
//-----------------------------------------------------------------------------
MCCOMErr TMFBaseWindow::QueryInterface(const MCIID& riid, void** ppvObj)
//-----------------------------------------------------------------------------
{
   if (MCIsEqualIID(riid, IID_IMFWindow))
   {
      TMCCountedGetHelper<IMFWindow> result(ppvObj);
      result = (IMFWindow*)this;
      return MC_S_OK;
   }
   return TMFNodePart::QueryInterface(riid, ppvObj);
}

//Use with type, no conditional affectation
void TMFBaseWindow::GetWindowFirstResponder(TMFResponder** outResponder)
const
{
   TMCCountedGetHelper<TMFResponder> result(outResponder);
   result = fFirstResponder;
}


//Use with type, conditional affectation *outDocument can be NULL at the

void TMFDocument::FindDocForFile(TMCFile* afile, TMFDocument**
outDocument)
{
   TMCCountedGetHelper<TMFDocument> result(outDocument);
   TMCCountedPtrArray<TMFDocument>::iterator
iter(TMFDocument::sDocumentList);
   for (TMFDocument* aDocument = iter.First(); iter.More(); aDocument =
iter.Next())
   {
      if (aDocument->FindDoc(afile))
      {
         result = aDocument;
```

©2001-2002 Eovia. All rights reserved.

```
        }
    }
}

//a more complex use of the GetHelper, you need to use both a TMCCountedPtr and
//a TMCCountedGetHelper because the type are different. If, like before, you were
//not using a TMCCountedPtr for 'component', the reference tracking would be
//wrong.
void TComponentUI::CloneComponent(IShParameterComponent **res)
{
    TMCCountedGetHelper<IShParameterComponent> result(res);
    TMCCountedPtr<TComponent> component;
    Clone(&component);
    result=static_cast<TComponentUI*>(*&component);
}
```

Example of TMCCountedCreateHelper

```
//example of TMCCountedCreateHelper
void TMFWindoidPart::Create(
    const TMCWindowInfo& inWindowInfo,
    const EWindoidTypeinWindoidType,
    TMFWindoidPart**outObject)
{
    TMCCountedCreateHelper<TMFWindoidPart> result(outObject);
    result = new TMFWindoidPart(inWindowInfo, inWindoidType); //Init count
to 1, no AddRef
}

//an other example of TMCCountedCreateHelper
void TBrushMask::Clone(TComponent **res)
{
    TMCCountedCreateHelper<TComponent> result(res);
    TComponentUI* newComp = new TBrushMask(); //Init count to 1, no AddRef
    CloneComponentUIData(newComp);
    result=newComp;
}
```

7.1.11 CountedPtr vs Interface Pointer

Always use CountedPtr in function cores but pay attention to cycle references when the pointers are contained in an object.

Here is an example of the correct way to use a CountedPtr:

```
        {
            TMCCountedPtr<interfaceType> ptr;

            QueryInterface(IID_interfaceType,(void**)&ptr);

            .....

            ptr->function(...);

`            ....
        }
```

The same code with interface Ptr:

```
        {
```

©2001-2002 Eovia. All rights reserved.

```
            interfaceType *ptr=NULL;   // don't forget to initialise to NULL
// !! (not necessary with counted ptr)

            QueryInterface(IID_interfaceType,(void**)&ptr);

            .....

            ptr->function(...);

            ....

            ptr->Release();   // do not forget this or will leak
        }
```

Two functions returning InterfacePtr

There are two ways to get an InterfacePtr:

```
           interfaceType *GetInterface();            // does no AddRef
            GetInterfaceType(interfaceType **);   //  does a AddRef USE THIS ONE
```

The first solution is the old one; before it was doing a AddRef but it should not anymore.

You should replace all the functions of the first type by functions of the second type because they are safer and you do not need to care about the AddRef. In both cases, you should always Release the pointer returned when you are done using it.

For example:

```
       {
            TMCCountedPtr<interfaceType> ptr;    // or interfaceType *ptr;

            obj->GetInterface(&ptr);

            ....

            ptr->function()

            ....
       }                                     // if you are not using CountedPtr
       , you need to add: ptr->Release();
```

Here is how to implement such a function (it should be done in QueryInterface too.):

```
      void GetInterface(interfaceType **ptr)
{
    if (*ptr)
    {
        (*ptr)->Release();
        *ptr = NULL;               // important in case of failure
    }

    (*ptr) = ...
    (*ptr)->AddRef();

}
```

Another possible prototype is:

©2001-2002 Eovia. All rights reserved.

```
void GetInterface(InstanceType *&ptr);
```

This works too but it not very easy to use with a countedPtr as you have to write: GetInterface(*(&ptr)); so you should use the InterfacetType ** style.

Never return a CountedPtr

For example:

```
interfaceType *Getfdfkdsjfk()
{
     TMCCountedPtr<..> ptr;
        ....
    return ptr;
}
```

What happens in this case is that the ptr will be released at the end of the function. So you will pass a reference to something that has just been released. This may cause crashes, specially in optimize compiled code, since the debugger often modifies the scope of variable for debugging purposes. Therefore those crashes are very difficult to find. If you happen to have crashes in optimize compiled code but not in debug compiled code, check the returned arguments for countedPtrs and local variables.

Creating objects

Objects should have a reference count of one when you just called their constructor. This is necessary for aggregation purposes. So if you write a CoCreateInstance function you should be careful to do a Release() after the QueryInterface or your reference count will be 2 when you get out of CoCreateInstance.

Aggregation

If an object keeps a reference to an interface to an other object with which it is aggregated, this pointed should not be counted (as it would make it impossible to delete the object).

However you usually need to call QueryInterface to get this internal reference which will do a AddRef on the aggregated object. So be sure to call Release just after the QueryInterface. This is possible as the reference count is initialized to 1.

Not all of the points concerning the aggregration are discussed here. You should at least be aware that QueryInterface and Release cannot be done in an aggregation as in a simple object.

©2001-2002 Eovia. All rights reserved.

Chapter 8 - Utilities

This chapter describes details about the utilities included with Carrara.

Many global utilities are available to extensions. They are initialized automatically by the SDK library when the Extension is loaded. You access these utilities through global variables.

Examples:

- File and streaming
- Memory allocation, off screen buffers, pixel iterators
- Menus, mouse action (for tracking), drag/drop
- Component utilities
- Etc.

8.1.Various Shell Utilities (IShUtilities)

Use the **IShUtilities** interface with the **gShellUtilities** pointer to access to various services from the Shell. These services include :

- Progress bar management
- Cooperative multi-tasking (YieldProcesses), time count (TickCount)
- Random number generation
- Low level alerts
- Image "pixel buckets"
- Open documents browsing
- MMX utilities (Windows only)
- Text metrics, cursor, fonts list, keyboard low level tests, etc.

8.2.Thread Utilities (IShTreadUtilities)

IShTreadUtilities and **gShellTreadUtilities** are used to launch and kill multi-tasking threads.

8.3.Component Utilities (IShComponentUtilities)

Use **IShComponentUtilities** and **gComponentUtilities** to manage components from your extension. The main purpose is to create components. You can then use QueryInterface to initiate a more personalized discussion with the component.

8.4.SMP Utilities (IShSMPUtilities)

Use **IShSMPUtilities** and **gShellSMPUtilities** for multi-tasking management (lauchning a new thread, killing one, etc.). See “SMP” on page 75 for details about multi-tasking.

8.5.Action Utilities (IShActionManager)

IShActionManager and **gActionManager** are used to post Actions and Mouse Actions. You can also test the status of modifier keys of the keyboard (Control, Alt, etc.).

©2001-2002 Eovia. All rights reserved.

8.6.Menu Utilities (IShMenuUtilities)

Use **IShMenuUtilities** and **gMenuUtilities** to manipulate menus: you can dynamically build menus, enable and disable menu items, etc.

8.7.Part Utilities (IShPartUtilities)

IShPartUtilities and **gPartUtilities** are used to create Parts (usually from a resource built in MCSketch). The most useful call of this interface is CreatePartByResource.

8.8.Resource Utilities (IShResourceUtilities)

Use **IShResourceUtilities** and **gResourceUtilities** to load any kind of resources stored in your DTA file by using GetResource or GetIndResource.

Use GetIndString to load string resources (STR#) with automatic Windows-to-MacOS or MacOS-to-Windows ASCII translation.

Before any resource loading operation, make sure that the current resource file is properly set on your DTA file by encapsulating your resource calls between SetupComponentResources and RestoreComponentResource.

8.9.File Handling (IShFileUtilities)

The utilities used to access and manipulate file streams are **IShFileUtilities**, **IShFileFormatUtilities** and **IShFileStream**. Another interface that you should become familiar with is IMCFile.

The global variables to use are gFileUtilities and gFileFormatUtilities.

8.9.1 Creating Files

When creating files, you never inherit from IMCFile. When you need to create a file, create it through IShFileUtilities::CreateIMCFile by doing the following:

```
TMCCountedPtr<IMCFile> myFile;

gFileUtilities->CreateIMCFile(&myFile); // Note the '&'

//Then you can use myFile at your leisure (don't forget it's a COM object)
myFile->....
```

Remember that when you see a Shell procedure with a double deferencing in its parameters list, it means a returned COM parameter, very much like in QueryInterface.

Examples:

```
IShUtilities::GetFontList(IMCFontList** outFontList);
IShMenuUtilities::CreateMenu(TMFMenuInfo& inInfo, IFMMenu** outMenu);
```

©2001-2002 Eovia. All rights reserved.

8.9.2 Opening Files

Once you created a IMCFile object, you can use it to manipulate a file. Link the interface to the physical file using SetWithPathname or SetWithFullPathname.

Example:

```
TMCCountedPtr<IMCFile> aFile;
gFileUtilities->CreateIMCFile (&aFile);

TMCDynamicString path;
gShell3DUtilities->GetLastImageOpenPath(path);
aFile->SetWithPathName(path);

gFileFormatUtilities->OpenFileDialog(DialogName, aFile,
   inTypes,Format1,Format2);
aFile->GetFileName(pathName);
```

8.9.3 Working with Streams

This example shows how to write out to a stream.

```
TMCfstream* stream = NULL;
uint32 nbInst = 0;
TMCCountedPtr<I3DShScene> scene;

   TMCString1023 fullPathName;
   file->GetFileFullPathName(fullPathName);
   stream = new TMCfstream(fullPathName.StrGet(),TMCiostream::out);
   ThrowIfNil(stream);



   //     FailNIL(stream);
   //     FailOSErr((int16)stream->InitFileStream(fullPathName, kSh-
   StreamOut));

   //stream->SetMacOSInfo('ttxt','TEXT'); // Needed on the Mac, Will not do
   anything on the PC

   WriteDXFBegin(stream);
```

8.10.Personality Utilities (IShPersonalityUtilities)

IShPersonalityUtilities and **gPersonalityUtilities** can be used to get the color settings of the UI. You will probably never need it.

8.11.Drag & Drop Utilities (IShDragAndDropUtilities)

IShDragAndDropUtilities and **gDragAndDropUtilities** and used for 2 things: creation of drop candidates and drop areas, and launching a drag-and-drop action.

©2001-2002 Eovia. All rights reserved.

8.12.Change Management (IChangeManager)

IChangeManager and **gChangeManager** are used to Change Management. Change Management is a powerful messaging feature in Carrara. It works by creating "channels" (see **IChangeChannel** interface), to which "listeners" (**IChangeListener**) can subscribe. Anyone can then "post" messages on this channel, all listeners will hear about them.

8.13.Clipboard Utilities (IMFClipboard)

IMFClipboard and **gClipboardUtilities** are used to implement copy/paste across extensions and across the Shell. A **IMFClipping** interface is used to encapsulate the copied data.

8.14.3D Utilities (I3DShUtilities)

I3DShUtilities and **gShell3DUtilities** give a lot of utility routines specific to Carrara (the previous utilities were more focused on the framework itself, not the 3D stuff). There are lots and lots of goodies in there, from documents, rooms and modules management, to animation utilities and shader utilities.

©2001-2002 Eovia. All rights reserved.

Chapter 9 - MCCommon References

This chapter introduces the MCCommon references included with Carrara. It provides an overview of the application framework. Some of the MCCommon classes are documented in the Reference.

The MCFrame application framework provides an open architecture that allows you to create a seamless integration of externally developed components. It includes a single user interface format (similar to the dta format on the Macintosh) for cross platform use. There is also a cross-platform tool provided for user interface data creation. The framework is composed of several groups of interfaces; each group is discussed in the sections that follow.

It provides basic classes with default behavior for most application-independent functionality such as windows, controls, dialogs, files, etc.

Generally, you will override the basic classes to provide application-specific functionality. For instance, Carrara overrides TMFApplication to TShellApplication, TMFFileDocument to TSceneDocument, and many dozens of parts.

You have access through extensions to much of the low and high level of the framework.

The framework is composed of:

- MCCore
- MCFrame
- MCImage

Resources are also stored in a platform-independent way. Carrara has a resource file. The resource file sits next to the application (or DLL) and contains almost all of the resources the application needs to run (the exceptions are platform dependent resources like BNDL and other such resources on the Mac). The data file is the same on both platforms and contains a dump of the Macintosh resource fork. The Macintosh Resource Manager is not used by MCCore; rather, MCCore reads resources itself. The tools “Data Resource Compiler” and “Res2Data Postlinker” translate data files to their Mac resource equivalents and vice versa.

Each class in the platform independent part generally begins with TMF. Each MCCore class begins with TMC. Each class in the platform dependent part generally begins with either TMAC_MC or TWIN_MC.

9.1.MCCore Interfaces

MCCore provides low-level I/O functionality such as files, streams, and memory. The associated files are the IMC* and MC* headers. MCCore provides a level of abstraction to hide platform-specific code.

There are currently only four interfaces:

- IMCGraphicContext: provides Draw, Fill, Blit, GetSystemGCPtr, etc.
- IMCFile: provides Get/Read/Write files
- IMCFont and IMCFontList: provide accessors to the fonts, individually or as a list.

There are also many types and utilities available. The types are available in the files in /Include/MCCommon/MC*.h Many of those files contains fully implemented utilities, like array templates in MCArray.h, strings templates in MCString.h, color, rect, matrix, etc.

©2001-2002 Eovia. All rights reserved.

9.2.MCF Interfaces

MCF provides user interface level functionality such as view system, menus, and widgets. The associated files are the IMFEx*, IMF*, and MF* headers. MCF provides a hierarchical view system (replacing the parts system) at the user interface level of the framework.

There are two main interfaces:

- IMFExPart: provides SelfDraw, SelfMouseDown, SelfMouseMoved, etc.
- IMFExResponder: provides Receive, SelfMenuAction, SelfPrepareMenu, SelfKeyDown/Up, etc.

User Interface Flexibility

MCF interfaces provide complete user interface flexibility to extensions.

Extension user interface is laid out using MCSketch. Refer to "MCSketch" on page 40 to learn more.

You can add a part extension for any visible user interface part component that you want to extend. When the shell loads that user interface element from a file, it sees the part extension, and connects the internal user interface part to the extension part. An extension can over-ride any method in the part to put in specialized behavior. An external part can also call back to internal part for standard behavior from its method (simulating inheritance!)

9.3.MCImage Interfaces

MCImage provides data storage and shared code area.

MCImage provides interfaces for:

- Color models and color pickers
- Application independent Import/Export (2D/3D specific Import/Export APIs too)
- RasterLayer Utilities (basic layer/channel/tile/pixel interface)
- MC Photoshop Host (ImageIO based component)

©2001-2002 Eovia. All rights reserved.

Chapter 10 - Failure Handling

10.1.Using the Failure Handling

Correctly handling the Failure Handling of Carrara will ensure that your extension will perform in the best conditions and protect your user for unpleasant situations...

The failure handling in Carrara is based on the standard C++ catch-and-throw scheme. Ray Dream Studio developers will have to replace their TRY, SUCCESS and CATCH macros by the standard C++ 'try' and 'catch' keywords.

10.2.General rules

- If anything goes wrong in your code, you are allowed to fail (by calling **throw**), Carrara will catch the failure event and displays the infamous "Program Error" dialog box. After that, normal life starts again, so your user can still save his/her work.
- Everytime you call a Shell routine, it can fail (especially when allocating memory). So your extension may want to be prepared for it to perform some clean-up during the Failure Handling phase, or even analyze the problem, treat it, and resume execution. You do this by implementing a **Failure Handler**, using the **catch** keyword.
- Since the scheme is not based on returning error codes, your code stays clean and readable. You implement Failure Handlers only where necessary.

10.3.A first simple example

Consider the following code. We are going to allocate 2 buffers, and we want to make sure that the first buffer is deleted if the allocation of the second one failed. We also want to delete them if anything goes wrong in the other procedures we call when using these buffers.

```
void FailureSample1()
{
   char* buffer1 = NULL;  // Always initialize to NULL
   char* buffer2 = NULL;  // Always initialize to NULL

   try              // This encapsulates the "protected" code
   {  //-- "Protected" code starts here --
      buffer1 = (char*) malloc(128000);  // malloc will fail if unsuccessful
      buffer2 = (char*) malloc(256000);  // Still here ? Go for the 2nd malloc!

       ..... do whatever you like here (it can trigger failures just fine)....

      free(buffer1); buffer1 = NULL;     // We are done. Note that the pointers
      free(buffer2); buffer2 = NULL;     // are set to NULL

   }    //-- End of "protected" code
   catch(...)
   {  //-- Failure Handler starts here
      // Something wrong happened either during the calls to malloc or during
      // the code in the middle or in sub procedures. Perform clean-up.
      if (buffer1) free(buffer1);
      if (buffer2) free(buffer2);
      throw;  // IMPORTANT ! This will pass the failure over
   }  //-- End of Failure Handler
}
```

©2001-2002 Eovia. All rights reserved.

Some comments on this code. As you can see, we do no test if *buffer1* is NULL after the malloc, because we know that if the allocation failed, we shall jump **directly** to the Failure Handler and not go any further. Likewise, if any subprocedure fails for one reason or another, it will jump directly to your Failure Handler, bypassing all the stack levels.

Overall, the code style is much more compact and readable than a classic error handling code, which would look like this:

```
long ClassicFailureSample()  // We need to return an error code :-(
{
   char*    buffer1 = NULL;
   char*    buffer2 = NULL;
   long error;

   buffer1 = (char*) malloc(128000);
   if (buffer1 == NULL) return -1;

   buffer2 = (char*) malloc(256000);
   if (buffer2 == NULL)
   {
      free(buffer1);
      return -1;
   }


   error = SubProc(...);
   if (error)
   {
      free(buffer1);
      free(buffer2);
      return error;
   }
   ...

   //-- Done at last
   free(buffer1);
   free(buffer2);

   return 0;
}
```

Typically you end up writing more error handling code than useful code, and your code becomes unreadable. Or you have to use the infamous `goto` to group your error handling at the end of the procedure. Moreover, you constantly have to pass over the error codes returned by sub procedures and test them. Most of the time, you end up not writing any error handling code at all...

Let go back to sample 1, and highlight a few things.

10.4.Initializing your local variables properly

It is very important that all local variables that are used in your Failure Handler are always in a “known state”, especially pointers. Remember that the processor can “jump” in your Failure Handler at any time until you are done in your protected code block.

- Everytime you declare a pointer, set it to NULL immediately.
- Everytime you free an object, set its pointer to NULL immediately after.

©2001-2002 Eovia. All rights reserved.

This way your Failure Handler can always safely test your local variables pointers and know if it has to do anything of not.

Local variables that are not used in the Failure Handler do not require anything special.

10.5.C++ Objects allocated in the stack

In Ray Dream Studio, C++ objects allocated in the stack were not destroyed during a failure. You had to allocate them through `new`, and implement a Failure Handler. C++ exception handling solves this problem, so Carrara developers do not have to worry about it anymore.

10.6.The Failure Handling Chain

Everytime you create a Failure Handler using `try`, it adds your Failure Handler at the top a global chain. When the `try` block is completed, it removes the Failure Handler from the chain.

This way, if each Failure Handler calls `throw` properly, each clean-up will be performed one after the other until the top level, where the Shell posts the cursed Program Error dialog.

10.7.Triggering a Failure yourself by using throw, ThrowIf-Nil(), ThrowIfNoMem(), and ThrowIfError()

It may be useful in some situations for your code to launch a Failure to abort in despair. To do this, you use `throw, ThrowIfNil(), ThrowIfNoMem()`, or `ThrowIfError()`:

Throwing an error will force the processor to jump to the first Failure Handler in the Failure Handling Chain. Any code located after `throw` will **never** be executed.

In addition, you can use ThrowIfNil(), ThrowIfNoMem(), and ThrowIfError() and other conditional throw routines.

Use ThrowIfNil to throw if the pointer passed as a parameter is NULL. A NIL pointer error message will appear.

Use ThrowIfNoMem to throw a NULL pointer as the result of a memory allocation failure. A "Not Enough Memory" error message will appear.

Use ThrowIfError() with an error code parameter, to throw if this error code is not zero. An appriopriate message (depending on the error code value) will appear.

©2001-2002 Eovia. All rights reserved.

10.8.Error codes

Table 6: Standard error codes of Carrara

Value	Meaning	Message
0	No error or silent error	*None*
-1	Generic Program Error	ÒProgram ErrorÓ
-2	Memory Manager error. Heap has been corrupted	ÒMemory suballocation error(-2)Ó
-3	Bus error	"Program Error(-3)"
-4	Bus error or OS exception	"Program Error(-4)"
-5	A NULL pointer has been found	"Null pointer error(-5)"
-6		"Program Error"
-7	Problem when reading file	"Wrong file format"
-8	Going off a list	"List out of bounds"
-9	Memory Manager error. Heap has been corrupted	"Memory suballocation error(-9)"
-34	Disk is full	ÒDisk fullÓ
-36	File Open error	"File already in use or left open"
-39	End of file error	"End of file"
-42	Too many files open	"Too many files open"
-43	File not found	"File not found"
-45	File is locked	"File locked"
-46	Volume is locked	"Disk locked"
-49	File is already open with write permission	"File already in use or left open"
-50	Parameter error	
-53	Volume is offline	"Disk not available"
-102	No object of that type in scrapbook	"the specified format is not in the edition"
-108	Out of memory	"Not enough memory"
-128	User has cancelled	*None*
-490	User Break	
-30001	Unsupported file format	"Problem opening this file. Unsupported file format"
-30002	Generic file i/o problem	"Problem reading this file"
-30003	Memory startup problem	"Not enough Memory to launch"
-30004	Windows Graphics space error	"Not enough Graphic memory space (GDI) to launch"

©2001-2002 Eovia. All rights reserved.

Table 6: Standard error codes of Carrara

Value	Meaning	Message
-30005	Windows User space error	"Not enough User memory space (windows, menus...) to launch"
-30006	File not found when reverting	"Disk copy deleted"
-30008	Error in Save Again	"Document currently open"
-30009	Wrong File Type	"Not the right kind of document"
-30010	Unimplemented feature	"Feature not yet implemented"
-30011	The C++ class could not be found	"\"^3\": Missing Sofware component";
-30012	The Class signature could not be found	"Missing Sofware component"
-30013	Drive mapping error	"Could not map network to local drive. All drives A: through Z: are in use"
-30014	External texture error	"External Texture Map or Movie not found"
-30015	Corrupted data error	"File had corrupted data"
-30016	DirectDraw init error	ÒDirect Draw driver needs to be installed"
-30017	MacOS system memory error	"Not enough memory left for the system (Finder)"
-30018	Help memory error	"Not enough memory left to open the Help file"

10.9.Message parameter coding

You can use the *message* filed of the TMCException class to have more control on the alert that will be posted.

Let *hiMessage* be the high 16 bits and *loMessage* the lower 16 bits of *message*.

Table 7: Message values

hiMessage	Meaning
0	*loMessage* is an Action Number (like in CMNU or TBAR resources). The Shell will try to find the corresponding name and display it in the alert. Example : Program Error in ÒSave As...Ó
0xFFFF	*loMessage* is the ID of a ALRT/DITL resource. If a DITL item contains a ^0 substituion tag, it will be replaced by the name of the error as found in the Errors table.
0xFFFE	*loMessage* is an error code used by Carrara to parse his errors string table.
other values	Carrara will post an alert containing the string located at the *loMessage* index in STR# (*hiMessage).*

10.10.Failing Silently

Sometimes, you will want to fail silently, i.e. abort without posting the Program Error dialog. To do this, pass 0 as the error code and message:

©2001-2002 Eovia. All rights reserved.

```
throw TMCException(0, 0);
```

All failure handlers will be processed, but no dialog will appear.

10.11.Recovering from a Failure

In some situation, your code can decide that it treated the Failure to its full extent and that normal execution of the code should resume. For example, you may have tried to allocate a very large buffer in RAM and it failed, but your code has an alternate solution (using a smaller buffer for example).

To do this, simply do **not** call `throw` at the end of your Failure Handler. The processor will exit the `catch` block normally and keep going. It is perfectly safe and legal to do so.

Some (lazy) programmers use this in combination with `throw TMCException(.,.)` to exit quickly from a deep part of their code and go back at the top level, using the “message” parameter of the `TMCException` to pass over some information of what happened.

©2001-2002 Eovia. All rights reserved.

Chapter 11 - SMP

This chapter introduces SMP (Symmetrical Multi-Processor) with Carrara.

Carrara supports multi-processor systems. Most of the time, implementing an extension can be done without any knowledge of that capability, since the Shell will handle most of the charge for SMP. But it is useful to understand what underlays Carrara, and what rules are expected to be followed.

Using SMP means that your extension might be simultaneously running on many (two or more) processors. Therefore, the Shell is going to use many clones of your extension, that is one clone for each processor. So if your extension is using global variables, static members, or those types of features, the clones will share and modify that information at the same time. That means they will influence each other. In other words, each processor will introduce bugs on the other processor's computations.

Do not use static members or global variables if you are using SMP.

Now, if you are familiar with SMP and want to write an extension that uses the full power of the platform in a specific way, Carrara's SDK provides interfaces to do so. If you are interested, look at the files /Include/MCCommon/IShSMP.h

©2001-2002 Eovia. All rights reserved.