

WINDOWS MEDIA FOUNDATION: GETTING STARTED IN C#

A Step-by-Step Guide to Writing Windows Media
Foundation Applications in C#

Nic Cyn

This book is available as a free PDF download or in print on Amazon at
nominal cost.

Windows Media Foundation

Getting Started in C#

*If you have a problem which requires
Windows Media Foundation to solve it,
well then, you really have two problems.*

There is many a true word spoken in jest and, joking aside, the same is true for any of the technologies you might use to solve that problem. DirectShow and Video Foundation for Windows, the Windows predecessors, are not really any easier and neither are the similar tools on the Linux or Apple platforms. All of them present a pretty steep learning curve for the novice. This book is an attempt to simplify the process; it will document the things you need to know and give you the skills which will enable you to write Windows Media Foundation (WMF) applications in the C# language.

For K, who puts up with a lot...

Copyright © 2019, OfItselfSo.com
All Rights Reserved

The source code accompanying this book is open source and released under the MIT License. This book is copyrighted and may not be distributed, or copied (in whole or in part) without permission of the author.

A print version of this book it is available on Amazon at nominal cost.

Version 1.1 March 2019

TABLE OF CONTENTS

Introduction	1
How the Problem Will Be Simplified	2
Why Was this Book Written.....	4
The Tanta Project	4
Overview of WMF.....	6
WMF And Windows Versions	7
MF.Net	8
Installing MF.Net.....	10
Windows Media Foundation Architecture	11
The Pipeline Architecture	15
The Pipeline Internals.....	19
The Reader-Writer Architecture	23
The Reader-Writer Synchronous Model	26
The Reader-Writer Asynchronous Model	26
The Hybrid Pipeline-Writer Architecture.....	27
Other Non-Pipeline WMF Components	28
The Transcode API.....	29
The Output Protection Manager	29
The IMFCaptureEngine and IMFSensorDevice	29
Which Architecture to Use?.....	29
MF.Net Programming Fundamentals	31
You must use an [MTAThread] Code Decoration	31
Enabling the Multi-Threaded Apartment Model	32
Coping with [MTAThread] Problems.....	33
Initializing Windows Media Foundation.....	34
Shutting Down Windows Media Foundation	34
Most WMF Objects are Interfaces	35

IUnknown	36
Getting an Interface from an Interface.....	37
The Difference between MFGetService and QueryInterface	38
WMF Object Creation is Indirect	39
Releasing COM Objects	40
Releasing Class Variables	42
GUIDs	44
About Attributes	46
PropVariant	47
Creating and Populating Attributes	49
Enumerating Attributes	52
Attribute Code Conversion from C++	53
HResults.....	54
The C++ vs C# Code Structure	55
The WMF Components	59
Fundamental Processing Objects	60
Media Sources	60
Media Sinks	61
An Interface Digression	61
Creating WMF Components	63
Creating a Media Source from a Device	64
Creating a Media Source from a Source Resolver	65
Creating a Media Sink from an Activator	66
Creating a Media Sink On a File.....	67
The Pipeline	68
A Simple Pipeline	69
A Pipeline with Two Branches	70
Reader-Writer Data Processing	73
The Hybrid Architecture Data Processing	74

Pipeline Errors and Events.....	75
Creating the Media Session	76
The Media Session Callback Object	77
Independent Pipeline Objects	79
Media Streams and the Presentation	79
Why Does a Media Source contain Multiple Streams	81
Obtaining Presentation and Stream Descriptors	84
Media Types and Sub-Types	87
Getting Media Types from the Stream Descriptor	88
Creating Your Own Media Type	89
Cloning a Media Type	93
Enumerating the Attributes of a Media Type.....	93
FOURCC Codes	94
Topologies.....	95
Topologies and Media Sub-Types	100
Partial Topologies.....	102
Creating a Topology Node for a Media Source.....	104
Creating a Topology Node for a Media Sink Using a Stream Sink	106
Creating a Topology Node for a Media Sink Using an Activator	107
Getting The WMF Object From a Topology Node	109
Transforms	110
Adding Transforms To a Topology	111
Registering Transforms.....	112
Finding Transforms.....	113
Synchronous and Asynchronous Transforms	113
Samples, Frames and Buffers.....	114
The Raw Media Data	115
Using Marshal to Access Raw Media Data	116
Accessing Raw Media Data with Unsafe Code	117

Manipulating Raw Media Data with Externs	118
The Media Buffer	119
Creating a Media Buffer	121
The Media Sample	122
Creating a New Media Sample	123
Callback Objects.....	127
The Media Session Callback Object	130
Source Reader and Sink Writer	135
Using The Source Reader	137
Obtaining Media Samples from a Source Reader	138
Synchronous vs Asynchronous Source Readers	138
Creating a Source Reader	139
Creating a Source Reader on a File	140
Creating a Source Reader on a Device	142
The Source Reader and Format Conversions	143
Sink Writer	144
Providing Media Samples to a Sink Writer	145
Creating a Sink Writer	146
Creating a Sink Writer On A File	147
The Sink Writer and Format Conversions	149
WMF – First Contact.....	150
The TantaVideoFormats Sample Application	151
The Video Picker Control.....	152
Enumerating the Video Capture Devices	154
Enumerating the Attributes of a Video Device	159
Practical WMF Architectures	167
Implementing the Pipeline Architecture.....	168
The Standardization of Pipeline Components.....	170
The Sample Pipeline Architecture Source.....	170

Starting up the Pipeline	178
Shutting Down the Pipeline	180
Errors in the Pipeline	182
Dealing with Multiple Streams	183
Creating an MP4 File Sink	184
Implementing the Reader-Writer Architecture	188
The Synchronous Reader-Writer Architecture	189
Dealing with Multiple Streams	198
The Asynchronous Reader-Writer Architecture	201
Implementing a Hybrid Architecture	205
Rendering Audio and Video	211
An Overview of the SAR.....	212
Audio Volume and Muting	214
An Overview of the EVR.....	217
Older Versions of the EVR	217
The EVR and Direct3D	217
Multiple Streams in the EVR	218
A Summary of the Capabilities of EVR	218
The TantaFilePlayback Samples	219
The ctlTantaEVRStreamDisplay Control	220
The ctlTantaEVRFilePlayer Control.....	224
The Video Window Drawing Surface.....	226
The Video Window Appearance	228
About Aspect Ratios	228
The EVR and Aspect Ratios	230
The EVR Video Window Size and Position.....	230
Software Magnification	232
Handling Size Change Events	232
Playback Control	233

Starting, Pausing and Stopping the EVR	233
Getting the Duration and the Current Progress	235
Seeking Forward and Back in the Stream	238
Fast Forwarding and Rewinding the EVR	241
Taking a Snapshot of the Video Display	247
Working With Transforms	253
The Tanta Transform Sample Projects	256
Basic Transform Operation	258
The Tanta Transform Base Classes	259
Transforms and Multiple Streams	261
Transform Streams and Media Types	263
Processing in the Transform	265
Events and Messages	267
Raw Data Handling in the Transform	267
Writing Text on a Video Frame	271
Adding Transforms To the Pipeline	274
Adding a Transform When You have the Source Code	275
Adding a Transform When You have An Activator	276
Adding a Transform When You have a Known GUID	277
Adding a Transform By Creating it from a GUID	277
Connecting Transform Nodes	278
Making a Transform Available	279
COM Interop Decorations	280
Registering A Transform Manually	282
Registering A Transform During Compilation	283
Registering A Transform via an Installer Application	284
Making a Transform Discoverable	284
Local Discoverability	286
Enumerating the Transforms on the System	288

The TantaTransformPicker Sample Application	288
Registry Based Transform Information	289
Enumerating the Transforms.....	292
Transform Based Information.....	296
Passing Information In and Out of a Transform	298
Information Exchange Via Direct Calls	299
Information Exchange Via Attributes	300
Information Exchange Via Reflection and Late Binding	305
Information Exchange Via COM and Marshal	310
Capturing Camera Data.....	311
Timestamp Rebasing	312
Capture with a Reader-Writer Architecture	313
Setting the Output Media Type on the Source Reader.....	313
Configuring an MP4 Sink Writer	318
Capture with a Hybrid Architecture	322
Creating a copy of a Media Sample	326
The Tanta Sample Code	328
Downloading the Tanta Sample Projects	329
Tanta Sample Applications	329
Converting Between C++ and C# Code Examples	332
Code Conversions in General Function Calls	333
Getting a Bool	333
Setting a Bool	333
Getting an Enum	334
Setting an Enum	334
Getting a GUID	335
Setting a GUID	335
Getting an Int	335
Setting an Int.....	336

Getting an IntPtr	336
Getting a Long:	337
Setting an MFlnt (DWORD Ptr)	337
Getting a String	338
Setting a String	338
Getting a Struct	339
Setting a Struct	339
Getting a Typed Object	340
Setting a Typed Object	340
Misc. Code Conversions	341
Converting a Byte[] to a Struct	341
Converting a Struct to a Byte[]	341
Copying the Data from an IntPtr	341
Getting the Size of a Struct	342
Converting an Array of Structs	342

Windows Media Foundation: Getting Started in C#

Chapter 1

INTRODUCTION

Working with Windows Media Foundation isn't really that difficult. It does seem complex at first but once you get the hang of it you'll find that you are mostly just connecting things together in various predictable ways. Ultimately WMF is just a collection of tools that do various specific things and once you get a sense of how everything works you can just bolt the components together in standard ways to achieve a result - much like you would build something with a set of Lego™ blocks.

The techniques illustrated in this book, along with the associated sample code, will provide you with the concepts and skills you need in order to write your own Windows Media Foundation applications.

As you acquire more experience with WMF, it soon becomes clear that much of the perceived complexity is only due to the fact that Windows Media Foundation takes a rather circuitous route to configuring the objects it uses. Thus you might see ten or fifteen lines of code (including error checking) used to configure an object where you would expect to see two lines in a normal C# program. Once you realize what is happening you can mentally resolve a large group of statements as a single action.

Remember the joke at the start of this book about how, if you have a problem that needs WMF, you really have two problems? Actually, since Windows Media Foundation

makes extensive use of Component Object Model (COM) technologies, you could more accurately say you really do have three problems. If you are a super expert in COM you will probably have a much easier time of it in WMF. In reality, many of the odd ways Windows Media Foundation does things are really just the odd COM ways of doing things. If you are a novice don't worry, we will not discuss COM any more than is absolutely necessary in this book. In truth, an understanding of COM is not that essential as long as every time you think something like "*why would they do that*" you are willing to just blame it on COM and move on. As an example, when we discuss things like Attributes and Activators later on, you will see what is meant by this – they are pure COM. The implementers of WMF just used these tools because they were there and they did not want to re-invent the wheel.

The above discussion of COM leads us into an interesting point. Another thing that can cause confusion in WMF is that there is often more than one way to achieve a particular goal. For example, it is possible to instantiate a WMF object by getting some other COM object (an Activator) to do it for you. Or you can make a call to a static function and have the COM layer build and return it. In other cases you simply specify a GUID identifier and the object will be dug out of the registry and created for you. Finally, sometimes, you can also just hand crank it and create the object manually. It should be noted that this last option (using the C# `new` operator) is only rarely available for objects of any significance. Not all methods are available for all WMF objects and if there are multiple methods available, some examples you will see on the Internet will do it one way and some will do it in another way. Typically, there is no one "*correct way*" – although some methods are easier than others in certain circumstances. After you understand the reasons why things are happening the way they are, and realize that there are probably other options available, then things will simplify considerably.

Once you get a few of the basics sorted out, and are a bit further along the learning curve, you will probably find that Windows Media Foundation is not so difficult as you first imagined and is a very powerful tool indeed.

HOW THE PROBLEM WILL BE SIMPLIFIED

Getting up-to-speed with any new technology is usually a non-trivial undertaking. There is a lot to take in and there are numerous concepts which must be learned first in order to make other ideas make sense. This, it would seem, is the major difficulty when learning from help files alone – they do not present the information as a logical progression in which each subsequent discussion uses the previously presented

material. Of course, you cannot really fault help files for that – they are intended to provide point solutions to specific problems - not tutorials.

This book is going to present, as far as is possible, the building blocks of the technology first. Once those are known, you will have enough background knowledge to understand the more complex techniques. Usually the Windows Media Foundation topics will be presented in multiple passes. If a concept is presented in too much detail at once, a reader unfamiliar with the topic simply will not possess enough reference points to make sense of the information. In other words, if things are too detailed, you will read the text but it will not “*stick*”. A very light overview on a first pass and then a subsequent section with more details provides a better learning experience. Of course, this method has the downside that a lot of information is repeated. Try to consider this repetition to be a “*feature*” rather than a “*bug*”.

We are also going to ignore large chunks of technology that, in theory, you should really know about but which, in reality, will divert mental resources away from the fundamentals of Windows Media Foundation. Given the earlier discussion in this chapter you probably think the previous sentence refers to COM – and you are right, it does. In practice, you don’t really need to know all that much about COM until you are ready to know about it. However, there are lots of other things that will be ignored too. For example, you also don’t really need to know the details of the NV12 or YUV video formats. When the time comes for you to dig about inside the guts of a YUV video frame you can just hit the Internet to find copious resources that explain it in detail – discussing it at this point will just get in the way. You may also be wondering how a C# application interacts with a C++ based system such as WMF? Well, the answer is yet another rather clever bit of technology called Interop Services. We are going to blithely ignore COM Interop in general and Marshaling in particular and just wave the whole lot away. It just works and going into the details of the magic will only serve to distract you.

There is a suite of Windows Media Foundation example projects written in C# which illustrate the various techniques available with this book. Not all of that code is discussed in this book as some projects are variations upon the same theme – but the intention is that they all illustrate some Windows Media Foundation concept. *The Tanta Sample Code* appendix at the end of this book has a detailed description of each sample application, a discussion of why it was written and the WMF techniques that C# Solution demonstrates. In addition, some of the more generally useful sections of code have been factored out into a common library. This library may be included in your own projects in order to save you re-writing code.

A significant amount of effort is going to be devoted to describing how to bolt the various components together. This, after all is the sort of practical information that gets applications written. There will be lots of source code examples and the location of every item of source code will be annotated with the relevant filename so that you can review the code section within the context of a C# class and project. In addition, you will be able to run the complete application and step through the code with a debugger (debugging works very well in the .NET version of WMF).

Of course, the Tanta Sample code should not be the only source code examples you should look at. When you begin writing your own code, you will probably want to hit your favorite search engine and do a bit of digging to get more information on a particular technique. When you do this, you will quickly become aware that the majority of the sample code you find will be written in C++ not C#. For the most part, translating the C++ code into its C# equivalent is pretty straightforward once you know how to do it. The *Converting Between C++ and C# Code Examples* appendix provides a detailed discussion of the more commonly required patterns.

WHY WAS THIS BOOK WRITTEN

When I was looking (Spring 2018), there did not seem to be anything intended for the beginner which brought all of the information together and presented specific novice friendly advice on how to get started in Windows Media Foundation programming. This book has been written in an attempt to address that issue¹. In other words, this book contains the information I would like to have had when I was first trying to figure out how to make anything at all work in WMF using C#.

THE TANTA PROJECT

Originally there was no intention of writing a book. The Tanta² project originated in the mid 2000's as a vehicle to use DirectShow to try out various digital signal processing techniques. I had just obtained the book *Digital Signal Processing 3rd Edition* by

¹ More accurately, it was my wife who made me write this book. She grew weary of hearing me rattle on about the lack of information on Windows Media Foundation in C# and basically said (as wives do) “*Well, write one yourself then*”. So I did.

² All my projects have silly code names, in fact I am uncomfortable working on one until it has a name. The name of this WMF demonstrator project is “*Tanta*” and there is no particular significance to the name.

Gonzalez and Woods and wanted to have a go at some of the methods they describe. What is more, I wanted to do it in C# and also have the code available to C# in a standard debugger. For that you need to get access to the stream of video data (either coming off a file or from a web camera) and somehow get it up into the user layer. I kind of got this working with DirectShow but was never terribly satisfied with it. So I put the project aside and never wrote it up or made it available on website (<http://www.OfItselfSo.com>).

Well, the years rolled on and I was starting in on another project (named *FPath*) one aspect of which requires a computer to make decisions based on video data. I looked at the old Tanta project to see what might be usable within it. As part of the background research, I dug around and discovered Microsoft had really moved DirectShow forward with its successor technology Windows Media Foundation. It seemed possible that a WMF component called a Transform might be much more useful than what I had before (spoiler alert: yes it is – Transforms are a very good fit for the requirements).

So, the old DirectShow Tanta project was scrapped and the name re-used for a new Windows Media Foundation based project with the same goal: *Make a demonstration project that provides a C# application with access to raw video data at the user level.* From the *FPath* project perspective, the *TantaTransformDirect* and *TantaCaptureToScreenAndFile* sample applications are the only two which are of interest. However, as part of the learning process, I wanted to thoroughly understand the usage and capabilities of the Windows Media Foundation technology and thus wrote the other sample applications (and this book).

Windows Media Foundation: Getting Started in C#

Chapter 2

OVERVIEW OF WMF

There is no point providing a lengthy discussion of the history of Windows Media Foundation. You can read Wikipedia as well as anybody and so there is little value in cut and pasting it in here.

Here is what you, as a C# programmer, really need to know: With WMF you get a really sweet set of tools that let you source, sink, manipulate or render (display) media data.

The tools involved are adaptable and extensible and you can write your own versions to slot in place of any of the Microsoft provided components if they don't do what you need (this is something of an advanced topic though). In addition, you can get access to the raw feed of media data as it passes through the system and this data will be up in the user layer (where normal C# programs run) so that a debugger can step you through the code. It should be noted that this debugging process is only operational on the code you write – you cannot debug down into the Windows Media Foundation functions. Those are written in C++ and the source code is not available.

Ultimately Windows Media Foundation provides you with a set of modular components which you can (if you know how) bolt together to meet just about any media processing requirement you may have. The tools and techniques in WMF are logically arranged and fairly consistent in the behavior – although, if you are just starting in on the learning process, you will probably not agree with that statement. WMF also contains a lot of technology devoted to playing Protected Media Path content (PMP) which will not be discussed here because it is something of a niche topic.

WMF AND WINDOWS VERSIONS

Windows Media Foundation was introduced with the Vista operating system and is intended to be a replacement for DirectShow. Realistically though, there is so much legacy DirectShow software in existence that both technologies will be available alongside one another for some time to come.

WMF is not, and never will be, available on Windows XP or earlier windows versions. Having said that, a lot of the Windows Media Foundation technology is kind of rocky on Vista and there are numerous reports of some functionality being missing without various patches and workarounds. As a standard, Windows 7 is a far better “*minimum baseline*” platform to work from – although you would be well advised to be on the latest service pack and fully patched. Most of the Tanta Sample Projects work on Windows 7 – however, Windows 10 was the development platform. It should be noted that the Tanta Sample code has never been tested on Windows 8 and its capabilities there, while assumed to be operational, are unproven.

While Windows Media Foundation is fully operational on Windows 7 there are issues with the codecs available by default. We have not discussed this issue in detail yet but some WMF components (such as the Sink Writer) will automatically attempt to load conversion DLLs in order to operate. The DLLs they load are dependent on the particular task being performed. This means that on Windows 7 some codec DLLs have to specifically be made available – they are present, just not “discoverable” by WMF. Most of the Tanta Sample Projects do this but it was not possible to get some ready in time for the publication of this book. In such cases, a warning will be issued.

Due to the non-default availability of some codecs on Windows 7, some WMF components will throw an error unless special configuration steps are taken. Windows 10

does not exhibit these problems and is the recommended development platform.

The version of the .NET Runtime you choose to use is pretty much up to you. The Tanta Sample Projects use .NET v4.6.1 and this appears to work well. In order to use this version of .NET you will need to use Visual Studio 2017 or 2015, as earlier releases do not support that .NET version. The free Visual Studio 2017 Community Edition was used in the development of all of the sample software for this book.

MF.NET

Ok, so you want to write a Windows Media Foundation application but long ago tired of writing things in C++. You are now a member of that league of awesomeness known as the C# programmer. So how can you use WMF which, fundamentally, is C++ software? Well, C# is equipped with a technology called Interop Services which enables C# programs to interact with compiled C++ binaries. This is done through a procedure called marshaling which moves data from the PC heap into the Managed Memory space of the C# environment and back again. Interop also handles function calls across this border and will “marshal” the parameters to that function. *“But”, you say, “Windows Media Foundation is COM based – how is that handled”?* Well there is also a thing called COM Interop which works the same and as long as you know *“how”* to use it you don’t need to concern yourself with the details of what it is doing.

In order to make a call from C# to a WMF function, you would have to provide a lot of definitions to COM Interop in order to tell C# how to set up marshaling. You would need one definition per WMF function and others for all of its polymorphic variants, plus you would need C# defines for all of the constants and enums and the inevitable multitude of other things. Wouldn’t it be nice if somebody did all that for you and rolled it up in a library you could just use?

Yes, indeed, such a thing would be very useful and, in fact, all of that has been done (and done well) in a library called MF.Net.

The MF.Net library can be found at the following address:

<https://sourceforge.net/projects/mfnet/>

The MF.Net library was written in C# by two anonymous programmers known by the handles of *nowinskie* and *snarfle* and it is fully open source. You can download the library DLL binary or, if you wish, download the complete code for the MF.Net library

and compile it up yourself. You can also, of course, look at the contents of it and if you do you will see things like the following.

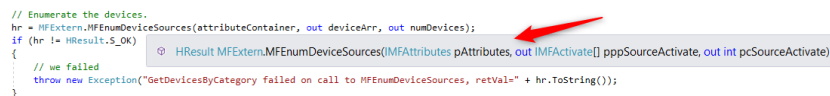
```
[DllImport("mf.dll", ExactSpelling = true), SuppressUnmanagedCodeSecurity]
public static extern HRESULT MFEnumDeviceSources(
    IMFAttributes pAttributes,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 2)] out IMFActivate[] pppSourceActivate,
    out int pcSourceActivate
);
Source: MediaFoundation-2010::Externs.cs
```

This is the code that defines how the `MFEnumDeviceSources()` static function is marshalled from C# to C++ and back. It looks complicated (and it kind of is) but you do not really have to deal with any of that. If you wish to make a `MFEnumDeviceSources()` function call to enumerate the audio and video devices on the system, all you need to write is something like the code below...

```
// Enumerate the devices.
IMFAttributes attributeContainer = null;
IMFActivate[] deviceArr;
int numDevices = 0;

hr = MFExtern.MFEnumDeviceSources(attributeContainer, out deviceArr, out numDevices);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("failed on call to MFEnumDeviceSources, retVal=" + hr.ToString());
};
Source: TantaCommon::TantaWMFUtils::GetDevicesByCategory
```

Ultimately, other than the definition of the variables for the parameters and the error checking, the call is a one-liner. Visual Studio will even tell you required parameter and



```
// Enumerate the devices.
hr = MFExtern.MFEnumDeviceSources(attributeContainer, out deviceArr, out numDevices);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("GetDevicesByCategory failed on call to MFEnumDeviceSources, retVal=" + hr.ToString());
}
```

return value definitions in the standard mouse-over tool tip as is shown in Figure 2.1.

Figure 2.1: Visual Studio Tool Tips Work on MF.Net Functions

The point is that you do not need to know about any of this marshaling stuff – all you need to know is that if you give it the proper parameters in the proper order then the function will perform the required operation and return the desired information. We will meet the `MFEnumDeviceSources()` call in more detail later on in the *WMF – First Contact* chapter when we use Windows Media Foundation to enumerate the video devices are present on the system. You can also step through this code by running the debugger on the *TantaVideoFormats* sample application.

The MF.Net library is truly a thing of beauty. It is stable, fast and appears to be quite bug free. Only one minor issue was found during the writing of this book and one of the maintainers (*snarfle*) responded quite quickly to the report.

INSTALLING MF.NET

Installing MF.Net is simple. Just download the library from <https://sourceforge.net/projects/mfnet/> and open it up (it is a zip file) and copy the `MediaFoundation.dll` file to wherever you wish. Once it is in place just reference the DLL in your application like you would any other C# library.

The Tanta Sample Projects (see *The Tanta Sample Code* chapter) simply place the `MediaFoundation.dll` file at the root of the repository and all of the applications reference it there. You could, if you wished, install the MF.Net library in the Global Assembly Cache. This process will not be discussed here – a quick search on “*install dll in the GAC*” will turn up very detailed information.

The full source code for the MF.Net library is available – so you can inspect and recompile it as necessary. There are also a number of C# sample projects available for MF.Net and it is worthwhile looking at that code as well. This book, however, only refers to the Tanta Sample Projects.

Windows Media Foundation: Getting Started in C#

Chapter 3

WINDOWS MEDIA FOUNDATION ARCHITECTURE

Ultimately the purpose of Windows Media Foundation is to transport media information from one or more objects that originate the data (called Media Sources) to one or more objects that consume the data (called Media Sinks). WMF provides a set of tools to manage and manipulate this flow of information and the way you structure these tools is the architecture of the software you are writing.

Allow me to let you in on the big “*secret*” regarding the architecture of WMF. Actually it’s not really so much of a secret as just one of those things that is obvious to everybody experienced in the technology but which nobody ever bothers to write down in order to explain it to new users.

In practice, there are two distinct Windows Media Foundation architectures. It is also possible to create hybrids between these two architectures and many of the examples (but not all) you may find on the Internet will be these hybrids.

This is an important point. You need to interpret every sample code, help file or blog post you read in that context otherwise you will become very confused. Let us take a look at what is going on.

Strictly speaking, there is only one Windows Media Foundation architecture. This is the Media Source, Media Session, Pipeline, Topology, Media Sink mechanism you may have read about elsewhere but which has not been discussed yet in this book. This is true, most things in WMF uses that structure and if you look online trying to get a sense of what is going on, this is pretty much the only discussion you will see. That is until you start looking at some of the sample code and find that a lot of it contains none of those things. No Media Source, no Pipeline or Media Session – just two components called a Source Reader and a Sink Writer.

In this book we will refer to the structure that uses the Media Session and Pipeline as the **Pipeline** Architecture and the structure that uses the Source Reader and Sink Writer components as the **Reader-Writer** Architecture.

The reality is that Media Sessions, Pipelines, Topologies et al are relatively complex things to set up and so in the interests of making things simpler and providing an easy win for people who *“just want to get stuff done”* Microsoft encapsulated the standard architecture into two objects. These objects are the Source Reader and the Sink Writer. The job of the Source Reader is to read data from a source (perhaps a file on disk or a webcam) and job of the Sink Writer is to write it out to a file. So, in theory, if you want to read data from a source and write it out to a file all you need to do is bolt these two tools together and the job is done. In such a situation the media data is handed from the Source Reader to the Sink Writer and you control the stream of data as it is processed. In other words your code sits in a loop and takes the packet of media data (called a Media Sample) from the Source Reader and gives it to the Sink Writer until some sort of stop signal is triggered. Of course you can modify or manipulate the Media Sample on the way through if you need to do so.

The Source Reader and the Sink Writer are very powerful and relatively simple to implement technologies and neither of these tools exposes a Media Session or a Pipeline or any of the standard WMF architecture components in a way that is usable by the implementer of the software. Actually they do use many of these things – but they are all internal and you, as the coder, never interact with them at all.

Powerful and simple they are but there is a trade-off (isn't there always). The Source Reader and the Sink Writer are not universally applicable to all situations. For example,

there is no way to use a Sink Writer to display media content (unless maybe you wrote a custom one yourself). The Sink Writer only writes to files. This means that in order to render media information such as playing video or sound you need to use the full Media Session and Pipeline Architecture. Similarly if you just want a simple and relatively easy way to save media data to a file (perhaps as an MP3 or MP4 file) then the Sink Writer is usually the preferred solution since its configuration is somewhat easier due to the encapsulation.

The Source Reader is not a Media Source and the Sink Writer is not a Media Sink. At this point, you probably do not really understand the significance of this – but for now just realize it means that there is no way to **directly** involve a Source Reader or Sink Writer in a Media Session and Pipeline.

The above statement is not meant to imply you cannot have an application with a Media Session and Pipeline and still use a Sink Writer. You can and this is commonly done – everybody wants to have their cake and eat it too. However, the Sink Writer can never be part of the Pipeline because the Media Session cannot interact with it. Rather a special Pipeline component called the Sample Grabber Sink copies the data it receives from the Pipeline and hands it off to the Sink Writer. In effect the Sample Grabber Sink pretends (as far as the Sink Writer is concerned) to be a Source Reader. The Sample Grabber is a true Media Sink and the Media Session can interact with it as required. In reality, the Sample Grabber just discards the media data once it has been handed off to the Sink Writer but the Media Session doesn't know or care about that.

An architecture which uses a combination of a Media Session and a Source Reader or Sink Writer will be referred to as a **Hybrid Architecture** in the following discussions.

Although it was said earlier that there are two distinct Windows Media Foundation architectures (the **Pipeline** and the **Reader-Writer**) the fact that there can be **Hybrids** between the two really means there are three ways of structuring a WMF application. Everything you read in this book or on the Internet and every bit of source code will be one of the three architectures.

It will greatly ease your understanding if the first thing you do when you come across an item of sample code or blog post is figure out what the architecture the author is

implementing. For example, if you have decided you need to implement a pure Pipeline Architecture for your application then some sample code which uses a Reader-Writer as a solution is not likely to be very useful other than as a cut-and-paste solution to a specific point problems (such as creating a copy of a Media Sample).

It should also be noted that within any specific architecture, there can be more than one possible sub-architecture. For example, a Reader-Writer application can operate in both Synchronous and Asynchronous mode. That particular topic will be covered in detail in the later on in this book, however, for now just realize that that things like this are rarely stated outright in the comments to whatever sample code you may come across.

The Tanta Sample Projects (see *The Tanta Sample Code* appendix) contain examples of all three of architectures and the architecture used is usually indicated in the name. For example, listed below are three of the code samples. All of these examples do the same thing - they copy an MP4 file with multiple streams (audio and video) from a source to a target.

TantaVideoFileCopyViaPipelineMP4Sink - uses a Pipeline Architecture to create a copy of a specified file.

TantaVideoFileCopyViaReaderWriter - uses a Synchronous Reader-Writer Architecture to create a copy of a specified file. For an example of an Asynchronous Reader-Writer see the *TantaCaptureToFileViaReaderWriter* application.

TantaVideoFileCopyViaPipelineAndWriter - implements a Hybrid Architecture to copy a specified file. This application is a mixture of the first two examples. Instead of an MP4 Sink, a Sample Grabber Sink consumes the Pipeline data and in the process feeds the data into a Sink Writer.

Other than as a demonstration of WMF technologies, these samples are not very useful – after all if you actually just wanted to copy a file from one place to another you would just drag and drop it in Windows Explorer. There are some similar examples for audio only files (MP3) which demonstrate the same copying process on a file with single media stream and this is about as simple as the process can get.

Another interesting Tanta Sample Code example (from an architectural point of view) is the *TantaCaptureToScreenAndFile* application which implements a Pipeline Architecture to display the video from a camera on the screen. This application also inserts a special object called a Transform into the Pipeline which intercepts the media data in the Pipeline and feeds it into a Sink Writer. Thus the video renderer (which is a Media Sink) still gets the data to display, but the Sink Writer also gets a copy to write to the disk in

order to provide a permanent record. This is an example of an application implementing both a full Pipeline and a Hybrid Architecture. It also serves very well to illustrate that things can get a bit muddy architecturally in WMF and that the three architectures discussed above are pretty adaptable entities.

In the following sections we will consider each of the three architectures in turn and also discuss the various aspects of each. Once those discussions are complete, the final section in this chapter (*Which Architecture to Use?*) will try to provide a bit of insight into which architectures are a good fit for a particular situation.

The following sections are quite significant and it is suggested that you read them carefully. The concepts described in them will be used extensively in future Chapters.

THE PIPELINE ARCHITECTURE

Recall that at the start of this chapter it was noted that the ultimate purpose of WMF was to transport media information from a source to a sink. Let us consider this problem in more detail and talk about the mechanics of one way in which this data transport might be performed. We'll do this by breaking things down into a simple list (some items of which are obvious) and along the way we will see how the structure of the Windows Media Foundation Pipeline Architecture provides this.

1. It is easy to see that there can be many different types of Media Source. For example a Media Source could be a video camera, microphone or an input file on a disk
2. Similarly, there can be many different types of Media Sink. For example a Media Sink could be a video display, a speaker or an output file on a disk.
3. Since the transport of media data is the point of WMF, we will note here that the data somehow has to move from the Media Sources to the Media Sinks. The data transmission path is known as the Pipeline.
4. Since the media data has to be manipulable as it moves through the Pipeline, and we wish to enable the maximum amount of user modification, logically the Pipeline will have to consist of

multiple steps. In other words, in most cases, the data will not go directly from the Media Source to the Media Sink.

5. Some of the objects which execute the processing steps in the Pipeline will be supplied by Windows Media Foundation or Microsoft as part of the platform on which the application is running. Other objects in the Pipeline will be supplied by third parties or custom written by the user.
6. We note however, that WMF is designed to provide for multiple sources and sinks. This implies that that Pipelines in general need not be just a simple, sequential, linear path. Instead, the Pipeline must be able to support multiple branching paths. For example, a file on a disk acting as a Media Source may contain both video and audio data. The video data may have to end up at a Media Sink which displays it and the audio data might need to be moved to a separate Media Sink which renders it as sound. The name for the structure of the branching paths in a Pipeline is called the Topology.
7. The objects which will form the steps in a Pipeline are represented as nodes in a Topology.
8. It is sometimes useful to think of a Topology as a design, (a blueprint if you wish) and a Pipeline as the built version of that design. More accurately though, the Topology is built out of various nodes which will end up in the Pipeline as instantiated objects when the Topology is converted into a Pipeline.
9. The Topology can be fully specified or partial. In certain circumstances (playback from a file for example) the Topology can be sparsely specified and something called the Topology Loader will automatically fill out the branches of a Topology with the required nodes based on Transform objects listed in a designated part of the registry. These Transform nodes may well have been placed there by third party applications.
10. It is also possible (and common) for an application to explicitly specify the Transforms it wishes to have in a Topology branch.
11. We also note that a Topology (and hence Pipeline) will never have to support loops. Data in the Pipeline will never return to any component that has previously processed it. This helps simplify things considerably.

12. The Topology may well have multiple branches which join together on a single object (usually only sinks). For example, separate Topology branches from a camera and a microphone might both terminate on the same Media Sink so that both the sound and video can both be written to the same file.
13. Taking a bit of a digression, we recognize that there are lots of different basic categories of media such as video, audio, still photo etc. These types are called the Media Major Types.
14. We also recognize that for any Media Major Type there can be any number of differing formats. This is called the Media Sub-Type. For example, the NV12 and YUY2 formats are not the same thing although they are both subtypes of the video Media Major Type. Furthermore, even within the NV12 format, video with a size 640x480 must be treated somewhat differently than video data sized as 1280x720.
15. Returning back to higher level concepts, we note that Media Sources might contain multiple identical Media Major Types. For example a DVD file might have video for the standard edition and video for the director's cut. Both are video – but only one can play at the same time. Similarly, there may be audio content in various languages and yet another Media Major Type for all the many and varied subtitles on offer. Each of these “*versions*” of a Media Major Type in one Media Source is known as a Media Stream.
16. Each Media Stream will have one or more Media Sub-Types and the Media Sub-Types need not be unique between streams. For example, both the theatrical and directors cuts of a video may both be offered in the same formats and only a human readable label known as a “Friendly Name” will differentiate them.
17. Usually only one Media Stream of each Media Major Type is used in a Topology. For example, it makes no sense to play audio in two languages simultaneously – although it might be desirable to play audio in one language and subtitles in another.
18. The data in a Media Stream is going to move through the Pipeline in small, easily manipulable chunks instead of all at once. This is why it is called a “Stream”.
19. A Pipeline consists of multiple components each of which process the stream of data in sequence. Theoretically, there are

two ways that data can move from component to component in a situation like that. One method is to have something in overall control of the Pipeline which picks up data, gives it to a component and then, when the component is done, the data is retrieved and given to the next component in the branch. The alternative is to have the components know about each other and then, once they are finished processing the data, the component itself hands the data on to the next one in the branch. Windows Media Foundation uses the first method and the object which is in overall control of the movement through the Pipeline is called the Media Session. DirectShow, WMF's predecessor, uses the second method. Both methods have their advantages, however, the Media Session method does remove the need for any particular component to know about any other. The Media Session alone has knowledge of the entire Pipeline and it takes care of managing the data processing by the various components.

20. If there are multiple Media Streams in a Media Source, their content will have to be delivered to their respective Media Sinks in a synchronized way. For example, the video and audio streams being played must be synchronized with each other otherwise the sound will not match what is happening on the screen. This is not as easy as it sounds since a Pipeline can have branches with many components. Different Media Streams from the same source will usually take different routes through the Pipeline. The data in the streams will also be of quite different sizes. For example, there will usually be much more video data than audio data. This means that relying on simple arrival timing to synchronize the streams will never work.
21. Furthermore, there are clearly situations in which there is an optimum rate of display. It would not do to have video and sound play faster than normal speed simply because the PC is capable of processing it at a faster rate. Alternatively there are occasions, such as writing to a file, in which the fastest possible rate would be preferred.
22. Clearly, if there are multiple branches in a Pipeline, some data is going to have to be delayed until other data is ready to be

rendered. This means that all branches in a Pipeline must be both throttled and synchronized.

23. A set of related Media Streams that share a synchronized arrival time is called a Presentation.
24. The Media Session will control all of the Media Streams in a Presentation and will also manage the data flow through the Pipeline to ensure that data in all branches arrives at their Media Sinks at the appropriate time. The Media Session also copes with issues such as Fast-forwarding, Rewinding, Pausing and Muting and other common events which interrupt or slow the Media Streams in the Pipeline.

Thus we see that, although there are quite a number of components in Windows Media Foundation, each component fulfills a specific purpose in the Pipeline Architecture. Turning to the internals of the Pipeline we see that there are a few more objects of which we must be aware.

THE PIPELINE INTERNALS

In order to provide a discussion of the internals of a Pipeline, we will once again resort to a list of concepts in order to ensure that each topic is only discussed after previous supporting concepts have been defined. It should be noted that the components discussed below which are containers for media data are also used in the other two architectures and those sections will reference this discussion in order to avoid repetition.

1. Recall from the previous discussion, that the Media Session is responsible for managing the flow of data through the Pipeline. Leaving aside the necessary synchronization requirements, the transport of data through the Pipeline mostly consists of the Media Session giving some data to a Pipeline component, retrieving the data when the component is done with it and then handing it on to the next component. This process repeats until the data eventually reaches the sink.
2. So what injects the data into the Pipeline in the first place? Well, this is also the responsibility of the Media Session. When the Pipeline is built from a Topology (called “resolving” a Topology) any given branch in the Pipeline is provided with a Media Source, a Media Sink and one or more intermediate components known generically as Transforms. The big difference between the three

is that Media Sources do not have inputs, Media Sinks do not have outputs and Transforms have at least one input and one output – but can have more. The Media Session simply treats a Media Source as a Pipeline component to which it does not have to provide data and just pulls data from the Media Source and feeds it into the Pipeline as required.

3. It is easy to see now that a Media Sink is treated by the Media Session as a Pipeline object to which it only has to give data and from which there is no need to retrieve data.
4. Since the data must be picked up from the Media Source and potentially passed in and out of multiple Transforms until it reaches the Media Sink you might imagine that the data is placed in a standardized container for easier transport. The name for this container is called a Media Sample.
5. Inside a Media Sample the data is held in buffers. A Media Buffer is simply a block of media data. If the media data represents a 2D surface such as a video frame or still picture, then the Media Buffer may well be something known as a 2D Buffer. All 2D Buffers are Media Buffers but not all Media Buffers are necessarily 2D Buffers.
6. Among other things, a Media Sample can be thought of as an object that contains an ordered list of zero or more buffers. Why more than one? Well, if the data is streaming in over a network, the Media Source might decide to place all of the data that it has into a single Media Sample. In such an event, the Media Source might not try to coalesce the data into a single buffer and will instead just place multiple buffers in the same sample.
7. Frames are the digital representations of individual pictures in a moving image sequence. Think of the old movie films in which each cell on the film reel had a picture slightly offset in time from the previous one. In general, for uncompressed video data, it would be very unusual to see more than one frame per sample.
8. Yes, the data in a Media Sample can be (and often will be) compressed. This is done for storage space or transmission speed reasons. In the event that a compressed stream is present, a special type of Transform called a Decoder will need to be inserted into the Topology to convert the compressed data into uncompressed data. Thus it is quite possible for a node in a

Pipeline to output much more data than it receives. This process is, of course, reversed if an Encoder Transform is used. There are lots of Codecs (the generic name for an Encoder/Decoder transform) available and Microsoft supplies some with the operating system and third party software supplies others. You can, of course, write your own.

9. The presence of compressed data in the Media Sample will depend on the Media Source used. As mentioned previously, there are some circumstances in which the Topology will automatically find the correct decoder for you, or you can find one yourself and add it to the Topology. Alternatively, you can simply not agree to use that sort of compressed format when adding the Media Source to the Topology and you may well find the Media Source can provide an uncompressed version you can use.
10. The concept of frames has also been extended to audio Media Streams and, although it is possible to see multiple frames in a single audio sample, you would not be likely to encounter an audio frame split across multiple media samples.
11. A Media Buffer is itself a container and the raw media data is only one of the things it carries. The Media Buffer will have two other pieces of information associated with it: the *current length* and the *maximum length*. The current length is the amount of memory currently in use by the buffer and the maximum length is the total amount of memory which can be used for the buffer. The two are different because some Transforms will perform in-place processing as they convert to and from formats and so the amount of data leaving a Transform may be more than was originally input. It is also possible to create a new buffer within the Transform, copy the input data across while modifying it and to return that new buffer (or buffers) instead.
12. A 2D Buffer, since it represents a video surface, can also return the stride of the video. The stride is the number of bytes from one row of pixels in memory to the next row of pixels in memory. If padding bytes are present, the stride is wider than the width of the image.
13. If you recall the earlier discussion of the Media Session you will remember that one of its jobs is to synchronize the flow of data

through the Pipeline so that each branch presents its data to the Media Sink at the appropriate time. The Media Sample also contains information to help with that. This information consists of a *time stamp* and a *duration*. The *time stamp* is used to determine the arrival time of the data at the Media Sink and the *duration* is the length of time the Media Sink should render it.

14. The presence of the *time stamp* and *duration* in the Media Sample makes it much easier for the Media Session to delay a sample in a particular Pipeline branch in order to ensure that the various branches stay synchronized. If you are writing a Transform (covered in the *Working With Transforms* chapter) and are creating new output Media Samples, you will need to take care to copy over the *time stamp* and *duration* values from the input sample in order to make sure things can stay synchronized.
15. Media Samples also can contain a multitude of other attributes – most of which are placed there by the Media Source. These will not be discussed here, but you can easily look them up if you need to.

The above is a whirlwind tour of how Windows Media Foundation Pipeline Architecture operates and the major actors within it. Each component of the above list will be discussed in detail in *The WMF Components* chapter and a step-by-step implementation guide and walk through of an example Pipeline Architecture from the Tanta Sample Projects is presented in the *Implementing the Pipeline Architecture* section in the *Practical WMF Architectures* chapter.

It should be noted that some of the components used in the Pipeline Architecture are common to all architectures. For example, objects such as the Media Samples, Media Buffers and Attributes are also used in the Reader-Writer and Hybrid Architectures. This makes sense – ultimately all of the architectures have to move data from a source to a sink and entities such as Media Samples and Media Buffers are nice generic containers designed specifically for that purpose. They are transferrable as well – a Media Sample is the same in every architecture. This is how the media data from a Pipeline is transferred to a Sink Writer in a Hybrid Architecture. The Media Sample (or sometimes Media Buffer) is just copied and handed over.

THE READER-WRITER ARCHITECTURE

If you have been reading this book linearly (instead of just jumping into the middle) you should, by now, have a pretty reasonable idea of how things work with the Windows Media Foundation Pipeline Architecture. So, of course, we are now going to ignore all of that and discuss the second major type of WMF model known as the Reader-Writer Architecture. As a recap, the Source Reader and Sink Writer are two of the weird and wonderful components (there are others – see the *Other Non-Pipeline WMF Components* section) provided by Microsoft that encapsulate much of the standard WMF Pipeline functionality which make it easier for the user to bolt together media applications.

So, why did Microsoft develop the Source Reader and Sink Writer? As you will see later on when we begin to look at C# code working with Windows Media Foundation in the *Implementing the Pipeline Architecture* section of the *Practical WMF Architectures* chapter, the setup of the Topology, Pipeline and Media Sources and Sinks can sometimes be quite lengthy and involve intricate configuration negotiations among the nodes as a workable sequence of codecs and media format conversion Transforms are resolved.

The team at Microsoft developing WMF recognized the difficulty of the Topology resolution issue. They also realized that probably one of the most common functions required on a Windows system is reading and writing media data to and from a file. In order to address this need, they built components which encapsulate that functionality and make life easier (in some specific cases). These components are called the Source Reader and Sink Writer and they are intended to provide faster more automated alternatives to the full Media Source, Media Sink, Media Session and Pipeline infrastructure.

Once again, we will make a nice list to discuss the Source Reader and Sink Writer.

1. If all you want to do is read a media file and get access to the raw data for your own purposes you probably do not want to have to set up the full Media Session and Pipeline. In this case, the Source Reader is the way to go. In order to use it all you need to do is point it at a file, choose the stream you wish to access, give it an output Media Sub-Type and you will get a flow of Media Samples containing the raw data in Media Buffers which you can use as you see fit.

2. The Source Reader is relatively simple to use and effectively all you need to do in order to use it is to give it a filename, choose a stream from the available Media Major Types and provide it with an output Media Sub-Type. The Source Reader it will do the rest of the configuration for you.
3. The Sink Writer performs a similar function for writing to files. If you have a stream of media data coming in from somewhere (perhaps over TCP/IP) you can just give the Sink Writer a file name, an input Media Type, an output Media Type and you will get all sorts of Media Sink type functionality wrapped up in one little nice package. After that, when the data arrives, all you need to do is stuff it into your own Media Samples hand it off to the Sink Writer for storage.
4. If the Source Reader is supplying the data, you will get the Media Samples and Media Buffers already created which saves you the need of creating your own.
5. People, being people, are also quick to note that devices can be treated as a kind of file and so why can't the Source Reader be used to read a device such as a video camera? Well, it turns out it can. The Source Reader, unlike the Sink Writer, can also access a physical device and present the data it emits to your code.
6. The Sink Writer can only deal with files not devices, so don't try to use it to display video or play sound – for that you need a Media Sink (and a Pipeline).
7. The Source Reader has its own internal Media Source and can contain zero or more conversion Transforms. If you specify an output Media Type different to that of the native Media Type on the stream you selected from the file or device, then the Source Reader is going to have to do some conversion for you.
8. The Sink Writer also has its own internal Media Sink and can also contain zero or more conversion transforms. If you specified an output Media Type different to that of the Input Media Type the Sink Writer is going to try to convert the data for you before it writes it out.
9. Neither the Source Reader nor the Sink Writer provide an easy way to specify a specific Transform to use. They are largely automated in this respect. For example, if necessary, the Sink Writer will dig around on the system (using the registry) and will

find a Transform which can convert the Media Samples of the input Media Type you specified into data that can be written in the output Media Type you specified. If it cannot find a suitable converter it will simply throw an error. This can sometimes cause problems as the Transform chosen may well vary from system to system depending on what is available. If you need more control than that, you should use a Pipeline or Hybrid Architecture and fully specify things yourself. If you make sure to specify identical input and output Media Types and take care to only give it data in that format, then the Sink Writer will not try to find a Transform to convert things.

10. As was noted at the beginning of this section, the Source Reader is not a Media Source and the Sink Writer is not a Media Sink. They each implement the appropriate functionality internally but neither exposes the Media Sink or Media Source functionality in a way the user can easily use.
11. Since they are neither Media Sources or Media Sinks, the Source Reader and Sink Writer cannot be directly included in a Pipeline.
12. Source Readers and Sink Writers can be used alongside a Pipeline. For example, it is possible to intercept the Media Samples as they pass through the Pipeline and hand them off to a Sink Writer. This is known (in this book) as a Hybrid Architecture and is discussed in a following section (see *The Hybrid Pipeline-Writer Architecture* section).
13. It is possible to have multiple Source Readers feeding the same Sink Writer. An example of such would be incoming feeds from a video and audio device being recorded to the same output file.
14. Neither the Source Reader nor the Sink Writer require each other. They are completely independent. However the two are commonly used together in a pattern in which the user writes what is effectively a loop which reads Media Samples from the Source Reader and gives them to the Sink Writer.
15. It is entirely possible to use a Source Reader to send media data to some other application (perhaps over TCP/IP). In such a case the outbound data will probably have to be stripped out of the Media Samples prior to transmission.
16. Similarly, it is possible for the Sink Writer to write data received from an external source. As you might imagine that would

probably require the raw media data to be placed into Media Buffers and Media Samples before the Sink Writer could accept it.

17. It is possible to create a Media Source and convert it into a Source Reader and similarly, it is possible to create a Media Sink and convert it into a Sink Writer. The use of the Media Source or Media Sink in a Pipeline after you have performed such a conversion is not documented and, while it *may*, be possible it is very likely to be problematic and is not recommended. The conversion of a Media Sink not involved in a Pipeline into a Sink Writer and then using the result as a Sink Writer is possible and is sometimes done. However there is little requirement to do this since the Sink Writer is generally easier to create and configure directly.
18. The reverse may or may not be possible. You *may* be able to dig a Media Source or Media Sink out of a Source Reader or Sink Writer but the subsequent use of those entities in a Pipeline is not documented. It is probably best to stick with the more well-known of the creation mechanisms unless you are super experienced or just like to cause trouble for yourself.
19. There are two fundamental types of pattern when using a Source Reader: Synchronous and Asynchronous.

THE READER-WRITER SYNCHRONOUS MODEL

20. When the Source Reader is used in Synchronous Mode, the user (once the Source Reader is configured) simply repeatedly calls `ReadSample()` on the Source Reader until there are no more Media Samples remaining. Once you have the Media Sample you can do what you want with it – including handing it off to a Sink Writer if you wish. The *TantaAudioFileCopyViaReaderWriter* example code provides a simple example of a Source Reader and Sink Writer used in Synchronous Mode.

THE READER-WRITER ASYNCHRONOUS MODEL

21. If you give the Source Reader a Callback Object, you are using the Source Reader in Asynchronous mode. In that case all you need to do is read the first sample (again using a `ReadSample()` call)

and a function in the Callback Object will receive the Media Sample. It is up to the Callback Function to request the next Media Sample before it exits. The Media Sample requested in the Callback Function will again be received by the same Callback Function. The Callback Function must also process the sample (including handing it off to a Sink Writer if you wish) as necessary. This looping process continues inside the Callback Function until no more Media Samples remain. The nice thing about using the Source Reader in Asynchronous mode is that, once started, the Callback Function operates in its own thread and the main part of the code is free to do other things such as manage the user interface. The *TantaCaptureToFileViaReaderWriter* sample file provides an example of a Source Reader and Sink Writer used in Asynchronous mode.

THE HYBRID PIPELINE-WRITER ARCHITECTURE

As mentioned at the start of this chapter, sometimes, you just want the best of both worlds. In other words, you want to have the ease of configuration of a component such as the Sink Writer but you also wish to have the fine grained functionality and control offered by the Pipeline Architecture. In such a case you will probably want to implement a Hybrid Architecture in which much of the processing is performed by a Media Session and Pipeline and then right at the end, once the data is suitably processed, the data is handed off to a Sink Writer.

This presents something of a problem since the Sink Writer cannot be included directly in a Pipeline. So, how is it done? Well the trick is to use a Pipeline component which, when it receives the Media Samples, creates a copy and then gives that copy to the Sink Writer. The original Media Sample is processed as normal and the Media Session, which is running the whole show, is none the wiser.

There are two fundamental ways of performing this “*extracting the data from the Pipeline on the way through*” sort of functionality. The first is to use something called a Sample Grabber Sink. The second is to write a custom Transform which, in effect, does the same thing as the Sample Grabber Sink, but which is a Transform not a Media Sink.

The Sample Grabber Sink is a Microsoft supplied component and is part of WMF. It is also, as its name suggests, an actual Media Sink and hence can be used in a Pipeline like

any other sink. Every system with a standard configuration will have one. The Sample Grabber Sink simply discards any data sent to it and lets the Media Session know that it can accept more. This keeps the Media Session nice and happy and the data travels through the Pipeline. The Sample Grabber Sink can also be given a user written Callback Object and a function in this object will be called before the data is discarded. The parameters to the Callback Function contain the Media Buffer (but not the Media Sample) of the data being processed. It is up to the writer of the Callback Function to process this data as they wish. In many cases this consists of wrapping the Media Buffer in a Media Sample and handing it off to a Sink Writer. In this manner, a copy of the media data can be presented to entities outside of the Pipeline. The *TantaAudioFileCopyViaPipelineAndWriter* Sample Project provides a simple example of this functionality.

The problem with the Sample Grabber Sink is that it discards the data it receives. What if it was desirable to have a sink in the Pipeline which actually performed a function such as rendering the data (perhaps displaying a video feed on the screen). How then might a copy of the data also be recorded in a file? In such a case a Tee Transform could be inserted into the Pipeline which creates two branches. One branch would feed the renderer sink and the other would feed the Sample Grabber Sink. The Sample Grabber Sink would, of course, hand off its data to a Sink Writer as in the previous example. This would work and the Media Session can easily handle such a branched topology. The downside is that branched topologies are a bit tricky to set up.

An alternative method would be to write a Transform located in between the Media Source and the Media Sink which renders the video data to the screen. The Transform could provide the sample grabbing functionality and feed a copy of the data to a suitably configured Sink Writer. This is the approach taken by the *TantaCaptureToScreenAndFile* sample. Both methods, (the Tee and the Sample Grabber Transform) have their advantages. As with most things WMF – there are usually multiple ways to implement any particular requirement.

OTHER NON-PIPELINE WMF COMPONENTS

In your travels you may find that you run across a few other WMF Components that do not use the Media Session and Pipeline Architecture. Two of these are the Transcode API and the Output Protection manager. There are also various API's dedicated to supporting of DirectX3D. These API's are "kinda-sorta" Pipeline related in that they can be used within a Media Session and Pipeline Architecture. However they can also be used with a Source Reader as well. The DirectX3D API's will not be discussed further

here and the others will be given only a brief coverage to note down the main points. In truth, if you are just learning WMF, you will probably not need to interact with any of these technologies for a while.

THE TRANSCODE API

Remember how the Reader-Writer Architecture was designed to make the read and write of media data easier in certain circumstances? Well, the Transcode API was designed for the process of conversion of a digital media file from one format to another. This particular technology does not seem to be very commonly used – the ability to add Transforms into the Pipeline provides a lot more functionality for only slightly more work. However, the Transcode API is there to use if you need it – it is not going to be discussed further in this book.

THE OUTPUT PROTECTION MANAGER

The Output Protection Manager (OPM) is a kind of copy-protection mechanism that enables an application to protect media content as it travels over from a physical source device to a sink device. Basically, it is an attempt to address the problem that no matter how encrypted or copy protected media data might be, it must inevitably be unencrypted or un-copy protected somewhere along the transport path in order for a renderer to display it. OPM technologies are used extensively inside the Protected Media Path (PMP) functionality which is a Pipeline based Digital Rights Management (DRM) type copy-protection mechanism. It is also available outside the Pipeline for applications that wish to use it. The OPM is quite an advanced topic and so will not be discussed further in this book.

THE IMFCAPTUREENGINE AND IMFSENSORDEVICE

These appear to be technologies designed to assist with the capture of video, audio and sensor data. They do not appear to be in common use and little information (other than the help files) is available online regarding them. They will not be discussed in this book.

WHICH ARCHITECTURE TO USE?

It is hard to provide a definitive prescription which can be used to select a particular architecture for an application - most outcomes can be achieved in any one of several ways. However, some generalities can be provided. If you need to transform or manipulate the media data in between acquiring it and dispensing with it then you will probably need to use the Pipeline or Hybrid Architectures. These would give you the

ability to inject Transforms into the Pipeline and modify the data as necessary. If you are creating your own media data, for example generating an animation, then the Sink Writer could be used to write the information to the disk since its configuration is generally much simpler. In the event that your media streams start on a physical device and end on a physical device without the need for any conversion or transformation, then any of the architectures would be suitable – although probably the Reader-Writer Architecture would be the simplest.

Windows Media Foundation: Getting Started in C#

Chapter 4

MF.NET PROGRAMMING FUNDAMENTALS

This chapter is intended to provide you with an overview of the basic building-block tools and concepts you will run into when you begin to program in Windows Media Foundation. As such it will be an assortment of “useful” things you will be glad to have run across before you start programming in Windows Media Foundation.

Having said that, in the discussion below, you will probably come across WMF entities which have only been briefly mentioned in in the previous chapters. Don’t worry too much about this – everything will be explained in much more detail in later on. However, it will be very useful for you to have been exposed to some of the following ideas by the time that happens and your ability to absorb the new information then will be greatly improved. So, let’s get an understanding of some fundamental concepts before we move on.

YOU MUST USE AN [MTAThread] CODE DECORATION

Internally the components in Windows Media Foundation use a number of threads. These components are also COM objects and there are two basic types of COM multi-threading model. These are “*Single-Threaded Apartment*” and “*Multi-Threaded*

Apartment". In reality, you do not need to know any of the gory details of this Apartment Model thing as long as you are aware that...

1. You usually have to enable the Multi-Threaded Apartment model or your WMF code will not work in many (if not most) cases.
2. The fact that you have enabled the Multi-Threaded Apartment model will cause problems with other COM components that use forms. The commonly used (Windows supplied) OpenFileDialog file picker component is an example of this.

ENABLING THE MULTI-THREADED APARTMENT MODEL

Most normal C# form based programs are launched from the `Main()` function in the `Program.cs` file. Enabling the COM Multi-Threaded Apartment model is as simple as placing the text `[MTAThread]` immediately above your `Main()` function in the `Program.cs` file of your C# application. The sample code section below shows an example of this.

```
/// <summary>
/// The main entry point for the application.
/// </summary>
///
/// SUPER IMPORTANT NOTE: You MUST use [MTAThread] here. If you use [STAThread]
/// you may get errors
[MTAThread]
//[STAThread]
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new frmMain());
}
```

Source: `TantaAudioFileCopyViaPipelineMP3Sink::Program.cs::Main()`

The `[MTAThread]` tag is called a "Code Decoration" and it is a signal which tells the compiler to change its behavior. The default model is `[STAThread]` and it is not at all obvious (or well documented) that you have to change it to `[MTAThread]` in order for Windows Media Foundation to work well. It should be noted that in many cases your WMF C# application will work just fine with the `[STAThread]` model – however it is possible that odd runtime errors could result and you will not get a sensible error message explaining what happened.

IMPORTANT: Always use an `[MTAThread]` tag just above your `Main()` function when using Windows Media Foundation. If you do not do this then you could introduce problems.

COPING WITH [MTATHREAD] PROBLEMS

Ok, so you have to use the Multi-Threaded Apartment model in most MF.Net applications. The downside is that a lot of the COM based components Windows supplies are not thread safe and only the `[STAThread]` model guarantees thread safety for them. Some examples of this are the Open File Dialog, the Directory Picker, the Web Browser control and things like Clipboard Drag and Drop. If you use these tools in an application based on the `[MTAThread]` model without taking special precautions then you may experience odd lockup behavior. Some components such as the `OpenFileDialog` will simply throw an error if you try to use them in a `[MTAThread]` based application.

So, does this mean you cannot use the `OpenFileDialog` component in your MF.Net application? Of course you can use it, but you just have to take some special precautions to get it to work. The code section below from the *TantaFilePlaybackAdvanced* Sample Project demonstrates this process.

[illegible]

The above code simply uses the `TantaOpenFileDialogInvoker` class in the *TantaCommon* project to put the `OpenFileDialog` temporarily back on a thread which uses an `[STAThread]` model. The actual code for the `TantaOpenFileDialogInvoker` class, while not overly complex, is too much of a digression for us to follow at this point. You can easily inspect that code in the Tanta Sample Projects to find out how it works.

INITIALIZING WINDOWS MEDIA FOUNDATION

The Windows Media Foundation substrate must be initialized before any of the WMF components will work. This is done with a call to the static `MFStartup()` function and is usually performed in the constructor of the applications main form. You would probably have figured this out within the first ten seconds of reviewing any sample code. However, for completeness, let's note down the process here.

```
// we always have to initialize MF. The 0x00020070 here is the WMF version
// number used by the MF.Net samples. Not entirely sure if it is appropriate
hr = MFExtern.MFStartup(0x00020070, MFStartup.Full);
if (hr != 0)
{
    LogMessage("Constructor: call to MFExtern.MFStartup returned " + hr.ToString());
}

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::frmMain
```

Not too amazing, but at least now you can't say you weren't told.

SHUTTING DOWN WINDOWS MEDIA FOUNDATION

In a similar way, Windows Media Foundation should be shut down when the application closes or no longer needs WMF. This is typically done in the `FormClosing()` handler of the applications main form. The procedure for closing down WMF is simple – just a straight forward call to the `MFShutdown()` static function.

```
private void frmMain_FormClosing(object sender, FormClosingEventArgs e)
{
    LogMessage("frmMain FormClosing");
    try
    {
        // do everything to close all media devices
        CloseAllMediaDevices();

        // Shut down MF
        MFExtern.MFShutdown();
    }
    catch
    {
    }
}

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::frmMain_FormClosing
```

Shutting down WMF may be simple - but there are other considerations. Note the call to `CloseAllMediaDevices()` in the above code section prior to the call to the `MFShutdown()` function. The `CloseAllMediaDevices` user written function releases the Windows Media Foundation components which are stored as class variables. This is not just a memory recovery process – some of the WMF components (such as file sinks) need to be formally shutdown in order to properly finish off writing their output data. The *Releasing COM Objects* section further on in this chapter contains a more detailed discussion on the releasing and shut down process.

MOST WMF OBJECTS ARE INTERFACES

Most non-trivial WMF objects are dealt with as interfaces not as the objects themselves. They actually are objects, of course, but you rarely know or care what type they are. What you will be interested in are the interfaces that object offers. For example, you will never deal with a `MediaSession` object directly. You will, though, regularly deal with some object, of some unknown type, which implements the `IMFSession` interface.

Do not expect to deal with objects of a certain type. You will almost always deal with some opaque entity of unknown type which implements a certain interface. Usually that object will also implement multiple other interfaces.

Windows Media Foundation interfaces are exactly the same thing in concept as C# interfaces and in MF.Net they *are* C# interfaces.

As a C# programmer you are much more familiar with the concept of interfaces than the C++ coders for whom the WMF classes were originally intended. This makes it much easier for the C# community to understand the idea. For example, in C# you would have no problem considering an object to be both a `Person` and `Comparable`. Similarly, an object can simultaneously be an `Organization` and `Comparable`. This does not mean a `Person` is an `Organization`. You could try and compare the two – but in most implementations the result would always be `False`.

In the above example, the `Comparable` interface is just a defined set of methods which form a contract. The actual mechanism of the comparison is left up to the implementer of the interface. Thus a `Person` object might compare names, gender, birthdates and national id number in order to assess equivalence. An `Organization` object would have an entirely different set of comparison criteria.

The point of interfaces is that the caller knows what to expect. It can take an object and, if that object implements a defined interface, then the caller will be able to interact with that object according to the specification of that interface.

You will constantly run across interfaces in Windows Media Foundation. For example, the Enhanced Video Renderer control which displays video is essentially just a collection of interfaces. In fact the EVR implements over 16 of them and each is dedicated to a different function. You never really interact with the Enhanced Video Renderer itself, instead you treat it, on different occasions, as an object of type

`IMFVideoDisplayControl`, `IMFVideoPresenter`, `IMFVideoRenderer` (and so on) and interact with it via the properties those interfaces expose.

Similarly the Media Session object is intrinsic to most of the Pipeline Architecture WMF programs you will write. Yet you will never interact directly with an object of type `MediaSession`. All you will ever do is work with an object which exposes the `IMFMediaSession` interface. In fact, you will never actually create this object – there is no concept of a new `MediaSession()` call in WMF. You get the Media Session object, with a call to a static function like `MFExtern.MFCreateMediaSession()`, which builds it for you and passes it back as an `out` variable. You have no idea what the type of the object you receive really is, or whatever else it might be. All you know is that it is some blob of code which implements the `IMFMediaSession` interface.

IUNKNOWN

Each interface in Windows Media Foundation is required to inherit and implement the `IUnknown` interface. The relationship between `IUnknown` and WMF interfaces is exactly analogous to the relationship between all C# objects and the base class `object`.

As with the `object` class, the `IUnknown` interface does not do all that much. The one thing it does do (that is relevant to the discussions here) is that it implements a `QueryInterface` function. This function will be discussed in a subsequent section.

The other thing the `IUnknown` interface does is that it allows a programmer to refer to some unknown WMF object as an `IUnknown`. This is quite useful when one wishes to be generic about things. In the example below, derived from the *TantaVideoFormats* sample code, it is possible to see a Callback Object being set in an Attribute.

```
// Set our Callback Object as an IUnknown pointer in the attribute container.
hr = attributeContainer.SetUnknown(
    MFAttributesClsid.MF_SOURCE_READER_ASYNC_CALLBACK,
    asyncCallbackHandlerIn);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("failed on call to MFCreateAttributes, retVal=" + hr.ToString());
}

Source: TantaCommon::TantaWMFUtils::CreateAsyncSourceReader
```

The `asyncCallbackHandlerIn` value passed in the above example is really an `IMFSourceReaderCallback` and pretty much everything in the code knows this – so why does it let the Attribute Container treat it as an `IUnknown`? Well, the reason is that Attributes just don't want to deal with that level of complexity. In exactly the same way that you might define a function to accept an object of type `object` and expect the code to figure it out later, WMF uses `IUnknown` interfaces to pass interfaces around. The implicit understanding is that whatever eventually reads this attribute will look at the

`MF_SOURCE_READER_ASYNC_CALLBACK` GUID, will know what it is and will be able to deal with it appropriately.

The really big takeaway from the above discussion is that when you see various help files and code samples constantly referring to `IUnknown` just realize they are really referring to the WMF interface equivalent of the `object` class.

GETTING AN INTERFACE FROM AN INTERFACE

Given that most Windows Media Foundation objects you interact with will actually offer multiple interfaces, the implementers of WMF have provided ways to ask an object for a specific interface.

Note that, in general, one cannot always cast an object from one interface type to another. Sometimes the original object implements the new interface directly and sometimes the original object simply contains an object of the new interface type. In other cases, the object does not actually implement a specific interface but can actually build and return an object that does. In all but the first case, a cast will simply fail.

The way you get an interface from an existing interface is either via a call to `MfExtern.MFGetService` or a call to `QueryInterface` on the interface itself.

The `MfExtern.MFGetService` call is the more commonly used of the two. This function is simply a helper function that eventually calls `IMFGetService::GetService` method. And just what is the `IMFGetService` interface you ask? Well, the `IMFGetService` interface is yet another interface implemented by most WMF entities and it allows you to query that object for other interfaces. The `IMFGetService::GetService` call queries the object for an `IMFGetService` interface and if that interface is present, calls `GetService` on the object and returns the results.

In any event, the `MfExtern.MFGetService` call delivers an object of the requested type from another object which knows about it or can create it. Its usage is pretty simple. Below is an example of the `IMFVideoDisplayControl` interface of the Enhanced Video Renderer object being obtained from a Media Session object.

```
// we need to get the active IMFVideoDisplayControl. The EVR presenter implements this
// interface and it controls how the Enhanced Video Renderer (EVR) displays video.

hr = MfExtern.MFGetService(
    mediaSession,
    MFServices.MR_VIDEO_RENDER_SERVICE,
    typeof(IMFVideoDisplayControl).GUID,
    out evrVideoService
```

```
    );  
    if (hr != HRESULT.S_OK)  
    {  
        throw new Exception("call to MFExtern.MFGetService failed. ");  
    }  
    if (evrVideoService == null)  
    {  
        throw new Exception("call to MFExtern.MFGetServicee failed");  
    }  
  
    // set the video display now for later use  
    IMFVideoDisplayControl evrVideoDisplay = evrVideoService as IMFVideoDisplayControl;  
  
    Source: TantaCommon::ctlTantaEVRFilePlayer::MediaSessionTopologyNowReady
```

As was mentioned previously in the *IUnknown* section, all interfaces in Windows Media Foundation ultimately derive from the *IUnknown* interface. One of the functions that the *IUnknown* interface requires the implementer to support is *QueryInterface*. The *QueryInterface* call is designed to accept a GUID and return an object which supports this interface. There are a number of strict rules regarding the behavior of the *QueryInterface* function. These rules are quite intricate and not really too relevant, given the discussion below and so they will not be discussed there.

However, some implementations of *QueryInterface* in WMF components were known to be problematic – particularly in early versions. Probably for this reason they are not used all that much. Certainly the MF.Net library source code contains a comment indicating that they have been troublesome.

The *QueryInterface* call takes a service identifier GUID and an *out* pointer to the returning object. It is important to be aware, as discussed earlier, that it can return another object that implements the interface instead of returning a pointer to the original object that is queried.

THE DIFFERENCE BETWEEN MFGETSERVICE AND QUERYINTERFACE

Ultimately both of these calls, if they succeed, return an object which implements the specified interface. They are generally equivalent, however, there is one technical difference. When *QueryInterface* returns an interface one of the strict conditions mentioned above is that it should be possible to query the returned interface and retrieve the original interface. There is no such requirement on the *GetService* method. It can, and will, return an object that knows nothing about the original creator. Most of the time you never need to know who created the object (you already know this) and the inability to get the original interface back again is not critical.

WMF OBJECT CREATION IS INDIRECT

Windows Media Foundation objects are almost never created directly. For example, you will rarely ever write a line of code like `WMFObject foo = new WMFObject()`. This is a consequence of dealing primarily with objects in the COM layer and is also associated with the fact that you are dealing largely with interfaces. This indirect creation process makes sense if you think about it. WMF will use COM and the registry to find and create many, if not most, of the WMF objects you will use. You will not know at compile time the type of this object – just the interfaces it supports. So, if you do not know the fundamental type of object a Media Session really is, then how can you create one directly? The simple answer is that you cannot – something else has to create it for you and all you will get back is an object of unknown type which you definitely know implements the `IMFMediaSession` interface.

Of course you have to get the media object which does the creation for you from somewhere else and that object in turn has to be generated from something else and so on. There are a variety of different creation methods, but if you walk a particular chain back to the source in most cases you will find that, ultimately, the ancestor WMF object was created by calling some static procedure in the `MFExtern` class which delivers it. Of course, you do not know any of the details of how the `MFExtern` call managed to do what it did.

Thus, in summary, an important thing to remember when dealing with Windows Media Foundation is that...

WMF objects are almost never directly created. Usually you call something else which builds it for you. In other words, you will never write `new MediaSession()` to create a session. It is always the equivalent of “*hey existing object, make something with an `IMFSession` interface for me and hand it over*” or “*hey static function, work some magic and make me an object with this interface*”.

If you keep the above idea in mind as you read the source code examples then things will make a lot more sense.

RELEASING COM OBJECTS

As discussed in the preceding *WMF Object Creation is Indirect* section, the process of creating a Windows Media Foundation object is rarely straightforward. For example, one of the ways of creating a Media Sink is to use something called an Activator. An Activator is an object which is usually obtained via a call to a static WMF function and which can automate the task of creating a Media Sink. Let's not worry too much about the details at this point and focus on the fact that in order to create a Media Sink we first had to obtain a temporary object which is no longer needed once the desired object exists. The point is, both the temporary object and the object of interest are COM entities and are not created within the memory managed by the .NET Common Language Runtime (CLR). The memory used to create the object is located on the unmanaged system heap and will need to be freed up.

The C# garbage collection mechanism cannot properly free up the memory of most of the objects your application obtains from WMF. It is up to you, the programmer to release these objects when you are done with them – otherwise you will get a memory leak.

The process of releasing a COM object closes the object down correctly, if necessary, and frees up the memory it is using. Of course, the two objects discussed above (the Activator and the Media Sink) are probably going to be released at different times. This makes sense if you think about it for a bit. The Activator is temporary and is only used briefly so it is reasonable to free it up as soon as possible. The Media Sink might be needed for some time and so it might only be released when the application finishes operations or closes.

Releasing an object is simple. You simply call the `Marshal.ReleaseComObject` COM function and pass the object in as a parameter. The code section below shows how an Activator might be released after it is given to a Topology Node (which will eventually create the EVR Renderer).

```
/// ++++++
/// <summary>
/// Create a topology node for EVR Video Renderer sink. The caller must
/// release the returned node.
/// </summary>
/// <param name="videoWindowHandle">the handle to the window on which
/// video streams will display</param>
/// <returns>the output stream node</returns>
/// <history>
/// 01 Nov 18 Cynic - Originally Written
/// </history>
public static IMFTopologyNode CreateEVRRendererOutputNodeForStream(IntPtr videoWindowHandle)
{
    HRESULT hr;
```

```

IMFTopologyNode outputNode = null;
IMFActivate pRendererActivate = null;

try
{
    // Create a downstream node.
    hr = MFExtern.MFCreateTopologyNode(MFTopologyType.OutputNode, out outputNode);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to MFCreateTopologyNode failed. Err=" + hr.ToString());
    }
    if (outputNode == null)
    {
        throw new Exception("call to MFCreateTopologyNode failed. outputNode == null");
    }

    // Create an activation object for the enhanced video renderer (EVR) media sink.
    hr = MFExtern.MFCreateVideoRendererActivate(videoWindowHandle, out pRendererActivate);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("MFCreateVideoRendererActivate failed. Err=" + hr.ToString());
    }
    if (pRendererActivate == null)
    {
        throw new Exception("failed. pRendererActivate == null");
    }

    // Set the IActivate object on the output node. Note that not all node types use
    // this object. On transform nodes this is IMFTransform or IMFActivate interface
    // and on output nodes it is a IMFStreamSink or IMFActivate interface. Not used
    // on source or tee nodes.
    hr = outputNode.SetObject(pRendererActivate);

    // Return the IMFTopologyNode pointer to the caller.
    return outputNode;
}
catch
{
    // If we failed, release the outputNode
    if (outputNode != null)
    {
        Marshal.ReleaseComObject(outputNode);
    }
    throw;
}
finally
{
    // Clean up.
    if (pRendererActivate != null)
    {
        Marshal.ReleaseComObject(pRendererActivate);
    }
}
}

```

Source: TantaCommon::TantaWMFUtils::CreateEVRRendererOutputNodeForStream

In the code section above, particularly note that an `IMFTopologyNode` is also created. The caller is expected to release the Topology Node object at the appropriate time. This sort of “*expected to release*” behavior is ubiquitous in Windows Media Foundation and you must pay attention to it in your code or your program will consume ever increasing amounts of memory until it crashes. It is fairly safe to say that every object given to you (large or small) will need to be released with a call to `Marshal.ReleaseComObject()`. The entity obtaining the WMF object is always responsible for cleaning it up.

Both a `catch` and `finally` block are used to ensure that the objects which are created, get properly released. The `catch` block is used to dispose of the Topology Node if there is an error and the `finally` block is used to properly release the Activator once it is passed into the Topology Node.

Also note how the Activator object is released once it has been given to the Topology Node. There is no indication that the `outputNode.SetObject()` call actually used or was finished with the Activator. In fact, it did not use the Activator and definitely was not finished with it. The Activator needs to be present in the Topology Node until the Topology is resolved (converted into a Pipeline) and that does not happen for quite a few lines of code beyond that point. So, how can the Activator be released at the bottom of the `CreateEVRRendererOutputNodeForStream` function if the Topology Node still needs it? Well, the `outputNode.SetObject()` call creates its own reference to the Activator object. Calling `Marshal.ReleaseComObject()` in the `finally` block just decrements the reference count and the Activator will not be formally released until the Topology Node itself lets it go with a call to `Marshal.ReleaseComObject()`. In general, most WMF objects work like this and this mechanism permits you to dispose of temporary objects immediately rather than having to implement some complex release procedure later.

You will occasionally see a call to `SafeRelease()` instead of `ReleaseComObject()`. `SafeRelease` is just a wrapper function which is part of MF.Net and is replicated in various classes within the Tanta Sample Projects. You can easily inspect this code and so it will not be reproduced here. However, it does perform quite a few checks before releasing - such as checking that the object to be released is actually a COM object. This is pretty much all of debatable utility since your application should really know the type of object it is releasing.

RELEASING CLASS VARIABLES

Obviously you can structure your code how you wish – but for reference, we'll discuss how the Tanta Sample Projects clean up and release the Windows Media Foundation objects they store in class variables.

When WMF is no longer needed, or concludes its operations, a call to `CloseAllMediaDevices()` is made. This call can be made at any time but is always triggered from the `FormClosing()` event of the application as well.

```

/// ++++++
/// <summary>
/// A centralized place to close down all media devices.
/// </summary>
/// <history>
/// 01 Nov 18 Cynic - Started
/// </history>
private void CloseAllMediaDevices()
{
    HRESULT hr;
    LogMessage("CloseAllMediaDevices");

    // close and release our Callback Object
    if (mediaSessionAsyncCallbackHandler != null)
    {

```

```

        // stop any messaging or events in the Callback Object
        mediaSessionAsyncCallbackHandler.ShutDown();
        mediaSessionAsyncCallbackHandler = null;
    }

    // close the session (this is NOT the same as shutting it down)
    if (mediaSession != null)
    {
        hr = mediaSession.Close();
        if (hr != HRESULT.S_OK)
        {
            // just log it
            LogMessage("call to mediaSession.Close failed. Err=" + hr.ToString());
        }
    }

    // Shut down the media source
    if (mediaSource != null)
    {
        hr = mediaSource.Shutdown();
        if (hr != HRESULT.S_OK)
        {
            // just log it
            LogMessage("call to mediaSource.Shutdown failed. Err=" + hr.ToString());
        }
        Marshal.ReleaseComObject(mediaSource);
        mediaSource = null;
    }

    // Shut down the media session (note we only closed it before).
    if (mediaSession != null)
    {
        hr = mediaSession.Shutdown();
        if (hr != HRESULT.S_OK)
        {
            // just log it
            LogMessage("call to mediaSession.Shutdown failed. Err=" + hr.ToString());
        }
        Marshal.ReleaseComObject(mediaSession);
        mediaSession = null;
    }

    // close the media sink
    if (mediaSink != null)
    {
        Marshal.ReleaseComObject(mediaSink);
        mediaSink = null;
    }
}

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::CloseAllMediaDevices

```

The actual contents of the `CloseAllMediaDevices()` function varies from application to application. In this particular example, taken from the *TantaAudioFileCopyViaPipelineMP3Sink* Sample Project, we can see that the Callback Object for the Media Session is the first thing shut down. Shutting down the Media Session is a two-step process. First the Media Session is closed with a call to its `Close()` function sometime later the Media Session is finally shut down with a call to its `Shutdown()` function. The call to `Close()` must always come first – it will trigger a `MESessionClosed` event which may be useful to whomever is looking for that sort of notice. Once the Media Session has been closed, very few of the functions on the `IMFMediaSession` interface are operational. Pretty much just the `Shutdown()` function and a few other informational ones are available. Note that the Media Source and Media Sink are also released in the `CloseAllMediaDevices()` function because they too are class variables. Every other WMF object obtained (however briefly) by the application was released as soon as it was no longer needed.

GUIDs

Simply put, GUID's are 128 bit numbers and they are used like a name to identify information. The term GUID is an acronym which stands for *Globally Unique Identifier*. Actually GUID is Microsoft's term for it, the more general name for a GUID is UUID which stands for *Universally Unique Identifier*.

To understand why GUID's are used, let us consider the problem they are trying to solve. Say, for example, you have some information which has to be stored somewhere - perhaps in the registry or in a data storage class like an Attribute (Attributes are discussed in much more detail in the *About Attributes* section below). In order to make the stored information useful it has to have a name. In normal computer code this name would just be the name of whatever variable you stuffed the value into. This mechanism does not really work when storing things external to the software – especially if it is intended to be read by multiple other programs. If just the value is placed in some memory location, how could some other program know what the value was and how it might be used? Maybe it could use the position of the value in the store – well that might work for a short while but if other things are adding and removing items the concept soon breaks down. Certainly, if the value is picked out of the store and passed around through various entities, each entity has to know before it receives it that a particular bit of data is of a particular named type – this too breaks down in distributed systems as each entity has to have intimate knowledge of the data it is processing.

So, what is done is to use a key-value pair. The key is a name and the value is whatever you want it to be. If the key is a name then what name should be used? You cannot have just everybody who wants to do so make up a text string and name something. For one thing, human nature being what it is, people would probably come up with remarkably similar names (`size` for example). A simple name string would generate naming collisions all over the place and the system would rapidly become unusable as the same name was used for different things. So, to fix that, one might set up some sort of universal naming mechanism – like an enum in concept. You start at 1 and hand out sequential values to anybody who asks. This would work in theory, but in reality, people working remotely would just make up their own numbers and soon you would have collisions all over the place again.

So what was done was to recognize that a randomly generated number of sufficient length is unlikely ever to be duplicated anywhere else. That is all a GUID (or UUID) is - a random 128 bit number which somebody generated and then named. It is theoretically possible that two randomly generated GUID's will be a match but the odds are infinitesimal – you can easily look up the math in any search engine if you wish.

You will see GUID's used all over the place in Windows Media Foundation. They sometimes visually look a lot like enums when used in code – but they are not – they are just tags or labels. Once you understand this, the available sample code becomes much more understandable. Also note that there is no central location in which all of the GUIDs in Windows Media Foundation are recorded. For example, the GUID `MFMediaType.Audio` is located in the `MFMediaType` class and the `MFAAttributesClsid.MF_EVENT_TOPOLOGY_STATUS` is located in the `MFAAttributesClsid` class.

C# has a dedicated `Guid` datatype which can deal with GUID's and so it is trivial to do things like compare GUIDs or pass them in and out of functions. It is important to note that GUIDs should never be sequential or related in any way and that if you ever need a new one you can use any one of a dozen online GUID generators. The `Guid` datatype also has a static function which can generate new, unique GUID's for you. It is a simple process - a call like `Guid myGuid = Guid.NewGuid()` will do it. Note that in C# the `Guid` datatype is actually a struct and in C# structs are value types. Thus you cannot declare a new `Guid` variable with a statement like `Guid myGuid = null`. If you do this you will get a compile time error. The proper declaration for an unused `Guid` would be `Guid myGuid = Guid.Empty`.

In general, you never need to know what the 128 bit content of a GUID is. The actual value is stored in a variable of type `Guid` and you use the names of these variables as you would an enum. So here is an important tip...

If you go around thinking of GUIDs as a user generated, distributed, always unique enum you will not go too far wrong.

You will also see the acronym CLSID being used. Technically, a CLSID is a GUID that identifies a COM object. In general, in order to create a COM object, you need to know its CLSID. Actually, the name CLSID is another handy tip for getting a handle on things in WMF...

In general, if you see a GUID referred to in Windows Media Foundation as a CLSID you know you are probably working in some way with a COM object and not some chunk of data being passed around the system.

ABOUT ATTRIBUTES

Windows Media Foundation uses Attributes to store configuration information. You will be dealing with them constantly and so it is a good idea to get an understanding of what they are and what they do.

An Attribute of an object is simply a key-value pair in which the key is stored in that object as a GUID and the data is stored in that object as a class type called a PropVariant. Since most WMF entities will need more than one configuration item, WMF objects will maintain a collection of these key-value pairs. It should be noted that, since the WMF objects configuration is effectively just a lookup-list of items based on the value of the key, the order in which you add Attributes to the WMF object is irrelevant. Similarly, if you are enumerating the Attributes in a WMF object, you cannot rely on there being any consistent order in which the Attributes are returned.

In Windows Media Foundation there is no `MFAAttribute` class. The Attribute itself is a key-value pair consisting of a GUID and a PropVariant and any object which maintains such a key-value pair is considered to have an Attribute.

As you might imagine, there is a considerable need to set, retrieve and store the Attributes in an object and so it is likely there is a standard collection class for this. In fact there is such a class, the actual type of which is unknown (see the discussion in the *Most WMF Objects are Interfaces* section), however the instantiated object does implement the `IMFAttributes` interface. So, how do you create this useful Attribute container class which will enable you to build your own collection of Attributes? The answer is that you use a call to the static `MFCreatAttributes()` function in the MF.Net `MFExtern` library. Note that the `MFCreatAttributes` function is somewhat misleadingly named – it does not actually create any Attributes – you have to do that yourself. What the `MFCreatAttributes` function actually does do is create a container to which you can add your own Attributes. It probably should be named something like `MFCreatAttributeContainer`, however, it is not, so you will just have to mentally translate that odd name.

There is a standard interface which manipulates a collection of Attributes and this is called `IMFAttributes`.

Note that many Windows Media Foundation objects will also implement the `IMFAttributes` interface. Some of them will meet the requirements of this interface

internally others will just create and maintain an internal `IMFAttributes` object as an Attribute store.

PROPVARIANT

In the following sections we are going to discuss Attributes in detail and the usage of this concept is ubiquitous in Windows Media Foundation. To understand WMF you will need to understand Attributes, and in order to understand Attributes, you will first need to have a good background in a supporting concept known as the PropVariant. Attribute data is stored in PropVariant datatypes and although this is the primary usage for the PropVariant construct in MF.Net, you will also occasionally see PropVariants used independently here and there in Windows Media Foundation code.

Before we go into what a PropVariant is, let's discuss some background as to why a PropVariant was thought to be necessary in the first place. Normally when a function is called the data in, or out, is strongly typed. You know perfectly well if you are passing in a `float` or receiving a `bool` or whatever. A PropVariant allows the system to pass around data in which the data type is self-described. In other words, the PropVariant functions in C++ kind of like an `object` base class construct does in C# except that a PropVariant really can only contain a specific list of value types.

A PropVariant is just a way of passing data around in a distributed multi-threaded system in which the data itself tells the recipient what data type it is.

C++ implements the PropVariant as a `struct` which has only two items of any importance. The first of these is a field containing a `VARTYPE` enum named `vt`. This field describes the type of data the structure contains. Besides a few reserved fields, the remainder of the `struct` is a `union` of numerous named fields – one for each data type. The code below shows the first part of the definition.

```
typedef struct PROPVARIANT {
    VARTYPE vt;
    WORD     wReserved1;
    WORD     wReserved2;
    WORD     wReserved3;
    union {
        CHAR          cVal;
        UCHAR          bVal;
        SHORT          iVal;
        USHORT         uiVal;
        ...
    }
};
```

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa380072\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa380072(v=vs.85).aspx)

If you are unfamiliar with C++ all you really need to know is that in a `union` the variables `cVal`, `iVal` etc. all use exactly the same memory – they overlay each other. That is what

the `union` operator does. Thus you can refer to `cVal` and get a character and `iVal` and get the same data as an integer. Of course, if the `PropVariant` really contains a four byte `iVal` and you access it as a 1 byte `cVal` you might get a very odd character indeed. The `PropVariant` does *not* convert or cast the data – it just returns whatever happens to be there as whatever datatype it is referred to.

C# does not possess an operator which matches the C++ `union` in function. Accordingly, the `PropVariant` is implemented as a class and the data is stored in individual variables which **do not** share memory. Thus, if you store an `int` value into a C# `PropVariant` class and subsequently try to access that stored data via the `GetUInt()` call you will not get back an unsigned version of the previous `int` value. Unlike C++, in the C# `PropVariant`, the two values are stored in entirely different memory locations. It is a good idea to keep this fundamental difference in mind if you are reading C++ sample code which uses `PropVariant` structs. This is important enough to write it out again and put it in a big warning notice – if you are translating C++ WMF source code to C# you need to be aware of this difference things will not work the same.

WARNING: In C# the `PropVariant` is implemented as a class and the data is stored in individual variables which do not share memory. Unlike C++ which uses a `union` structure, in the C# `PropVariant`, the two values are stored in entirely different memory locations.

Mostly, however, it is rare when working with Windows Media Foundation to need to look at the data content of a `PropVariant`. Typically, you are populating one with some data and sending it off to WMF to process. In such cases, MF.Net will automatically Marshal the C# `PropVariant` into a C++ `PropVariant` on your behalf and you do not need to concern yourself with the details.

The C# `PropVariant` class contains numerous overloaded constructors and creating one with the `new` operator is enough to set the data and the record the datatype internally. An example of this creation process would be `new PropVariant((Int64)presentationTime)` which creates a `PropVariant` class with a stored `Int64` bit value.

WARNING: Take very great care to get the datatype correct when creating a `PropVariant` object. The recipient will almost certainly check the datatype for correctness and if you get it wrong it will throw an error.

This error is more subtle than it seems because the usual compile time checks are invalidated. The example below demonstrates how a data type error can occur with a `PropVariant`.

```
// Get the presentation time as a UInt64
UInt64 presentationTime = TantaWMFUtils.GetPresentationTimeFromSession(mediaSession);

// the line below will not work and you will not get
// a compile time error. mediaSession.Start() expects an
// Int64 not a UInt64 value.
// mediaSession.Start(Guid.Empty, new PropVariant(presentationTime));

// the cast here makes it work
mediaSession.Start(Guid.Empty, new PropVariant((Int64)presentationTime));

Source: None
```

In the above code section note how the `PropVariant` which is passed into the `mediaSession.Start()` call is a `UInt64`. The problem arises because the `mediaSession.Start()` call actually expects and `Int64`. Since in C#, the `PropVariant` `UInt64` storage does not share the same memory like it does in C++, the value you input cannot just “*be treated as a UInt64*”. If you do not perform the cast (as shown in the second example) so as to activate the `Int64` version of the constructor, an invalid value will be passed into WMF when MF.Net converts the `PropVariant` to a C++ version for the WMF layer to use.

CREATING AND POPULATING ATTRIBUTES

The code section below illustrates the creation and population of an `IMFAttributes` object which is then used to provide configuration parameters to another WMF function.

```
IMFMediaSource videoSource = null;
HRESULT hr = 0;
IMFAttributes attributeContainer = null;

try
{
    // Initialize an attribute store. We will use this to
    // specify the enumeration parameters.
    hr = MFExtern.MFCreateAttributes(out attributeContainer, 2);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("failed on call to MFCreateAttributes, retVal=" + hr.ToString());
    }
    if (attributeContainer == null)
    {
        // we failed
        throw new Exception("attributeContainer == IMFAttributesClsid.null");
    }

    // setup the attribute container, it is always a VIDEO SOURCE here
    hr = attributeContainer.SetGUID(
        IMFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE,
        CLSID.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_GUID);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("failed setting up the attributes, retVal=" + hr.ToString());
    }

    // set the formal (symbolic name) name of the device as an attribute.
```

```

    hr = attributeContainer.SetString(
        MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_SYMBOLIC_LINK,
        symbolicLinkStr);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("failed setting up the symboic name, retVal=" + hr.ToString());
    }

    // get the media source from the symbolic name
    hr = MFExtern.MFCreateDeviceSource(attributeContainer, out videoSource);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("failed on call to MFCreateDeviceSource, retVal=" + hr.ToString());
    }
}
finally
{
    // make sure we release the attribute memory
    if (attributeContainer != null)
    {
        Marshal.ReleaseComObject(attributeContainer);
    }
}
}
Source: Source: TantaWMFUtils::GetVideoSourceFromSymbolicName

```

The purpose of the above code is to get a Media Source object (in this case a USB camera) from WMF. However don't worry too much about that particular detail just now. What we need to focus on in the above is the method in which the parameters of the required Media Source are supplied to the `MFExtern.MFCreateDeviceSource` function which subsequently finds that Media Source for us.

As can be seen above, a call to `MFExtern.MFCreateAttributes()` creates an `IMFAttributes` object.

```

// Initialize an attribute store.
hr = MFExtern.MFCreateAttributes(out attributeContainer, 2);

```

We also state at creation time that we will be using two Attributes, hence the number 2 in that call. The two Attributes needed are the media type and a string which is the name of the device. It is useful to take a bit of time and note carefully how this works. Each Attribute is a key-value pair and the key is always a GUID. The value can be anything and, in the source type Attribute, the value is itself a GUID. The code is reproduced below.

```

// setup the attribute container, it is always a VIDEO SOURCE here
hr = attributeContainer.SetGUID(
    MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE,
    CLSID.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_GUID);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("failed setting up the attributes, retVal=" + hr.ToString());
}

```

So basically what the above code is saying is that *“This Attribute specifies a video source type since the key is a `MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE` GUID and the value for this key is itself a GUID and this value is `MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_GUID`”*.

Note that the `MFFExtern.MFCreateDeviceSource` function expects this. It expects to see a key of the `MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE` GUID and it expects the value for that key to be a GUID. If this key is not present it will throw an error. If the value GUID is invalid, or cannot be found, it will also throw an error.

Let's have a look at the creation of second Attribute. The code section is reproduced below.

```
// set the formal (symbolic name) name of the device as an attribute.
hr = attributeContainer.SetString(
    MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_SYMBOLIC_LINK,
    symbolicLinkStr);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("failed setting up the symboic link, retVal=" + hr.ToString());
}
```

In this Attribute we are supplying the name of the video capture device – which is, for some reason, called a symbolic link in Windows Media Foundation. This name was obtained earlier by enumerating all the video capture devices on the system. As a key for the Attribute, we supply the standard GUID that indicates the value is a symbolic link and then we supply the name of the device itself as a string.

It is at this point that we begin to see the usefulness of Attributes as configuration items rather than as just a complicated way of passing in parameters to a function. We needed the first Attribute to indicate that it was a video capture device that was of interest. Once we have done that we specified the device name via another Attribute. If we had specified that we require an audio capture device in the first attribute (using a `MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_AUDCAP_GUID`) we would be required to use a different GUID and value in the second Attribute. In this example, in order to obtain an audio source device we could use a device endpoint id found by enumerating the audio devices on the system and the `MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_AUDCAP_ENDPOINT_ID` GUID. The Device Endpoint ID for an audio capture device is also a string – but it is not a nice readable “friendly name” as it is in the case of the video capture device.

So, we can create either a video capture device or an audio capture device simply by changing the contents of the `IMFAttributes` object we pass in as a parameter on the `MFFExtern.MFCreateDeviceSource` function call. Even if the types of data and parameters were different, the Attribute object can handle this. It is illustrative to imagine how this multiple-case example might look if the configuration data was supplied as standard parameters in a C# call. It is possible to see how this sort of requirement might be met via polymorphism – however things would start to get a bit tricky if, as in the case above, two scenarios used the same data types. In that event you

have to start specifying an Enum describing type of data for any one parameter position and then you are right back to a key-value pair type situation.

ENUMERATING ATTRIBUTES

The above scenarios discussed in detail the population of an `IMFAttributes` object for the purposes of providing configuration information to a function call. It is also possible to enumerate all of the Attributes in an object. The code below illustrates the process of examining each attribute in an object implementing the `IMFAttributes` interface in order to better understand the capabilities of that Media Type.

```
public static HRESULT EnumerateAllAttributesAsText(
    IMFAttributes attributesContainer,
    List<string> attributesToIgnore,
    int maxAttributes,
    out StringBuilder outSb)
{
    Guid guid;
    PropVariant pv = new PropVariant();

    // we always return something here
    outSb = new StringBuilder();

    // sanity check
    if (attributesContainer == null) return HRESULT.E_FAIL;

    // loop through all possible attributes
    for (int attrIndex = 0; attrIndex < maxAttributes; attrIndex++)
    {
        // get the attribute from the mediaType object
        HRESULT hr = attributesContainer.GetItemByIndex(attrIndex, out guid, pv);
        if (hr == HRESULT.E_INVALIDARG)
        {
            // we are all done, outSb should be updated
            return HRESULT.S_OK;
        }
        if (hr != HRESULT.S_OK)
        {
            // we failed
            return HRESULT.E_FAIL;
        }
        string outName = TantaWMFUtils.ConvertGuidToName(guid);
        // are we ignoring certain ones
        if ((attributesToIgnore!=null) && (attributesToIgnore.Contains(outName))) continue;
        outSb.Append(outName + ",");
    }

    return HRESULT.S_OK;
}
```

Source: `TantaWMFUtils::EnumerateAllAttributesAsText`

The `EnumerateAllAttributesAsText` static function in the `TantaWMFUtils` library takes a `IMFAttributes` object and uses the `GetItemByIndex` function in a for loop to examine each attribute in turn. Once it has the attribute, the function converts it to a nice human readable form and appends it to a string. Since the output is probably intended for display it also appends a comma between each attribute name and also has the capability to filter out certain attribute names that are not desired in the output.

Thus it is a relatively simple task to list all of the attributes contained in an object such as a Media Type.

```

public static HRESULT EnumerateAllAttributeNamesInMediaTypeAsText(
    IMFMediaType mediaTypeObj,
    bool ignoreMajorType,
    bool ignoreSubType,
    int maxAttributes,
    out StringBuilder outSb)
{
    // we always return something here
    outSb = new StringBuilder();

    // sanity check
    if (mediaTypeObj == null) return HRESULT.E_FAIL;
    if ((mediaTypeObj is IMFAttributes) == false) return HRESULT.E_FAIL;

    // set up to ignore
    List<string> attributesToIgnore = new List<string>();
    if (ignoreMajorType == true) attributesToIgnore.Add("MF_MT_MAJOR_TYPE");
    if (ignoreSubType == true) attributesToIgnore.Add("MF_MT_SUBTYPE");

    // just call the generic TantaWMFUtils Attribute Enumerator
    return TantaWMFUtils.EnumerateAllAttributesAsText(
        (mediaTypeObj as IMFAttributes),
        attributesToIgnore,
        maxAttributes,
        out outSb);
}

```

Source: **TantaMediaTypeInfo::EnumerateAllAttributeNamesInMediaTypeAsText**

The important point to realize is that the only reason the `EnumerateAllAttributeNamesInMediaTypeAsText` function works in the above code section is because a Media Type object also implements the `IMFAttributes` interface and thus the `GetItemByIndex()` call is available.

As well as providing a useful container which enables Attribute objects to be added or removed, any object implementing the `IMFAttributes` interface will enable the Attributes it contains to be enumerated and examined. This is a requirement of the `IMFAttributes` interface and is another useful feature of the Attribute mechanism.

Sometimes the object with Attributes does not implement the `IMFAttributes` interface directly and you then have to get the internal Attribute container from that object in order to access the attributes it contains. In many cases, as described above, the object itself just explicitly implements the `IMFAttributes` interface in which case you can just directly query the object.

ATTRIBUTE CODE CONVERSION FROM C++

As you look around on the Internet you will probably find much more help and sample code online for WMF in C++ and almost nothing in C#. This means you will have to translate the code. Sections of code containing Attributes are particularly prone to this requirement since they deal with such a wide variety of data types. For example, how do you translate C++ code in which a string is passed in as a pointer to a null terminated

memory location into a C# call using a standard string object? Usually this sort of thing is pretty straight forward if you know how and there is an entire section at the end of this document which describes the most common cases – please see the *Converting Between C++ and C# Code Examples* appendix for details.

HRESULTS

Remember how, in a previous section (*GUIDs*), it was mentioned that it would be awkward and problematic to use a specific set of numbers as a universal naming mechanism. Well, simply put, HRESULT's are really just a version of that. HRESULT codes are a bit of a historical legacy and are an attempt to use defined number ranges to represent error codes and return values. In MF.Net they are implemented as an enum and there are around 350 values. Of course, this is just the MF.Net HRESULT enum – other systems would have others and they could have the same numerical values as the ones in the WMF HRESULT enum. This, as discussed previously, is the downside of using a central authority to name things.

You will see HRESULT values everywhere in Windows Media Foundation programming – just about every procedure call you see will output an HRESULT as a return code. Typically in WMF, one does not see an object being used as return value as one sometimes does in other systems. For example, a Presentation Descriptor being created from a Media Session would be coded as follows...

```
HRESULT hr = mediaSource.CreatePresentationDescriptor(out sourcePresentationDescriptor);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to CreatePresentationDescriptor failed. Err=" + hr.ToString());
}
if (sourcePresentationDescriptor == null)
{
    throw new Exception("call to CreatePresentationDescriptor failed. srcPD == null");
}

Source: TantaCommon::ctlTantaEVRFilePlayer::OpenVideoFileAndPrepareSessionAndPlay
```

Note that the return value is an HRESULT and the actual object being created is returned in an `out` variable. Of course, one has to check the return code and, because of defensive programming, it is also a good idea to check the output object (`sourcePresentationDescriptor` in this case) for null as well.

A return of `HRESULT.S_OK` is the standard acknowledgement of success. `HRESULT.S_OK` equates to zero in the HRESULT enum and any other non-zero value is considered a fail. The HRESULT codes returned by any call are documented. For example, the `CreatePresentationDescriptor` call above can return the following HRESULT values...

```
S_OK           The method succeeded.
MF_E_SHUTDOWN  The media source's Shutdown method has been called
```


Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms702261\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms702261(v=vs.85).aspx)

The MF.Net library provides some tools that can be used to process HR values and render them into human readable form. For example, you will see the `GetExceptionForHR` and `ThrowExceptionForHR` calls used fairly often in the available MF.Net sample code. The Tanta Sample Projects avoid the use of these calls preferring to explicitly document the return value and throw an exception when necessary. Remember, `HResult` is just an enum so if you just want to know the enum label text for use in an error message a simple `hr.ToString()` will do the job nicely.

Note: For those of you who know a bit about the Component Object Model (COM) will probably realize that a COM function must return an `HResult` – the standard requires it. Most of the WMF objects that you will use will actually be interfaces exposed by COM objects so the return values from calls into their functions will always be `HResults`. Although the usage of an `HResult` is not mandatory in calls to internal Windows Media Foundation functions (such as the ones in the `MFExtern` library), those return values are used extensively for consistency reasons.

THE C++ VS C# CODE STRUCTURE

The Tanta Sample projects will probably be quite useful to you – however, you are going to want to dig around on the Internet and review other code samples. A lot of these will be C++ examples that you will have to translate (see the *Converting Between C++ and C# Code Examples* Chapter for assistance on this). In that, and in the MF.Net C# samples, which are a direct port, you will probably see the use of the `Succeeded()` function testing the `HResult` return value used in a staircase manner as illustrated in the example code below.

```
HResult ConfigureEncoder(
    EncodingParameters eparams,
    IMFMediaType pType,
    IMFSinkWriter pWriter,
    out int pdwStreamIndex
)
{
    HResult hr = HResult.S_OK;

    IMFMediaType pType2 = null;

    hr = MFExtern.MFCreateMediaType(out pType2);

    if (Succeeded(hr))
    {
        hr = pType2.SetGUID(MFAttributesClsid.MF_MT_MAJOR_TYPE, MFMediaType.Video);
    }
}
```

```

    if (Succeeded(hr))
    {
        hr = pType2.SetGUID(MFAttributesClsid.MF_MT_SUBTYPE, eparams.subtype);
    }

    if (Succeeded(hr))
    {
        hr = pType2.SetUINT32(MFAttributesClsid.MF_MT_AVG_BITRATE, eparams.bitrate);
    }

    if (Succeeded(hr))
    {
        hr = CopyAttribute(pType, pType2, MFAttributesClsid.MF_MT_FRAME_SIZE);
    }

    if (Succeeded(hr))
    {
        hr = CopyAttribute(pType, pType2, MFAttributesClsid.MF_MT_FRAME_RATE);
    }

    if (Succeeded(hr))
    {
        hr = CopyAttribute(pType, pType2, MFAttributesClsid.MF_MT_PIXEL_ASPECT_RATIO);
    }

    if (Succeeded(hr))
    {
        hr = CopyAttribute(pType, pType2, MFAttributesClsid.MF_MT_INTERLACE_MODE);
    }

    pdwStreamIndex = 0;
    if (Succeeded(hr))
    {
        hr = pWriter.AddStream(pType2, out pdwStreamIndex);
    }

    SafeRelease(pType2);

    return hr;
}

```

Source: `MFaptureToFile-2010::Ccapture.cs::ConfigureEncoder`

The above code, taken from the standard MF.Net samples, is a faithful C# port of the C++ Windows Media Foundation code examples. It is not wrong, per-se, but it does do things a lot more awkwardly than is necessary in C# which has constructs like `try-catch-finally` blocks. In C# you would probably code this up as something like the following ...

```

HRESULT HRESULT ConfigureEncoder(
    EncodingParameters eparams,
    IMFMediaType pType,
    IMFSinkWriter pWriter,
    out int pdwStreamIndex
)
{
    HRESULT hr = HRESULT.S_OK;

    IMFMediaType pType2 = null;

    try
    {
        hr = MFExtern.MFCreateMediaType(out pType2);
        if (hr != HRESULT.S_OK)
        {
            .. throw an exception
        }

        hr = pType2.SetGUID(MFAttributesClsid.MF_MT_MAJOR_TYPE, MFMediaType.Video);
        if (hr != HRESULT.S_OK)
        {
            .. throw an exception
        }
    }
}

```

```

    hr = pType2.SetGUID(MFAttributesClsid.MF_MT_SUBTYPE, eparams.subtype);
    if (hr != HRESULT.S_OK)
    {
        .. throw an exception
    }

    ... and so on

    pdwStreamIndex = 0;
    if (Succeeded(hr))
    {
        hr = pWriter.AddStream(pType2, out pdwStreamIndex);
    }
}
finally
{
    SafeRelease(pType2);
}
return hr;
}

```

Source: Non-working Pseudo-code Example

Obviously it is up to you how you structure your code. The point being made here is that C# offers a lot of extra functionality over C++ in regards to error handling and trapping. Therefore there is no reason why your code needs to exactly follow the C++ structuring method. In particular, note that the `pType2` object is always released in both of the previous example code blocks – error or not.

Similarly, many C++ applications are designed to pass information around the system by putting messages on the Windows Message Processor (a.k.a. the Message Pump). Various user written entities then hook that process in order to receive notice of events. This sort of thing is also sometimes seen in C# code, (for example, the MF.Net Samples), which are a direct port of the equivalent C++ code. There really is no need to do this sort of thing in a C# program and all functionality of that type can be replaced with the far simpler C# Delegate/Event mechanisms.

Windows Media Foundation: Getting Started in C#

Chapter 5

THE WMF COMPONENTS

The *Windows Media Foundation Architecture* chapter provided a whirlwind tour of the architecture and the discussion of each of the three architecture types briefly mentioned each major WMF component and its place in the system. This section will list each component and provide more detail on each one. This chapter will cover much of the same material again but this time with more depth and provide a focus on the relationships between the components.

This section will not, however, provide an intricately detailed discussion of some of the components – particularly if later sections are devoted to concepts which cover the topic in more depth. For example, the section below on Transforms (*Transforms*) is intended to only provide enough background to provide context and, hopefully, make subsequent discussions more understandable. Diving into the internal details of each object at this point would distract from the main flow and would probably not be too useful from a learning perspective.

One of the major difficulties in a discussion of this sort is that some topics invariably require the references to concepts not yet covered. The sections below have (hopefully) been organized in such a way which minimizes this effect. However, you may occasionally see some references to concepts which have not yet been discussed - or

which have been discussed and you have forgotten about in the onslaught of all this information. So be prepared for a bit of that and also for some duplicate coverage as attempts are made to provide context.

It is highly advisable to read (or re-read) the *Windows Media Foundation Architecture* chapter before starting in on this chapter as that information will provide a broad overview of the WMF system and assist your understanding of how all the various parts fit together.

FUNDAMENTAL PROCESSING OBJECTS

There are three fundamental types of data processing entity in Windows Media Foundation: Media Sources, Media Sinks and Media Transforms. These are the objects that originate the media data, consume the media data or modify the media data. All of the other entities used in WMF could be broadly classified as supporting infrastructure which assists with the transport or control of the data as it moves through the data processing entities. You will note that we are ignoring the Source Reader and Sink Writer objects here as they contain their own internal Media Source and Media Sink respectively.

MEDIA SOURCES

A Media Source originates media data and it has no inputs from other Windows Media Foundation entities - although it can obtain raw data from various devices. WMF components that present microphone and a video camera data to the system are examples of streaming Media Sources and the input data to both is provided by the Windows Device Drivers. A Media Source is, in general, specific to the type of input and it knows how to interact with its device. An example of a non-streaming Media Source is a WMF entity which reads a file on a disk - in that case the input data is provided by the operating system.

For any one Media Source there is typically only one input of raw data. Thus, if your application is recording both sound and video, then you will have two Media Sources in your application. A file being read from disk may well contain both sound and video. This is still only one input, however, and it is the job of the Media Source to split these two media types into separate streams. It really does not matter to WMF if the raw data

originated from two independent Media Sources (microphone and camera) or if the two streams were split from one Media Source (the file on disk).

You will also see references to a WMF entity named the Source Reader. The Source Reader object is not a true Media Source – although it does contain a Media Source and functions similar to one. The purpose of the Source Reader entity is to encapsulate a lot of common Media Source functionality and present it as one easy to use object.

MEDIA SINKS

Similarly, Media Sinks consume data and produce no output for other WMF objects. In general, there are two types of Media Sink – the renderers which present the data to the user and the recorders (or archivers) which store media data. The Enhanced Video Renderer (discussed in the *An Overview of the EVR* section of the *Rendering Audio and Video* chapter) is an example of the former while the MPEG-4 file sink supplied as part of WMF is an example of the latter.

Media Sinks can accept multiple streams and it is their job to combine these streams in an appropriate way. In the case of an archive sink, it means ensuring that both streams are correctly encoded into the same file in the specified format.

Similar to the earlier discussion of Media Sources, you will also see references to a Windows Media Foundation entity named the Sink Writer which provides a wrapping mechanism for Media Sinks. The Sink Writer object is also not a true Media Sink – although it does contain a Media Sink and can be used in a similar way

AN INTERFACE DIGRESSION

Let's take a bit of time to review our previous discussion of interfaces (see the *Most WMF Objects are Interfaces* section in the *MF.Net Programming Fundamentals* chapter). You will need a solid knowledge of interfaces in order to properly understand the material in the following sections so it is worth repeating.

In order to ensure applications interact consistently with objects of a specific type, Windows Media Foundation has formalized the actions of each object it implements. This formalized behavior manifests itself as a standardized and documented set of function calls, parameters and return values. In general, the possible actions of every WMF object are highly constrained and specified in considerable detail.

The WMF Components

In Windows Media Foundation a set of rigidly defined behaviors an object must implement is called an Interface. In MF.Net the WMF interfaces are physically implemented as C# Interfaces because their intent and behavior is identical. We will use Media Sources and Media Sinks as an illustrative example in this discussion because we want to mention Interfaces in subsequent sections. It is important to realize that each and every WMF object implements one or more types of C# Interface and these translate exactly into the Windows Media Foundation Interfaces you will see defined in the online help files.

This implies that there is a specifically defined interface for both the Media Source and Media Sink – and, indeed, there is: `IMFMediaSource` and `IMFMediaSink`. Any object which implements the `IMFMediaSink` interface is a Media Sink and, similarly, any object which implements the `IMFMediaSource` interface is a Media Source. As an example, if you write your own Media Sink you will have to ensure that the object fully supports the `IMFMediaSink` interface. If it does, it can be used anywhere in the Pipeline that a Media Sink can be used. All the Media Session is looking for in a sink object is that it implements the `IMFMediaSink` interface. It really does not care about anything else that object may be able to do. Please be aware that writing your own Media Sink (or Media Source) is an advanced topic and it will not be discussed in this book – so don't bother digging around looking for it.

There are Interfaces for every conceivable collection of actions and some (most) Windows Media Foundation objects implement multiple interfaces. For example, the Enhanced Video Renderer (EVR) which displays video on the screen is an `IMFMediaSink`. It is also (among other things) an `IEVRTrustedVideoPlugin`, an `IMFDesiredSample`, an `IMFVideoDisplayControl`, an `IMFVideoMixerBitmap`, an `IMFVideoMixerControl`, an `IMFVideoPresenter`, an `IMFVideoProcessor`, an `IMFVideoRenderer` and an `IMFVideoSampleAllocator`. All-in-all, the EVR implements well over sixteen additional interfaces and probably numerous others which are not documented.

By now you have probably figured out that all Windows Media Foundation Interfaces start with the initials “IMF”. This is true, but you will sometimes see other interfaces in use such as `IActivate`. Interfaces with names like that will be COM interfaces (not WMF) but you will see them used here and there. As a side note, the interface name is a great thing to search on if you need more information on a topic since it does not collide with anything non-WMF in most search engines.

It should be noted that neither the Source Reader nor the Sink Writer objects implement the `IMFMediaSink` and `IMFMediaSource` interfaces. The Source Reader implements the `IMFSourceReader` interface and the Sink Writer implements an

`IMFSinkWriter` interface. This, ultimately, is why you cannot add a Source Reader or Sink Writer to the Pipeline. The Media Session requires an `IMFMediaSource` or an `IMFMediaSink` (or an `IMFTransform`) and nothing else will do.

CREATING WMF COMPONENTS

We are not going to cover the topic of creating Windows Media Foundation Objects in detail here since there is an entire section of this book devoted to the topic of creating WMF objects (see the *WMF Object Creation is Indirect* section of the *MF.Net Programming Fundamentals* chapter). However, having said that, it is important for your understanding of the material that follows that you have a basic understanding of the way WMF goes about creating the objects it needs.

Remember earlier in this book how it was said that pretty much every WMF object is also a COM object. Well, one does not simply create COM objects as you would a normal object. In other words, if you need an Enhanced Video Renderer object you just don't write a line of code like ...

```
EVRRenderer myEVR = new EVRRenderer();
```

This concept simply does not exist in COM (and hence in WMF). For one thing, you do **not** know the class name of the EVR object (the `EVRRenderer()` class name was just made up in the previous sentence for illustration purposes). If you don't, and never, know the name of the object you want to create then how can you create it? You cannot use the Interface name - writing something like `new IMFMediaSink()` is meaningless and, even if it did work, could get you any one of a dozen different Media Sinks.

There are several ways you can go about creating a particular WMF object.

1. You can use a special unique key (called a GUID) and request the COM system find and create the correct object for you.
2. You can acquire something called an Activator (which is itself a COM object) which can then be used to find and create the correct object for you.
3. You can, in some cases, use something called a Resolver (which works a lot like an Activator) to create the object for you. This appears to be the WMF equivalent of a COM Activator.
4. You can call a static function which creates the correct object for you.

Call me a cynic (and people do) but probably the static function just uses one of the first two previous methods and is really only intended to leave you blissfully ignorant of the gory details.

So let us look at some specific examples of how a Media Source can be created.

CREATING A MEDIA SOURCE FROM A DEVICE

The code section below demonstrates the process of creating a Media Source from the Symbolic Name of a device (a string which is a kind of URL or Path for devices, see the *WMF – First Contact* chapter for more details). Ultimately this process is a variant on the Static Function call method – however the parameters are passed in via an Attribute container.

```
/// ++++++
/// <summary>
/// Returns a media source from the contents of a TantaMFDevice
/// </summary>
/// <param name="sourceDevice">the source device</param>
/// <returns>a IMFMediaSource or null for fail</returns>
/// <history>
///     01 Nov 18   Cynic - Started
/// </history>
public static IMFMediaSource GetMediaSourceFromTantaDevice(TantaMFDevice sourceDevice)
{
    IMFMediaSource videoSource = null;
    HRESULT hr = 0;
    IMFAttributes attributeContainer = null;

    try
    {
        if (sourceDevice == null)
        {
            // we failed
            throw new Exception("sourceDevice == null");
        }
        if ((sourceDevice.SymbolicName == null) || (sourceDevice.SymbolicName.Length == 0))
        {
            // we failed
            throw new Exception("failed null or bad symbolicLinkStr");
        }
        if (sourceDevice.DeviceType == Guid.Empty)
        {
            // we failed
            throw new Exception("GetMediaSourceFromTantaDevice DeviceType == Guid.Empty");
        }

        // Initialize an attribute store. We will use this to
        // specify the enumeration parameters.
        hr = MFExtern.MFCreateAttributes(out attributeContainer, 2);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("failed MFCreateAttributes, retVal=" + hr.ToString());
        }
        if (attributeContainer == null)
        {
            // we failed
            throw new Exception("failed attributeContainer == null");
        }

        // setup the attribute container, it is always a
        // MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE here
        hr = attributeContainer.SetGUID(
            MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE,
            sourceDevice.DeviceType);
        if (hr != HRESULT.S_OK)
        {

```

```

        // we failed
        throw new Exception("failed setting up the attributes, retVal=" + hr.ToString());
    }

    // set the formal (symbolic name) name of the device as an attribute.
    hr = attributeContainer.SetString(
        MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_SYMBOLIC_LINK,
        sourceDevice.SymbolicName);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("failed symbolic name, retVal=" + hr.ToString());
    }

    // get the media source from the symbolic name
    hr = MFExtern.MFCreateDeviceSource(attributeContainer, out videoSource);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("failed MFCreateDeviceSource, retVal=" + hr.ToString());
    }
}
finally
{
    // make sure we release the attribute memory
    if (attributeContainer != null)
    {
        Marshal.ReleaseComObject(attributeContainer);
    }
}
return videoSource;
}

```

Source: TantaCommon::TantaWMFUtils::GetMediaSourceFromTantaDevice

The code sets up an Attribute container and populates it with two attributes – the first of which describes the type of device. This is the `MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE` and `sourceDevice.DeviceType` key value pair. The second attribute is the Symbolic Name of the device and it uses a `MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_SYMBOLIC_LINK` key. Once the Attribute container is populated, the static function `MFExtern.MFCreateDeviceSource` is called to create the Media Source. If all of the above code just seems like a long winded replacement for what really could be a very simple static function call like...

```

hr = MFExtern.MFCreateDeviceSource(sourceDevice.DeviceType,
                                   sourceDevice.SymbolicName,
                                   out deviceSource);

```

... well you probably have a point. However, you had better just get used to technique – you will be seeing it a lot. WMF does things the COM way and in this, as with so many things in life, nobody much cares what you or I might think.

CREATING A MEDIA SOURCE FROM A SOURCE RESOLVER

The code section below shows a Source Resolver being used to create a Media Source from a file.

```

// As with so many things WMF, the creation of the media source is indirect
// We now create a source resolver which will be used to create a media source
// from a URL, filename or byte stream. The call below returns an
// IMFSourceResolver interface pointer.
hr = MFExtern.MFCreateSourceResolver(out pSourceResolver);
if (hr != HRESULT.S_OK)
{

```

The WMF Components

```
        throw new Exception("MFExtern.MFCreateSourceResolver failed. Err=" + hr.ToString());
    }
    if (pSourceResolver == null)
    {
        throw new Exception("pSourceResolver == null");
    }

    // here we use our source resolver to create the media source
    MFObjectType objectType = MFObjectType.Invalid;
    hr = pSourceResolver.CreateObjectFromURL(
        mediaFileName,           // URL (file path and name) of the source.
        MFResolution.MediaSource, // Create a source object.
        null,                   // Optional property store.
        out objectType,         // Receives the created object type.
        out pSource             // Receives a pointer to the media source.
    );
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to CreateObjectFromURL failed. Err=" + hr.ToString());
    }
    if (pSource == null)
    {
        throw new Exception("CreateObjectFromURL failed. pSource == null");
    }
    // Cast the output into our media source object
    mediaSource = (IMFMediaSource)pSource;

    // make sure we clean up
    if (pSourceResolver != null)
    {
        Marshal.ReleaseComObject(pSourceResolver);
    }
}

Source: TantaCommon::TantaWMFUtils::GetMediaSourceFromFile
```

The above code uses a static function call to create the Source Resolver and the Source Resolver is used (along with the file name) to create the Media Source.

CREATING A MEDIA SINK FROM AN ACTIVATOR

How about a Media Sink? Here is a code section that uses a static function call to get an Activator. The Activator is later handed off to a WMF Topology Node which will eventually create the Media Sink (in this case an EVR Video Renderer) when the Pipeline is created.

```
// Create an activation object for the enhanced video renderer (EVR) media sink.
hr = MFExtern.MFCreateVideoRendererActivate(videoWindowHandle, out pRendererActivate);
if (hr != HRESULT.S_OK)
{
    throw new Exception("MFExtern.MFCreateVideoRendererActivate failed. Err=" + hr.ToString());
}
if (pRendererActivate == null)
{
    throw new Exception("pRendererActivate == null");
}

// Set the IActivate object on the output node. Note that not all node types use
// this object. On transform nodes this is IMFTransform or IMFActivate interface
// and on output nodes it is a IMFStreamSink or IMFActivate interface. Not used
// on source or tee nodes.
hr = outputNode.SetObject(pRendererActivate);

Source: TantaCommon::TantaWMFUtils::CreateRendererOutputNodeForStream
```

Don't focus too much for the moment on what the `outputNode.SetObject()` call does. Just realize that once it has the Activator it can use that object to create the EVR File Renderer when it needs to do so.

Hopefully the above two code blocks, which create the Media Source and Media Sink from a Source Resolver or Activator, are a lot more understandable to you now - even if you are still a bit dubious about why it is necessary to do it that way. It should be noted that there are other methods of creating these two objects (indeed the Source Resolver method is only appropriate for file based Media Sources). However, you can be sure that no matter which creation process you use, you are going to be calling some other object that does the creation for you. In other words, don't be surprised when you see this sort of multi-step creation process going on. It is just the way it is.

CREATING A MEDIA SINK ON A FILE

As will be discussed in *The Standardization of Pipeline Components* section of the *Practical WMF Architectures* chapter, the creation process for Media Sinks is usually specific to the type of Media Sink. The creation process for Media Sinks does not seem to be consistent like it is for Media Sources. Below is a section of code which demonstrates how to create an MP3 file sink.

```
private IMFMediaSink OpenMediaFileSink(string outputFileName)
{
    HRESULT hr;
    IMFMediaSink workingSink = null;
    IMFByteStream outbyteStream = null;

    if ((outputFileName == null) || (outputFileName.Length == 0))
    {
        // we failed
        throw new Exception("OpenMediaFileSink: Invalid filename specified");
    }

    try
    {
        // Create the media sink. We use the filename to create a byte stream and
        // then create the sink from that. The types configure the output

        // first we need a bytestream
        hr = MFExtern.MFCreateFile(MFFileAccessMode.ReadWrite,
            MFFileOpenMode.DeleteIfExists,
            MFFileFlags.None, outputFileName,
            out outbyteStream);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("Failed on call to MFCreateFile, retVal=" + hr.ToString());
        }
        if (outbyteStream == null)
        {
            // we failed
            throw new Exception("Failed to create Sink bytestream, Nothing will work.");
        }

        // note the MP3 File sink does not seem to require the media type as part of its
        // configuration. The reasons for this are unknown. However it may be
        // due to the fact that the MP3 file spec is pretty fixed in structure and
        // as long as the input stream actually _is_ MP3 the media sink can write
        // it to the output file.
        hr = MFExtern.MFCreateMP3MediaSink(outbyteStream, out workingSink);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("Failed MFCreateMPEG3MediaSink, retVal=" + hr.ToString());
        }
        if (workingSink == null)
        {
            // we failed
        }
    }
}
```

The WMF Components

```
        throw new Exception("OpenMediaFileSink: Failed to create media sink");
    }
}
catch (Exception ex)
{
    // note this clean up is in the Catch block not the finally block.
    // if there are no errors we return it to the caller. The caller
    // is expected to clean up after itself
    if (workingSink != null)
    {
        // clean up. Nothing else has this yet
        Marshal.ReleaseComObject(workingSink);
        workingSink = null;
    }
    workingSink = null;
    throw ex;
}

return workingSink;
}
```

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::OpenMediaFileSink

There are several things to note in the above code. The first, and most obscure, is that the `MFCreatemp3MediaSink` static function call wants a byte stream not a file name. For some reason, there is no override which takes a filename. No problem, the code above creates a byte stream from the filename and hands that to the `MFCreatemp3MediaSink` static function and the MP3 file sink is created. The other item of particular interest is that if there are errors the Media Sink is released in the `catch` block of the function. Under normal conditions it is passed back for the caller to release at some other appropriate time.

The creation process for an MP4 file sink is slightly different and will be discussed in the *Creating an MP4 File Sink* section of the *Practical WMF Architectures* chapter.

THE PIPELINE

If you have been doing a bit of digging in your quest to understand Windows Media

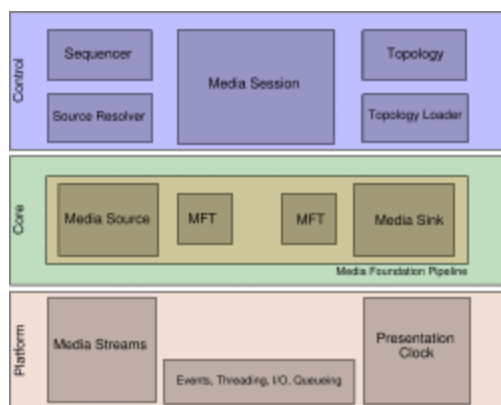


Figure 5.1: A not very helpful diagram of the Pipeline.
Source: Wikipedia

Foundation you will doubtless have seen a diagram similar to the one below in Figure 5.1. Variations of this diagram are ubiquitous and so, if we are going to be considered a proper bit of WMF literature, we better include a version of it.

Very probably, after looking at the diagram in Figure 5.1, you are probably not all that much more enlightened than you were previously. The problem is that graphical representations of the Pipeline do not really seem to provide a

good sense of what is happening. So let's change things up a bit and make a sequential list of the things going on in the Pipeline.

A SIMPLE PIPELINE

Below is a simple, linearly ordered, sequence of the events Windows Media Foundation might follow in order to process a packet of media data known as a Media Sample (an object implementing the `IMFMediaSample` interface). This represents the Pipeline. Let's leave aside, for the moment, how the whole thing is originally set up and focus on the flow of information through the system.

1. The Media Source acquires the next Media Sample of data from the device it is reading.
2. The Media Session notices the Media Source has data ready.
3. The Media Session notices that the Media Sink can accept data.
4. The Media Session requests the data from the Media Source.
5. The Media Session hands the data to the Media Sink.
6. The Media Sink writes the data to the device (video display or file etc.).
7. GOTO step 2

The Media Source and Media Sink each operate independently. They have their own internal threads and will continually source and sink data as fast as they can. The Media Session does not wait for the Media Source to fetch the data and it does not wait around while the Media Sink writes out the data. It just hands off the Media Sample and goes on about its business. This is why Step 7 refers back to Step 2 and not Step 1 – by the time Step 6 is finished, the Media Source has already completed Step 1 in parallel.

It is up to the Media Session to pick the data up from the source when it is available and give it to the sink when it can accept it.

This implies that the Media Session controls the throttle on the data flow and that opens up all kinds of interesting possibilities. The Media Session can, if it wishes, stop the flow of information simply by not reading any more data from the Media Source. In effect, we have an easy to implement Pause function. The Media Source may well have more data to give and the Media Sink may well have the ability to write, but unless the Media Session moves it along, the data does not go anywhere. For similar-*ish* reasons, most of this sort of “*flow control*” functionality such as rewind, fast forward and skipping is the purview of the Media Session. More accurately, (in WMF speak), it can be

said that the `Media Session` object implements both the `IMFMediaSession` and `IMFRateControl` interfaces and the application can interact with these interfaces to implement various types of flow control. In fact, you can see a great deal of this sort of thing going on in the *TantaFilePlaybackAdvanced* Sample Project. An extensive discussion of the techniques involved can also be found in the *Playback Control* section of the *Rendering Audio and Video* chapter.

It should be noted that it is sometimes said that the Pipeline operates on a “Pull Model” in which the Media Sink pulls the data through the Pipeline. This is sort of true, however, mostly it is really only due to the fact that the Media Sink is usually the slowest component in the Pipeline. The Media Session is usually spinning around waiting for the Media Sink to tell it that it is ready for more data and that is where the “Pulling” effect is coming from.

A PIPELINE WITH TWO BRANCHES

Let’s extend the previous concepts to see how multiple media streams might be handled by a Media Session. Take, for example, an application that plays an MP4 file. The file contains both video and audio data. Each type of media data in the file is going to have to be processed and the items of each data type will appear on the Media Source in their own distinct sequences known as Media Streams. In order to provide the viewer with a good experience, both of the Media Streams must be rendered simultaneously. Since the Enhanced Video Renderer (EVR) video sink is distinct from the Streaming Audio Renderer (SAR) audio sink this implies that there is going to have to be two branches in the Pipeline.

Now that we see the need for multiple Media Streams, let us recognize that multiple Media Streams cause the Media Session two additional problems. Firstly, the media data must not be displayed as fast as the Media Source can provide it otherwise the images on the screen might just be a fast forward blur. Secondly, the play of the audio data must be correlated in time with the play of the video data – otherwise the sounds will not match the images on the screen. These two issues mean that the data must be both throttled and synchronized. We have seen earlier how the throttling might be done – the Media Session just stops requesting data from the single stream on the Media Source. In this case, there are two streams and the Media Session can just stop requesting data from either one as is appropriate. The Windows Media Foundation object inside the Media Source that keeps track of the “current place” in the streams is called the Presentation Clock. The Media Session will interact with the Media Source and use the Presentation Clock to tune its request rate on the Media Streams. Here is how it works...

1. The Media Source video stream acquires the next Media Sample of video data from the device it is reading.
2. The Media Source audio stream acquires the next Media Sample of audio data from the device it is reading.
3. The Media Session notices that either of Media Streams on the Media Source have data ready.
4. The Media Session notices that either of the Media Sinks can accept data.
5. The Media Session checks the time in the Presentation Clock to see if it can process any of the media streams. Each Media Sample taken from the file has a timestamp encoded in it and this timestamp corresponds to the time at which the sample should be rendered.
6. If a Media Stream can be processed and the corresponding Media Sink can accept data, then the Media Session requests the data from the stream on Media Source.
7. The Media Session hands the data to the appropriate Media Sink.
8. The Media Sink writes the data to the device (video display or speaker etc.).
9. GOTO step 3

In reality it is all very logical. Notice that the Media Session is controlling the two streams and is using the Presentation Clock within the Media Source to synchronize the timing of this.

It should be noted (before you start getting worried) that you, as the programmer, do not have to code up any of the above logic when using a Media Session. The whole sequence of events is automatic and entirely controlled by the Media Session. True, you have to create the Media Session, the Media Source, the Media Sink and set up the streams – but once you wire everything up and set it rolling, the process is automatic. You, as the programmer, do not interact with any of the Pipeline components. See the *TantaAudioFileCopyViaPipelineMP3Sink* sample code for a simple example of single stream MP3 version of this process. The *TantaVideoFileCopyViaPipelineMP4Sink* example project performs a similar function on a two stream MP4 file.

A careful examination of the above process should reveal that if there were two distinct Media Sources and only one Media Sink then the Media Session mechanism and readily cope with only minor variations in operation. For example, such a scenario could occur a

The WMF Components

when a video camera and microphone are recording to an MP4 file. Each independent Media Source would have one stream and the single Media Sink would have two streams. Since a file is the target, the Media Session would probably not need to throttle anything and would send the data to the Media Sink as soon as the Media Sink could accept it.

Other extensions are possible. Let's enhance the first simple Pipeline example to include an object in the Pipeline between the Media Source and Media Sink. Such objects are known as Media Transforms (they implement the `IMFTTransform` interface). Transforms have one or more input streams and one or more output streams. Transforms are designed to perform manipulations on the data and they will be discussed in detail in the *Working With Transforms* chapter. For now, just let's just assume that the Transform we are discussing does something benign like counting the Media Sample as it arrives and then passes that input sample on to the output unchanged.

1. The Media Source acquires the next Media Sample of data from the device it is reading.
2. The Media Session notices the Media Source has data ready.
3. The Media Session notices that the Transform can accept data.
4. The Media Session requests the data from the Media Source.
5. The Media Session hands the data to the Transform.
6. The Media Session notices that the Media Sink can accept data.
7. The Media Session notices that the Transform has data ready.
8. The Media Session requests the data from the Transform.
9. The Media Session hands the data to the Media Sink.
10. The Media Sink writes the data to the device (video display or file etc.).
11. GOTO step 2

The Media Transform is just another link in the chain and it too is operating in an independent thread. This means a lot of the steps in the above list are happening simultaneously. The Media Session can be fetching more data at the same time as the Media Transform is processing it.

You, as the author of the application, can write your own Transforms. In fact, there are two examples in the Tanta Samples (`TantaTransformDirect` and `TantaTransformInDLL`) that demonstrate how to do this and an entire chapter in this book (*Working With Transforms*) discussing the process.

In order to make things simple, so that you don't have to do thread programming inside your Transform, the Media Session maintains a queue of worker threads. The Transform processes the data inside of this worker thread and all you have to do is supply the code.

It should also be noted that the input format of the stream into the Transform does not have to be the same as the format on the output stream. In fact, such format conversions are the entire point of a large class of Media Transforms. Similarly, the amount of data entering the Transform does not have to be the same as the amount leaving. Compression and decompression Transforms (codecs) will always behave like this. Lastly, it is quite possible to emit copies of the data and split the branch inside the Transform. Thus the Media Session, if you set it up correctly, can have one stream entering and two or more streams leaving. All of those streams must terminate on a Media Sink though.

It should be noted that the Transform in the above process is operating in Synchronous Mode. Asynchronous Mode Transforms are multi-threaded internally and their interaction with the Media Session is somewhat more complicated. Ultimately though, you don't have to worry about it. If you ever write an Asynchronous Mode transform (and you will probably never have to) you simply add it to the Pipeline like you would any other Transform and the Media Session figures out what to do with it.

READER-WRITER DATA PROCESSING

The Source Reader and Sink Writer architecture may not include a Media Session but it does contain a data transfer mechanism which could be described as a Pipeline of sorts. In this case you, the programmer, handle the flow and transfer of the Media Samples. Here, once again, is a list detailing how a Reader-Writer Architecture might work.

1. The Source Reader is configured.
2. The Sink Writer is configured
3. The application gets the next Media Sample from the Source Reader
4. The application gives the Media Sample to the Sink Writer
5. GOTO step 3

Simple isn't it - the application is the Pipeline. In this example, there is a lot of blocking going on. Unlike with the Media Session, the data transfer process will wait in for the Source Reader to provide the data and then it will wait until the Sink Writer finishes writing it. You can see a working example of a Synchronous Mode Reader-Writer

Architecture (which is what is described above) in the *TantaVideoFileCopyViaReaderWriter* sample application and the process is detailed in the *Implementing the Reader-Writer Architecture* section of the *Practical WMF Architectures* chapter. The logical steps involved in the Asynchronous Mode Reader-Writer Architecture are pretty much the same and so they will not be discussed here. The major difference would be that only the read of first Media Sample would happen in the application. Thereafter, Steps 3 and 4 would happen in a designated Callback Object which would process the data in a similar fashion. This has the advantage that the application is free to do other things while the information is being processed since the Callback Object operates in its own thread.

THE HYBRID ARCHITECTURE DATA PROCESSING

For completeness, we had better discuss the structure of a Hybrid Architecture – although by now you can probably figure out how it is going to work. Listed below is what the flow of information through simple Pipeline which implements a Sample Grabber Sink and a Sink Writer might look like.

1. The Media Source acquires the next Media Sample from the device it is reading.
2. The Media Session notices the Media Source has data ready.
3. The Media Session notices that the Sample Grabber Sink can accept data.
4. The Media Session requests the data from the Media Source.
5. The Media Session hands the data to the Sample Grabber Sink.
6. The Sample Grabber Sink gives a copy of the data to a Sink Writer object which writes it out to disk.
7. The Sample Grabber Sink discards the original data.
8. GOTO step 2

This process is discussed in the *Implementing a Hybrid Architecture* section of the *Practical WMF Architectures* chapter and you can see a working example in the *TantaAudioFileCopyViaPipelineAndWriter* and *TantaVideoFileCopyViaPipelineAndWriter* Sample Projects. The *TantaCaptureToScreenAndFile* sample application also has an interesting variation on this architecture in which the Media Sink is actually an Enhanced Video Renderer and a custom written Transform copies the Media Samples on their way through the Pipeline and hands them off to a Sink Writer.

PIPELINE ERRORS AND EVENTS

In the following discussion we will reference only the Pipeline Architecture in order to avoid complicating the information with multiple strands of ideas. The Reader-Writer Architecture has a pretty simple error and event mechanism and that will be discussed in the *Implementing the Reader-Writer Architecture* section. The Hybrid Architecture error and event handling is pretty much identical to the one discussed below in the Pipeline Architecture since any errors in the Sink Writer just propagate back via the normal Media Session channels.

As we have seen, the Media Session controls the flow of information and the entire Pipeline rolls along until the Media Sources run out of data or there is an error or the application signals a stop.

Any events or errors originating from any Pipeline object will be passed to your application via the Media Session. This provides a sort of “*single point of contact*” between your application and a wide variety of WMF objects.

Since they both interact with devices (which can be problematic) Media Sources and Media Sinks have quite a comprehensive internal mechanism to send and receive events or errors. This does not matter, the Media Session handles any such events or errors. In a similar way, any event or error originating inside of a Transform will be presented to your application via the Media Session. In practical terms, for user written Transforms in MF.Net, this means that all you have to do is throw a standard C# exception if something that you do not like happens inside the Transform. You will see this pattern in use everywhere in the error checking areas of the various Transform based Tanta Samples (see the *TantaTransformDirect* project). By way of illustration the code block below is from the output Media Sample processing function in the demonstrator Grayscale Conversion Transform.

```
// Get the data buffer from the input sample. If the sample contains more than one buffer,
// this method copies the data from the original buffers into a new buffer, and replaces
// the original buffer list with the new buffer. The new buffer is returned in the
// inputMediaBuffer parameter. If the sample contains a single buffer, this method
// returns a pointer to the original buffer.
// In typical use, most samples do not contain multiple buffers.
hr = InputSample.ConvertToContiguousBuffer(out inputMediaBuffer);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to ConvertToContiguousBuffer failed. Err=" + hr.ToString());
}

Source: TantaTransformDirect::MFTTantaGrayscale_Sync::OnProcessOutput
```

As mentioned previously, any events or exceptions will eventually be presented to the application by the Media Session – but how does this information get passed from the

Media Session to the application? Well, when you create a Media Session you also give it a Callback Object. We have not discussed Callback Objects in detail yet - for now, just realize that a Media Session Callback Object is a user written class implementing the `IMFAsyncCallback` interface that the Media Session can call when it needs to notify anything about an event or error. The Callback Object can be written to communicate with the application. In essence, the Callback Object is the interface between the Media Session and the application and you will use them extensively. Let's take a look at the Media Session creation process and then we will look at the operation of the Callback Objects in a bit more detail.

CREATING THE MEDIA SESSION

```
// reset everything
CloseAllMediaDevices();

// Create the media session.
hr = MFExtern.MFCreateMediaSession(null, out mediaSession);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to MFExtern.MFCreateMediaSession failed. Err=" + hr.ToString());
}
if (mediaSession == null)
{
    throw new Exception("call to MFExtern.MFCreateMediaSession failed. mediaSession == null");
}

// set up our media session Callback Object.
mediaSessionAsyncCallbackHandler = new TantaAsyncCallbackHandler();
mediaSessionAsyncCallbackHandler.Initialize();
mediaSessionAsyncCallbackHandler.MediaSession = mediaSession;
mediaSessionAsyncCallbackHandler.MediaSessionAsyncCallbackError =
    HandleMediaSessionAsyncCallbackErrors;
mediaSessionAsyncCallbackHandler.MediaSessionAsyncCallbackEvent =
    HandleMediaSessionAsyncCallbackEvent;

// Register the Callback Object with the session and tell it that events can
// start. This does not actually trigger an event it just lets the media session
// know that it can now send them if it wishes to do so.
hr = mediaSession.BeginGetEvent(mediaSessionAsyncCallbackHandler, null);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to mediaSession.BeginGetEvent failed. Err=" + hr.ToString());
}

Source: TantaVideoFileCopyViaPipelineMP4Sink::frmMain::PrepareSessionAndTopology
```

We will return to our discussion of the Callback Object shortly – but for now let's indulge in a bit of digression and discuss the creation of a Media Session. The Callback Object is an intimate part of that creation process.

The Media Session can support Pipelines with both Protected Media Path (PMP) content and also content of the regular (unprotected) kind. Since, as mentioned previously, we are not going to concern ourselves in this book with PMP content, there is only really one way to create a Media Session and that is by calling the static `MFCreateMediaSession()` function. This function just goes off, presumably interacts with COM, and delivers a working Media Session object to you. Your application should remember the Media Session object (perhaps as a class variable) as it will need it later

when time comes to shut things down. That's basically it, creating the Media Session is trivial. It is the configuration of the Pipeline that is the tricky bit.

THE MEDIA SESSION CALLBACK OBJECT

Windows Media Foundation components use Callback Objects pretty much any time they need to notify another component or object about events or errors which occur. In this example, we are talking about the Callback Object associated with the Media Session which is an object which implements the `IMFAsyncCallback` interface. Be aware that not all Callback Objects are the same. Especially, do not confuse this Callback Object with the Callback Object used by the Source Reader component in Asynchronous Mode (which is an `IMFSourceReaderCallback`). The purpose may be somewhat similar but in reality they are quite different implementations.

Note that the Callback Object the Media Session expects to use is an interface (`IMFAsyncCallback`) not an object. This means your C# program has to define an object which implements the `IMFAsyncCallback` interface. In the Tanta Sample Code, this object has been made generic and is present in the *TantaCommon* library as the `TantaAsyncCallbackHandler` class. This makes it usable by any application that references the library. It should be noted that the `TantaAsyncCallbackHandler` also inherits from the OIS base classes so you can use statements like `LogMessage()` and `DebugMessage()` within it to record things to the log file.

To be succinct, the major function in the Callback Object is the `Invoke()` call. This function gets called when there is an event of interest - this can be either an informational status event or an error.

```
try
{
    ... more code

    // Get the event type. The event type indicates what happened to trigger the event.
    // It also defines the meaning of the event value.
    hr = eventObj.GetType(out meType);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to IMFMediaEvent.GetType failed. Err=" + hr.ToString());
    }

    // Get the event status. If the operation that generated the event was successful,
    // the value is a success code. A failure code means that an error condition
    // triggered the event.
    hr = eventObj.GetStatus(out hrStatus);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to IMFMediaEvent.GetStatus failed. Err=" + hr.ToString());
    }

    // Check if we are being told that the the async event succeeded.
    if (hrStatus != HRESULT.S_OK)
    {
        // The async operation failed. Notify the application
        if (MediaSessionAsyncCallBackError != null)
        {
            MediaSessionAsyncCallBackError(this, "Error Code =" + hrStatus.ToString(), null);
        }
    }
}
```

The WMF Components

```
    }
    else
    {
        // we are being told the operation succeeded and therefore the
        // event contents are meaningful. Switch on the event type.
        switch (meType)
        {
            // we let the app handle all of these. There is not really much we can do here
            default:
                MediaSessionAsyncCallbackEvent(this, eventObj, meType);
                break;
        }
    }
}
catch (Exception ex)
{
    // The async operation failed. Notify the application
    if (MediaSessionAsyncCallbackError != null)
    {
        MediaSessionAsyncCallbackError(this, ex.Message, ex);
    }
}
}

Source: TantaCommon::TantaAsyncCallbackHandler::MFAsyncCallback::Invoke
```

Reading down through the code we see that for any particular event, that the Media Session cares to notify the application about, there are two possibilities. Either the `hrStatus` parameter of the `eventObj.GetStatus()` call contains `HResult.S_OK` - in which case the event is a normal status change event. If the `hrStatus` parameter contains some other `HResult` code, then the event is a notification of an error.

Depending on type of event we are dealing with, the Callback Object immediately hands off the call to one of two C# delegate/events. If the Callback Event is a normal status change then `MediaSessionAsyncCallbackEvent` gets called and if the Callback Event is an error then `MediaSessionAsyncCallbackError` gets called. These two C# events are populated shortly after the Media Session is created. This is what the following lines of code in the Media Session creation code section were doing.

```
// set up our media session Callback Object.
mediaSessionAsyncCallbackHandler = new TantaAsyncCallbackHandler();
mediaSessionAsyncCallbackHandler.Initialize();
mediaSessionAsyncCallbackHandler.MediaSession = mediaSession;
mediaSessionAsyncCallbackHandler.MediaSessionAsyncCallbackError =
    HandleMediaSessionAsyncCallbackErrors;
mediaSessionAsyncCallbackHandler.MediaSessionAsyncCallbackEvent =
    HandleMediaSessionAsyncCallbackEvent;

Source: TantaVideoFileCopyViaPipelineMP4Sink::frmMain::PrepareSessionAndTopology
```

The `HandleMediaSessionAsyncCallbackEvent` and `HandleMediaSessionAsyncCallbackErrors` are functions in the application. Thus, these two application functions will get called whenever the Media Session has an event or error to report. It is important to realize that when those two functions execute they will **not** be on the form thread. This means that if you take any actions within them that interact with forms or controls you will need to get back on the main form thread first.

That, ladies and gentlemen, is the way that your application is notified of events and errors in the Pipeline. In fact, that is the *only* way your application interacts with the

Pipeline components - other than perhaps with your own user-written transforms. Recall that earlier it was mentioned that both Media Sources and Media Sinks send plenty of events and error messages – these are all swept up by the Media Session and will appear in the Callback Object in due course.

You will see this pattern of Callback Object and C# delegate/events used throughout the Tanta Sample applications whenever a Pipeline Architecture is used. We will be discussing the actions of those two event handlers in considerable detail in the *Implementing the Pipeline Architecture* section of the *Practical WMF Architectures* chapter. If you wish to review the code for those two handlers before then, have a look at the *TantaAudioFileCopyViaPipelineMP3Sink* sample application. It should be emphasized, that these two C# delegates and events are just the Tanta Sample Projects way of doing things. If you look in other sample code you will probably see the event and error handling mechanism implemented differently.

INDEPENDENT PIPELINE OBJECTS

The components in the Pipeline are all COM objects and they do not really know anything about each other. This means they can be used independently of the Media Session as long as you feed the data in and pull the data out in the correct manner. The classic example of this is the Enhanced Video Renderer (EVR). The EVR, as you know, is a Media Sink which displays video data. As long as the EVR object is given the correct information in the correct way, it does not care if a WMF Media Session is providing that data or if something else entirely is activating it. The EVR can be called from completely different architectures and, in fact, the same EVR component you get in Windows Media Foundation is also used to display video in DirectShow graphs. For more information on the EVR please see *An Overview of the EVR* section of the *Rendering Audio and Video* chapter. This book will not provide any discussion about the use of WMF components outside of their “normal” habitat. That sort of thing is a very advanced proposition.

MEDIA STREAMS AND THE PRESENTATION

The previous section (*The Pipeline*) noted how the raw data originating from Media Sources and terminating at Media Sinks is divided up into streams. Also noted was the concept of a Presentation Clock in which the speed at which the streams are rendered is both throttled and synchronized by the Media Session. The throttling is necessary to ensure that the video or audio being rendered does not play too fast and the synchronization makes certain that, if there are multiple streams in the Pipeline, they

The WMF Components

both render their information at the proper time. This makes the video the user sees appear to be correct according to the sounds they hear.

It should be noted that even if only one type of data is being processed there is always a stream. Similarly, even if the intent is to transport the data as fast as possible (a data copying operation) there is always a Presentation Clock in use. If you are using the Pipeline Architecture you will always be dealing with these two entities. The Reader-Writer Architecture uses streams but does not have the concept of a Presentation Clock – you would have to provide that sort of throttling and synchronization yourself if it was required.

The expected behavior of streams originating on a Media Source has also been formalized as an interface named `IMFMediaStream`. Every source Media Stream will (amongst other things) be an object of type `IMFMediaStream`. There is not a lot of functionality in the `IMFMediaStream` interface – but what is present is really important. Using the `IMFMediaStream` interface you can request a new sample, get something called the Stream Descriptor which enables you to discover the Media Sub-Types the Media Stream supports or, if you need to, you can get the Media Source object which originated the stream.

It should be noted that Media Sinks also implement streams in order to accept input data. These streams are conceptually the same thing as Media Streams except that they are called Stream Sinks and they use the `IMFStreamSink` interface. This is a mildly annoying example of an inconsistent naming convention: streams on Media Sources are Media Streams and implement the `IMFMediaStream` interface; streams on Media Sinks are Stream Sinks and implement the `IMFStreamSink` interface. Streams into and out of Transforms are just called streams and they do not implement any interfaces at all – the `IMFTransform` interface takes care of the configuration of those streams.

Returning to the concept of the Presentation Clock where, as you might imagine, there is an object which describes the synchronized and timed presentation of multiple streams. This object is called a Presentation Descriptor and it also has its own interface which defines how an application interacts with it – a Presentation is an `IMFPresentationDescriptor`. The Presentation Descriptor describes all of the possible Media Streams on a Media Source which are related by a common presentation time.

Here is an overview of how basic process works. We will go into more detail further on...

1. You have a Media Source.
2. You obtain a Presentation Descriptor from the Media Source.

3. You look at each Media Stream (there is usually more than one of these) in the Presentation Descriptor and get its Stream Descriptor.
4. Look at the Stream Descriptor and make sure it implements the Media Type you are interested in – there will probably be more than one Media Type. If not, find the next Media Stream. If the Media Type is suitable, enable the stream (“*select*” it in the Stream Descriptor) and then enable the Media Type (make it “*current*” in the Media Stream).

Seem simple enough doesn't it? However, there is more going on here than meets the eye at first glance. There can be (and usually are) a number of Media Streams present in any one Media Source. Essentially, the Presentation Descriptor presents a sort of virtual buffet of the available Media Streams from which your application can choose. Usually the selection process consists of your application sitting in a loop enumerating every possible Media Stream in the Presentation Descriptor and taking a careful look at the details of that stream to see if it is suitable for your needs.

The details of a Media Stream are made available in a Windows Media Foundation container known as a Stream Descriptor. The Stream Descriptor object will implement the `IMFStreamDescriptor` interface and your application can use the function calls defined by this interface to discover the underlying details of the Media Stream it represents. Ultimately, your application will choose a Media Stream (or Media Streams) from the Presentation Descriptor based on the contents of a Stream Descriptor. However, be aware that there are usually a lot of options within a Stream Descriptor and once you have chosen the Media Stream you still have to configure the stream by enabling only one of the Media Types it contains.

It is the Stream Descriptor that is used to configure the Topology and the objects in the Pipeline. Once the Stream Descriptors have been obtained, the Presentation Descriptor plays no further role in the configuration of the Pipeline. Once the Pipeline has been created, the Stream Descriptors are also no longer needed.

WHY DOES A MEDIA SOURCE CONTAIN MULTIPLE STREAMS

Before we carry on, you may well be wondering why a Media Source might contain multiple Media Streams. There is much more complexity here than is immediately apparent. Let's consider an MP4 file. This file will probably have both video and audio – and that is two distinct streams right there. Media Streams of these types would be said to have a Media Major Type of `MFMediaType.Video` or `MFMediaType.Audio`. Less

obviously, there could also be other Media Major Types such as sub-titles, static pictures and many more.

It should be noted that the `MFMediaType` which describes the Media Major Type is just a C# static class with a collection of `Guid` constants. This collection effectively acts as an enum. This means that in Visual Studio, the easy way to inspect it is just to open up any Tanta Sample Project, put your cursor on `MFMediaType` name and press `F12` to see the list. Also, in case you missed it, note that the name of the `MFMediaType` class does not start with an “I”—it is not an interface. When you see some WMF entity that you have never seen before (and you will), you can, in general, use the naming convention as a clue to figure out what you are dealing with.

Also be aware that the Media Type object being referred to above is not just the `MFMediaType` value. A Media Type object implements the `IMFMediaType` interface and one of the many configuration attributes it can contain is a Media Major Type (in other words it contains an `MFMediaType` value). Just to get the nomenclature sorted out be aware that, in general, the other attributes the Media Type contains are collectively referred to as the Media Sub-Type.

You are very unlikely to see Media Types with different Media Major Type values involved in the same Media Stream. Theoretically it is possible – but in reality it does not happen. This is why it is said that a Media Stream has a “*Media Major Type*”. In reality this is just the Media Major Type of the currently enabled Media Type in the Stream Descriptor. Since all Media Types in the stream will have an identical Media Major Type, this is a convenient short-cut.

It is possible to have multiple Media Streams in a Presentation which implement the same Media Major Type. For example, there could be separate audio streams in various languages or separate video streams containing both the theatrical and directors cut of a film.

Within a Media Stream representing a Media Major Type you will see multiple Media Type objects representing a variety of options. As mentioned previously, these attributes are collectively called Media Sub-Types and they will describe a wide variety of options like encoding formats (NV12 or YUV) or frame sizes (640x480 or 1280x720) and many others. We will discuss this in more detail further on but you should be aware at this point that the Media Type objects and their content are discoverable by asking the Stream Descriptor for an object known as a Type Handler and then asking the Type Handler for the details. Yes, that’s correct, you cannot directly ask the Stream Descriptor for the Media Types it offers. You have to get a Type Handler object and use it for that

purpose. Very probably, the same object is that `IMFStreamDescriptor` is also the `IMFTypeHandler` – but you should not just cast these directly. You should call `GetMediaTypeHandler()` on the Stream Descriptor and use the object returned from that call.

All-in-all you should expect to see multiple Media Streams present in any one Media Source. In fact even Media Sources such as a webcam, which you might think would only have one stream, can present multiple streams. Some webcams will present normal uncompressed video on one stream and compressed video on another. Of course, within any one Media Stream there can be multiple dozens of various Media Sub-Types and formats. You can find a more detailed description of this in the *Media Types and Sub-Types* section below and also in the *WMF – First Contact* chapter which discusses discovering the Media Types of the video devices on a system.

Another thing to note is that most Media Sources embrace the concept of a default Media Stream. In such a case the default stream of each Media Major Type will be “selected”. In this case “selected” is just a word in WMF speak which means “enabled”. Of course you can de-select or select any of the Media Streams you wish by calling `DeselectStream()` and `SelectStream()` functions on the Presentation Descriptor (these calls are part of the `IMFPresentationDescriptor` interface). A stream which is not selected will not have media data generated for it by the Media Source. In reference to a previous example, the Media Source may well have a video stream for both the theatrical and directors cut of a film but only one of those Media Streams will be selected by default.

This concept extends down into the Media Stream itself. Once you have your Media Stream and its Stream Descriptor you can look at each one of the Media Sub-Types that stream contains. Only one Media Sub-Type can be “current”. In other words it may be possible that there are lots of encoding types (RGB, NV12 or YUV etc.) in the stream, lots of formats (640x480 or 1280x720 etc.) and lots of other options but only one Media Sub-Type can be “current” and that is the type of media data the Media Stream is going to generate.

Let us recap how this works with a non-WMF example. Say, for example, you decide you wish to eat a meal and you are in a city which offers a lot of options. First you choose one street from many which has a variety of restaurants – this is the equivalent of choosing your Media Source. Each restaurant on that street offers a different type of food – these are your Media Streams and the food type is your Media Major Type. Your scanning of the street and deciding on the type of food you wish to eat is the equivalent of looking at each and every Media Stream in the Presentation Descriptor and making a

decision based on the Media Major Type. Once you choose you “select” the Media Stream. After you choose the restaurant, you sit down and look at the menu. This is the equivalent of asking the Stream Descriptor to tell you the Media Sub-Types. Now the restaurant only offers “*set items*” from the menu. In other words, you cannot pick and choose. You can have option #5 or you can have option #17 but not both and you certainly cannot have a bit of #5 with parts of #10 and extra egg rolls. It is just not possible. Choosing your menu item is the equivalent of choosing the Media Type in the Stream Descriptor based on the Media Sub-Type details. When you choose you menu option – you make it “current”. When the Pipeline starts up, that is the type of food which is going to be delivered. You could have chosen a different item, a different restaurant or a different street but ultimately, whichever Media Stream you choose, you are picking a fixed item from a menu.

From the Presentation Descriptor you “select” the Media Stream – this is usually based on the Media Major Type. From the Media Stream you make one of the available Media Sub-Types “current”. This will be the encoding and format data in which the Media Stream will be presented.

OBTAINING PRESENTATION AND STREAM DESCRIPTORS

Let’s look at the process of obtaining a Presentation Descriptor and selecting a Media Stream. The code block below is from the *TantaAudioFileCopyViaPipelineMP3Sink* example application and it does not do much more than simply choose the first selected audio stream (`MFMediaType.Audio`) that it finds.

```
IMFPresentationDescriptor sourcePresentationDescriptor = null;
int sourceStreamCount = 0;
IMFStreamDescriptor audioStreamDescriptor = null;
bool streamIsSelected = false;
IMFMediaType currentAudioMediaType = null;
int audioStreamIndex = -1;

// A presentation is a set of related media streams that share a common presentation time.
// We now get a copy of the media source's presentation descriptor. Applications can use
// the presentation descriptor to select streams and to get information about the source
// content.
hr = mediaSource.CreatePresentationDescriptor(out sourcePresentationDescriptor);
if (hr != HRESULT.S_OK)
{
    throw new Exception("CreatePresentationDescriptor failed. Err=" + hr.ToString());
}
if (sourcePresentationDescriptor == null)
{
    throw new Exception("sourcePresentationDescriptor == null");
}

// Now we get the number of stream descriptors in the presentation. Each presentation
// descriptor contains a list of one or stream descriptors. These describe the streams
// in the presentation. Streams can be either selected or deselected. Only the
// selected streams produce data. Deselected streams are not active and do not produce
// any data.
hr = sourcePresentationDescriptor.GetStreamDescriptorCount(out sourceStreamCount);
```

```

if (hr != HRESULT.S_OK)
{
    throw new Exception("CreatePresentationDescriptor failed. Err=" + hr.ToString());
}
if (sourceStreamCount == 0)
{
    throw new Exception("sourceStreamCount == 0");
}

// Check each stream in the Presentation Descriptor. Choose the first audio one
// we find irregardless of any sub formats it may use.
for (int i = 0; i < sourceStreamCount; i++)
{
    // we require the major type to be audio
    Guid guidMajorType = TantaWMFUtils.GetMajorMediaTypeFromPresentationDescriptor
        (sourcePresentationDescriptor, i);
    if (guidMajorType != MFMediaType.Audio) continue;

    // we also require the stream to be enabled
    hr = sourcePresentationDescriptor.GetStreamDescriptorByIndex
        (i, out streamIsSelected,
        out audioStreamDescriptor);

    if (hr != HRESULT.S_OK)
    {
        throw new Exception("GetStreamDescriptorByIndex failed. Err=" + hr.ToString());
    }
    if (audioStreamDescriptor == null)
    {
        throw new Exception("audioStreamDescriptor == null");
    }
    // if the stream is selected, leave now we will release the audioStream descriptor later
    if (streamIsSelected == true)
    {
        audioStreamIndex = i; // record this
        break;
    }

    // release the one we are not using
    if (audioStreamDescriptor != null)
    {
        Marshal.ReleaseComObject(audioStreamDescriptor);
        audioStreamDescriptor = null;
    }
    audioStreamIndex = -1;
}

// by the time we get here we should have a audioStreamDescriptor if
// we do not, then we cannot proceed
if (audioStreamDescriptor == null)
{
    throw new Exception("audioStreamDescriptor == null");
}
if (audioStreamIndex < 0)
{
    throw new Exception("GetStreamDescriptorByIndex failed. audioStreamIndex < 0");
}

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::PrepareSessionAndTopology

```

That is a lot of code to take in all at once, but if we walk through it step by step it will not seem that hard. The first thing we do is obtain the Presentation Descriptor from the Media Session. Disregarding all the error checking, that process is just one line...

```
hr = sourcePresentationDescriptor.GetStreamDescriptorCount(out sourceStreamCount);
```

Next we obtain the count of the number of streams in the Presentation Descriptor. The only reason we need this is to set up a C# `for` loop over the streams.

```
hr = sourcePresentationDescriptor.GetStreamDescriptorCount(out sourceStreamCount);
```

After that, we enter a `for` loop and check each Media Stream in the Presentation Descriptor to see if it suits our purpose. In this particular case all we require is that the

stream has a Media Major Type of `MFMediaType.Audio`. We use a call to the `GetMajorMediaTypeFromPresentationDescriptor` static function in the Tanta Library for this purpose. Don't worry too much at this point how the Media Major Type is dug out of the Presentation Descriptor – we will cover that topic in more detail in the following *Media Types and Sub-Types* section.

```
// we require the major type to be audio
Guid guidMajorType = TantaWMFUtils.GetMajorMediaTypeFromPresentationDescriptor
    (sourcePresentationDescriptor, i);
if (guidMajorType != MFMediaType.Audio) continue;
```

The next thing we have to check is that the stream is selected. As noted previously in the *Why Does a Media Source contain Multiple Streams* section, the word “selected” when used on Media Samples is just WMF speak which means “enabled”. The sample code does not select or deselect Media Streams and, if there are none selected by default, an error is returned.

By the time the loop finishes we will have a Stream Descriptor of the audio stream we wish to use. We also keep the index of that Stream Descriptor in the Presentation Descriptor around since a lot of the Tanta Common Library utility functions use that to reference the Media Stream.

Of particular note is the `ReleaseComObject()` call at the bottom of the loop. Up to this point we have not specifically provided examples of releasing Windows Media Foundation objects. The *Releasing COM Objects* section in the *MF.Net Programming* chapter discusses the release of COM objects (which is what WMF objects are) in more detail. It is important to realize that if we obtain an object from Windows Media Foundation it is up to us to release once we are done with it or we will get a memory leak. Take careful note how the logic works in that loop – if we find a Stream Descriptor we want to use, we store it in a local variable and leave the loop before the `ReleaseComObject()` function is called. If you check the `PrepareSessionAndTopology` function in the *TantaAudioFileCopyViaPipelineMP3Sink* example application you will see that we take great care to release the chosen Stream Descriptor as well. You will also note that since we obtained the Presentation Descriptor object from WMF (the Media Source gave it to us) we also have to release that object. In general, every object we receive from Windows Media Foundation must be released. If the object is something we need to keep around for the duration of the process we typically store it in a class variable and take care to release it when operations are complete. If you wish to see how this sort of release can be done, it would be instructive to follow the treatment of the Media Session object in the *TantaAudioFileCopyViaPipelineMP3Sink* sample application.


```

finally
{
    // Clean up
    if (sourcePresentationDescriptor != null)
    {
        Marshal.ReleaseComObject(sourcePresentationDescriptor);
    }
    if (audioStreamDescriptor != null)
    {
        Marshal.ReleaseComObject(audioStreamDescriptor);
    }
    ... <more objects released here>
}
Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::PrepareSessionAndTopology

```

MEDIA TYPES AND SUB-TYPES

The *Media Streams and the Presentation* section has already provided some background on the usage of Media Type objects within the context of Media Streams. This section will provide more details on Media Types and will discuss, in some detail, how to get the media type information from a stream and how to create and populate your own Media Type object.

You will see later on in the *Topologies* section, Media Types are used extensively when configuring the various objects that make up a Topology – after all just about everything that processes media data will need to know the details of the encoding and format of that data.

As was previously noted, a Media Type is an object that implements the `IMFMediaType` interface. A Media Stream offered by a Media Source probably contains multiple Media Type objects. Only one of those will be current (enabled) and the attributes of that Media Type object will determine the format and encoding of the data in the Media Stream.

One of the confusing aspects of configuring a Topology (and hence Pipeline) is that sometimes, when you configure objects you can just provide a Stream Descriptor and sometimes you have to provide a Media Type object. Be aware that in the first instance, the object being configured is probably just discovering the Media Type information from the current Media Type object in the Stream Descriptor. The occasions when we have to supply a Media Type directly are not usually too much trouble since, unless we are changing things like the format, we can often just dig the current Media Type out of the Stream Descriptor and use that. This is why you will sometimes see a completely new Media Type being generated, sometimes you will just see the Source Descriptor

being used and sometimes you will just see the Media Type from the Source Stream Descriptor being used.

GETTING MEDIA TYPES FROM THE STREAM DESCRIPTOR

The Stream Descriptor object cannot be used to access the Media Types contained within the Media Stream. Instead, as mentioned previously, an object of type `IMFMediaTypeHandler` has to be obtained from the Stream Descriptor. The reasons for this particular arrangement are unknown – probably the implementers of Windows Media Foundation just wanted to separate the functionality into distinct components. It is very likely that the object which is the `IMFStreamDescriptor` is also the object which is the `IMFMediaTypeHandler`. You should not cast these directly though – always use the `GetMediaTypeHandler()` call on the Stream Descriptor for this purpose. The code below shows how the Media Type objects can found in a Stream Descriptor

```
public static IMFMediaType GetMediaTypeFromStreamDescriptorById(
    IMFStreamDescriptor streamDescriptor,
    int mediaTypeId)
{
    HRESULT hr;
    IMFMediaTypeHandler typeHandler = null;
    IMFMediaType outMediaType = null;

    if (streamDescriptor == null)
    {
        throw new Exception("No source stream descriptor provided");
    }

    // Getting the media type from a stream has to be done by first fetching a
    // IMFMediaTypeHandler from the stream descriptor and then asking that about the
    // media type. The type handler also has to be cleaned up afterwards. This is a
    // pretty commonly required, multi-step, operation so it
    // has been factored off here as a useful bit of building block code.

    try
    {
        // Get the media type handler for the stream. IMFMediaTypeHandler
        // interface is a standard way of getting or
        // setting the media types on an object
        hr = streamDescriptor.GetMediaTypeHandler(out typeHandler);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to GetMediaTypeHandler failed. Err=" + hr.ToString());
        }
        if (typeHandler == null)
        {
            throw new Exception("call to GetMediaTypeHandler failed. typeHandler == null");
        }

        // Now we have the handler, get the media type.
        hr = typeHandler.GetMediaTypeByIndex(mediaTypeId, out outMediaType);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to GetMediaTypeByIndex failed. Err=" + hr.ToString());
        }
        if (outMediaType == null)
        {
            throw new Exception("call to GetMediaTypeByIndex failed. outMediaType == null");
        }

        // return this
        return outMediaType;
    }
    finally
    {
        // Clean up.
    }
}
```

```

        if (typeHandler != null)
        {
            Marshal.ReleaseComObject(typeHandler);
        }
    }
}
Source: TantaCommon::TantaWMFUtils::GetMediaTypeFromStreamDescriptorById

```

Note that the above code cleans up after itself as much as it can but the Media Type object which is returned must be released by the caller.

Enumerating the Media Types (perhaps to check each one for suitability) in a Media Stream would be a process similar to that of enumerating the Media Streams in a Presentation Descriptor. All that is necessary is to call the `GetMediaTypeCount()` function on the Type Handler object to get a total count of number of Media Type objects the Stream Descriptor contains and then loop through each with a call to the `TantaWMFUtils` function `GetMediaTypeFromStreamDescriptorById` described above.

CREATING YOUR OWN MEDIA TYPE

It should be noted that streams are not the only place to obtain a Media Type objects. It is entirely possible to create and populate your own Media Type. In theory, creating a Media Type is simple – obtaining a new Media Type is pretty much just a matter of calling the static `MFCreateMediaType()` function. Usually it is the subsequent population of the empty Media Type with all the attributes it needs which causes the trouble. The attributes associated with any particular Media Type are highly specific to the Media Major Type, Media Sub-Type, encoding and formats required.

Before we get into the details of how to populate a Media Type object we need to undertake a small digression and discuss how the Media Sub-Type data is stored in a Media Type. Windows Media Foundation rarely uses “hardcoded” variable names when it stores data. For example, inside the Media Type object there is no variable named “majorMediaType” which is populated with the `MFMediaType` enum value. WMF just does not work that way. What Windows Media Foundation does is create a key-value pair mechanism which stores the data and also defines an interface which can manipulate that information. The key-value pair is called an Attribute, an object that maintains a collection of Attributes is called an Attribute Container and the interface used to manipulate this collection is named `IMFAttributes`. There is an extensive discussion of this topic in the *About Attributes* section of the *MF.Net Programming Fundamentals* chapter – however it is important to mention it again here in order to give you a sense of what is happening when the Media Sub-Type data is populated in the code section below.

The WMF Components

There are a variety of ways Windows Media Foundation will implement Attributes in a particular type of object. Sometimes the `IMFAttributes` object is separate and distinct and you have to get the Attribute Container by calling the `QueryInterface()` function on whatever object you are working with. Other times the object actually inherits from `IMFAttributes` (as well as other interfaces) and so you can just cast the object to `IMFAttributes` and use the interface calls directly. The third option which is sometimes seen is to have an interface itself inherit from `IMFAttributes`. This last technique is the behavior of the Media Type object. The `IMFMediaType` interface directly inherits from `IMFAttributes` and so you can always treat any Media Type object as an Attribute Container without the need to cast it or specifically dig the Attribute Container out of the object.

Now that we have a bit of an understanding of Attributes we can see why the code section below can treat the Media Type object as an Attribute Container. The code below creates a new Media Type and populates it with a variety of Media Sub-Type data so that it can be used to configure the output stream of a Sink Writer object.

```
// now configure the SinkWriter. This sets up the sink writer so that it knows what format
// the output data should be written in. The format we give the writer does not
// need to be the same as the format it receives as input - however to make life
// easier for ourselves we will copy a lot of the settings from the videoType retrieved above

// create a new empty media type for us to populate
hr = MFExtern.MFCreateMediaType(out encoderType);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed on call to MFCreateMediaType, retVal=" + hr.ToString());
}

// The major type defines the overall category of the media data. Major types include
// video, audio, script & etc.
hr = encoderType.SetGUID(MFAttributesClsid.MF_MT_MAJOR_TYPE, MFMediaType.Video);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed setting the MF_MT_MAJOR_TYPE, retVal=" + hr.ToString());
}

// The subtype GUID defines a specific media format type within a major type. For
// example, within video, the subtypes include MFMediaType.H264 (MP4),
// MFMediaType.WMV3 (WMV), MJPEG & etc. Within audio, the
// subtypes include PCM audio, Windows Media Audio 9, & etc.
hr = encoderType.SetGUID(MFAttributesClsid.MF_MT_SUBTYPE, MEDIA_TYPTETO_WRITE);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed setting the MF MT SUBTYPE, retVal=" + hr.ToString());
}

// this is the approximate data rate of the video stream, in bits per second, for a
// video media type. The choice here is somewhat arbitrary but seems to work well.
hr = encoderType.SetUINT32(MFAttributesClsid.MF_MT_AVG_BITRATE, TARGET_BIT_RATE);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed setting the MF MT AVG BITRATE, retVal=" + hr.ToString());
}

// populate our new encoding type with the frame size of the videoType selected earlier
hr = TantaWMFUtils.CopyAttributeData(incomingVideoMediaType, encoderType,
                                     MFAttributesClsid.MF_MT_FRAME_SIZE);
if (hr != HRESULT.S_OK)
{

```

```

        // we failed
        throw new Exception("Failed copying the MF_MT_FRAME_SIZE, retVal=" + hr.ToString());
    }

    // populate our new encoding type with the frame rate of the video type selected earlier
    hr = TantaWMFUtils.CopyAttributeData(incomingVideoMediaType, encoderType,
                                         MFAttributesClsid.MF_MT_FRAME_RATE);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("Failed copying the MF_MT_FRAME_RATE, retVal=" + hr.ToString());
    }

    // populate our new encoding type with the pixel aspect ratio of the video type selected earlier
    hr = TantaWMFUtils.CopyAttributeData(incomingVideoMediaType, encoderType,
                                         MFAttributesClsid.MF_MT_PIXEL_ASPECT_RATIO);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("Failed copying the PIXEL_ASPECT_RATIO, retVal=" + hr.ToString());
    }

    // populate our new encoding type with the interlace mode of the video type selected earlier
    hr = TantaWMFUtils.CopyAttributeData(incomingVideoMediaType, encoderType,
                                         MFAttributesClsid.MF_MT_INTERLACE_MODE);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("Failed copying the MF_MT_INTERLACE_MODE, retVal=" + hr.ToString());
    }

    // add a stream to the sink writer for the output Media type. The
    // incomingVideoMediaType specifies the format of the samples that will
    // be written to the file. Note that it does not necessarily need to
    // match the input format.
    hr = workingSinkWriter.AddStream(encoderType, out sinkWriterVideoStreamId);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("Failed adding the output stream(v), retVal=" + hr.ToString());
    }
}

Source: TantaCaptureToScreenAndFile:MFTTantaSampleGrabber_Sync::StartRecording

```

Let's consider a few lines in detail in order to see what is going on. The line below just creates the empty Media Type object using a static WMF function call. This is pretty straight forward.

```
hr = MFExtern.MFCreateMediaType(out encoderType);
```

The next line sets the Media Major Type (in this case `MFMediaType.Video`). In particular, note that the `MFAttributesClsid.MF_MT_MAJOR_TYPE` key is a GUID. In other words, it is a known 128 bit random integer which is always used (in WMF) as a key for the Media Major Type. You set the Media Major Type using this key and everything else knows to find it using this key.

```
hr = encoderType.SetGUID(MFAttributesClsid.MF_MT_MAJOR_TYPE, MFMediaType.Video);
```

The `SetGUID()` function call is part of the `IMFAttributes` interface. As mentioned previously, the only reason the above line works on a Media Type (an `IMFMediaType` object) is because the `IMFMediaType` interface directly inherits from `IMFAttributes`. You may see this being done differently elsewhere and that is probably because the person coding it up did not realize that a Media Type is also an Attribute Container. In

The WMF Components

truth, this particular relationship is not documented all that well and it is not all that common to see other Windows Media Foundation interfaces inheriting in that way.

The next line sets the encoding format.

```
encoderType.SetGUID(MFAttributesClsid.MF_MT_SUBTYPE, MEDIA_TYPTO_WRITE);
```

In this particular case, the `MEDIA_TYPTO_WRITE` constant is defined as `MFMediaType.H264` which is the desired output format. The `MFAttributesClsid.MF_MT_SUBTYPE` key is also a defined and well known GUID which is always used for the Media Sub-Type encoding format.

The following line sets the maximum bit rate at which the video the data can be played. This is an integer and so we use a different call (`SetUINT32` instead of `SetGUID`) to set this particular Attribute value.

```
hr = encoderType.SetUINT32(MFAttributesClsid.MF_MT_AVG_BITRATE, TARGET_BIT_RATE);
```

We also have to set the frame size and when doing this we introduce an interesting technique. Things like frame sizes are typically stored as Attributes with one key (`MF_MT_FRAME_SIZE`) but as a 64 bit integer in which the upper 32 bits contain the width and the lower 32 bits contain the height. This keeps the two data items (width and height) together and associated under one key.

```
hr = TantaWMFUtils.CopyAttributeData(incomingVideoMediaType, encoderType,  
                                     MFAttributesClsid.MF_MT_FRAME_SIZE);
```

In order to set the frame size we would have to build up our own 64 bit integer from the two values and use the `SetUINT64()` call to set the data. The requirement to do this is a bit of an irritation and since this sort of thing happens often enough, there are static function calls like `MFSetAttributeSize()` which takes the Attribute Container object, the GUID key value and the width and height as `UInt32` values and will do the job for you. You still have to obtain the width and height as `UInt32` values though and this would be done via a similar `MFGetAttributeSize()` function call. This, of course, is not difficult but it is more coding than is necessary in this case. In the end all we really want is the same frame size used in the source Media Type (which we have access to) to be present in the output Media Type. So we take a bit of a short-cut and just copy it across using the `CopyAttributeData()` function call located in the *TantaCommon* library. Here is the code for that copy process – in case you would like to see how it works

```
public static HRESULT CopyAttributeData(IMFAttributes srcAttr, IMFAttributes tgtAttr, Guid key)  
{  
    PropVariant var = new PropVariant();  
    HRESULT hr = HRESULT.S_OK;  
  
    if (srcAttr == null) return HRESULT.S_FALSE;  
    if (tgtAttr == null) return HRESULT.S_FALSE;  
  
    // get the source data
```

```

    hr = srcAttr.GetItem(key, var);
    if (hr != HRESULT.S_OK) return hr;

    // get the target data
    hr = tgtAttr.SetItem(key, var);
    return hr;
}

```

Source: TantaCommon::TantaWMFUtils::CopyAttributeData

Returning back to the creation and population of the Media Type, we see that the remainder of the configuration consists of copying Attribute values from the source Media Type to the output Media Type since we wish them to be identical.

```

hr = workingSinkWriter.AddStream(encoderType, out sinkWriterVideoStreamId);

```

At the end, we give the populated Media Type to the Sink Writer object so it can configure its output stream. Strictly speaking this is not part of the creation and population of the Media Type but it is nice to see how the Media Type is used once it has been built.

CLONING A MEDIA TYPE

Cloning a Media Type is a pretty straight forward process, just create it and copy the items over. The code section below documents this so that you have it for future reference.

```

public static IMFMediaType CloneMediaType(IMFMediaType inType)
{
    IMFMediaType outType = null;
    HRESULT hr;

    if (inType != null)
    {
        hr = MFExtern.MFCreateMediaType(out outType);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to MFCreateMediaType failed. Err=" + hr.ToString());
        }
        hr = inType.CopyAllItems(outType);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to CopyAllItems failed. Err=" + hr.ToString());
        }
    }
    return outType;
}

```

Source: TantaCommon::TantaWMFUtils::CloneMediaType

Basically because the IMFMediaType interface inherits directly from IMFAttributes you can just call the CopyAllItems() function to clone the data in the source Media Type over to the destination Media Type.

ENUMERATING THE ATTRIBUTES OF A MEDIA TYPE

If you have a Media Type object (perhaps obtained directly from a Media Stream) it might be useful to look at the Attributes it contains. This is called “enumerating” the

Attributes and is discussed in detail in the *Enumerating Attributes* section of the *MF.Net Programming Fundamentals* chapter.

We will leave the in-depth discussion of the enumeration process to that chapter, however, to set the stage, it is interesting to outline what you would need to do. Here is the basic process...

1. You have a Media Type which contains an unknown number of Attributes.
2. You find the number of Attributes in the Media Type using the `GetCount()` call of the `IMFAttributes` interface.
3. You sit in a `for` loop and obtain each Attribute in turn with a `GetItemByIndex()` call this gives you the key (always a GUID) and an object called a `PropVariant` which can store multiple data types.
4. You see if the key is the one you are interested in (`MF_MT_MAJOR_TYPE`, `MF_MT_SUBTYPE`, `MF_MT_FRAME_SIZE` or many others etc.). If it is, you can obtain the value from the `PropVariant`. This can be one of many data types and the `PropVariant` contains a number of access functions to get you the data in the correct type. You *have* to know the type to get. For example, if you are looking for the target bit rate and the key is `MF_MT_AVG_BITRATE` then you must use the `GetUInt()` function on the `PropVariant`. If the key represented a string, you would have to use the `GetString()`. `PropVariants` are discussed in detail in the *PropVariant* section of the *MF.Net Programming Fundamentals* chapter. It should be noted that the behavior of `PropVariants` in `MF.Net` is different than the behavior of `PropVariants` in `C++` and if you are a `C++` programmer you should read that section very carefully.

Attributes are designed to store data of a varying quantity and type and they do that very well. However the downside is if you are retrieving information from them you have to know exactly what you are looking for (the key) and the format in which it will be present (the storage type in the `PropVariant`).

FOURCC CODES

There are a variety of digital video and audio formats with wildly differing names and capabilities. So it is only natural that somebody would try to make sense of all the chaos

and define a consistent set of acronyms by which any particular codec or compression standard might be known. Thus the concept of the FOURCC code was developed. FOURCC means “*four character code*” and it is an identifier for a specific digital Media Format. You will probably already have seen some of these without realizing the significance of the four digits in the acronym. For example, some common video format examples are YUY2, UYVY and NV12 - but there are many more. Each four letter code specifies a distinct encoding method for the media data and each encoding method will have its own four digit code.

As you might expect, each FOURCC code will be treated internally inside Windows Media Foundation as a distinct Media Sub-Type and that subtype will be represented by a unique GUID. In order to make things simple, an algorithm was defined so that if you have the FOURCC code then you know the GUID for the subtype and vice-versa. The algorithm is simple, just convert the FOURCC code to hexadecimal, reverse the order and append a standard, known sequence of hexadecimal numbers (“-0000-0010-8000-00AA00389B71”) to complete the GUID. Thus in Windows Media Foundation, every Media Sub-Type will be in the format `XXXXXXXX-0000-0010-8000-00AA00389B71` where `XXXXXXXX` is the hexadecimal values of the FOURCC code in reverse order. As an example for the YUY2 codec, the hexadecimal value of “Y” = 0x59, “U” = 0x55, and “2” = 0x32, so “YUY2” in reverse order is “2YUY” and the hexadecimal equivalent of that string would be 0x32595559. Thus, the Media Type GUID for the YUY2 format is always 32595559-0000-0010-8000-00AA00389B71.

The MF.Net library contains a class named `FourCC` which enables the FOURCC code information to be stored and the equivalent Media Sub-Type GUID for it to be automatically generated. So, if you see a section of code that looks like the one below, then you should now be able to understand exactly what it is doing.

```
FourCC FOURCC_YUY2 = new FourCC('Y', 'U', 'Y', '2');
FourCC FOURCC_UYVY = new FourCC('U', 'Y', 'V', 'Y');
FourCC FOURCC_NV12 = new FourCC('N', 'V', '1', '2');

Guid[] m_MediaSubtypes m_MediaSubtypes = new Guid[] {
    FOURCC_NV12.ToMediaSubtype(),
    FOURCC_YUY2.ToMediaSubtype(),
    FOURCC_UYVY.ToMediaSubtype() };

Source: TantaTransformDirect::MFTTantaGrayscale_Sync
```

TOPOLOGIES

Previous chapters and sections (*The Pipeline Architecture*, *The Pipeline*) discussed how Windows Media Foundation uses the Pipeline to transfer media data from the Media Sources to the Media Sinks. It was also mentioned that things can get quite complex and it is possible to have a Pipeline with multiple Media Sources, multiple Media Sinks and

The WMF Components

complex things like splits (Tee's) and joins in multiple branches. Just about the only mechanism that is not possible (or at least not supported) are loops in the branches.

Clearly Pipelines are a powerful processing tool for media data. In addition, as if that were not enough, the Media Session will automatically manage the data flow through the Pipeline and will use a concept called a Presentation to implement throttling and synchronization amongst the branches if necessary. Basically all you, as the programmer, need to do is configure a Pipeline and set it running. After that, an enormous number of extremely complex operations happen in the background and the amount of code you have to write is minimal.

So, how does one configure a Pipeline? Well, one creates a map (essentially a blueprint) containing all of the Media Sources, Media Sinks and other components in the branches of the Pipeline. This map also identifies how the components are connected together. The name for this map is called a Topology and the objects in it are represented by Topology Nodes. There is one Topology Node for each component in the Pipeline. Furthermore, the Topology Nodes (which are themselves WMF objects) can easily be connected in ways which model the branches of the Topology. Once the Topology has been configured, passing the Topology object as a parameter on a call to the `SetTopology()` function of the Media Session will generate the Pipeline. This is called “resolving” the Topology.

A Topology is a Windows Media Foundation object and it is easily created using a static function call. The code section below shows the process.

```
hr = MFExtern.MFCreateTopology(out pTopology);
```

It worth explicitly noting at this point that “creating” the Topology object is not the same thing as “resolving” the Topology. When you create a Topology you just get an empty `IMFTopology` object, once you populate it, the Media Session can use it (“resolve it”) and make a Pipeline out of it.

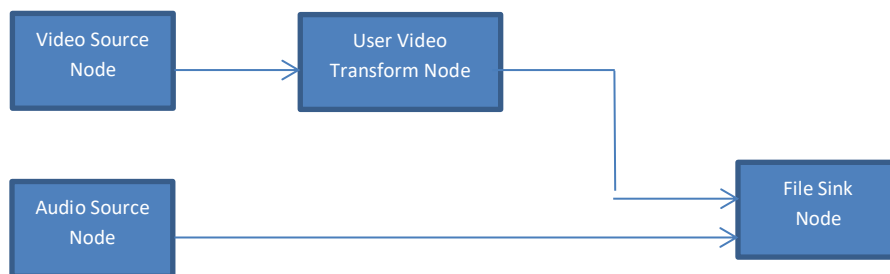


Figure 5.2: A Sample Topology with Two Branches

The image in Figure 5.2 above shows a representation of a Topology with two branches. You will sometimes hear a Topology referred to as a “graph” although that is a DirectShow term which occasionally carries over into WMF. Each Topology Node represents an object. Each arrow represents a stream of data and, even though each stream starts on its own Media Source we would say that there are two branches in the Topology. In this case each branch consists of a stream of one of two Media Major Types (video or audio). It should be noted that this does not always have to be the case – the Topology could be mixing two streams of the same type. Both branches in the Topology terminate on the same node. In this example the File Sink Node is probably something like the MP4 File Sink which writes both audio and video to the file. The User Video Transform Node is present through choice. The user specifically configured it into the Topology in order to perform some function.

The Topology Nodes and their relationship with the components and each other are the essential part of the Topology. The Topology object itself is mostly just a container for the Topology Nodes.

In order to get an additional perspective on what is going on, let’s think about the things a Topology Node might need to be configured with in order for the Media Session to set up a Pipeline.

One of the first things a Topology Node will need to know is whether the node is originating data (called a source node), rendering data (called a sink node) or transforming data (called a Transform Node). As you might imagine, the way the Media Session will interact with each type of node is quite different and so it needs to know what sort of object it is dealing with. When you create a Topology Node you will always be required to tell it the node type. The node type is always one of four `MFTTopologyType` enum values (`OutputNode`, `SourcestreamNode`, `TransformNode` or `TeeNode`). It should be noted that the name “`MFTTopologyType`” is a bit unfortunate since the name implies that it represents a type of Topology (it doesn’t - there is no such concept) rather than a type of Topology Node. A name like `MFTTopologyNodeType` for the enum would have been more descriptive – but it is what it is.

Another thing a Topology Node is going to need to know is the underlying WMF object which will be in the Pipeline at that point. Remember how previously it was said that Windows Media Foundation is based on COM. This means that a most of the ways you “tell” a Topology Node about the underlying object are the COM ways of doing this. Let’s run through the options.

1. You can just give the Topology Node the instantiated object. This is easily done for the user written Transform objects (see the *TantaTransformDirect* Sample Project). This action is performed by the calling the `SetObject()` function of the `IMFTopologyNode` interface and is only useable on Transform or Output nodes. It should be noted that, for things like Media Sources and Media Sinks, you mostly do not ever have the instantiated object itself before you resolve the Topology.
2. A Topology node can accept a GUID value which will allow it to use COM to create the object. This is typical for most of the standard Microsoft supplied WMF objects such as the various Media Sub-Type modification or encoding/decoding Transforms since the GUID values for these are well known and documented. You can also create your own GUID and Transform and the Topology Node will cheerfully use that as long as it can find the object in the Registry and use COM to build it. The *TantaTransformInDLL* and the *TantaTransformInDLLClient* Sample Project pair demonstrate this process. The actual procedure for setting the GUID is performed by calling the `SetGuid()` function on the `IMFAttributes` interface of the Topology Node. Like the `IMFMediaType` interface, the `IMFTopologyNode` interface directly inherits from `IMFAttributes`. Calling `SetGuid()` with the known key of `MF_TOPONODE_TRANSFORM_OBJECTID` and the GUID of the Transform as a value will enable the Topology Node to create the object when the Topology is resolved.
3. A Topology Node can also accept an object called an Activator. An Activator is a COM object that knows how to create the object. You could, theoretically, use COM and the Activator yourself to create your Pipeline Object and then hand it in to the Topology Node directly. However, just giving the Topology Node the Activator is far easier to do and is a lot less trouble. Typically, you obtain an Activator from a static WMF function call. For example, the common way of creating an Enhanced Video Renderer (which is a Media Sink) is by using a call to the WMF function `MFCreatVideoRenderActivate()` which returns an Activator to you. You then give this Activator to the Topology node by the calling the `SetObject()` function of the

`IMFTopologyNode` interface and, it should be noted, that Activators are only useable on Transform or Output nodes.

4. When configuring Source Topology Nodes you have to provide the node with the Media Source object, Presentation Descriptor and Stream Descriptor. This is done by setting those objects as Attributes via a `SetUnknown()` call on the `IMFAttributes` interface on the Topology Node. The *Creating a Topology Node for a Media Source* section below demonstrates this process using the `MF_TOPONODE_SOURCE`, `MF_TOPONODE_PRESENTATION_DESCRIPTOR` and `MF_TOPONODE_STREAM_DESCRIPTOR` GUIDs.
5. In the case of output nodes a Stream Sink object can be supplied to the Topology Node. You can get a Stream Sink object by asking the Input Media Stream on the Media Sink for it with a call to the `GetStreamSinkByIndex()` function. You can give this object to the node by calling the `SetObject()` function of the `IMFTopologyNode` interface. The *Creating a Topology Node for a Media Sink Using a Stream Sink* section below discusses this process in some detail.

So, in summary, the creation methods of a Topology Node, and the items you need to use to populate it, vary widely depending on the type of Topology Node you are creating (source, sink, transform or tee). Some types of Topology Node can use the same creation and population techniques and some cannot. Mostly, they all have multiple ways of obtaining the same result. You just have to understand what is required, look at some example code, and do it that way. Things will go much easier for you if you do not expect any consistency.

Another thing a Topology Node is going to need to know is how it is connected to the other nodes. This is fairly straightforward. The Topology Node implements the `IMFTopologyNode` interface and that interface contains (among many other things) a `ConnectOutput()` function call. Once you have all of your Transform Nodes built, you just start at the source and connect them up node-by-node for each branch. Here is a simple example of a source node connecting directly to an output node.

```
// Connect the output stream from the source node to the input stream of the output
// node. The parameters are:
//   dwOutputIndex - Zero-based index of the output stream on this node.
//   *pDownstreamNode - Pointer to the IMFTopologyNode interface of the node to connect to.
//   dwInputIndexOnDownstreamNode - Zero-based index of the input stream on the other node.
hr = sourceAudioNode.ConnectOutput(0, outputSinkNode, 0);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to pSourceNode.ConnectOutput failed. Err=" + hr.ToString());
}
```

```
}  
Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::PrepareSessionAndTopology
```

Note that in the above `ConnectOutput()` function call that there is an output stream index and an input stream index. In order to incorporate the concept of Tee nodes, there has to be a way to connect two output streams on the Tee to separate input streams on other nodes. In other words, in order to form a split in the Topology, output stream 0 might be configured to connect to input stream 0 on one node and output stream 1 might connect to input stream 0 on another. These stream index values are only relevant to Topology Nodes and are distinct from the Stream Descriptor ID you earlier used when setting up the streams on a Media Source.

It should also be noted that the Topology provides quite a versatile mechanism to describe a branched network. There are many other connection options available on the `IMFTopologyNode` interface besides the `ConnectOutput()` function call. We will not discuss these here because it would divert us into some very advanced topics. The documentation is readily available online should you ever feel the need to make some innovative connections.

TOPOLOGIES AND MEDIA SUB-TYPES

In the above discussion of Topologies, we have kind of glossed over the treatment of the Media Sub-Types used to represent the data as it flows through the Pipeline. Recall (see the *Media Streams and the Presentation* section) that a specific Media Type is selected (made current) when the output stream of a Media Source is chosen. This is the format in which the data will appear. Similarly, the input stream of a file sink is also configured with a Media Type. In addition, in certain types of Media Sink such as those that write files to disk the ultimate output of the Media Sink is also configurable with a Media Type.

The Media Types used in each node of a Pipeline branch do not need to be identical. The Media Sub-Type representation can change at each stage and this must be considered when you configure your Topology. The image below shows a Topology in the process of being configured and the input and output Sub-Type information at each stage.



Figure 5.3: A Topology with Mis-Matched Media Sub-Types

In the above diagram in Figure 5.3 we can see that the video source is configured with an output Media Sub-Type of RGB and a frame size of 800x600. The Video Renderer (a Media Sink) has its input stream configured to accept YUV and a frame size of 640x480. There are a number of ways to fix this – let’s make a list

1. You, as the programmer, could choose to use a compatible Media Sub-Type and format from the Video Source. In other words if YUV (640x480) is available on the Media Source you could simply make that Video Type current in the Stream Descriptor and then the output of the Media Source will match the input Media Type on the Media Sink.
2. Alternatively, you could change the Media Type used to configure the input stream on the Media Sink to match that of the Media Source. As with the Media Source, this may or may not be configurable.
3. A sort of negotiation is also possible. Your application can go back and forth between the available Media Types on offer in the Media Source and Media Sink and find a good compromise that they both support.
4. You could insert a conversion transform in the branch between the Media Source and Media Sink. This does not necessarily have to be a custom Transform written by yourself – Microsoft supply a number of Transforms, as standard, which can be used to perform such tasks. A web search on “*Windows Media Foundation Color Converter DSP*” should take you to the appropriate place in the documentation.

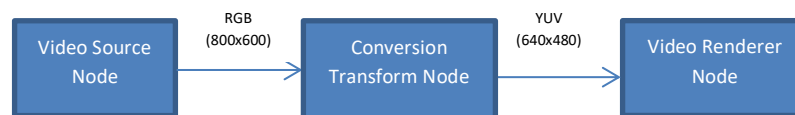


Figure 5.4: A Topology with a Conversion Transform

The above diagram shows the Topology with a Conversion Transform in place. This Transform would be inserted into the Topology in the normal way. A Transform Node would be created and the GUID of the Conversion Transform (in this example `CLSID_CColorConvertDMO`) would be supplied to it. When the Topology is resolved the Conversion Transform will be created and added to the Pipeline.

It should be noted that many Transforms can accept a variety of input Media Types and output Media Types. You, as the programmer, do not specifically choose these Media Types on the Transform like you do with the Media Sources and Media Sinks. Instead, when the Topology is resolved, the list of acceptable inputs or outputs is matched to the Media Type currently on the branch at that point. If there are two Transforms in sequence in the Topology the output Media Type of the first one is just matched to the input Media Type of the next until something that works is found. Since most Media Sub-Types are pretty standard this is not as complicated as it seems.

Also, be aware that all of this Media Type negotiation only happens on Transforms. The Topology will never change the Media Type you set the source stream to output or the Media Types you have configured on the Media Sink.

PARTIAL TOPOLOGIES

If you have been following along in the above discussion, you are now probably well aware that there is a considerable scope for negotiation on the input and output Media Types of the components that make up the Pipeline. In addition, it is up to the application to find the appropriate Transforms and add them to the Topology branch in order to get everything to work.

Does the application always have to do all this work to figure out how to match the Media Types when it sets up a branch in the Topology? In some circumstances the answer is no – it is sometimes possible to automate the addition of the conversion Transforms to the Pipeline.

If you are implementing a Pipeline in a media playback type application (i.e. you are using a renderer style sink) then it is possible to have Transforms automatically added to the Topology when the Topology is resolved.

It should be emphasized that this “auto configuration” does not work on non-playback applications. In other words, if you are writing to something like a file sink you will have to fully configure the Topology.

The component that performs this automatic resolution is called the Topology Loader. It was thought that the most common applications of Windows Media Foundation would be for the playing of media information and the Topology Loader was introduced in order to automate the construction of Pipelines in that situation. Apparently Microsoft decided that rather than have various 3rd party applications perform their own

negotiations in a buggy and indeterminate kind of way they would rather design something that can take the various bits of the Topology and figure it all out automatically. One presumes that they thought applications that wrote to files would need to have much stricter controls on the content format as it passed through the Pipeline and so did not enable the Topology Loader on such Pipelines.

A Topology in which the Media Sub-Types and formats of the components in the various branches are inconsistent is called a Partial Topology. In other words, a Partial Topology is just a normal Topology (in a playback situation) which may or may not need to have Transforms automatically inserted in it to make the Pipeline work.

The automatic resolution of the Topology seems like a fine and wonderful thing – but as you might imagine there are drawbacks. If the Topology loader is busy inserting Transforms into a branch on your behalf, you will never really quite know what the Pipeline looks like. In effect, the Pipeline becomes a kind of black box into which you put data at one end and receive it in a different format at the other.

In addition, the Topology Loader digs about in the Registry to find the Transforms it needs. It does not take any particular care to use only the Microsoft supplied ones and so any third party Transform which is registered might be used. This can introduce odd behaviors into the application and also the Pipeline that is built may well be different from machine to machine or operating system version. When writing your own Transforms you should take care not to leave them in the Registry (unless that is the intent) in case they get picked up and used by the Topology Loader.

It should be noted that the Topology Loader is enabled by default. If you are building a Topology in a playback situation and you call the `SetTopology()` on the Media Session, the Topology Loader will be invoked. You can turn the Topology Loader off if you wish so that the application is forced to fully specify the Topology and the nodes it contains. You can also write your own Topology Loader if you feel you can do a better job than the WMF default one – but that is a topic way more advanced than this book is prepared to discuss.

The connection of the nodes in a Partial Topology is performed in exactly the same way as for a fully specified Topology. You simply call `ConnectOutput()` on the upstream node to connect it to the next node in the sequence. What you do not do, in a Partial Topology, is take any particular care to make sure that the two nodes represent components whose streams use identical Media Types. The Topology Loader will break

the connections and insert a Transform (or Transforms) in order to make that branch of the Pipeline work.

It probably isn't obvious from the above discussion but the Media Types, Sub-Types and formats any one Transform is prepared to accept on its input or produce on its output are essentially "*hard-coded*" into the Transform. This makes sense if you think about it – the Transform is intimately involved with the processing of the data that passes through. It has to know how to deal with that data on a very fundamental and basic level. This means that the each Media Sub-Type it can support must necessarily be backed up by some very dedicated code. There can, of course, be more than one Media Sub-Type (and frame size etc.) on an input or output but all of these are specifically declared as supported. The Transform cannot just "*figure it out*" if it gets an unknown Media Type.

There are other situations in which Transforms may be automatically invoked. Recall that it was earlier mentioned that some Media Sinks (such as a file sinks) can have a different Media Type on their input than that which is actually written to the disk? Well, if the Media Types are different, then something has to do the conversion. The Media Sink will, in a way which is completely transparent to you, find and use a Conversion Transform. Similarly a Sink Writer (even though it is not part of a Pipeline) will find a use a Conversion Transform for the same purpose.

CREATING A TOPOLOGY NODE FOR A MEDIA SOURCE

The following code block demonstrates how to create a Topology Node from a Media Source. Note that the creation process also requires a Presentation Descriptor and a Stream Descriptor. Both of these components are also obtained from the same Media Source – but since they are usually also used elsewhere they are passed in to this function. See the `PrepareSessionAndTopology` function in the `frmMain` class of the *TantaAudioFileCopyViaPipelineMP3Sink* Sample Project for details.

```
/// ++++++
/// <summary>
/// Create a topology node for a source stream. The source node must contain
/// pointers to the media source, the presentation descriptor, and the
/// stream descriptor. This code is just an encapsulated way of doing
/// that. It looks way more complicated than it is.
/// </summary>
/// <param name="pSource">the media source</param>
/// <param name="sourcePresentationDescriptor">the source presentation descriptor</param>
/// <param name="streamDescriptor">the source stream descriptor</param>
/// <returns>the source stream node</returns>
/// <history>
///     01 Nov 18   Cynic - Originally Written
/// </history>
public static IMFTopologyNode CreateSourceNodeForStream(IMFMediaSource pSource,
                                                         IMFPresentationDescriptor
                                                         sourcePresentationDescriptor,
                                                         IMFStreamDescriptor streamDescriptor)
{
```

```

HRESULT hr;
IMFTopologyNode outSourceNode = null;

if (pSource == null)
{
    throw new Exception("No media source object provided");
}
if (sourcePresentationDescriptor == null)
{
    throw new Exception("No source presentation descriptor provided");
}
if (streamDescriptor == null)
{
    throw new Exception("No source stream descriptor provided");
}

try
{
    // A source node represents one stream from a media source. The source
    // node must contain pointers to the media source, the presentation
    // descriptor, and the stream descriptor. This code is just an encapsulated
    // way of doing that.

    // Create the empty structure of the source-stream node.
    hr = MFExtern.MFCreateTopologyNode(MFTopologyType.SourcestreamNode, out outSourceNode);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("GetStreamDescriptorByIndex failed. Err=" + hr.ToString());
    }
    if (outSourceNode == null)
    {
        throw new Exception("outSourceNode == null");
    }

    // Set attribute: Pointer to the media source.
    hr = outSourceNode.SetUnknown(MFAttributesClsid.MF_TOPONODE_SOURCE, pSource);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("Set MF_TOPONODE_SOURCE failed. Err=" + hr.ToString());
    }

    // Set attribute: Pointer to the presentation descriptor.
    hr = outSourceNode.SetUnknown(
        MFAttributesClsid.MF_TOPONODE_PRESENTATION_DESCRIPTOR,
        sourcePresentationDescriptor);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("MF_TOPONODE_PRESENTATION_DESCRIPTOR. Err=" + hr.ToString());
    }

    // Set attribute: Pointer to the stream descriptor.
    hr = outSourceNode.SetUnknown(MFAttributesClsid.MF_TOPONODE_STREAM_DESCRIPTOR,
        streamDescriptor);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("MF_TOPONODE_STREAM_DESCRIPTOR failed. Err=" + hr.ToString());
    }

    // Return the IMFTopologyNode pointer to the caller.
    return outSourceNode;
}
catch
{
    // If we failed, release the pnode
    if (outSourceNode != null)
    {
        Marshal.ReleaseComObject(outSourceNode);
    }
    outSourceNode = null;
    throw;
}
}

Source: TantaCommon::TantaWMFUtils::CreateSourceNodeForStream

```

For consistency you might expect to just be able to pass in the Media Source with a `SetObject()` call like one does for Transforms and the Media Session could figure it out

for itself. However this is not possible (and the documentation does say so). This is logical if you think about it – there is only one Presentation in a Media Source but there can be any number of Media Streams in a Presentation. The Media Session would not know which Media Stream is to be associated with that Topology Node when the time comes to resolve it.

CREATING A TOPOLOGY NODE FOR A MEDIA SINK USING A STREAM SINK

The code section below demonstrates how to create a Topology Sink Node from a Media Sink. Recall that a Media Sink can have multiple inputs (a file sink writing audio and video to the same file). For this reason, the Topology Node requires the index of the Media Stream on the Media Sink. This stream must be previously configured before the Topology Node is created. See the `OpenMediaFileSink` function in the `frmMain` class of the *TantaAudioFileCopyViaPipelineMP3Sink* Sample Project for details.

```
/// ++++++
/// <summary>
/// Create a topology node for a sink stream. The sink node must contain
/// pointers to the media sink and the stream it is using. This code is just
/// an encapsulated way of doing that.
/// </summary>
/// <param name="pSink">the media sink</param>
/// <param name="streamIndex">the stream index</param>
/// <returns>the sink stream node</returns>
/// <history>
/// 01 Nov 18 Cynic - Originally Written
/// </history>
public static IMFTopologyNode CreateSinkNodeForStream(IMFMediaSink pSink, int streamIndex)
{
    HRESULT hr;
    IMFTopologyNode outSinkNode = null;
    IMFStreamSink pStream = null;

    if (pSink == null)
    {
        throw new Exception("CreateSinkNodeForStream No media sink object provided");
    }

    try
    {
        // A sink node represents one stream from a media sink. The sink node must
        // to the media sink and the stream it is using.

        // Create the empty structure of the sink-stream node.
        hr = MFExtern.MFCreateTopologyNode(MFTopologyType.OutputNode, out outSinkNode);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to MFCreateTopologyNode failed. Err=" + hr.ToString());
        }
        if (outSinkNode == null)
        {
            throw new Exception("call to MFCreateTopologyNode failed. outSinkNode == null");
        }

        // get the StreamSink
        hr = pSink.GetStreamSinkByIndex(streamIndex, out pStream);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to GetStreamSinkByIndex failed. Err=" + hr.ToString());
        }
        if (pStream == null)
        {
            throw new Exception("call to GetStreamSinkByIndex failed. pStream == null");
        }
    }
}
```

```

        // Set the object pointer to the media stream sink
        hr = outSinkNode.SetObject(pStream);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to SetObject on node failed. Err=" + hr.ToString());
        }

        // Return the IMFTopologyNode pointer to the caller.
        return outSinkNode;
    }
    catch
    {
        // If we failed, release the pnode
        if (outSinkNode != null)
        {
            Marshal.ReleaseComObject(outSinkNode);
        }
        outSinkNode = null;
        throw;
    }
}

Source: TantaCommon::TantaWMFUtils::CreateSinkNodeForStream

```

In the above code a Stream Sink object (an `IMFStreamSink` interface) is obtained from the Media Sink. The `SetObject()` call is used on the Topology Node. When the Topology is resolved, the Media Session will notice that the node is an output node, the contained object is an `IMFStreamSink` and it will deal with it appropriately. In particular, note how the Stream Sink knows about the Media Stream on the Media Sink that it represents.

CREATING A TOPOLOGY NODE FOR A MEDIA SINK USING AN ACTIVATOR

The code below will demonstrate the process of creating an output Topology Node using an Activator. The Activator is obtained via a call to a static WMF function. The code below, clipped from the `TantaWMFUtils` sample code, is written to handle the creation of a Topology Node for an Enhanced Video Renderer (EVR). There is a similar version for the Streaming Audio Renderer (SAR).

```

/// ++++++
/// <summary>
/// Create a topology node for EVR Video Renderer sink. The caller must
/// release the returned node.
/// </summary>
/// <param name="videoWindowHandle">the handle to the window
///         on which video streams will display</param>
/// <returns>the output stream node</returns>
/// <history>
///     01 Nov 18   Cynic - Originally Written
/// </history>
public static IMFTopologyNode CreateEVRRendererOutputNodeForStream(IntPtr videoWindowHandle)
{
    HRESULT hr;
    IMFTopologyNode outputNode = null;
    IMFActivate pRendererActivate = null;

    try
    {
        // Create a downstream node.
        hr = MFExtern.MFCreateTopologyNode(MFTopologyType.OutputNode, out outputNode);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("MFCreateTopologyNode failed. Err=" + hr.ToString());
        }
        if (outputNode == null)
        {

```

The WMF Components

```
        throw new Exception("outputNode == null");
    }

    // There are two ways to initialize an output node
    //      1) From a pointer to the stream sink.
    //      2) From a pointer to an activation object for the media sink.
    // since we do not have a stream sink at this point we are going to go the
    // activation object route. This is what we are doing below.

    // Create an activation object for the enhanced video renderer (EVR) media sink.
    hr = MFExtern.MFCreateVideoRendererActivate(videoWindowHandle, out pRendererActivate);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("MFCreateVideoRendererActivate failed. Err=" + hr.ToString());
    }
    if (pRendererActivate == null)
    {
        throw new Exception("pRendererActivate == null");
    }

    // Set the IActivate object on the output node. Note that not all node types use
    // this object. On transform nodes this is IMFTransform or IMFActivate interface
    // and on output nodes it is IMFStreamSink or IMFActivate interface. Not used
    // on source or tee nodes.
    hr = outputNode.SetObject(pRendererActivate);

    // Return the IMFTopologyNode pointer to the caller.
    return outputNode;
}
catch
{
    // If we failed, release the pNode
    if (outputNode != null)
    {
        Marshal.ReleaseComObject(outputNode);
    }
    throw;
}
finally
{
    // Clean up.
    if (pRendererActivate != null)
    {
        Marshal.ReleaseComObject(pRendererActivate);
    }
}
}
```

Source: TantaCommon::TantaWMFUtils::CreateEVRRendererOutputNodeForStream

Note that renderer objects typically only have one primary input stream so the stream id is simply assumed by the Topology Node. This is yet another inconsistency you need to be aware of when creating output Topology Nodes. It should be noted that the Enhanced Video Renderer (EVR) has the capability to accept multiple secondary Media Streams and things like overlays and Window-In-Window are possible. However, these are advanced concepts and they will not be discussed in this book. In addition, the EVR renderer is able to handle most of the common input Media Sub-Types and formats so their specification at creation time is not terribly relevant. Of course, since this is a playback situation, the Topology Loader may well automatically add additional Transforms to the Pipeline in order to make things work.

It should be noted that if you use an Activator to create the Media Sink you do not, when creating the Topology Node, get the actual renderer object returned to you. You may need this component later on (for the EVR in particular) in order to adjust things like the screen aspect ratio or to get a static snapshot of the display. There are two basic

ways to obtain this object – both are only available to you after the Topology has been resolved. You can use the general technique described in the *Getting The WMF Object From a Topology Node* section below to directly obtain the object backing up the Topology Node. However, this has complications with the EVR. Alternately, in the case of the Enhanced Video Renderer, you can dig it out of the Media Session using the static `MFGetService` call with the `MR_VIDEO_RENDER_SERVICE` GUID value. See the *Getting an Interface from an Interface* section in the *MF.Net Programming Fundamentals* chapter for more information on this.

GETTING THE WMF OBJECT FROM A TOPOLOGY NODE

When a Windows Media Foundation object (of any type) is added to a Pipeline, a Topology Node is always created for it. There are a variety of ways of indirectly creating the object (for example you have a GUID or an Activator) and so your code may not have ever have handled the actual instantiated object. If you have access to the node (perhaps you saved a reference during the creation process), the Topology Node itself provides a way to access the underlying WMF object in the Pipeline.

This technique is used in the *Information Exchange Via Attributes* section of the *Working With Transforms* chapter to get the Transform object belonging to a Topology Node. Since the method of getting an instantiated Transform object from a Topology node was not the main focus of that discussion - it may well be missed. The code section below will detail the general procedure for reference purposes.

```
public static object GetObjectFromTopologyNode(IMFTopologyNode topoNode)
{
    HRESULT hr;
    object nodeObject;

    if (topoNode == null) return null;

    // get the object from the node
    hr = topoNode.GetObject(out nodeObject);
    if (hr != HRESULT.S_OK) return null;
    return nodeObject;
}
```

Source: `TantaCommon::TantaWMFUtils::GetObjectFromTopologyNode`

The line to note is `hr = topoNode.GetObject(out transformObject)`. If you have the Topology Node then, assuming the Topology has been resolved, you can always get the underlying WMF object that node represents. Note that in the above code section, the component is initially returned as an `object` and it should be subsequently tested to see if it actually implements the required interface before it is used.

TRANSFORMS

The previous section in this chapter discussed Transforms in the context of the Topology and Pipeline. This section will provide a more in-depth perspective on Transforms. In order to provide sufficient background to make the concepts understandable, there will be a bit of repetition of some of the previous topics. The *Working With Transforms* chapter towards the end of this book offers a more comprehensive discussion of the process of creating your own Transforms.

The Pipeline (managed by the Media Session) transports one or more streams of data from Media Sources to Media Sinks. In between the Media Source and the Media Sink for each stream, there can be one or more binary objects which process the data as it moves through the Pipeline. These objects may be decoders, encoders, multiplexers, effect processors or a multitude of other types. Any object which processes the data in the stream, and which is not a Media Source or Media Sink, is known as a Windows Media Foundation Transform (MFT). There are many different categories of Transform and, on any one system, there can be multiple Transforms fulfilling the same function. You may also hear Transforms referred to as Codecs if they are intended for encoding/decoding (compressing/decompressing) a stream of data or as a Digital Signal Processor (DSP) if they are related to converting the stream of data into a different format. In all cases, within Windows Media Foundation, Codecs and DSP's are Transforms and they will all implement the `IMFTTransform` interface.

Most WMF playback applications do not know or care about Transforms – they simply want an end result. In recognition of this, Microsoft has made it possible for playback operations to automatically build a working Topology using a clever bit of code called the Topology Loader. Essentially this means a Topology is created, the Media Source, Media Sink and desired Transforms are added in to it and then the Topology Loader automatically connects them all up when a `SetTopology()` call is made on the Media Session. The collection of incompletely connected Media Sources, Sinks and Transforms is called a Partial Topology. If the Transforms supplied in a Partial Topology are unable to connect to each other because of mismatched input and output Media Types, the Topology Loader may well add in other Transforms in order to make the connections work. Note that the automatic resolution of a Topology is only available for playback operations – file encoding and other types of Pipelines will not automatically add Transforms to a Partial Topology in order to resolve it.

In playback situations this means that most of the time the Media Session, when given a Media Source and a Media Sink, will automatically add to the Topology (and hence the Pipeline) the Transforms it needs in order to get the whole process to work. As

mentioned previously, when building a topology for situations other than playback, you will have to fully specify each source, sink and transform and connect them up to each other. In general, applications which must specify a Full Topology tend to ship with a complete set of their own Transforms with the occasional addition of a standard Microsoft supplied one.

The automatic resolution process can be very dynamic and the Topology that results from such an operation can be highly dependent on the configuration of that particular machine and the Transforms available on it. For example, take the case where a Media Source is providing a stream with encoded information. In that event, a decoder will need to be added before the data can be used. If the decoder offers an output format the Media Sink can accept then all is well – the Media Session will simply connect the Source to the Decoder and the Decoder to the Sink and the Topology is complete. However, it may happen that the only available decoder does not have an output format that the Media Sink can accept and so one or more format conversion Transforms will automatically be added to the Topology in order to provide a viable Pipeline. The Topology Loader does not care if the Transform it uses is a third party version installed by some other software vendor or if it is one of the Windows standard ones. It just uses the first Transform it comes to that will match the criteria. This can make for somewhat variable performance and reliability if different machines have different collections (or versions) of Transforms.

ADDING TRANSFORMS TO A TOPOLOGY

Transforms, as discussed above, can be automatically added to a Topology in certain situations. However they can also be explicitly specified and, as is typical with WMF, there are a number of equivalent ways to do this. There is an extended discussion of each of these methods in the *Working With Transforms* chapter and so only a summary will be given below.

1. If you have the source code for the Transform you can include it with the application source, compile it up, instantiate it with the `C# new` operator and just give it to the topology as a binary via a `SetObject()` call on a Transform Node.
2. If you have the Activator object for the Transform (perhaps because you enumerated the available Transforms on the system) you can simply give the Activator to the Topology by creating a Transform Node and calling `SetObject()` with the Activator object as a parameter.

3. If you know the GUID of a Transform registered on your system you can create a Transform Node for it and get that node to instantiate it with a call to `SetGUID()` on the node.
4. If you know the GUID of a Transform registered on your system. You can instantiate it yourself via COM with a call to `CoCreateInstance()` and then give the object to the Topology by creating a Transform Node and calling `SetObject()` with the newly instantiated object as a parameter.
5. It is possible for Source Reader or Sink Writer objects, which do not explicitly use a Media Session or Pipeline, to automatically load a Transform if they happen to need that functionality. The only Transforms available in such a situation are those discoverable because they are registered and enumerable.

REGISTERING TRANSFORMS

As mentioned in the previous section, several methods of adding a Transform to a Topology involve specifying a known GUID which makes it possible for an application to find an available Transform and use it. This poses the interesting question: *“How does one register a Transform and make it available via a GUID?”*

Transforms can be made available to an application in one of two modes – internally inside the process or globally which means it is available to all applications on the system. Both of these methods use the Registry. The internal process-specific method simply records the registration in a temporary part of the Registry which is only visible to that process and only present while the process is running.

The primary thing to realize is that a Transform which is to be made generally available is a COM object, it is contained in a DLL library and that DLL is recorded in the Registry like any other C# COM object. Typically, this registration process requires the use of the `RegAsm` tool. Registering the DLL for use with COM is enough to make it available to Windows Media Foundation if the application knows the GUID beforehand. However, COM registration alone is not enough to enable WMF to *discover* the Transform if it does not know the GUID.

There are lots of COM objects on the system which have nothing to do with Windows Media Foundation Transforms and so, if Transforms are to be discovered, a special area of the registry must be set aside specifically for their listing and categorization. For the discovery process to happen, the Transform must also be registered with a call to the WMF functions `MFTRegister` or `MFTRegisterLocal`. Once the Registration is complete,

the applications on the system will be able to see it when they enumerate the Transforms available on the system.

There are several automated ways of registering a Transform and the *Working With Transforms* section contains a detailed review of the options.

FINDING TRANSFORMS

The process of enumerating the available Transforms registered on a system is quite straightforward and the Transforms are helpfully organized by category in order to make the selection process easier. The MF.Net library contains a static `MFTEnumEx()` function specifically for this purpose. The procedure for enumerating the Transforms on a system is discussed in detail in the *Working With Transforms* section of this book and the *TantaTransformPicker* sample application contains a full set of working code for demonstration purposes.

SYNCHRONOUS AND ASYNCHRONOUS TRANSFORMS

When Windows Media Foundation was first introduced with Windows Vista, the only type of Transform available was what is now known as the Synchronous Transform. With the Synchronous model, the Media Session makes repeated calls to the `ProcessInput()` and `ProcessOutput()` functions of the Transform. Of course there are situations where more input is needed before more output can be provided and, similarly, circumstances in which no more input can be accepted until the output has been taken. By and large, however, the information flows through the Transform in a linear sequence and the Transform simply blocks while it is processing data. There is no mechanism in a Synchronous Transform for the parallel processing of media data.

The Synchronous model works quite well. However, the developers of Windows Media Foundation recognized that there are situations in which the parallel processing of the Media Samples would be useful and introduced the concept of Asynchronous Transforms with Windows 7.

With the Asynchronous model, it is possible to have multiple threads working on different input data at the same time. The Media Samples to process and the threads that work on them are controlled by an internal queue inside the Transform. It should be recognized that, even though the various threads that are processing the data are working in parallel, the output data must still be emitted in the correct order. This means the output of some worker threads may have to block until others which are processing earlier Media Samples are finished. As you might imagine, there is some complexity in the scheduling all of this.

In addition, while a Synchronous MFT can send and receive events, these events are all in-band. In other words, the events are inserted into the data flow and are acted on as the data is processed. Since the data processing in an Asynchronous Transform is necessarily split between multiple threads, an event model like that could never work. Thus an Asynchronous MFT implements an out of band event mechanism in which the events are sent and received separately from the data. Since this event mechanism is already present and functional, the Asynchronous Transform also uses it to request more data or to tell the client that it has output ready. The communications between an Asynchronous Transform and the client are much more intricate and more tightly choreographed. It is often said that the Synchronous Transform uses a “push-pull” model and the Asynchronous version is “event driven”.

The Asynchronous Transform is a very powerful concept and there is a base class in the Tanta Library (*TantaMFTBase_Async*) greatly simplifies its implementation.

However, unless you need multiple threads, you will probably find the Synchronous versions much easier to implement - even with the help of the base classes to factor out most of the work. This book will not discuss Asynchronous Transforms in detail. It should be noted here that there is also a base class in the Tanta Library for Synchronous Mode Transforms (*TantaMFTBase_Sync*) which also greatly simplifies their implementation.

SAMPLES, FRAMES AND BUFFERS

The whole point of Windows Media Foundation is that data is transferred from a source to a sink. To achieve this goal, the media data is sent as a continuous sequence of “chunks” rather than all at once. This particular architectural decision nicely takes care of live streaming applications and also reduces the resource requirements in situations where all of the data could theoretically be delivered at once. The “chunk” of media data is called a Media Sample. Note that the Media Sample is not the media data – it is a container for another container (called a Media Buffer) which actually contains the actual raw media data.

Unless you are writing your own Transforms it is unlikely you will have to deal with the content of a Media Sample. If you use the Reader-Writer or Hybrid Architectures you may have to deal with Media Samples as a unit – but not with their internal contents.

The processing of Media Samples is slightly different depending on the WMF architecture your application is using. If you are using the Pipeline model and have not introduced any user written Transforms, the transmission and processing of the Media Samples is performed entirely in the background. In such situations your application never handles a Media Sample or its contents. With the Reader-Writer model you, as the application coder, are responsible for taking the Media Sample from the Source Reader and giving it to the Sink Writer component. In the Hybrid Architecture the Media Samples are hidden in the Pipeline until they reach a special Sample Grabber Sink or user written Transform at which time your application may need to process them in order to pass them on to a Sink Writer.

THE RAW MEDIA DATA

Ultimately the raw media data is a sequence of ones and zeros and, in order for it to make sense to anybody, it has to have a defined structure. The bit layout of this structure is entirely determined by the Media Sub-Type of the data and adheres to a strictly defined standard. These standards can get quite detailed. For example, one of the simplest is video data stored in RGB32 format. This is basically a series of 32 bit integers in which the least significant 8 bits stores the Red color intensity, the next 8 bits stores the Green color intensity and the 8 bits after that store the Blue color intensity. The final 8 bits are either not used or are used to represent a Transparency value in some implementations. Other formats such as YUYV are designed to take human perception into account and divide the color space up into one luminance (Y) and two chrominance components (U and V). Essentially the Y represents the brightness and the U and V represent the colors. This book will not discuss the details of these formats – there is plenty of information available online.

Be aware that that if you manipulate the raw media data, your code must necessarily be designed to interact with a specific video format.

Specific examples of manipulations of this type can be seen in the *MFTTantaWriteText_Sync* and *MFTTantaVideoRotator_Sync* example Transforms implemented as part of the *TantaTransformDirect* Sample Project.

In MF.Net the raw video data will be located on the system heap. In other words it is not located within the Managed Memory space of the .NET Common Language Runtime (CLR) and thus your program cannot directly access it. At most you will get something known as an *IntPtr* which is basically just a memory address. So how does your program

get access to the raw media data? Well, there are two basic ways and two other ones which are not really ways at all ...

1. You can use `Marshal.Copy()` and copy the data at the `IntPtr` location from the heap to managed memory. Of course you then may have to copy it back again once you are done with it.
2. You can use “*unsafe*” code and manipulate the memory directly.
3. If your underlying data is contained in nice large sequences (for example planes in NV12 format data) you can sometimes just use the `CopyMemory` and `FillMemory` externs to deal with the memory.
4. If you don’t want to make changes to the raw media data and just want to give something else access to it, in some circumstances you may not need to copy the data at all you can just hand over the `IntPtr` value.

USING MARSHAL TO ACCESS RAW MEDIA DATA

Let’s be honest here. Yes, using `Marshal` is the recommended way of accessing the raw media data in MF.Net, however it is slow. Probably you will find that the `Marshal.Copy()` method too slow for applications which are processing large amounts of data. In the Tanta Sample Projects the use of `Marshal.Copy()` is used to for simple, one-off, things like getting a structure or other data on the WMF heap into managed memory. The sample code section below is part of a routine which creates a bitmap snapshot from the contents of the Enhanced Video Renderer display. You can find it in the *TantaFilePlaybackAdvanced* Sample Project.

```
... more code

// get the image on the screen now. This will give us the image data and the
// bitmap info header. However, be aware that there are two headers associated
// with every .bmp file. The first is a file header (which we have to build
// ourselves) and the second is an info header which we are given in the call below.
// Also note that the memory for the bitmapData variable we receive here needs to be freed

hr = evrVideoDisplay.GetCurrentImage(workingBitmapInfoHeader, out bitmapData, out bitmapDataSize,
out bitmapTimestamp);
if (hr != HRESULT.S_OK)
{
    throw new Exception("buttonTakeSnapshot_Click failed. Err=" + hr.ToString());
}

// bitmapData is an IntPtr. Use Marshal to copy the video data out into a byte array
// bitmapDataSize is the length of bitmapData
byte[] managedArray = new byte[bitmapDataSize];
Marshal.Copy(bitmapData, managedArray, 0, bitmapDataSize);

... more code

Source: TantaCommon::ctlTantaEVRFilePlayer::buttonTakeSnapshot_Click
```

The need to create a bitmap only occurs when the user triggers the operation with a button press and so the process is not too time critical.

ACCESSING RAW MEDIA DATA WITH UNSAFE CODE

The word “unsafe” used in this context is a rather unfortunate term. In C# “unsafe” simply means accessing *“a block of memory not managed by the .NET runtime”*. Accessing memory in C# via “unsafe” pointers is exactly the same thing as doing regular pointer access in C++. In other words, if you are careful (and you have to be careful in C++ too), unsafe pointer access in C# is no more and no less dangerous than doing regular pointer access in any other language. The upside is that it is much, much faster.

In order to use unsafe code your function must be decorated with the unsafe keyword and you also have to check the *“Allow unsafe code”* option on the Build Tab of your projects properties otherwise you will get a compile time error. The code section below demonstrates how to do some pointer based math inside an *“unsafe”* C# function.

```
/// ++++++
/// <summary>
/// Copy a YUY2 formatted image to an output buffer while converting to grayscale.
/// </summary>
/// <param name="pDest">Pointer to the destination buffer.</param>
/// <param name="lDestStride">Stride of the destination buffer, in bytes.</param>
/// <param name="pSrc">Pointer to the source buffer.</param>
/// <param name="lSrcStride">Stride of the source buffer, in bytes.</param>
/// <param name="dwWidthInPixels">Frame width in pixels.</param>
/// <param name="dwHeightInPixels">Frame height, in pixels.</param>
/// <history>
/// 01 Nov 18 Cynic - Ported In
/// </history>
unsafe private void TransformImageOfTypeYUY2(
    IntPtr pDest,
    int lDestStride,
    IntPtr pSrc,
    int lSrcStride,
    int dwWidthInPixels,
    int dwHeightInPixels
)
{
    // This routine uses unsafe pointers for performance reasons.
    // If you don't know what unsafe pointers are then you
    // should look it up. Spoiler alert: they are not as
    // "unsafe" as the word would imply - unsafe is just a
    // specific c# technical term.

    // Remember the actual data is down in unmanaged memory
    // there does not seem to be an efficient "safe" way to copy
    // unmanaged memory to unmanaged memory without spooling it
    // through a temporary managed store - and this is slow.

    ushort* pSrc_Pixel = (ushort*)pSrc;
    ushort* pDest_Pixel = (ushort*)pDest;
    int lMySrcStride = (lSrcStride / 2); // lSrcStride is in bytes and we need words
    int lMyDestStride = (lDestStride / 2); // lSrcStride is in bytes and we need words

    for (int y = 0; y < dwHeightInPixels; y++)
    {
        for (int x = 0; x < dwWidthInPixels; x++)
        {
            // Byte order is Y0 U0 Y1 V0
            // Each WORD is a byte pair (Y, U/V)
            // Windows is little-endian so the order appears reversed.

            pDest_Pixel[x] = (ushort)((pSrc_Pixel[x] & 0x00FF) | 0x8000);
        }
    }
}
```

The WMF Components

```

        pSrc_Pixel += lMySrcStride;
        pDest_Pixel += lMyDestStride;
    }
}

Source: TantaTransformDirect::MFTTantaGrayscale_Sync::TransformImageOfTypeYUY2

```

The above code is designed to convert a video frame in YUY2 format to grayscale - don't concern yourself too much with the details though. For now just note that the `unsafe` key word is used in the function header and that the code is doing what is really only some pretty standard pointer math on the underlying data.

MANIPULATING RAW MEDIA DATA WITH EXTERNS

It is possible to define C# externs to functions in external DLLs to perform block memory manipulation operations for you. Two of these are the `CopyMemory` and `FillMemory` functions in the `Kernel32.dll`. The definitions for these two externs are shown in the code section below.

```
#region Externs

[DllImport("Kernel32.dll"), System.Security.SuppressUnmanagedCodeSecurity]
private static extern void CopyMemory(IntPtr Destination, IntPtr Source, int Length);

[DllImport("kernel32.dll"), System.Security.SuppressUnmanagedCodeSecurity]
private static extern void FillMemory(IntPtr destination, int len, byte val);

#endregion

Source: TantaTransformDirect::MFTTantaGrayscale_Sync
```

It is pretty rare one runs across a situation in which one can use these two calls to perform operations as efficiently requires that there are large sections of data present that need to be copied or filled. However, having said that, the conversion of an NV12 video frame to grayscale is one of those cases. It just so happens that the NV12 format associates the video data into specific groups called “planes”. In the case of NV12, in order to convert the image to grayscale, all we need to do is copy one of the planes and fill in the other with dummy data. This job, as is shown in the code section below, is ideal for the `CopyMemory` and `FillMemory` externs.

[illegible]


```

    int dwHeightInPixels
    )
{
    // in this code we do not need to indulge in "unsafe" pointer
    // manipulations because NV12 is planar ( Y plane, followed
    // by packed U-V plane), we can just copy the planes we need
    // and fill the rest with dummy data

    // Y plane
    for (int y = 0; y < dwHeightInPixels; y++)
    {
        CopyMemory(pDest, pSrc, dwWidthInPixels);
        pDest = new IntPtr(pDest.ToInt64() + lDestStride);
        pSrc = new IntPtr(pSrc.ToInt64() + lSrcStride);
    }

    // U-V plane
    for (int y = 0; y < dwHeightInPixels / 2; y++)
    {
        FillMemory(pDest, dwWidthInPixels, 0x80);
        pDest = new IntPtr(pDest.ToInt64() + lDestStride);
    }
}

Source: TantaTransformDirect::MFTTantaGrayscale_Sync::TransformImageOfTypeNV12

```

Note how the above function does not require the use of the `unsafe` keyword in the function header. The calling of extern functions is not considered “unsafe” by C#. Admittedly, it is rare to find a circumstance like this where the memory block manipulation externs can be used – however, keep the technique in mind. It may come in useful one day.

THE MEDIA BUFFER

Clearly the raw media data will have a length. Since the data storage area is reused by Windows Media Foundation there will also be a maximum possible length associated with the raw media data. The length of the raw media data can be less than the total amount of memory allocated. Any WMF entity that manipulates or processes this data will need to know basic things like the total length of the data and the maximum allocated memory size. The easy way to enforce this association is to create a container whose payload is both the raw media data and the basic information such as length and allocation.

The name for the container that has both the raw media data and other basic information regarding the raw data size is called a Media Buffer. A Media Buffer implements the `IMFMediaBuffer` interface. Although all of the previous examples in this section referred to video, it should be noted that audio data (and other Media Major Types) is also contained within Media Buffer objects. Media Buffers are a generic multi-use media data container.

So what functionality do you get in the `IMFMediaBuffer`? Just some pretty basic things really such as the ability to find out the size of the data, the maximum size and the ability to lock and unlock access to the raw data. It is the call to the `Lock()` function of

The WMF Components

the `IMFMediaBuffer` interface that returns an `IntPtr` to the raw data. The `Lock()` function also returns the size of the raw data and the maximum size as well. It is very important to call the `Unlock()` function when you are done with the data since nothing else will be able access or re-use the Media Buffer until you do.

```
public static void LockIMFMediaBufferAndGetRawData(IMFMediaBuffer mediaBuffer,
                                                    out IntPtr rawDataPtr,
                                                    out int maxLen,
                                                    out int currentLen)
{
    // init
    rawDataPtr = IntPtr.Zero;
    maxLen = 0;
    currentLen = 0;

    if (mediaBuffer == null) throw new Exception("mediaBuffer == null");

    // must call an UnlockIMFBuffer
    HRESULT hr = mediaBuffer.Lock(out rawDataPtr, out maxLen, out currentLen);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to mediaBuffer.Lock failed. Err=" + hr.ToString());
    }
    return;
}
```

Source: `TantaCommon::TantaWMFUtils::LockIMFMediaBufferAndGetRawData`

The code section above demonstrates the usage of the `Lock()` function on a Media Buffer. Note how an `IntPtr` to the raw data is returned along with the two length values.

A length and a maximum length are nice generic things to know about the raw data in a Media Buffer. However video data (which WMF deals with a lot) has its own additional requirement. Video is presented on the screen as a sequence of pixels of a certain length. The length of any one line of video pixels is called the “stride”. It is possible for the stride to be longer than the actual pixel width on the display if there are padding bytes added to the end. In order to accommodate video data in Media Buffers, the Windows Media Foundation team created an entity called a 2D Media Buffer. A 2D Media Buffer implements the `IMF2DBuffer` interface and this interface inherits from `IMFMediaBuffer`. In other words, a 2D Media Buffer is always a Media Buffer but not all Media Buffers are 2D Buffers.

Only video data is found in 2D Buffers and each 2D buffer will have one frame of video data. The `IMF2DBuffer` interface contains more functionality and its lock and unlock functions are called `Lock2D` and `Unlock2D`. If you call `Lock2D()` you will get the stride returned to you instead of the raw data length and maximum length. Note if you call `Lock2D()` you must call `Unlock2D()` to release the lock – calling the base class `Unlock()` on a buffer locked with a `Lock2D()` call will not do the job properly.

```
public static void LockIMF2DBufferAndGetRawData(IMF2DBuffer media2DBuffer,
                                                  out IntPtr rawDataPtr,
                                                  out int bufferStride)
{
```

```
// init
rawDataPtr = IntPtr.Zero;
bufferStride = 0;

if (media2DBuffer == null) throw new Exception("media2DBuffer == null");

// must call an UnlockIMF2DBuffer
HRESULT hr = media2DBuffer.Lock2D(out rawDataPtr, out bufferStride);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to media2DBuffer.Lock2D failed. Err=" + hr.ToString());
}
return;
}

Source: TantaCommon::TantaWMFUtils::LockIMF2DBufferAndGetRawData
```

The documentation recommends that, if you are dealing with an `IMF2DBuffer`, that you always use the `Lock2D` and `Unlock2D` locking functions – they are more efficient in dealing with non-contiguous data. There are also a number of other functions in the `IMF2DBuffer` interface which deal with the copy of the data to a contiguous format – these will not be discussed in this book. You can look them up if sufficiently interested.

CREATING A MEDIA BUFFER

It should be noted, in case it is not clear, that it is often possible to “*process in place*”. In other words, you don’t always have to copy the data to another buffer if you change it.

If you configure your Transform to use In-Place Processing (look up the `MFTInputStreamInfoFlags.ProcessesInPlace` flag) you can just make your changes to the existing raw data and do not have to create a new Media Buffer for the output. The `MFTTantaWriteText_Sync` Transform is an example of a Transform that does in-place processing and the `MFTTantaGrayscale_Sync` Transform is an example of one which copies the data to a new output buffer which is provided to it by the Media Session. Both of these Transforms are found in the `TantaTransformDirect` sample code.

However, there are occasions when you do have to create a new Media Buffer. There are several static WMF functions which can create a new Media Buffer for you but most of them are designed for pretty specialized circumstances. We will not discuss those oddball functions here – the one which is pretty much universally used is the `MFCCreateMemoryBuffer()` static function.

```
... more code

IMFMediaBuffer outputBuffer = null;

// Allocate an output buffer.
hr = MFExtern.MFCCreateMemoryBuffer(sourceSampleSize, out outputBuffer);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to MFCCreateMemoryBuffer failed. Err=" + hr.ToString());
}
if (outputBuffer == null)
{
    throw new Exception("call to MFCCreateMemoryBuffer failed. outputBuffer == null");
}
```

```
... more code  
Source: TantaCommon::TantaWMFUtils::CreateMediaSampleFromIntPtr
```

The code section above shows this function in operation. Note that you have to specify the size of the buffer when you create it. This is what the `sourceSampleSize` parameter (it is an `int`) is doing.

An `IMF2DBuffer` is similarly created with a call to the `MFCreat2DMediaBuffer()` static function. This function takes few more parameters such as the frame width, height and a FOURCC code but the usage is similar and will not be reproduced here.

THE MEDIA SAMPLE

One of the primary functions of the Media Session (see *The Pipeline* section in *The WMF Components* chapter) is to synchronize the flow of data through the Pipeline so that each branch presents its data to the Media Sink at the appropriate time. This ensures that the audio the user hears matches the video currently on display on the screen. It is easy to see that information regarding the “time of display” and “duration of display” information of the Media Buffer would be quite useful to have available as the media data makes its way through the Pipeline.

The “time of display” and “duration” type of presentation information is really not appropriate to store in the Media Buffer since the Media Buffer is purely designed to facilitate the transport of the raw media data. The “time of display” and “duration” meta-data are too high level and very dependent on the Media Major Type and Media Sub-Type.

The solution was to implement another container which wraps a Media Buffer (or buffers) and associates the Presentation type information with the contained Media Buffers. The name of this container is called a Media Sample and all Media Samples implement the `IMFMediaSample` interface. Note that unlike Media Buffers there is not a special Media Sample for video information. There is no need, a Media Sample is also an Attribute Container and any significant configuration details are easily stored there if required.

A Media Sample can be thought of as an object that contains an ordered list of zero or more Media Buffers and the Presentation meta-data describing them. Why might more than one buffer be present in a Media Sample? Well, if the data is streaming in over a network, the Media Source might decide to place all of the data that it has into a single Media Sample. In such an event, the Media Source might not try to collect all the data into a single buffer and will instead just place multiple buffers in the same sample.

Other than functions to get or set the sample time and duration, the `IMFMediaSample` interface mostly concerns itself with the manipulation of the Media Buffers it contains. There are functions to add a buffer, remove a buffer, count them, enumerate them and, possibly most usefully, coalesce all of the buffers into one. To be realistic it is unusual to see more than one Media Buffer in a Media Sample. In addition, if you are dealing with video data, it would be unusual to see a Media Buffer that contained more than one video frame. It is possible, and the standard requires support for other types of behavior, but people rarely like to introduce such complexity into their applications.

Media Samples can also contain a multitude of Attributes which specify the intricate details of how the Media Buffers are to be processed. These Attributes are all mostly pretty obscure and so they will not be discussed here - you can easily look them up if you need to.

CREATING A NEW MEDIA SAMPLE

Theoretically, the creation of a Media Sample is a simple process – just call the Windows Media Foundation static function `MFCreatSample`. In reality, while the creation is a simple, the process of populating the new Media Sample can get quite complex. The code section below shows the process of creating and populating a Media Sample with data from a provided `IntPtr`.

```
/// ++++++
/// <summary>
/// Creates a IMFSample from a IntPtr to Buffer data. Will throw an exception
/// for anything it does not like. Always creates a sample with an
/// IMFMediaBuffer
/// </summary>
/// <param name="sourceSampleFlags">the sample flags</param>
/// <param name="sourceSampleSize">the sample size</param>
/// <param name="sourceSampleDuration">the sample duration</param>
/// <param name="sourceSampleTimeStamp">the sample time</param>
/// <param name="sourceSampleIntPtr">the source IntPtr</param>
/// <param name="sourceAttributes">the attributes for the sample - we copy these</param>
/// <history>
///     01 Nov 18  Cynic - Ported in
/// </history>
public static IMFSample CreateMediaSampleFromIntPtr(int sourceSampleFlags,
                                                    long sourceSampleTimeStamp,
                                                    long sourceSampleDuration,
                                                    IntPtr sourceSampleIntPtr,
                                                    int sourceSampleSize,
                                                    IMFAttributes sourceAttributes)
{
    HRESULT hr;
    IMFSample outputSample = null;
    IMFMediaBuffer outputBuffer = null;
    IntPtr destRawDataPtr = IntPtr.Zero;

    try
    {
        // Create a new sample
        hr = MFExtern.MFCreatSample(out outputSample);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to MFCreatSample failed. Err=" + hr.ToString());
        }
        if (outputSample == null)
        {

```

The WMF Components

```
        throw new Exception("call to MFCreatSample failed. outputSample == null");
    }

    // Allocate an output buffer.
    hr = MFExtern.MFCreatMemoryBuffer(sourceSampleSize, out outputBuffer);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to MFCreatMemoryBuffer failed. Err=" + hr.ToString());
    }
    if (outputBuffer == null)
    {
        throw new Exception("call to MFCreatMemoryBuffer failed. outputBuffer == null");
    }

    // add the buffer to the sample
    hr = outputSample.AddBuffer(outputBuffer);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to AddBuffer failed. Err=" + hr.ToString());
    }

    // get a pointer to the raw data from the output buffer. We need this in
    // order to copy the input raw data across
    int maxLen = 0;
    int currentLen = 0;
    TantaWMFUtils.LockIMFMediaBufferAndGetRawData(outputBuffer,
                                                    out destRawDataPtr,
                                                    out maxLen, out currentLen);

    // now that we have the input data and a pointer to the destination area
    // do the work to copy it across.
    CopyMemory(destRawDataPtr, sourceSampleIntPtr, sourceSampleSize);

    // Set the data size on the output buffer.
    hr = outputBuffer.SetCurrentLength(sourceSampleSize);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to SetCurrentLength failed. Err=" + hr.ToString());
    }

    // Set the sample time stamp
    hr = outputSample.SetSampleTime(sourceSampleTimeStamp);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to SetSampleTime failed. Err=" + hr.ToString());
    }

    // set the sample duration
    hr = outputSample.SetSampleDuration(sourceSampleDuration);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to SetSampleDuration failed. Err=" + hr.ToString());
    }

    // set the attributes
    if (sourceAttributes != null)
    {
        if ((outputSample is IMFAttributes) == true)
        {
            sourceAttributes.CopyAllItems((outputSample as IMFAttributes));
        }
    }
}
finally
{
    TantaWMFUtils.UnlockIMFMediaBuffer(outputBuffer);

    if (outputBuffer != null)
    {
        Marshal.ReleaseComObject(outputBuffer);
        outputBuffer = null;
    }
}
return outputSample;
}
```

Source: TantaCommon::TantaWMFUtils::CreateMediaSampleFromIntPtr

Now that you understand the “nested” container mechanism of Media Samples it can be seen that the above code proceeds fairly logically. First the new Media Sample is created and then a Media Buffer is created for the new data.

```
hr = MFExtern.MFCreateMemoryBuffer(sourceSampleSize, out outputBuffer);
```

Note that the use of the `MFCreateMemoryBuffer()` call implies that a generic `IMFMediaBuffer` is being used here. If you needed a 2D Buffer you would have to edit this line to call `MFCreate2DMediaBuffer()`. The size of the Media Buffer is specified as a parameter on the call to the `MFCreateMemoryBuffer()`. Since the Media Sample needs to contain the Media Buffer, a call to `AddBuffer()` on the newly created output sample places it there.

```
hr = outputSample.AddBuffer(outputBuffer);
```

The only way to get access to the `IntPtr` of the destination memory in the newly created Media Buffer is to lock it and this is done with a call to the `LockIMFMediaBufferAndGetRawData()` wrapper function in the *TantaWMFUtils* library. Recall from our earlier discussion that the locking method is also different for `IMF2DBuffers`, if you were using those you would have to change this line to call `LockIMF2DBufferAndGetRawData()`. Now that we have a source and destination `IntPtr` (the source `IntPtr` is supplied as a parameter), the next line simply calls the `CopyMemory` extern.

```
CopyMemory(destRawDataPtr, sourceSampleIntPtr, sourceSampleSize);
```

This is a very good use for this external function call since it excels at shifting large blocks of memory quickly. The next lines set the Sample Time and the Duration on the Media Sample. The association of these two items with the Media Buffer is essential – the Media Sample is useless without them.

```
hr = outputSample.SetSampleTime(sourceSampleTimeStamp);  
hr = outputSample.SetSampleDuration(sourceSampleDuration);
```

If any Attributes were supplied as parameters to the function call, the final section of code copies them over into the Attributes of the new Media Sample. The `CopyAllItems()` call is part of the `IMFAttributes` interface.

```
sourceAttributes.CopyAllItems((outputSample as IMFAttributes));
```

Note that the `Unlock` call is placed in the `finally` block of the function. This ensures that it always gets called even if exceptions are thrown. In particular, observe that the newly created Media Buffer is also released in the `finally` block – this must be done or you will get a memory leak.

```
Marshal.ReleaseComObject(outputBuffer);
```

The WMF Components

Don't worry that the Media Sample is still using the Media Buffer you just released, the act of adding the Media Buffer to the Media Sample caused an additional reference to be placed on it. The Media Buffer will not actually go anywhere until that reference is also released by the Media Sample. Of course, the new Media Sample will have to be released as well – the caller of the function is responsible for handling that.

The `TantaWMFUtils` class also contains a function entitled `CreateMediaSampleFromBuffer` which can create a new Media Sample from an existing Media Buffer.

```
/// ++++++
/// <summary>
/// Creates a IMFSample from IMFMediaBuffer data. Will throw an exception
/// for anything it does not like
/// </summary>
/// <param name="sourceSampleFlags">the sample flags</param>
/// <param name="sourceSampleSize">the sample size</param>
/// <param name="sourceSampleDuration">the sample duration</param>
/// <param name="sourceSampleTimeStamp">the sample time</param>
/// <param name="sourceSampleBuffer">the sample buffer</param>
/// <param name="sourceAttributes">the attributes for the sample - we copy these</param>
/// <history>
/// 01 Nov 18 Cynic - Ported in
/// </history>
public static IMFSample CreateMediaSampleFromBuffer(int sourceSampleFlags,
                                                    long sourceSampleTimeStamp, long sourceSampleDuration,
                                                    IMFMediaBuffer sourceSampleBuffer, int sourceSampleSize,
                                                    IMFAttributes sourceAttributes)
{
    IntPtr srcRawDataPtr = IntPtr.Zero;
    bool srcIs2D = false;
    int srcStride;

    // in C# the actual video data is down in the unmanaged heap. We have to get
    // an intptr to the data in order to copy it. In C# this involves a bit
    // of marshaling
    try
    {
        // Lock the input buffer. Use the IMF2DBuffer interface
        // (if available) as it is faster
        if ((sourceSampleBuffer is IMF2DBuffer) == false)
        {
            // not an IMF2DBuffer - get the raw data from the IMFMediaBuffer
            int maxLen = 0;
            int currentLen = 0;
            TantaWMFUtils.LockIMFMediaBufferAndGetRawData(sourceSampleBuffer,
                                                         out srcRawDataPtr, out maxLen, out currentLen);
            // now make the call to the version of this function which accepts
            // only IntPtrs as the source
            return CreateMediaSampleFromIntPtr(sourceSampleFlags,
                                                sourceSampleTimeStamp, sourceSampleDuration,
                                                srcRawDataPtr, sourceSampleSize, sourceAttributes);
        }
        else
        {
            // we are an IMF2DBuffer, we get the stride here as well
            TantaWMFUtils.LockIMF2DBufferAndGetRawData((sourceSampleBuffer as IMF2DBuffer),
                                                         out srcRawDataPtr, out srcStride);
            srcIs2D = true;

            // now make the call to the version of this function which accepts
            // only IntPtrs as the source
            return CreateMediaSampleFromIntPtr(sourceSampleFlags, sourceSampleTimeStamp,
                                                sourceSampleDuration, srcRawDataPtr,
                                                sourceSampleSize, sourceAttributes);
        }
    }
    finally
    {
        if (srcIs2D == false) TantaWMFUtils.UnlockIMFMediaBuffer(sourceSampleBuffer);
        else TantaWMFUtils.UnlockIMF2DBuffer((sourceSampleBuffer as IMF2DBuffer));
    }
}
```



```
}
Source: TantaCommon::TantaWMFUtils::CreateMediaSampleFromIntPtr
```

As can readily be observed in the above code, the `CreateMediaSampleFromBuffer` function does little more than call the appropriate lock function to obtain an `IntPtr` for the raw data in the Media Buffer. Once it has that, the Media Sample can readily be created with a call to the previously discussed `CreateMediaSampleFromIntPtr` function. The operation of the `LockIMFMediaBufferAndGetRawData` function used above was described in the previous *Creating a Media Buffer* section and will not be discussed here.

CALLBACK OBJECTS

To start this section, it is probably best to discuss what a Callback Object is and why they are needed. Once some background is provided, the discussion of the use of Callback Objects in Windows Media Foundation will make a lot more sense.

You probably already know how C# has a really clever and useful Delegate/Event system which can permit other objects to “*sign up*” to receive alerts or information from some other component in the application? Well, C++ does not really have that functionality and Windows Media Foundation is primarily designed to be used with C++. What C++ does have is the ability to pass the address of an object to another component. If this object is coupled with an interface definition, then the calling object will know exactly the name and number of functions available in the object and the type of parameters to supply to each. Thus, if the calling component calls a specified function when an event or other message is to be passed, then the information will make it back into a function in the Callback Object and the interested party can probably use it or take action based on it. That’s essentially what a Callback is – the address of an object to call and the knowledge of the interface that specifies the functions in the Callback Object. Think of it as a less refined Delegate/Event mechanism where the names and parameters of the functions are hard coded as an interface.

Windows Media Foundation uses Callback Objects (sometimes called Callback Handlers) for a variety of purposes and it is important to realize that there are numerous interfaces defined for this purpose. For example, Callback Objects used by the Media Session must implement the `IMFAsyncCallback` interface, Callback Objects used by the Source Reader (in Asynchronous Mode) must implement the `IMFSourceReaderCallback` and Callback Objects used by the Sample Grabber Sink must implement the `IMFSampleGrabberSinkCallback` interface. These are just the

The WMF Components

ones used in the Tanta Sample Applications – there are many others. The point here is that ...

Windows Media Foundation defines a large number of Callback interfaces. They are not the same and, in general, are specific to a single component. When you work with Callbacks, do not get them mixed up with each other.

Sometimes the use of a Callback Object is mandatory. For example, the Media Session requires an `IMFAsyncCallback` Object and the component will not work without it. Other times the Callback Object is optional. For example, if you do not specify an `IMFSourceReaderCallback` Callback Object to a Source Reader it will just assume you wish to work in Synchronous Mode.

Be aware that in much of the example source code you see on the Internet, the object that supplies the Callback Object is itself the Callback Object. In other words the object supplies its own address and implements the appropriate Callback interface functions. This makes for a curious sort of re-entrant looping code the logic of which can be hard to follow. The Tanta Samples do not use this structure. In the Tanta Sample Projects, the Callback Object is always a separate object (see the `TantaAsyncCallbackHandler`, `TantaSourceReaderCallbackHandler` and `TantaSampleGrabberSinkCallback` classes). Those classes are designed to be as easy as possible to understand and the Callback Object generally passes events and information back to the owner application via standard C# Events and Delegates. Not everybody does it this way – keeping this fact in mind will greatly simplify your understanding of other example code.

Another important issue to be aware of, when dealing with actions triggered by a Callback Object, is that the calling entity almost certainly is working on its own pool of threads. The code in the Callback Object will not be executing on the thread that originally configured the Callback Object.

Calls made into the functions of a Callback Object will almost certainly be on a different thread than your applications main thread. This means in an action triggered by a function in a Callback Object, you MUST take all multi-thread based precautions. For example, do not access forms or controls from within it and watch out for locking issues.

This is not to say that you can never update the screen from a Callback Event. All it really means is that you have to use the standard C# `Invoke` constructs to ensure that you are back on the main form thread before you do. The code section below taken from the *TantaAudioFileCopyViaPipelineAndWriter* Sample Project illustrates how errors in the Sample Grabber Sink are presented to the screen.

```

/// ++++++
/// <summary>
/// Handles error reports from the AsyncCallbackHandler. Note you CANNOT
/// assume this is called from within the form thread.
/// </summary>
/// <param name="caller">the Callback Object obj</param>
/// <param name="errMsg">the error message</param>
/// <param name="ex">the exception (if there is one) that generated the error</param>
/// <history>
/// 01 Nov 18 Cynic - Started
/// </history>
public void HandleSampleGrabberAsyncCallbackErrors(object caller, string errMsg, Exception ex)
{
    // log it - the logger is thread safe!
    if (errMsg == null) errMsg = "unknown error";
    LogMessage("HandleSampleGrabberAsyncCallbackErrors, errMsg=" + errMsg);
    if (ex != null)
    {
        LogMessage("HandleSampleGrabberAsyncCallbackErrors, ex=" + ex.Message);
        LogMessage("HandleSampleGrabberAsyncCallbackErrors, ex=" + ex.StackTrace);
    }

    // Ok, you probably already know this but I'll note it here because this is so important
    // You do NOT want to update any form controls from a thread that is not the forms main
    // thread. Very odd, intermittent and hard to debug problems will result. Even if your
    // handler does not actually update any form controls do not do it! Sooner or later you
    // or someone else will make changes that calls something that eventually updates a
    // form or control and then you will have introduced a really hard to find bug.

    // So, we always use the InvokeRequired...Invoke sequence to get us back on the form thread
    if (InvokeRequired == true)
    {
        // call ourselves again but this time be on the form thread.
        Invoke(new TantaSampleGrabberSinkCallback.SampleGrabberAsyncCallbackError Delegate(
            HandleSampleGrabberAsyncCallbackErrors),
            new object[] { this, errMsg, ex });
        return;
    }

    // if we get here we are assured we are on the form thread.

    // do everything to close all media devices
    CloseAllMediaDevices();
    buttonStartStopCopy.Text = START_COPY;
    // re-enable our screen controls
    SyncScreenControlsToCopyState(false, null);

    // tell the user
    if (ex != null) OISMessageBox("There was an error processing.\n\n" + ex.Message);
    else if (errMsg != null)
    {
        OISMessageBox("There was an error processing the audio stream.\n\n" + errMsg);
    }
    else
    {
        OISMessageBox("There was an unknown error processing the audio stream ");
    }
}

```

Source: *TantaAudioFileCopyViaPipelineAndWrite::frmMain::HandleSampleGrabberAsyncCallbackErrors*

Just for completeness, and so you have an example which may be useful in your own code, let's review how the `HandleSampleGrabberAsyncCallbackErrors` function shown above is called. The function is located in the `frmMain` class of the

The WMF Components

TantaAudioFileCopyViaPipelineAndWriter Sample Project and it is set as an event handler on the Sample Grabber Sink Callback Object.

```
sampleGrabberSinkCallback.SinkWriter = workingSinkWriter;
sampleGrabberSinkCallback.InitForFirstSample();
sampleGrabberSinkCallback.SampleGrabberAsyncCallBackError =
    HandleSampleGrabberAsyncCallBackErrors;
```

Source: *TantaAudioFileCopyViaPipelineAndWrite::frmMain::PrepareSessionAndTopology*

Inside the *TantaSampleGrabberSinkCallback* class, the *SampleGrabberAsyncCallBackError* event is structured as a standard C# Event and Delegate as shown below

```
// our error reporting delegate
public delegate void SampleGrabberAsyncCallBackError_Delegate(object obj,
                                                             string errMsg, Exception ex);
public SampleGrabberAsyncCallBackError Delegate SampleGrabberAsyncCallBackError = null;
```

Source: *TantaAudioFileCopyViaPipelineAndWrite::TantaSampleGrabberSinkCallback*

The major processing function in the *TantaSampleGrabberSinkCallback* class is *OnProcessSampleEx()* function and any errors in it will trigger a call to the *SampleGrabberAsyncCallBackError* event as shown below

```
... more code

// we have all the information we need to create a new output sample
outputSample = TantaWMFUtils.CreateMediaSampleFromIntPtr(sampleFlags, sampleTimeStamp,
sampleDuration, sampleBuffer, sampleSize, null);
if (outputSample == null)
{
    string errMsg = "Error on call to CreateMediaSampleFromBuffer outputSample == null";
    SampleGrabberAsyncCallBackError(this, errMsg, null);
    return HRESULT.E_FAIL;
}

... more code
```

Source: *TantaAudioFileCopyViaPipelineAndWrite::TantaSampleGrabberSinkCallback*

The *OnProcessSampleEx()* function is certainly not on the main form thread, and neither will be the handler when the error event is called. A careful review of the code in the *HandleSampleGrabberAsyncCallBackErrors* function will show how the *InvokeRequired()* function detects this and a subsequent *Invoke()* call is used to recursively call the *HandleSampleGrabberAsyncCallBackErrors* function again – this time on the main form thread.

THE MEDIA SESSION CALLBACK OBJECT

Media Session is essentially asynchronous – it performs its processing on multiple threads and you, the programmer, have no access to this. So, if the Media Session is merrily working along in the background, how then can it provide information about events of interest to the application? Well, you have probably guessed it – the Media

Session uses a Callback Object. The alternative would be to have your application poll the Media Session for event or status changes - nobody wants to do that.

Callback Objects intended for use with Media Sessions must implement the `IMFAsynCallback` interface. It should be noted that the Tanta Sample Applications use the `TantaAsynCallbackHandler` class which is derived from, and implements, that interface. All projects in the Tanta Sample code that use a Media Session, use this Callback Object – it is generic.

The `IMFAsynCallback` interface defines a number of functions but the only one of interest to us here is named `Invoke`. Please note the name “*Invoke*” has nothing to do with the C# cross-threading mechanism also named “*Invoke*” – in this interface it is just an unfortunate naming collision. Unlike many other Callback Objects which expect the programmer to undertake some processing, the Media Session uses its Callback Object purely to pass messages. Any error, thrown by any object, anywhere in the Pipeline will appear in the `Invoke` function of the Media Sessions Callback Object. Similarly, any event of note such as `MESessionStarted`, `MESessionPaused`, `MESessionStopped`, `MESessionClosed` and over a hundred others (see the `MediaEventType` enum in MF.Net library) will also appear here. Most of these events can be ignored, most of the time, and you will find that the Tanta Sample programs only intercept the events they really need.

There are lots of threads going on inside the Media Session and it takes care not to call its Callback Object function simultaneously. If you take certain precautions you can be sure you will always only get one event at a time. You can get quite a sequential blizzard of them however, so any processing you do which is triggered by the Media Session Callback Object should be pretty quick. The event process is regulated by the calls your applications makes to the Media Sessions `BeginGetEvent()` and `EndGetEvent()` functions and via a C# `Lock()` you obtain. You call `BeginGetEvent()` to let the Media Session that you can receive events and you call `EndGetEvent()` functions to let it know that it can no longer send events and you do this inside of a `Lock()` to make sure that nothing slips by. If we have a look at the `Invoke` function of the `TantaAsynCallbackHandler` class we can see this happening.

```
/// ++++++
/// <summary>
/// Part of the IMFAsynCallback interface. This is called when an
/// asynchronous operation is completed.
/// </summary>
/// <param name="pResult">Pointer to the IMFAsyncResult interface. </param>
/// <returns>S_OK for success, others for fail</returns>
/// <history>
/// 01 Nov 18 Cynic - Originally Written
/// </history>
HRESULT IMFAsynCallback.Invoke(IMFAsyncResult pResult)
{
```

The WMF Components

```
HResult hr;
IMFMediaEvent eventObj = null;
MediaEventType meType = MediaEventType.MEUnknown; // Event type
HResult hrStatus = 0; // Event status

lock (this)
{
    try
    {
        if (MediaSession == null) return HResult.S_OK;

        // Complete the asynchronous request this is tied to the previous
        // BeginGetEvent call and MUST be done. The output here is a pointer to the
        // IMFMediaEvent interface describing this event. Note we MUST
        // release this interface
        hr = MediaSession.EndGetEvent(pResult, out eventObj);
        if (hr != HResult.S_OK)
        {
            throw new Exception("call to EndGetEvent failed. Err=" + hr.ToString());
        }
        if (eventObj == null)
        {
            throw new Exception("call to EndGetEvent failed. eventObj == null");
        }

        // Get the event type. The event type indicates what happened to trigger the event.
        // It also defines the meaning of the event value.
        hr = eventObj.GetType(out meType);
        if (hr != HResult.S_OK)
        {
            throw new Exception("call to GetType failed. Err=" + hr.ToString());
        }

        // Get the event status. If the operation that generated the event was successful,
        // the value is a success code. A failure code means that an error condition
        // triggered the event.
        hr = eventObj.GetStatus(out hrStatus);
        if (hr != HResult.S_OK)
        {
            throw new Exception("call to GetStatus failed. Err=" + hr.ToString());
        }

        // Check if we are being told that the the async event succeeded.
        if (hrStatus != HResult.S_OK)
        {
            // The async operation failed. Notify the application
            if (MediaSessionAsyncCallbackError != null)
                MediaSessionAsyncCallbackError(this, "Error Code =" + hrStatus.ToString(), null);
        }
        else
        {
            // we are being told the operation succeeded and therefore the event
            // contents are meaningful. Switch on the event type.
            switch (meType)
            {
                {
                    // we let the app handle all of these. There is not really
                    // much we can do here
                    default:
                        MediaSessionAsyncCallbackEvent(this, eventObj, meType);
                        break;
                }
            }
        }
    }
    catch (Exception ex)
    {
        // The async operation failed. Notify the application
        if (MediaSessionAsyncCallbackError != null)
            MediaSessionAsyncCallbackError(this, ex.Message, ex);
    }
    finally
    {
        // Request another event if we are still operational.
        if (((meType == MediaEventType.MESessionClosed)
            || (meType == MediaEventType.MEEndOfPresentation)) == false)
        {
            // Begins an asynchronous request for the next event in the queue
            hr = MediaSession.BeginGetEvent(this, null);
            if (hr != HResult.S_OK)
            {
                throw new Exception("call to BeginGetEvent failed. Err=" + hr.ToString());
            }
        }
    }
}
```

```

        // release the event we just processed
        if (eventObj != null)
        {
            Marshal.ReleaseComObject(eventObj);
        }
    } // bottom of lock(this)

    return HRESULT.S_OK;
}

```

Source: `TantaAudioFileCopyViaPipelineMP3Sink::TantaAsyncCallbackHandler::Invoke`

The first thing the `Invoke` function in the Media Session Callback Object does is create a lock and it uses itself as a handy lock object

```

lock (this)
{
    ... code here
}

```

The next step is to call the Media Session `EndGetEvent()` function to prevent any more events being generated.

```
hr = MediaSession.EndGetEvent(pResult, out eventObj);
```

The `EndGetEvent()` function does more than just inhibit any more event generation – it also gives us an event object. This object is of type `IMFMediaEvent` and we can use it to determine whether we are dealing with a “notification” type event or with an “error” type event. We do this by calling the `GetStatus()` function of the `IMFMediaEvent` interface.

```
hr = eventObj.GetStatus(out hrStatus);
```

We also call the `GetType()` function of the `IMFMediaEvent` interface for use later. We don’t need it if the event is an error but we will if the event is of the “notification” variety. Once we have the Event Type and Event Status we can branch on the output of the `GetStatus()` function (the `hrStatus` value in the above code).

```

if (hrStatus != HRESULT.S_OK)
{
    // The async operation failed. Notify the application
    if (MediaSessionAsyncCallbackError != null)
        MediaSessionAsyncCallbackError(this, "Error Code =" + hrStatus.ToString(), null);
}

```

If the `hrStatus` value is not `HRESULT.S_OK` we are dealing with an error and simply call the error `MediaSessionAsyncCallbackError` handling event. If we are not dealing with an error, the code simply calls the `MediaSessionAsyncCallbackEvent` event.

```

switch (meType)
{
    // we let the app handle all of these. There is not really much we can do here
    default:
        MediaSessionAsyncCallbackEvent(this, eventObj, meType);
        break;
}

```

As you can see the call to the `MediaSessionAsyncCallbackEvent` event is wrapped in a switch statement but this is mostly for later use. The function in the application which

The WMF Components

processes the `MediaSessionAsyncCallbackEvent` event determines which events it will take notice of and which it will not. The two functions in the application which handle the error and notification events will not be discussed in this section – the *Implementing the Pipeline Architecture* section in the *Practical WMF Architectures* chapter will provide a more detailed discussion.

It should be noted that the only reason the `TantaAsyncCallbackHandler` class knows about the event handlers is because it was told that information just prior to the Callback Object being given to the Media Session.

```
// set up our media session Callback Object.
mediaSessionAsyncCallbackHandler = new TantaAsyncCallbackHandler();
mediaSessionAsyncCallbackHandler.Initialize();
mediaSessionAsyncCallbackHandler.MediaSession = mediaSession;
mediaSessionAsyncCallbackHandler.MediaSessionAsyncCallbackError =
HandleMediaSessionAsyncCallBackErrors;
mediaSessionAsyncCallbackHandler.MediaSessionAsyncCallbackEvent =
HandleMediaSessionAsyncCallBackEvent;

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::PrepareSessionAndTopology
```

Please realize that this is just the way the Tanta Sample Applications pass information back from the Callback Object. Not all applications do it this way (particularly the C++ ones).

However, the `Invoke` function of the `TantaAsyncCallbackHandler` class is still not complete. The events may have been passed on the application but as far as the Media Session is concerned it still cannot send any. We resolve this issue by making a `BeginGetEvent()` call to let the Media Session that another event can be processed – if required.

```
// Request another event if we are still operational.
if ((meType == MediaEventType.MESessionClosed)
    || (meType == MediaEventType.MEEndOfPresentation)) == false)
{
    // Begins an asynchronous request for the next event in the queue
    hr = MediaSession.BeginGetEvent(this, null);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("Invoke call to BeginGetEvent failed. Err=" + hr.ToString());
    }
}
```

Note that we only turn events on again in the Media Session if the Media Session is operational. Thus any `MESessionClosed` or `MEEndOfPresentation` event will inhibit further event processing (after those events are sent to the application of course).

So, one remaining question is “*how did the events get enabled on the Media Session in the first place*”? Well, if you look at the code where the Media Session is built (the `PrepareSessionAndTopology` function in the Tanta Sample Applications) you will see a `BeginGetEvent()` call as is shown below.

```
// Register the Callback Object with the session and tell it that events can
// start. This does not actually trigger an event it just lets the media session
```



```
// know that it can now send them if it wishes to do so.
hr = mediaSession.BeginGetEvent(mediaSessionAsyncCallbackHandler, null);
if (hr != HRESULT.S_OK)
{
    throw new Exception("all to mediaSession.BeginGetEvent failed. Err=" + hr.ToString());
}

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::PrepareSessionAndTopology
```

It should be pointed out, in case it did not leap out at you in the code block above, that the initial `BeginGetEvent()` call also registers the Callback Object in the Media Session and from that point on the Media Session will send events to it. One of the most important of these will be the `MESessionTopologyStatus` event. Among other things, this event can tell us that the Pipeline has started – but that is a topic for another chapter.

One final thing to note in the example `Invoke` function above is that the `BeginGetEvent()` call is placed in a `finally` block. This enables us to cope with errors – forgetting to enable the next event will effectively just lock up the Media Session.

```
// release the event we just processed
if (eventObj != null)
{
    Marshal.ReleaseComObject(eventObj);
}
```

As shown in the code block above, also note that the event object obtained from the initial call to `EndGetEvent()` is released. Don't forget to do things like this or you will get memory leaks. The rule is *"if you obtain an object from Windows Media Foundation (however you do it) then you are responsible for cleaning it up"*. Since you are new to this, and likely a bit confused at the moment, also note that the `meType` variable is a simple enum (of `MediaEventType`), not a COM object, and so does not need to be released.

SOURCE READER AND SINK WRITER

Up to this point we have not talked very much about the components used in the alternative Windows Media Foundation Reader-Writer or Hybrid Architectures. The sections below which discuss the Source Reader and Sink Writer components will rectify that.

It appears that the Source Reader and Sink Writer components were originally introduced to provide media access to the Universal Windows Platform (UWP) API. UWP is a technology provided by Microsoft which enables an application to be universally executable on any platform on which Windows runs. In other words, you would write your application once and then it will run on PC's, tablets, phones and anything else running Windows. As a "universal" program, the visual interface and internal

components are expected to adapt to a wide range of environments. This “universal” requirement has pretty serious consequences for a technology like the Media Session and Pipeline which is highly specific to the PC Windows environment.

Clearly a Universal Windows Platform will require methods of media access – after all, lots of people want to play or record videos on their phone. The WMF Pipeline Architecture was thought to be far too complex to ever work on all platforms. So the solution was to write a general, relatively easy to use, component which could act as a source for media data and supply that information in a specified Media Type regardless of the format a physical device might provide it in. This is, of course, is the Source Reader. A similar requirement exists for an easy to use component which can write data to a disk file in a specified format without regard to the Media Type in which the data was supplied. The Sink Writer was created to fill that need.

The Source Reader and Sink Writer were introduced to fill a specific need in the Universal Windows Platform technology for media components that could supply or sink a stream of data. In order to make things simple, these components were also designed to contain a variety of internal mechanisms which can automatically modify the input or output data according to a specified Media Type.

Since the Source Reader and Sink Writer exist on the Windows platform to support UWP and they deal with complex media operations in a simple way, there is no real good reason not to associate them with Windows Media Foundation. After all they use many of the same technologies (such as Media Samples and Media Types), they are useful and they can provide some interesting functionality which enhances the technology.

So, there really is a perfectly good explanation as to why the Source Reader and Sink Writer are present in Windows Media Foundation even though they are so wildly different than the rest of the architecture. One issue to be aware of is that, because the Source Reader and Sink Writer components were relatively easy to use, people tended to make them their first choice to solve their problems. This is a good thing since the problems got solved, but a bad thing in that much of the sample code you may find on the Internet implements just those two components. You will find many fewer examples available which use the Media Session and Pipeline Architecture – although that situation is changing somewhat now.

USING THE SOURCE READER

The Source Reader is designed to interact with a device or file and provide you with a stream of Media Samples containing Media Buffers with the raw data. The Source Reader can support multiple streams so it is possible to have video and audio data coming off the same file.

With the Source Reader you get a stream of media data in the form of standard Media Samples - it is up to you to deal with this data as you wish.

One common destination for the Source Reader data stream is to give it to a Sink Writer. Thus it is possible to read from a video device (such as a webcam) and write that data to a file. The *TantaCaptureToFileViaReaderWriter* Sample Project does exactly this. Note, however, that your application could do anything with the data once the Source Reader has handed it over - it does not have to immediately give it to a Sink Writer. For example, your application could display the data on the screen or play it via audio. You would have to write custom components for this though – there is no Microsoft supplied Sink Writer component which can do either of these things. The Sink Writer only writes to files.

On the PC platform, the Source Reader internally opens and maintains a standard WMF Media Source. This is why, when you configure the Source Reader, things look slightly similar (but not identical) to the way you would interact with a Media Source. In reality, the Source Reader is just passing the operations through to its own internal Media Source. This can be a source of frustration for people new to Windows Media Foundation when they see one example program obtaining media data in a one way and a second doing something completely different. In reality, one example of the sample code is probably using a Media Source and the other is using a Source Reader – they are not the same thing.

A Source Reader component is not a Media Source. It offers the `IMFSourceReader` interface not the `IMFMediaSource` interface. It cannot be used in a Pipeline and the Media Session cannot work with it.

A Source Reader is an entirely standalone entity and is most useful in situations where your application needs a simple way of getting a stream of data from a device or file. It can sometimes be a viable alternative to using the Media Session and the Microsoft

Media Foundation Pipeline to process media data. The Source Reader encapsulates a lot of functionality you would otherwise need to handle yourself.

OBTAINING MEDIA SAMPLES FROM A SOURCE READER

Once the Source Reader has been configured, how do you get access to the media data? Well, ultimately it is simple. Your application simply makes repeated calls to the `ReadSample()` function on the `IMFSourceReader` interface. The following incomplete code section shows this process.

```
// we sit in a loop here and get the sample from the source reader and write it out
// to the sink writer. An EOS (end of sample) value in the flags will signal the end.
while (true)
{
    int actualStreamIndex;
    MF_SOURCE_READER_FLAG actualStreamFlags;
    long timeStamp = 0;
    IMFSample workingMediaSample = null;

    // Request the next sample from the media source. Note that this could be
    // any type of media sample (video, audio, subtitles etc). We do not know
    // until we look at the stream ID. We saved the stream ID earlier when
    // we obtained the media types and so we can branch based on that.
    hr = sourceReader.ReadSample(
        TantaWMFUtils.MF_SOURCE_READER_ANY_STREAM,
        0,
        out actualStreamIndex,
        out actualStreamFlags,
        out timeStamp,
        out workingMediaSample
    );
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("Failed on ReadSample, retVal=" + hr.ToString());
    }

    ... more code
}

Source: TantaVideoFileCopyViaReaderAndWriter::frmMain::CopyFile
```

The above code block is incomplete as it is rather lengthy and the part not displayed is mostly associated with the detection of the end of the video stream. However, particularly note that the `ReadSample()` function returns the ID of the stream it is reading but not the `IMFMediaStream` object itself – that belongs to the internal Media Source and is hidden. This is done because you may have opened up multiple streams on the Source Reader and may have to handle the data in them differently. The stream index (`actualStreamIndex`) allows you to direct the Media Sample to the correct location. It would be a good idea to review the full code in the `CopyFile` function of the *TantaVideoFileCopyViaReaderAndWriter* Sample Application to get a sense of how it works.

SYNCHRONOUS VS ASYNCHRONOUS SOURCE READERS

If your application is sitting in a loop and repeatedly calling the `ReadSample()` then it is likely that it will appear to be frozen or locked up to external users. From a design point

of view this is rarely ok. Most of the time you want the `ReadSample()` processing to continue in a separate thread so that the GUI of the application remains operational. There are multiple ways of doing this in C# – not the least of which is placing the loop code in a separate thread. However, recall that the Source Reader is designed for UWP and so clearly something more generic was required. Thus the Asynchronous Source Reader model was developed.

When the application obtains Media Samples directly from the Source Reader, the Source Reader is said to be operating in the *Synchronous Mode*. If a Callback Object implementing the `IMFSourceReaderCallback` interface is given to the Source Reader, then a call to `ReadSample()` will trigger a call to a function in the Callback Object to handle the processing of the Media Sample. The Callback Object itself is expected to trigger the next the `ReadSample()` call so that the process can continue. In this mode the Source Reader is said to be operating *Asynchronously*.

The Asynchronous Mode was designed to provide a simple method of processing the Media Samples in a separate thread (and the Source Reader provides the thread). That requirement is somewhat redundant in an environment like C#, which has ample multi-threading capabilities of its own, but Asynchronous Mode is still used. In fact the *TantaCaptureToFileViaReaderWriter* Sample Project implements this architecture for reference purposes.

CREATING A SOURCE READER

There are three ways of creating a Source Reader. You can...

1. Create the Source Reader directly from a file name using the static `MFCreatSourceReaderFromURL` function.
2. Create a Source Reader from an existing Media Source using the static `MFCreatSourceReaderFromMediaSource` function.
3. Create a Source Reader from a byte stream using the static `MFCreatSourceReaderFromByteStream` function.

The sections that follow will demonstrate the process of creating a Source Reader on a file and on a device. The creation of a Source Reader from a byte stream will not be

discussed. That particular technique is not commonly required and none of the Tanta Sample Projects use it.

All of the methods of creating a Source Reader listed above are also applicable to the creation of a Source Reader in Asynchronous Mode. The sole difference is that a Callback Object is provided in the Attributes parameter in any of the `MFCCreateSourceReaderFrom...` static functions and you will get back an object implementing the `IMFSourceReaderAsync` interface. Don't worry, an `IMFSourceReaderAsync` object is still an `IMFSourceReader` – the `IMFSourceReaderAsync` interface inherits directly from the `IMFSourceReader`. You can observe how the Callback Object is passed in as an Attribute in the *Creating a Source Reader on a File* section below. It is not explicitly stated in the documentation (so it will be stated here) that if you do not provide a Callback Object when the Source Reader is created, then the Source Reader is, by default, a Synchronous Mode Source Reader.

CREATING A SOURCE READER ON A FILE

If you have a file, and have decided to use a Source Reader to process it, then a call to the `MFCCreateSourceReaderFromURL` function is the way to go. The *TantaVideoFileCopyViaReaderWriter* Sample Project provides a full demonstration of this and the sample code block below shows the creation process.

```
/// ++++++
/// <summary>
/// Opens the Source Reader object
/// </summary>
/// <param name="inFileName">the filename we write read from</param>
/// <param name="wantAllowHardwareTransforms">if true we allow hardware transforms</param>
/// <returns>a IMFSourceReader object or null for fail</returns>
/// <history>
/// 01 Nov 18 Cynic - Started
/// </history>
public static IMFSourceReader CreateSourceReaderSyncFromFile(string inputFileName,
                                                            bool wantAllowHardwareTransforms)
{
    HRESULT hr;
    IMFSourceReader workingReader = null;
    IMFAttributes sourceReaderAttributes = null;

    if ((inputFileName == null) || (inputFileName.Length == 0))
    {
        // we failed
        throw new Exception("CreateSourceReaderSyncFromFile: Invalid filename specified");
    }

    try
    {
        // create the attribute container we use to create the source reader
        hr = MFExtern.MFCCreateAttributes(out sourceReaderAttributes, 1);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("Failed MFCCreateAttributes, retVal=" + hr.ToString());
        }
        if (sourceReaderAttributes == null)
        {
            // we failed
            throw new Exception(": Failed to create Source Reader Attributes");
        }
    }
}
```

```

        hr = sourceReaderAttributes.SetUINT32(
            MFAttributesClsid.MF_READWRITE_ENABLE_HARDWARE_TRANSFORMS,
            wantAllowHardwareTransforms ? 1 : 0);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("Failed on call to SetUINT32, retVal=" + hr.ToString());
        }

        // Create the SourceReader. This takes the URL of an input file
        // creates a media source internally.
        hr = MFExtern.MFCreateSourceReaderFromURL(inputFileName,
            sourceReaderAttributes, out workingReader);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("Failed MFCreateSourceReaderFromURL, retVal=" + hr.ToString());
        }
        if (workingReader == null)
        {
            // we failed
            throw new Exception("Failed to create Source Reader");
        }
    }
    catch (Exception ex)
    {
        // note this clean up is in the Catch block not the finally block.
        // if there are no errors we return it to the caller. The caller
        // is expected to clean up after itself
        if (workingReader != null)
        {
            // clean up. Nothing else has this yet
            Marshal.ReleaseComObject(workingReader);
            workingReader = null;
        }
        workingReader = null;
        throw ex;
    }
    finally
    {
        if (sourceReaderAttributes != null)
        {
            Marshal.ReleaseComObject(sourceReaderAttributes);
        }
    }
    return workingReader;
}

```

Source: TantaCommon::TantaWMFUtils::CreateSourceReaderSyncFromFile

The creation process is quite straightforward really. Note that, in this particular example, the function has the capability of setting an Attribute which enables or disables the use of hardware transforms. The provision of this Attribute at creation time is entirely optional and, if it is not present, a default will be assumed. The Source Reader will automatically load hardware or software Transform objects (if it can) to ensure that the media data it reads from the input file or device is output in the correct Media Type. *The Source Reader and Format Conversions* section below discusses this topic in more detail.

Just to emphasize, observe that in the above code block, no Callback Object was provided and hence the Source Reader will operate in Synchronous Mode. This is why the newly created object is returned as an `IMFSourceReader`.

CREATING A SOURCE READER ON A DEVICE

If a Source Reader is to be created on a physical device the usual route is to create a Media Source on the device and then generate a Source Reader from that. This is usually the only time a Source Reader is generated from a Media Source. While it certainly is possible to create a Source Reader from a Media Source when interacting with a file – it is just a lot more cumbersome and most people don't bother. Be aware that you will sometimes see the Source Reader from a file Media Source technique used in WMF example code available on the Internet – but none of the Tanta Samples will do this.

The *TantaCaptureToFileViaReaderWriter* Sample Project creates a Source Reader from a temporary Media Source based on a webcam physical device. This process is shown in the code block below.

```

/// ++++++
/// <summary>
/// Opens the up a SourceReader in asynch mode.
///
/// NOTE: It is the callers responsibility to clean up and properly dispose
///       of the SourceReader object returned here.
///
/// </summary>
/// <param name="sourceDevice">the Device we use for the source</param>
/// <param name="asyncCallbackHandlerIn">the Callback Object for async mode</param>
/// <returns>an IMFSourceReaderAsync object or null for fail</returns>
/// <history>
///     01 Nov 18  Cynic - Started
/// </history>
public static IMFSourceReaderAsync CreateSourceReaderAsyncFromDevice(
    TantaMFDevice sourceDevice,
    IMFSourceReaderCallback asyncCallbackHandlerIn)
{
    HRESULT hr = HRESULT.S_OK;
    IMFMediaSource mediaSource = null;

    IMFAttributes attributeContainer = null;

    if (sourceDevice == null)
    {
        throw new Exception(": Null source device specified. Cannot continue.");
    }
    if (asyncCallbackHandlerIn == null)
    {
        throw new Exception("asyncCallbackHandlerIn != null");
    }
    try
    {
        // use the device symbolic name to create the media source for the device.
        // Media sources are objects that generate media data.
        mediaSource = TantaWMFUtils.GetMediaSourceFromTantaDevice(sourceDevice);
        if (mediaSource == null)
        {
            throw new Exception("mediaSource == null. Cannot continue.");
        }

        // Initialize an attribute store. The 2 is the number of initial
        // attributes which can be stored
        hr = MFExtern.MFCreateAttributes(out attributeContainer, 2);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("failed MFCreateAttributes, retVal=" + hr.ToString());
        }

        // Set our Callback Object as an IUnknown pointer in the attribute container.
        hr = attributeContainer.SetUnknown(MFAttributesClsid.MF_SOURCE_READER_ASYNC_CALLBACK,

```



```

                                asyncCallbackHandlerIn);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("failed SetUnknown, retVal=" + hr.ToString());
    }
    // Create the SourceReader. We will no longer need our media source object
    // after this, the SourceReader will maintain its own pointer into it
    // and will clean it up properly when it is closed down.
    IMFSourceReader sourceReader;
    hr = MFExtern.MFCreateSourceReaderFromMediaSource(mediaSource,
                                                    attributeContainer, out sourceReader);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("failed creating source reader, retVal=" + hr.ToString());
    }
    return (IMFSourceReaderAsync)sourceReader;
}
finally
{
    // make sure we release the attribute memory
    if (attributeContainer != null)
    {
        Marshal.ReleaseComObject(attributeContainer);
        attributeContainer = null;
    }

    // close and release the source device
    if (mediaSource != null)
    {
        Marshal.ReleaseComObject(mediaSource);
        mediaSource = null;
    }
}
}
}

Source: TantaCommon::TantaWMFUtils::CreateSourceReaderAsyncFromDevice

```

In the above code block a Callback Object was provided and hence the Source Reader will operate in Asynchronous Mode and the newly created object is returned as an `IMFSourceReaderAsync`. The Callback Object is provided as an Attribute. The `MF_SOURCE_READER_ASYNC_CALLBACK` GUID is the key and the object itself is the value. This is the purpose of the lines of code below...

```

hr = attributeContainer.SetUnknown(MFAttributesClsid.MF_SOURCE_READER_ASYNC_CALLBACK,
                                asyncCallbackHandlerIn);

```

Also note the cast of returned value to an `IMFSourceReaderAsync`. The `MFCreateSourceReaderFromMediaSource` static function always returns an `IMFSourceReader` object even though it really is an `IMFSourceReaderAsync`.

THE SOURCE READER AND FORMAT CONVERSIONS

If you need to perform complex or custom operations on the media data (transform it) or render it to a video or audio device, then you should probably use a Media Source and the Pipeline Architecture.

Having said that, the Source Reader is perfectly prepared to convert media data from one format to another if it can find the correct Transforms which will do the job. This is what makes the Source Reader relatively easy to use – lots of things are done for you automatically.

The Transforms used are determined by the Media Type of the input data and the Media Type you have specified for the output format. Their use is entirely automatic, internal to the Source Reader, and you have no control over them. For example, if the underlying Media Source delivers compressed data, the Source Reader may well need to decode the data to match the specified output format. In that case, the Source Reader will find and load the correct decoder and manage the data flow between the Media Source and the decoder (internally it would act like a Media Session). The Source Reader can also perform some limited video DSP type processing such as a conversion from NV12 to YUV format and similar.

The process of configuring the Media Types on the Source Reader is discussed in the *Implementing the Reader-Writer Architecture* section of the *Practical WMF Architectures* chapter and so will not be reproduced here.

SINK WRITER

Given all the knowledge you now have from the sections above regarding the Source Reader, you can probably guess where we are going with the Sink Writer. The Sink Writer is pretty much an analogue of the Source Reader except that it writes out data instead of sourcing it.

Similarly, the Sink Writer is not an `IMFMediaSink` – it implements the `IMFSinkWriter` interface and maintains its own Media Sink internally. It will also load its own Transforms, if it can, to automatically convert between the specified input format and the format being written to disk. As mentioned previously, there is no Sink Writer which renders data (video to the screen or audio to the speaker). The Sink Writer only operates on files.

The Sink Writer exists only to consume data and write it to a file on disk. Your application is required to provide this data. The Sink Writer is incompatible with the Media Session and Pipeline.

The Sink Writer is often teamed up with the Source Reader and, in such cases, your application simply runs in a loop pulling the data off of the Source Reader and handing it over to the Sink Writer. It should be emphasized that the use of a Source Reader to provide data to the Sink Writer is optional. Your application could generate the data (perhaps it is an animation application) or it could implement a Media Session and Pipeline which has a special component (the Sample Grabber Sink) which copies the

Media Samples as they pass through and gives them to the Sink Writer. In that case you will have implemented the Hybrid Architecture. Note that the use of a Sink Writer in a Hybrid Architecture does not imply that the Sink Writer is part of the Pipeline - it is not. The Sink Writer is still very much outside the Pipeline and is simply being manually provided with Media Samples by a Pipeline object.

Be aware that there is no concept of a synchronous or asynchronous Sink Writer – those are purely Source Reader concepts. All Sink Writers are effectively synchronous.

PROVIDING MEDIA SAMPLES TO A SINK WRITER

Once you have the Media Sample, then writing the data out to the Sink Writer is a pretty straightforward process. You just call the `WriteSample()` function on the `IMFSinkWriter` interface. Of course, there is more to it than that though. A proper Sink Writer loop needs to perform specific actions on the first Media Sample written to the Media Sink. It also needs to watch for signals which indicate the stream is complete - otherwise the loop would be endless and the output file would not get shutdown properly. The code section below shows a part of a loop which writes Media Samples to a Sink Writer.

```
// the sample may be null if either end of stream or a stream tick is returned
if (workingMediaSample == null)
{
    // just ignore, the flags will have the information we need.
}
else
{
    // the sample is not null
    if (actualStreamIndex == sourceReaderAudioStreamId)
    {
        // audio data
        // ensure discontinuity is set for the first sample in each stream
        if (audioSamplesProcessed == 0)
        {
            // audio data
            hr = workingMediaSample.SetUINT32(
                MFAttributesClsid.MFSampleExtension_Discontinuity, 1);
            if (hr != HRESULT.S_OK)
            {
                // we failed
                throw new Exception("Failed SetUINT32 on the sample, retVal=" + hr.ToString());
            }
            // remember this - we only do it once
            audioSamplesProcessed++;
        }
        hr = sinkWriter.WriteSample(sinkWriterOutputAudioStreamId, workingMediaSample);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("Failed WriteSample on the writer, retVal=" + hr.ToString());
        }
    }

    // release the sample
    if (workingMediaSample != null)
    {
        Marshal.ReleaseComObject(workingMediaSample);
        workingMediaSample = null;
    }
}
}
```

Source: `TantaAudioFileCopyViaReaderWriter::frmMain::CopyFile`

The WMF Components

The first thing to note is that the Media Sample can be null. In the above sample code, the Media Samples are being generated by a Source Reader. Besides the Media Sample, the read of the data on a Source Reader returns a set of flags and the Stream ID. If the stream is at an end (no more data) then the Media Sample will be null and the flags will indicate the event status.

If a non-null Media Sample is present, the `actualStreamIndex` value is checked to see if it matches the index of the audio stream. This stream ID will also have been returned when the Media Sample is read. If only one stream has been opened, then this check is somewhat redundant – every Media Sample will be on the single open stream. It is possible for there to be multiple streams though (see the *TantaVideoFileCopyViaReaderWriter* Sample Project).

The other major check is for the first Media Sample on the stream. The Sink Writer (or rather more likely the internal Media Sink) expects a “Discontinuity” flag to be set for the very first Media Sample it sees. It needs this to trigger some internal set-up for the data. As you can see, the Discontinuity Flag is set as an Attribute on the Media Sample itself.

```
hr = workingMediaSample.SetUINT32(MFAttributesClsid.MFSampleExtension_Discontinuity, 1);
```

If there are multiple streams, be sure to check for and set the Discontinuity flag on the first Media Sample in each stream.

The actual write of the Media Sample is just a simple call to the `WriteSample()` function.

```
hr = sinkWriter.WriteSample(sinkWriterOutputAudioStreamId, workingMediaSample);
```

Note that the ID of the stream is also passed in on this call. In some Internet example code you will sometimes see this value hard coded to 0. This is not really a good practice as it assumes there will never be more than one output stream on the Sink Writer.

CREATING A SINK WRITER

There are two ways of creating a Sink Writer. You can...

1. Create the Sink Writer directly from a file name using the static `MFCCreateSinkWriterFromURL` function.
2. Create a Sink Writer from an existing Media Sink using the static `MFCCreateSinkWriterFromMediaSink` function.

In reality, the requirement to create a Sink Writer from a Media Sink is a pretty rare occurrence. Since the Sink Writer only operates on files and the

`MFCCreateSinkWriterFromURL` function is so much easier to use, everybody pretty much does it that way. You may see the odd example on the Internet doing it the hard way but none of the Tanta Sample Projects do so. The `MFCCreateSinkWriterFromMediaSink` mechanism will not be discussed in this book.

It should be noted that the `MFCCreateSinkWriterFromURL` function can also accept a pointer to a byte stream – so it is also useful for transporting media data by that mechanism.

CREATING A SINK WRITER ON A FILE

The sample code block below clipped from the *TantaAudioFileCopyViaReaderWriter* Sample Project demonstrates the process of creating a Sink Writer on a file.

```
//+=====  
/// <summary>  
/// Opens the Sink Writer object  
/// </summary>  
/// <param name="outputFileName">the filename we write out to</param>  
/// <param name="wantAllowHardwareTransforms">if true we allow hardware transforms</param>  
/// <returns>an IMFSinkWriter object or null for fail</returns>  
/// <history>  
///     01 Nov 18   Cynic - Started  
/// </history>  
public static IMFSinkWriter CreateSinkWriterFromFile(string outputFileName,  
    bool wantAllowHardwareTransforms)  
{  
    HRESULT hr;  
    IMFSinkWriter workingWriter = null;  
    IMFAttributes sinkWriterAttributes = null;  
  
    if ((outputFileName == null) || (outputFileName.Length == 0))  
    {  
        // we failed  
        throw new Exception("CreateSinkWriterFromFile: Invalid filename specified");  
    }  
  
    try  
    {  
        // create the attribute container we use to create the source reader  
        hr = MFExtern.MFCreateAttributes(out sinkWriterAttributes, 1);  
        if (hr != HRESULT.S_OK)  
        {  
            // we failed  
            throw new Exception("Failed MFCreateAttributes, retVal=" + hr.ToString());  
        }  
        if (sinkWriterAttributes == null)  
        {  
            // we failed  
            throw new Exception("Failed on Attributes, Nothing will work.");  
        }  
        hr = sinkWriterAttributes.SetUINT32(  
            MFAttributes.Clsid.MF_READWRITE_ENABLE_HARDWARE_TRANSFORMS,  
            wantAllowHardwareTransforms ? 1 : 0);  
        if (hr != HRESULT.S_OK)  
        {  
            // we failed  
            throw new Exception("Failed on call to SetUINT32, retVal=" + hr.ToString());  
        }  
  
        // Create the sink writer. This takes the URL of an output file or a pointer  
        // to a byte stream and creates the media sink internally. You could also  
        // use the more round-about MFCreateSinkWriterFromMediaSink which takes a  
        // pointer to a media sink that has already been created by the application.  
        // If you are using one of the built-in media sinks, the MFCreateSinkWriterFromURL  
        // function is preferable, because the caller does not need to configure  
        // the media sink.  
        hr = MFExtern.MFCreateSinkWriterFromURL(outputFileName, null,
```

The WMF Components

```
        sinkWriterAttributes, out workingWriter);

    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("Failed MFCreatSinkWriterFromURL, retVal=" + hr.ToString());
    }
    if (workingWriter == null)
    {
        // we failed
        throw new Exception("Failed to create Sink Writer, Nothing will work.");
    }
}
catch (Exception ex)
{
    // note this clean up is in the Catch block not the finally block.
    // if there are no errors we return it to the caller. The caller
    // is expected to clean up after itself
    if (workingWriter != null)
    {
        // clean up. Nothing else has this yet
        Marshal.ReleaseComObject(workingWriter);
        workingWriter = null;
    }
    workingWriter = null;
    throw ex;
}
finally
{
    if (sinkWriterAttributes != null)
    {
        Marshal.ReleaseComObject(sinkWriterAttributes);
    }
}
return workingWriter;
}

Source: TantaCommon::TantaWMFUtils::CreateSinkWriterFromFile
```

The code above is pretty straight-forward and is easy to follow. The only oddity is that this particular function also has a flag to enable or disable hardware Transforms.

```
hr = sinkWriterAttributes.SetUINT32(MFAttributesClsid.MF_READWRITE_ENABLE_HARDWARE_TRANSFORMS,
    wantAllowHardwareTransforms ? 1 : 0);
```

As with the Source Reader, the provision of this Attribute at creation time is entirely optional and if it is not present a default will be assumed. The Sink Writer will automatically load hardware or software Transform objects (if it can) to ensure that the media data it accepts on its input is converted to the correct format on the output file according to the configured Media Type. *The Sink Writer and Format Conversions* section below discusses this topic in more detail.

One important point to note, which is not referenced in the above code blocks, is that a call to the `BeginWriting()` function of the Sink Writer must be made before any data is given to it with a `WriteSample()` call.

```
// begin writing on the sink writer
hr = sinkWriter.BeginWriting();
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("CopyFile: failed on call to BeginWriting, retVal=" + hr.ToString());
}

Source: TantaAudioFileCopyViaReaderWriter::frmMain::CopyFile
```

The `BeginWriting()` call must be made after the input streams are configured but before the any data is sent to the Sink Writer. You will see this in use when the full Reader-Writer Architecture is discussed in the *Implementing the Reader-Writer Architecture* section of the *Practical WMF Architectures* chapter.

THE SINK WRITER AND FORMAT CONVERSIONS

The Sink Writer is designed to be simple to use. You tell it the Media Type of the data you are giving it and you tell it the Media Type of the data you wish to have written to the output file. The Sink Writer will automatically convert the input media data for you if it can find the correct Transforms which will do the job.

Like the Source Reader, the Transforms used by the Sink Writer are determined by the Media Type of the input data and the Media Type you have specified for the output format. Their use is entirely automatic, internal to the Sink Writer, and you have no control over them.

The process of configuring the Media Types on the Sink Writer is discussed in the *Implementing the Reader-Writer Architecture* section of the *Practical WMF Architectures* chapter and so will not be reproduced here.

Windows Media Foundation: Getting Started in C#

Chapter 6

WMF – FIRST CONTACT

In this section we will make the first contact with a full Windows Media Foundation application – previous to this the only examples you have seen were short code sections illustrating various WMF concepts.

The discussion in this section revolves around the *TantaVideoFormats* Sample Application which is designed to enumerate the Video Capture Devices (webcams) on the system and allow the user to choose one. In addition, the application will dig into the selected Video Capture Device and display the many Media Types and formats offered by each.

For those of you whose first language is not English, strictly speaking, “*enumerate*” means “*to count*”. However, when used in the programming sense, it usually means “*find all of*” or “*look at each one of*”.

This chapter will make use of the WMF tools (Interfaces, Attributes, HResults and GUIDs etc.) discussed in the *MF.Net Programming Fundamentals* chapter. It will also make extensive use of some of the concepts (Media Source, Media Streams, Media Types etc.) which you encountered in *The WMF Components* chapter. If you are not familiar with any of those items you should probably review the relevant chapter now.

THE TANTAVIDEOFORMATS SAMPLE APPLICATION

In the discussion that follows, the *TantaVideoFormats* sample application will be used to demonstrate the required concepts. The full source code for this application, including a C# solution, is available for download – see the discussion in *The Tanta Sample Code* appendix for more details.

Here is a general overview of what is going to happen...

1. All of the functionality will take place within a control named `ctlTantaVideoPicker`. This permits other applications to simply drop that control onto a form in order to offer the ability to “pick” a Video Capture Device and select a format from those on offer by that device.
2. The Video Capture Devices on the system will be enumerated on application startup and the list of the names of those devices will be used to populate a dropdown combo box on the control.
3. The first Video Capture Device in the list will be the default, but the user can pick a different device if they wish.
4. When the user picks a device, a temporary Media Source will be built on that device.
5. The Media Source will provide a Presentation Descriptor.
6. The Presentation Descriptor can be enumerated to find a list of the Media Streams on offer.
7. The first Media Stream with a Media Major Type of “video” will be chosen. In order to keep things simple, this application ignores other Media Streams.
8. A Media Type Handler will be obtained on the first video Media Stream.
9. The Media Type Handler will be used to enumerate the Media Types offered by the stream.
10. Information on each Media Type will be stored in a container and the Media Type will be released.
11. The containers of media type information will be collected in a list.
12. The list of media type information will be presented to the user in a ListView from which they can select a format.

This approach has the advantage of introducing many of Windows Media Foundation concepts in a fairly simple way. Ok, let’s be realistic, if you are new to WMF programming it probably will not look all that simple to you. In reality though, it is just a

sequence of logical steps – none of which are particularly complex. You may wish to take some comfort from the fact that your first attempt at WMF programming does not involve setting up a Pipeline and Topology and that the concepts you learn in this section will actually be of considerable use later on.

THE VIDEO PICKER CONTROL

Since the discovery of video source devices and their capabilities is a useful thing, the code for this operation has been embedded into a control named `ctlTantaVideoPicker` in the `TantaCommon` library so that it can be re-used by other applications.

The `ctlTantaVideoPicker` control is designed to be easily placed on the screen to provide a “pick-list” of device and format options for the user. The `TantaVideoFormats` application does not do anything with the picked options – it just displays them. This is

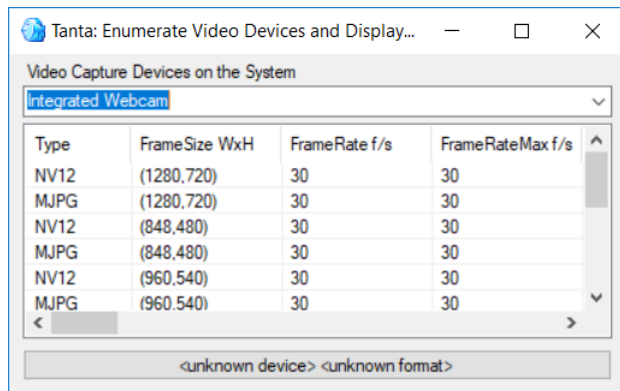


Figure 6.1: The TantaVideoFormats Sample Application

just fine for our purposes since, at this point, we are much more interested in how to obtain this information rather than how to use it once we do have it. As you look at the other Tanta Sample Projects, you will find that several of them use the `ctlTantaVideoPicker` control to enable the user to select a video device and format. The `ctlTantaVideoPicker` control will not be discussed in those sections.

Like all C# form based programs, the main form for the `TantaVideoFormats` sample application is launched from the `Main()` function in the `Program.cs` file. Note that the `Main()` function in the project is decorated with the `[MTAThread]` tag. This is not obvious and `[STAThread]` is the default. If you leave the `[STAThread]` tag in place, pretty much nothing will work and you will not get a sensible error message explaining why.

IMPORTANT: Always use an `[MTAThread]` tag just above your `Main()` function when using Windows Media Foundation. If you do not do this then very little else will work.

If you need a more in-depth discussion of reasons for this then review the *You must use an [MTAThread] Code Decoration* section in the *MF.Net Programming Fundamentals* chapter. The code for the `Main()` function of the *TantaVideoFormats* application is shown below.

```
/// <summary>
/// The main entry point for the application.
/// </summary>
///
///
///
// SUPER IMPORTANT NOTE: You MUST use [MTAThread] here. If you use [STAThread] you
// will get the following error
//   Unable to cast COM object of type 'System. ComObject' to interface type
//   'MediaFoundation.Alt.IMFSourceReaderAsync'. This operation failed because the
//   QueryInterface call on the COM component for the interface with IID
//   '{70AE66F2-C809-4E4F-8915-BDCB406B7993}' failed due to the following error: No such
//   interface supported (Exception from HRESULT: 0x80004002 (E_NOINTERFACE)).
[MTAThread]
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new frmMain());
}

Source: TantaVideoFormats:Main
```

The constructor for the *TantaVideoFormat* sample application main form starts off with the configuration of the standard Tanta sample application logging substrate and the only other operation of consequence it performs is to initialize Windows Media Foundation. This is *absolutely* necessary.

```
// we always have to initialize MF. The 0x00020070 here is the WMF version
// number used by the MF.Net samples. Not entirely sure if it is appropriate
hr = MFExtern.MFStartup(0x00020070, MFStartup.Full);
if (hr != 0)
{
    LogMessage("Constructor: call to MFExtern.MFStartup returned " + hr.ToString());
}

Source: TantaVideoFormats:frmMain:frmMain
```

It is the `frmMain_Load()` event, triggered when the form is fully available, that initiates the process of obtaining the Video Capture Device information.

```
private void frmMain_Load(object sender, EventArgs e)
{
    LogMessage("frmMain_Load");
    ctlTantaVideoPicker1.DisplayVideoCaptureDevices();
}

Source: TantaVideoFormats:frmMain:frmMain_Load
```

As can be seen in the above code section, the call to the `DisplayVideoCaptureDevices()` function in the `ctlTantaVideoPicker1` control transfers the thread of execution into that object and it is there that the real work is performed. It is to this sequence of code that we will now turn our attention. The contents of the `DisplayVideoCaptureDevices()` member function is shown in a future section (after a short digression).

For completeness it should also be noted that the `frmMain_Closing()` handler is automatically invoked to deal with the shutdown of MF.Net when the application closes. That function also calls, as all Tanta Sample Applications do, the `CloseAllMediaDevices()` function - although in this particular case there is nothing that needs to be done.

```
private void frmMain_FormClosing(object sender, FormClosingEventArgs e)
{
    LogMessage("frmMain_FormClosing");
    try
    {
        // do everything to close all media devices
        CloseAllMediaDevices();

        // Shut down WMF
        MFExtern.MFShutdown();
    }
    catch
    {
    }
}

Source: TantaVideoFormats::frmMain::frmMain_Closing
```

Important though it is, this is the last time we are going to mention the need to use and `[STAThread]` in your WMF applications, WMF initialization and WMF shutdown. From now on you should simply assume that every Tanta Sample Application implements these (because they all do).

ENUMERATING THE VIDEO CAPTURE DEVICES

A Windows system on a modern laptop computer will typically have one Video Capture Device – this is usually the built-in camera, however, a computer can have multiple devices or none at all. If there are multiple Video Capture Devices, usually at least one is a plug-in USB video camera. In Windows Media Foundation it does not matter if the Video Capture Device is built-in or is a plug-in via USB - the treatment of the device is identical. WMF interacts with the device driver via standard protocols and physical characteristics of the device connection are largely abstracted away.

As mentioned previously, the process of enumerating the devices on the system has been split into two parts. The first part is a call to the `DisplayVideoCaptureDevices()` member function which, after calling the `GetDevicesByCategory()` static function, simply populates a combo box with the returned list of the Video Capture Devices.

```
public void DisplayVideoCaptureDevices()
{
    StringBuilder sb = new StringBuilder();

    // Query MF for the devices, can also use MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_AUDCAP_GUID
    // here to see the audio capture devices
    List<TantaMFDevice> vcDevices = TantaWMFUtils.GetDevicesByCategory(
        MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE,
        CLSID.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_GUID);
    if (vcDevices == null) return;
```

```

foreach (TantaMFDevice mfDevice in vcDevices)
{
    sb.Append("FriendlyName:" + mfDevice.FriendlyName);
    sb.Append("\r\n");
    sb.Append("Symbolic Name:" + mfDevice.SymbolicName);
    sb.Append("\r\n");
    sb.Append("\r\n");
}
// add all known devices
comboBoxCaptureDevices.DataSource = vcDevices;
}
Source: TantaCommon::ctlTantaVideoPicker::DisplayVideoCaptureDevices

```

The actual acquisition of the Video Capture Devices on the system is a pretty straight forward procedure. It is the subsequent examination of the contents of that device and its multiple capabilities that gets somewhat intricate.

When we enumerate a device on the system we do not actually get the device object itself, what we receive is an object, called an Activator (see the *WMF Object Creation is Indirect* section in the *MF.Net Programming Fundamentals* chapter for more information), that knows how to create the enumerated device. We could use the Activator to build the object immediately – and, indeed, many of the WMF sample programs do exactly that. However, we want the user to be able to pick from multiple devices and so what we will do is extract two text tags from the Activator and once we have these values, we release the Activator. We can always acquire a new Activator for a specific device later using the label tags.

One tag is known as the “*Friendly Name*” and it is just a text string suitable for display to the user. The other tag is called the “*Symbolic Name*” and it is effectively a link or registry address which tells WMF and the COM layer how to find the dedicated Activator for the device should it be necessary. Since there can be multiple devices on any one system, the two tags are stored in the *TantaMFDevice* class. All this class really does is act as a container to keep the Friendly Name and Symbolic Name for a device together so they can be easily be passed around the system without losing the association between the two.

It is the call to the static `GetDevicesByCategory()` function in the *TantaWMFUtils* class that really does the work and we shall examine that method shortly. For now, have a look back at the previous source code section and note that the two parameters to the `DisplayVideoCaptureDevices()` member function call are the GUID `MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE` and the GUID `CLSID.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_GUID`. The relevant section of code is shown below.

```

List<TantaMFDevice> vcDevices = TantaWMFUtils.GetDevicesByCategory(
    MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE,
    CLSID.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_GUID);

```

If you recall the much earlier discussion on the usage of GUID values, you will realize that these two parameters are just keys to tell WMF which information is required. In this case all we are really saying is “*look among the source devices*” and “*we want video capture devices*”. We could just as well have passed in the `MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_AUDCAP_GUID` GUID and retrieved a list of audio source devices (microphones).

In any event, the `GetDevicesByCategory()` function is just a wrapper for the multi-step process of configuring the request to enumerate the specified devices. The source code for this function is listed below. Remember that Windows Media Foundation code always looks a lot more complicated than it really is. The steps in the code section below, if performed in a more typical C# fashion, would probably be achieved in about six lines of code.

```
public static List<TantaMFDevice> GetDevicesByCategory(Guid attributeType, Guid filterCategory)
{
    // our return value
    List<TantaMFDevice> outList = new List<TantaMFDevice>();
    IMFActivate[] deviceArr;
    int numDevices=0;
    HRESULT hr = 0;
    IMFAttributes attributeContainer = null;

    try
    {
        // Initialize an attribute store. We will use this to
        // specify the enumeration parameters.
        hr = MFExtern.MFCreateAttributes(out attributeContainer, 1);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("failed on call to MFCreateAttributes");
        }
        if (attributeContainer == null)
        {
            // we failed
            throw new Exception("attributeContainer == MFAttributesClsid.null");
        }

        // populate the attribute container
        hr = attributeContainer.SetGUID(attributeType, filterCategory);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("failed setting up the attributes, retVal=" + hr.ToString());
        }

        // Enumerate the devices.
        hr = MFExtern.MFEnumDeviceSources(attributeContainer, out deviceArr, out numDevices);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("failed on call to MFEnumDeviceSources");
        }
        if (deviceArr == null)
        {
            // we failed
            throw new Exception("deviceArr == MFAttributesClsid.null");
        }

        // add the devices to our list as TantaMFDevices
        for (int i = 0; i < numDevices; i++)
        {
            // extract the friendlyName and symbolicLinkName
            string symbolicLinkName = GetStringForKeyFromActivator(
                deviceArr[i],
                MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_SYMBOLIC_LINK);
        }
    }
}
```

```

        string friendlyName = GetStringForKeyFromActivator(
            deviceArr[i],
            MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_FRIENDLY_NAME);

        // create the new TantaMFDevice
        outList.Add(new TantaMFDevice(friendlyName, symbolicLinkName,
            filterCategory));

        // clean up our activator
        Marshal.ReleaseComObject(deviceArr[i]);
    }
}
finally
{
    // make sure we release the attribute memory
    if (attributeContainer != null)
    {
        Marshal.ReleaseComObject(attributeContainer);
    }
}
return outList;
}

```

Source: TantaCommon::TantaWMFUtils::GetDevicesByCategory

Recall that, ultimately, the point of all the above code is to get a list of devices from Windows Media Foundation. Before we can do that, we have to tell WMF which devices we wish to have. In this case, the type of the devices we wish to look at are supplied as parameters to the `GetDevicesByCategory()` function call. These two parameters are the two GUID values discussed earlier and they actually form a key-value pair. In other words, they are an Attribute. In order to pass information in to WMF we need to formally load that information into an `IMFAttributes` container since most WMF functions will not accept a key-value pair as parameters. So we first make a container to hold our Attribute data, then we populate the Attribute container with our GUID key-value pair and then we pass that container in as a parameter on the `MFEnumDeviceSources` call.

```

// Initialize an attribute container
hr = MFExtern.MFCreateAttributes(out attributeContainer, 1);
// populate the attribute container
hr = attributeContainer.SetGUID(attributeType, filterCategory);

```

Thus, all we are really doing in the first three groups of code, is building a container for the Attribute (the `MFCreateAttributes()` call) and then setting the GUIDs in that container as an Attribute key-value pair (the `SetGUID()` call). Of course, including comments and error checking, this takes twenty lines of code – but don't worry about that too much. Once you get the hang of it you will soon not notice the verbosity.

```

// Enumerate the devices.
hr = MFExtern.MFEnumDeviceSources(attributeContainer, out deviceArr, out numDevices);

```

Once we have the populated our attribute container, a simple call to the `MFEnumDeviceSources` static function provides us with the appropriate list of devices. These devices are returned to us in the `deviceArr` variable as an array of Activator objects (an `IMFActivate[]`). At that point it is a simple matter to sit in a `for` loop and

extract both the Friendly Name and Symbolic Name for the device from the Activator object and store those tags in a `TantaMFDevice` class.

```
// clean up our activator
Marshal.ReleaseComObject(deviceArr[i]);
```

In particular, note how the Activator object is released inside the `for` loop with a call to `Marshal.ReleaseComObject()`. It is very important to do this or you will get a memory leak - the documentation does mention it as well. If you were keeping the Activator around you would have to remember to release it later or you would get the same problem.

In addition, note how the `IMFAttributes` container (the `attributeContainer` variable) is also released in the finally block. This is typical of Windows Media Foundation. If WMF retains the attribute container internally it will add a reference and release it itself. Thus there is a firm rule that if you create a WMF object, you are responsible for releasing it as well. Note that getting an array of Activators from the `MEnumDeviceSources` function also counts as creating them. After all, Windows Media Foundation has no idea when or where you might need to release those Activator objects so it is up to you to do so.

While we are on the subject of releasing things, note that the strings containing the Friendly Name and Symbolic Name do not have to be released. The act of digging it out of the Activator copied it into a proper C# string and releasing the Activator will take care of the original.

```
public static string GetStringForKeyFromActivator(IMFActivate activatorContainer, Guid guidKey)
{
    string allocatedStr = "";
    HRESULT hr = 0;
    int iSize = 0;

    if (activatorContainer == null) return "";

    // get it now.
    hr = activatorContainer.GetAllocatedString(
        guidKey,
        out allocatedStr,
        out iSize
    );
    if (hr != HRESULT.S_OK) return "";

    // sanity check
    if (allocatedStr == null) allocatedStr = "";
    return allocatedStr;
}
```

Source: `TantaCommon::TantaWMFUtils::GetStringForKeyFromActivator`

Before we proceed onto other things, we should probably discuss the contents of the `GetStringForKeyFromActivator` function. This call is the one we use to fetch the Friendly Name and Symbolic Name from the Activator. This function is, as can readily be seen in Figure 6.2, just a simple wrapper for a call to the Activators

`GetAllocatedString` call. This static utility function just avoids the need to place two

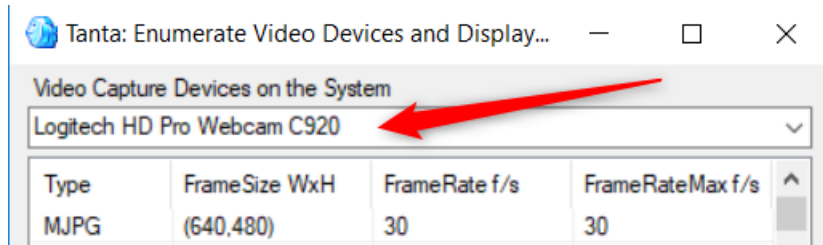


Figure 6.2: The Video Capture Devices Dropdown ComboBox

copies inline in the for loop of the `GetDevicesByCategory()` function

Now that we have a list of all the Video Capture Devices on

the system we can offer them to the user as a choice. This is exactly what the `ctlTantaVideoPicker` control does – it takes the list returned by the `DisplayVideoCaptureDevices()` function and populates a drop down ComboBox control with them.

ENUMERATING THE ATTRIBUTES OF A VIDEO DEVICE

Most Video Capture Devices will offer a multitude of Media Types and formats for the video stream they produce. At this point, if the user were to use the only Video Capture Device information to set up a Media Stream, they would probably just get the first Media Type and format on offer and this would most likely be a fairly low quality generic default choice. There is no reason why we should limit the user to that option. We can further interrogate each Video Capture Device to find out all the Media Types and options it offers. Once we have those options, we can present them in a nice list and then then let the user choose from amongst them.

In the `ctlTantaVideoPicker` control, the act of picking the Video Capture Device in the dropdown ComboBox initiates the process of discovering the video formats associated with that particular device. This process is driven out of the `comboBoxCaptureDevices_SelectedIndexChanged` method in the `ctlTantaVideoPicker1` control. That code will not be reproduced here since all it really does is call the `DisplayVideoFormatsForCurrentCaptureDevice()` function which does the real work of obtaining the Media Type and format data for the current Video Capture Device and displaying it in a standard ListView control. This function is show in the code section below – it is rather lengthy – but it does do all of the rest of the work in the application.

```
private void DisplayVideoFormatsForCurrentCaptureDevice()
{
    IMFPresentationDescriptor sourcePresentationDescriptor = null;
    int sourceStreamCount = 0;
    bool streamIsSelected = false;
    IMFStreamDescriptor videoStreamDescriptor = null;
    IMFMediaTypeHandler typeHandler = null;
```

```

int mediaTypeCount = 0;

List<TantaMFVideoFormatContainer> formatList = new List<TantaMFVideoFormatContainer>();
HRESULT hr;
IMFMediaSource mediaSource = null;

try
{
    // clear what we have now
    listViewSupportedFormats.Clear();
    // reset this
    listViewSupportedFormats.ListViewItemSorter = null;

    // get the currently selected device
    TantaMFDevice currentDevice = (TantaMFDevice)comboBoxCaptureDevices.SelectedItem;
    if (currentDevice == null)
    {
        throw new Exception("currentDevice == null");
    }

    // use the device symbolic name to create the media source for the video
    // device. Media sources are objects that generate media data.
    // For example, the data might come from a video file, a network stream,
    // or a hardware device, such as a camera. Each media source contains one
    // or more streams, and each stream delivers data of one type, such as audio or video.
    mediaSource = TantaWMFUtils.GetMediaSourceFromTantaDevice(currentDevice);
    if (mediaSource == null)
    {
        throw new Exception("call to mediaSource == null");
    }

    // A presentation is a set of related media streams that share a
    // common presentation time. we don't need that functionality in
    // this app but we do need to presentation descriptor to find out
    // the stream descriptors, these will give us the media types on offer
    hr = mediaSource.CreatePresentationDescriptor(out sourcePresentationDescriptor);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("CreatePresentationDescriptor failed. Err=" + hr.ToString());
    }
    if (sourcePresentationDescriptor == null)
    {
        throw new Exception("failed, sourcePresentationDescriptor == null");
    }

    // Now we get the number of stream descriptors in the presentation.
    // A presentation descriptor contains a list of one or more
    // stream descriptors.
    hr = sourcePresentationDescriptor.GetStreamDescriptorCount(out sourceStreamCount);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("GetStreamDescriptorCount failed. Err=" + hr.ToString());
    }
    if (sourceStreamCount == 0)
    {
        throw new Exception("GetStreamDescriptorCount failed. sourceStreamCount == 0");
    }

    // look for the video stream
    for (int i = 0; i < sourceStreamCount; i++)
    {
        // we require the major type to be video
        Guid guidMajorType = TantaWMFUtils.GetMajorMediaTypeFromPresentationDescriptor(
            sourcePresentationDescriptor, i);
        if (guidMajorType != MFMediaType.Video) continue;

        // we also require the stream to be enabled
        hr = sourcePresentationDescriptor.GetStreamDescriptorByIndex(i,
            out streamIsSelected, out videoStreamDescriptor);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("GetStreamDescriptor) failed. Err=" + hr.ToString());
        }
        if (videoStreamDescriptor == null)
        {
            throw new Exception("failed. videoStreamDescriptor == null");
        }
        // if the stream is not selected (enabled) look for the next
        if (streamIsSelected == false)
        {
            Marshal.ReleaseComObject(videoStreamDescriptor);
        }
    }
}

```

```

        videoStreamDescriptor = null;
        continue;
    }

    // Get the media type handler for the stream. IMFMediaTypeHandler
    // interface is a standard way of looking at the media types on an object
    hr = videoStreamDescriptor.GetMediaTypeHandler(out typeHandler);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("GetMediaTypeHandler failed. Err=" + hr.ToString());
    }
    if (typeHandler == null)
    {
        throw new Exception("GetMediaTypeHandler failed. typeHandler == null");
    }
    // Now we get the number of media types in the stream descriptor.
    hr = typeHandler.GetMediaTypeCount(out mediaTypeCount);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("GetMediaTypeCount failed. Err=" + hr.ToString());
    }
    if (mediaTypeCount == 0)
    {
        throw new Exception("GetMediaTypeCount failed. mediaTypeCount == 0");
    }

    // now loop through each media type
    for (int mediaTypeId = 0; mediaTypeId < mediaTypeCount; mediaTypeId++)
    {
        // Now we have the handler, get the media type.
        IMFMediaType workingMediaType = null;
        hr = typeHandler.GetMediaTypeByIndex(mediaTypeId, out workingMediaType);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("GetMediaTypeByIndex failed. Err=" + hr.ToString());
        }
        if (workingMediaType == null)
        {
            throw new Exception("workingMediaType == null");
        }
        TantaMFVideoFormatContainer tmpContainer =
            TantaMediaTypeInfo.GetVideoFormatContainerFromMediaTypeObject(
                workingMediaType, currentDevice);
        if (tmpContainer == null)
        {
            // we failed
            throw new Exception("failed tmpContainer == null");
        }
        // now add it
        formatList.Add(tmpContainer);
        Marshal.ReleaseComObject(workingMediaType);
        workingMediaType = null;
    }

    // NOTE: we only do the first enabled video stream we find.
    // it is possible to have more but our control
    // cannot cope with that
    break;
}

// now display the formats
foreach (TantaMFVideoFormatContainer videoFormat in formatList)
{
    ListViewItem lvi = new ListViewItem(new[]
    { videoFormat.SubTypeAsString,
      videoFormat.FrameSizeAsString,
      videoFormat.FrameRateAsString,
      videoFormat.FrameRateMaxAsString, videoFormat.AllAttributes
    });
    lvi.Tag = videoFormat;
    listViewSupportedFormats.Items.Add(lvi);
}

listViewSupportedFormats.Columns.Add("Type", 70);
listViewSupportedFormats.Columns.Add("FrameSize WxH", 100);
listViewSupportedFormats.Columns.Add("FrameRate f/s", 100);
listViewSupportedFormats.Columns.Add("FrameRateMax f/s", 100);
listViewSupportedFormats.Columns.Add("All Attributes", 2500);
}
finally
{

```

```

        // close and release
        if (mediaSource != null)
        {
            Marshal.ReleaseComObject(mediaSource);
            mediaSource = null;
        }
        if (sourcePresentationDescriptor != null)
        {
            Marshal.ReleaseComObject(sourcePresentationDescriptor);
            sourcePresentationDescriptor = null;
        }
        if (videoStreamDescriptor != null)
        {
            Marshal.ReleaseComObject(videoStreamDescriptor);
            videoStreamDescriptor = null;
        }
        if (typeHandler != null)
        {
            Marshal.ReleaseComObject(typeHandler);
            typeHandler = null;
        }
    }
}

```

Source: TantaCommon::ctlTantaVideoPicker::DisplayVideoFormatsForCurrentCaptureDevice

The code for the `DisplayVideoFormatsForCurrentCaptureDevice` function is shown above. The important point to realize here is that we cannot get the video format information directly from the Video Capture Device. It just does not have the capability to provide that. There are two basic ways of getting the Media Type and format information.

1. We could use the Video Capture Device information to create a Media Source and then use that to give us a list of all of the video formats offered by a specific Video Capture Device.
2. Alternatively, we could use the Video Capture Device to create a Source Reader and interrogate that to find the video formats.

Recall that a Source Reader actually contains a Media Source so the Source Reader route is just a slightly different (and some would say easier) way of interrogating a Media Source. The Source Reader method will not be discussed in this book – but if you are interested you can have a look at the `ctlTantaVideoPickerViaReader` control in the *TantaCommon* project. This control does exactly the same thing as the `ctlTantaVideoPicker` control – it just uses a Source Reader enumeration mechanism. It should be noted that pretty much all of the examples you will find on the Internet use the Source Reader route – but they will not provide you with any experience interacting with a Media Source, Presentation and Media Stream. They get the job done but, from a learning perspective, they are something of a dead end.

The first thing we do in the `DisplayVideoFormatsForCurrentCaptureDevice` function is to create the Media Source from the currently selected Video Capture Device. The creation of the Media Source is done with a call to the static `GetMediaSourceFromTantaDevice` function in the *TantaWMFUtils* library.

```
// get the currently selected device
TantaMFDevice currentDevice = (TantaMFDevice)comboBoxCaptureDevices.SelectedItem;
if (currentDevice == null)
{
    throw new Exception("DisplayVideoFormatsForCurrentCaptureDevice currentDevice == null");
}

// use the device symbolic name to create the media source for the video device.
mediaSource = TantaWMFUtils.GetMediaSourceFromTantaDevice(currentDevice);
if (mediaSource == null)
{
    throw new Exception("failed, mediaSource == null");
}
```

The `GetMediaSourceFromTantaDevice` function takes a single parameter of type `TantaMFDevice`. Recall that a `TantaMFDevice` is pretty much just a Friendly Name and Symbolic Name pair for a specific Video Capture Device.

```
public static IMFMediaSource GetMediaSourceFromTantaDevice(TantaMFDevice sourceDevice)
{
    IMFMediaSource mediaSource = null;
    HRESULT hr = 0;
    MFAttributes attributeContainer = null;

    try
    {
        if (sourceDevice == null)
        {
            // we failed
            throw new Exception("GetMediaSourceFromTantaDevice sourceDevice == null");
        }
        if ((sourceDevice.SymbolicName == null) || (sourceDevice.SymbolicName.Length == 0))
        {
            // we failed
            throw new Exception("failed null or bad symbolicLinkStr");
        }
        if (sourceDevice.DeviceType == Guid.Empty)
        {
            // we failed
            throw new Exception("GetMediaSourceFromTantaDevice DeviceType == Guid.Empty");
        }

        // Initialize an attribute store. We will use this to
        // specify the device parameters.
        hr = MFExtern.MFCreateAttributes(out attributeContainer, 2);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("failed MFCreateAttributes, retVal=" + hr.ToString());
        }
        if (attributeContainer == null)
        {
            // we failed
            throw new Exception("failed, attributeContainer == null");
        }

        // setup the attribute container, it is always a MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE
        hr = attributeContainer.SetGUID(
            MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE,
            sourceDevice.DeviceType);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("failed setting up the attributes, retVal=" + hr.ToString());
        }

        // set the formal (symbolic name) name of the device as an attribute.
        hr = attributeContainer.SetString(
            MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_SYMBOLIC_LINK,
            sourceDevice.SymbolicName);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("failed symbolic name, retVal=" + hr.ToString());
        }

        // get the media source from the symbolic name
        hr = MFExtern.MFCreateDeviceSource(attributeContainer, out mediaSource);
    }
}
```

```

        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("failed MFCreatDeviceSource, retVal=" + hr.ToString());
        }
    }
    finally
    {
        // make sure we release the attribute memory
        if (attributeContainer != null)
        {
            Marshal.ReleaseComObject(attributeContainer);
        }
    }
    return mediaSource;
}

```

Source: TantaCommon::TantaWMFUtils::GetMediaSourceFromTantaDevice

Remember that at this point we do not have a Video Capture Device or even an Activator for one. What we do have is a Friendly Name and Symbolic Name pair for a specific Video Capture Device of interest. So the first thing we need to do is get a Media Source from this information. This process, since it is likely to need to be re-used, is encapsulated into a static routine in the Tanta libraries which accepts a `TantaMFDevice` and returns an appropriate Media Source built from it.

Really the ultimate goal of the above code is to call the WMF `MFCreatDeviceSource` function and get it to give us back a Media Source device. However the `MFCreatDeviceSource` function it is not prepared to accept a symbolic name as a parameter – it wants an Attribute and it wants that Attribute to be stored in an Attribute container. So we run through the, by now familiar pattern, of getting an attribute store with an `MFCreatAttributes` call and then populating it with the information the `MFCreatDeviceSource` function is going to need.

```

// Enumerate the devices.
hr = MFExtern.MFEnumDeviceSources(attributeContainer, out deviceArr, out numDevices);

// setup the attribute container, it is always a MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE here
hr = attributeContainer.SetGUID(MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE,
                                sourceDevice.DeviceType);

// set the formal (symbolic name) name of the device as an attribute.
hr = attributeContainer.SetString(
    MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_SYMBOLIC_LINK,
    sourceDevice.SymbolicName);

```

The code is mostly self-explanatory and so will not be discussed further here other than to note that the attribute container requires two Attributes to be set in this particular case.

Returning to the `GetMediaSourceFromTantaDevice` code, once we have a populated Attribute container we can easily create a Media Source using the static `MFCreatDeviceSource()` function call.

```

// get the media source from the symbolic name
hr = MFExtern.MFCreatDeviceSource(attributeContainer, out mediaSource);

```

Now that we have a Media Source, we return to the `DisplayVideoFormatsForCurrentCaptureDevice` function. Once we have the Media Source it is a simple task to acquire a Presentation Descriptor and the count of the Media Streams in that Presentation Descriptor.

```
// A presentation is a set of related media streams that share a common presentation time.
// we don't need that functionality in this app but we do need to presentation descriptor
// to find out the stream descriptors, these will give us the media types on offer
hr = mediaSource.CreatePresentationDescriptor(out sourcePresentationDescriptor);

// Now we get the number of stream descriptors in the presentation.
hr = sourcePresentationDescriptor.GetStreamDescriptorCount(out sourceStreamCount);
```

After we have the count of the streams in the Presentation Descriptor (`sourceStreamCount`) we look at each one in turn until we find the first enabled stream whose Media Major Type is video.

```
// look for the video stream
for (int i = 0; i < sourceStreamCount; i++)
{
    // we require the major type to be video
    Guid guidMajorType = TantaWMFUtils.GetMajorMediaTypeFromPresentationDescriptor(
        sourcePresentationDescriptor, i);
    if (guidMajorType != MFMediaType.Video) continue;

    // we also require the stream to be enabled
    hr = sourcePresentationDescriptor.GetStreamDescriptorByIndex(i,
        out streamIsSelected, out videoStreamDescriptor);

    // if the stream is not selected (enabled) look for the next
    if (streamIsSelected == false)
    {
        Marshal.ReleaseComObject(videoStreamDescriptor);
        videoStreamDescriptor = null;
        continue;
    }

    ... more code
}
```

The process of extracting the Media Major Type from a stream has been discussed elsewhere in this book – it will not be discussed here other than to note the entire process has been factored out into a function in the `TantaWMFUtils` class of the *TantaCommon* library. Note the release process for the streams we do not use.

Once we have an enabled video stream, we need to look at the Media Types and formats it supports - Video Capture Devices usually offer a large number of options. This means we have to enumerate the Media Types in the stream and the way this is done is to acquire a Type Handler object from the Media Stream. Quite why we need to acquire a Type Handler to do this instead of just asking the Media Stream for them directly is something of a mystery. However, it is what it is and the Type Handler is the way it is done.

```
// Get the media type handler for the stream. IMFMediaTypeHandler
// interface is a standard way of looking at the media types on a stream
hr = videoStreamDescriptor.GetMediaTypeHandler(out typeHandler);

// Now we get the number of media types in the stream descriptor.
hr = typeHandler.GetMediaTypeCount(out mediaTypeCount);

... more code
```

Once we have the Type Handler we enumerate the Media Types in the stream in exactly the same way we used the Presentation Descriptor to discover the Media Streams in a Media Source.

```
// now loop through each media type
for (int mediaTypeId = 0; mediaTypeId < mediaTypeCount; mediaTypeId++)
{
    // Now we have the handler, get the media type.
    IMFMediaType workingMediaType = null;
    hr = typeHandler.GetMediaTypeByIndex(mediaTypeId, out workingMediaType);

    TantaMFVideoFormatContainer tmpContainer =
        TantaMediaTypeInfo.GetVideoFormatContainerFromMediaTypeObject(
            workingMediaType, currentDevice);

    // now add it
    formatList.Add(tmpContainer);
    Marshal.ReleaseComObject(workingMediaType);
    workingMediaType = null;
}
```

The operation of the enumeration in the above for loop is simple enough, the first action is to get the Media Type from the Type Handler. Once we have a media type, we make a call to a static `TantaWMFUtils` function named `GetVideoFormatContainerFromMediaTypeObject` which acts as a container for the Media Type information and also separates certain selected Attribute details out into

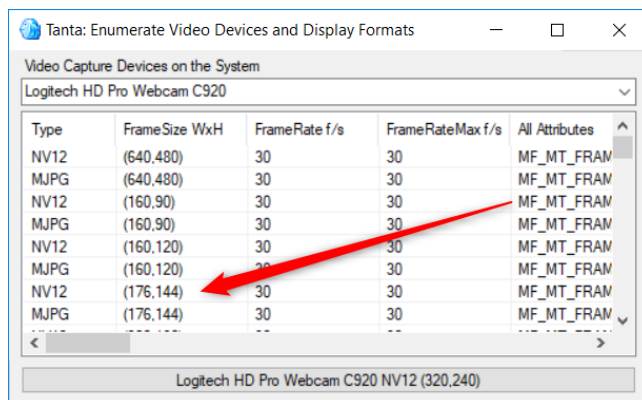


Figure 6.3: The Media Type Information Presented on the Screen

human readable form for display purposes. The information we are mostly interested in is the video format type, the screen size and the typical frame rates. Note that there is typically a lot of other information stored in the Attributes of a Media type. These details are not processed – but the names of the Attributes are collected as a string for display purposes.

The contents of the `GetVideoFormatContainerFromMediaTypeObject` function will not be displayed or discussed further here as it is not particularly educational and you can easily look that code up if you wish to have more details.

Once each Media Type supported by the Media Source (and hence ultimately by the Video Capture Device) has been processed into a `TantaMFVideoFormatContainer` and returned as an item in a generic list the `DisplayVideoFormatsForCurrentCaptureDevice` function can use it to present that information to the user. In this case a simple list view is populated as shown in Figure 6.3.

Windows Media Foundation: Getting Started in C#

Chapter 7

PRACTICAL WMF ARCHITECTURES

As discussed in the *Windows Media Foundation Architecture* chapter, there are two basic types of Windows Media Foundation Architecture and a third which is a hybrid between the first two. The names given to these architectures (in this book) are the Pipeline Architecture, the Reader-Writer Architecture and the Hybrid Architecture. Each of these will be discussed in detail in the sections below.

It is worth taking a bit of time to read and understand the concepts presented here. The subsequent chapters, which discuss working applications and sample techniques, will not repeat these ideas in any great detail. They will, for example, simply assume that you know what is meant by a statement like *“The application uses a standard Pipeline Architecture running from a MP4 Media Source to an EVR Renderer”*. There is a great deal of implied context embedded in a statement like that and if you are going to be able to successfully “unpack” it you will need to understand the concepts below.

IMPLEMENTING THE PIPELINE ARCHITECTURE

The Pipeline is arguably the primary architecture of Windows Media Foundation – although you could be forgiven for thinking otherwise if you look at the available sample code on the Internet.

Pipelines seem complex but, if you take a broad enough overview, they always follow the same general prescription. Let's take the simple case of a Pipeline which will contain one Media Source and one Media Sink. The set of steps taken to create a Pipeline Architecture in that situation are as follows...

1. A Media Session is created.
2. A Media Source is created.
3. A Media Stream and Media Type on that stream are selected.
4. A Media Sink is created.
5. A Topology is created
6. The Media Source is added to the Topology as a Node.
7. The Media Sink is added to the Topology as a Node.
8. The source Topology Node is connected to the sink Topology Node.
9. The Topology is resolved.
10. The Pipeline is created and the media data flows.

Of course, it is not quite that simple and there are plenty of other things to consider. Let's take another run at it, this time with more detail.

1. A Media Session is setup and it is given a Callback Object so that it can communicate events to the application.
2. A Topology object is created.
3. A Media Source is created.
4. The Media Source will have one or more Media Streams. We obtain a Stream Descriptor and choose one of those streams for our Pipeline. This is called "*selecting*" it.
5. The Media Stream will probably offer multiple Media Types. Inside the Media Stream we choose the Media Type we wish to use. This is called making the Media Type "*current*" on the stream.
6. We create a Media Sink and give it the Media Type it will be dealing with as an input. This does not have to be the same Media Type as output by the source stream.

7. We may, for some types of Media Sink have to tell it the Media Type of the output it should use.
8. We create a Topology Node for the Media Source and tell it about the source Media Stream using the Stream Descriptor.
9. We create a Topology Node for Media Sink and tell it which of the input streams (on the sink) the node will reference. If there is only a single stream this will always be stream 0. If there is more than one stream we will provide a specific index.
10. We add the two Topology Nodes to the Topology object.
11. If the output Media Type of the stream on the Media Source is not the same as the Media Type of the input stream on the Media Sink, then we have two options. The first option is to find a conversion Transform and add that to the Topology. Alternatively, if we are in a file playback situation (i.e. we are using the Enhanced Video Renderer or Streaming Audio Renderer as sinks) we can just ignore the Media Type mismatch.
12. We now connect up the Topology Nodes. If we have only source and sink nodes we connect them up. If we have a Transform, we connect the source node to the transform node and the transform node to the sink node. If we have mismatched Media Types, we just connect the source and sink node anyways. If the Media Type used as output on one node does not match the input Media type on the next we say we have created a *“Partial Topology”* rather than a *“Full Topology”*.
13. We ask the Media Session to resolve the Topology. If there is a Partial Topology, the Topology Object will adjust the Topology and automatically add Transforms to make it all work. This is only available in a file playback scenario.
14. Immediately after the Topology is resolved, the Pipeline is created. The Media Session sends events through its Callback Object and the media data can start moving through the Pipeline from the source to sink.

It is obvious from the above list that there are considerably more steps – but the fundamental process is identical. We could make yet another list with even more steps - but there is no point. We would just be writing out words the operations a block of example code would make explicitly clear. The complete source code for a simple Pipeline will be shown shortly. First, however, we need to make a brief digression to discuss Media Sinks and their sometimes unique requirements.

THE STANDARDIZATION OF PIPELINE COMPONENTS

Media Sources are pretty standard and your interaction with them will largely be the same even if they represent wildly different devices such as webcams, microphones or files. This means that if you find some example code on the Internet using one type of Media Source then you should be able to convert it to another type of Media Source with only modest changes.

The real area of confusion is Media Sinks - each Media Sink tends to have its own set-up requirements. Although they are broadly similar in most respects, the creation and configuration of each type of Media Sink does vary.

For example, there is no need to tell the MP3 file sink its output format – it only writes MP3 files. This is definitely not so with something like the MP4 file sink which can store a variety of internal formats and media types (H.264/AVC video, AAC audio, MP3 audio).

The Enhanced Video Renderer, which displays video data on the screen, also does not have an output format. The EVR is a very sophisticated component and can also accept a wide variety of input formats. This means usually no need to consider the EVR when choosing a Media Type on the source stream. Also, since the EVR is a renderer, Transforms will be automatically added to the Topology to convert the format if necessary.

That is just the way it is with Media Sinks, sometimes you need to care about the input Media Type, sometimes you need to care about the output Media Type and sometimes the sink more or less just figures it out for itself. Fortunately, there are not many Media Sinks on offer and so learning the idiosyncrasies of each is not too hard. The Tanta Sample Projects only use five types of sink (MP4, MP3, EVR, SAR and Sample Grabber). Just be aware that if you are trying to convert some sample code which uses an MP3 sink to use an MP4 sink you will probably have to make changes. Media Sinks are, in general, not “drop-in” replacements for each other.

THE SAMPLE PIPELINE ARCHITECTURE SOURCE

The code block below shows a complete end-to-end build of a Pipeline Architecture. This example code is taken from the *TantaAudioFileCopyViaPipelineMP3Sink* Sample

Project. The project is designed to copy an MP3 file using the Pipeline. This project is about as simple as it is possible to get – the output format is identical to the input format (so there is no need to worry about Media Type conversion) and there is only one stream of data to deal with (audio). Let's have a look at the code. As you walk through it, remember the above list of steps, you will see it follows the sequence very closely.

```

/// ++++++
/// <summary>
/// Opens prepares the media session and topology and opens the media source
/// and media sink.
///
/// Once the session and topology are setup, a MESSessionTopologySet event
/// will be triggered in the Callback Object. After that the events there
/// trigger other events and everything rolls along automatically.
/// </summary>
/// <param name="sourceFileName">the source file name</param>
/// <param name="outputFileName">the name of the output file</param>
/// <history>
/// 01 Nov 18 Cynic - Originally Written
/// </history>
public void PrepareSessionAndTopology(string sourceFileName, string outputFileName)
{
    HRESULT hr;
    IMFSourceResolver pSourceResolver = null;
    IMFTopology pTopology = null;
    IMFPresentationDescriptor sourcePresentationDescriptor = null;
    int sourceStreamCount = 0;
    IMFStreamDescriptor audioStreamDescriptor = null;
    bool streamIsSelected = false;
    IMFTopologyNode sourceAudioNode = null;
    IMFTopologyNode outputSinkNode = null;
    IMFMediaType currentAudioMediaType = null;
    int audioStreamIndex = -1;

    LogMessage("PrepareSessionAndTopology ");

    // we sanity check the filenames - the existence of the path and if the file already exists
    // should have been checked before this call
    if ((sourceFileName == null) || (sourceFileName.Length == 0))
    {
        throw new Exception("source file name is invalid. Cannot continue.");
    }

    if ((outputFileName == null) || (outputFileName.Length==0))
    {
        throw new Exception("output file name is invalid. Cannot continue.");
    }

    try
    {
        // reset everything
        CloseAllMediaDevices();

        // Create the media session.
        hr = MFExtern.MFCreateMediaSession(null, out mediaSession);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to MFCreateMediaSession failed. Err=" + hr.ToString());
        }
        if (mediaSession == null)
        {
            throw new Exception("call to MFCreateMediaSession failed. mediaSession == null");
        }

        // set up our media session Callback Object.
        mediaSessionAsyncCallbackHandler = new TantaAsyncCallbackHandler();
        mediaSessionAsyncCallbackHandler.Initialize();
        mediaSessionAsyncCallbackHandler.MediaSession = mediaSession;
        mediaSessionAsyncCallbackHandler.MediaSessionAsyncCallbackError =
            HandleMediaSessionAsyncCallBackErrors;
        mediaSessionAsyncCallbackHandler.MediaSessionAsyncCallbackEvent =
            HandleMediaSessionAsyncCallBackEvent;
    }
}

```

```
// Register the Callback Object with the session and tell it that events can
// start. This does not actually trigger an event it just lets the media session
// know that it can now send them if it wishes to do so.
hr = mediaSession.BeginGetEvent(mediaSessionAsyncCallbackHandler, null);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to BeginGetEvent failed. Err=" + hr.ToString());
}

// Create a new topology.
hr = MFExtern.MFCreateTopology(out pTopology);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to MFCreateTopology failed. Err=" + hr.ToString());
}
if (pTopology == null)
{
    throw new Exception("call to MFCreateTopology failed. pTopology == null");
}

// ####
// #### we now create the media source, this is an audio file
// ####

// use the file name to create the media source for the audio device.
mediaSource = TantaWMFUtils.GetMediaSourceFromFile(sourceFileName);
if (mediaSource == null)
{
    throw new Exception("PrepareSessionAndTopology call to mediaSource == null");
}

// We now get a copy of the media source's presentation descriptor.
// Applications can use the presentation descriptor to select streams
hr = mediaSource.CreatePresentationDescriptor(out sourcePresentationDescriptor);
if (hr != HRESULT.S_OK)
{
    throw new Exception("CreatePresentationDescriptor failed. Err=" + hr.ToString());
}
if (sourcePresentationDescriptor == null)
{
    throw new Exception("failed. sourcePresentationDescriptor == null");
}

// Now we get the number of stream descriptors in the presentation.
hr = sourcePresentationDescriptor.GetStreamDescriptorCount(out sourceStreamCount);
if (hr != HRESULT.S_OK)
{
    throw new Exception("GetStreamDescriptorCount failed. Err=" + hr.ToString());
}
if (sourceStreamCount == 0)
{
    throw new Exception("GetStreamDescriptorCount failed. sourceStreamCount == 0");
}

// Look at each stream, there can be more than one stream here
// Usually only one is enabled. This app uses the first "selected"
// stream we come to which has the appropriate media type
for (int i = 0; i < sourceStreamCount; i++)
{
    // we require the major type to be audio
    Guid guidMajorType =
        TantaWMFUtils.GetMajorMediaTypeFromPresentationDescriptor(
            sourcePresentationDescriptor, i);
    if (guidMajorType != MFMediaType.Audio) continue;

    // we also require the stream to be enabled
    hr = sourcePresentationDescriptor.GetStreamDescriptorByIndex(i,
        out streamIsSelected, out audioStreamDescriptor);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("GetStreamDescriptorByIndex failed. Err=" + hr.ToString());
    }
    if (audioStreamDescriptor == null)
    {
        throw new Exception("failed. audioStreamDescriptor == null");
    }
    // if the stream is selected, leave now we will release the
    // audioStream descriptor later
    if (streamIsSelected == true)
    {
        audioStreamIndex = i; // record this
    }
}
```

```

        break;
    }

    // release the one we are not using
    if (audioStreamDescriptor != null)
    {
        Marshal.ReleaseComObject(audioStreamDescriptor);
        audioStreamDescriptor = null;
    }
    audioStreamIndex = -1;
}

// by the time we get here we should have a audioStreamDescriptor if
// we do not, then we cannot proceed
if (audioStreamDescriptor==null)
{
    throw new Exception("audioStreamDescriptor == null");
}
if(audioStreamIndex < 0)
{
    throw new Exception("failed. audioStreamIndex < 0");
}

// ####
// #### we now create the media sink, we need the type from the stream to do
// #### this which is why we wait until now to set it up
// ####

currentAudioMediaType = TantaWMFUtils.GetCurrentMediaTypeFromStreamDescriptor(
    audioStreamDescriptor);
if (currentAudioMediaType == null)
{
    throw new Exception("call to currentAudioMediaType == null");
}

mediaSink = OpenMediaFileSink(outputFileName);
if (mediaSink == null)
{
    throw new Exception("PrepareSessionAndTopology call to mediaSink == null");
}

// ####
// #### we now make up a topology branch for the audio stream
// ####

// Create a source node for this stream.
sourceAudioNode = TantaWMFUtils.CreateSourceNodeForStream(mediaSource,
    sourcePresentationDescriptor, audioStreamDescriptor);
if (sourceAudioNode == null)
{
    throw new Exception("failed. pSourceNode == null");
}

// Create the output node - this is a file sink in this case.
outputSinkNode = TantaWMFUtils.CreateSinkNodeForStream(mediaSink);
if (outputSinkNode == null)
{
    throw new Exception("outputSinkNode == null");
}

// Add the nodes to the topology. First the source
hr = pTopology.AddNode(sourceAudioNode);
if (hr != HRESULT.S_OK)
{
    throw new Exception("AddNode(sourceAudioNode) failed. Err=" + hr.ToString());
}

// then add the output
hr = pTopology.AddNode(outputSinkNode);
if (hr != HRESULT.S_OK)
{
    throw new Exception("AddNode(outputSinkNode) failed. Err=" + hr.ToString());
}

// Connect the output stream from the source node to the input stream of
// the output node.
hr = sourceAudioNode.ConnectOutput(0, outputSinkNode, 0);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to ConnectOutput failed. Err=" + hr.ToString());
}

```

```

        // Set the topology on the media session.
        // If SetTopology succeeds, the media session will queue an
        // MESSessionTopologySet event.
        hr = mediaSession.SetTopology(0, pTopology);
        MFCError.ThrowExceptionForHR(hr);

        // Release the topology
        if (pTopology != null)
        {
            Marshal.ReleaseComObject(pTopology);
        }
    }
    catch (Exception ex)
    {
        LogMessage("Error: " + ex.Message);
        OISMessageBox(ex.Message);
    }
    finally
    {
        // Clean up
        if (pSourceResolver != null)
        {
            Marshal.ReleaseComObject(pSourceResolver);
        }
        if (sourcePresentationDescriptor != null)
        {
            Marshal.ReleaseComObject(sourcePresentationDescriptor);
        }
        if (audioStreamDescriptor != null)
        {
            Marshal.ReleaseComObject(audioStreamDescriptor);
        }
        if (sourceAudioNode != null)
        {
            Marshal.ReleaseComObject(sourceAudioNode);
        }
        if (outputSinkNode != null)
        {
            Marshal.ReleaseComObject(outputSinkNode);
        }
        if (currentAudioMediaType != null)
        {
            Marshal.ReleaseComObject(currentAudioMediaType);
        }
    }
}

```

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::GetMediaSourceFromTantaDevice

Don't be too concerned about the length of the above code. It was written from a demonstration point of view. This means that a lot of the things that might have been placed in subroutines have been deliberately written in-line so the process can be observed from end to end. Let's follow it down section by section and discuss what each part is doing. You may well wish to refer back to *The WMF Components* chapter if your memory of the role of each component is hazy.

The first thing we do, other than a bit of error checking and administration, is to create a Media Session.

```

// Create the media session.
hr = MFExtern.MFCreateMediaSession(null, out mediaSession);

```

No great surprises there - it is just a simple call to the external `MFCreateMediaSession` static function. In the next block we create and configure a Callback Object and give it to the Media Session.


```
// set up our media session Callback Object.
mediaSessionAsyncCallbackHandler = new TantaAsyncCallbackHandler();
mediaSessionAsyncCallbackHandler.Initialize();
mediaSessionAsyncCallbackHandler.MediaSession = mediaSession;
mediaSessionAsyncCallbackHandler.MediaSessionAsyncCallBackError =
    HandleMediaSessionAsyncCallBackErrors;
mediaSessionAsyncCallbackHandler.MediaSessionAsyncCallBackEvent =
    HandleMediaSessionAsyncCallBackEvent;

// Register the Callback Object with the session and tell it that events can
// start. This does not actually trigger an event it just lets the media session
// know that it can now send them if it wishes to do so.
hr = mediaSession.BeginGetEvent(mediaSessionAsyncCallbackHandler, null);
```

The error and event handlers in the code block above are standard C# delegate/events which relay messages back to the main application from the Callback Object. They were extensively discussed in the *Callback Objects* section of *The WMF Components* chapter and that information will not be repeated here.

The `BeginGetEvent()` call on the Media Session registers the Callback Object with the Media Session and also lets it know that it can start sending messages if it needs to do so. In reality, Media Session will not do anything until the Topology is resolved. The next action, which creates the Topology object, is similarly standard.

```
// Create a new topology.
hr = MFExtern.MFCreateTopology(out pTopology);
```

Note that the creation of the Topology object here is *not* the same as “resolving” it. There is a good deal of configuration to do before we get to that action. After the Topology is created, the Media Source is created from the filename.

```
// use the file name to create the media source for the audio device.
mediaSource = TantaWMFUtils.GetMediaSourceFromFile(sourceFileName);
```

The creation of a Media Source from a file or URL is a common multi-step process. In order to simplify things visually (not to mention promoting code-reuse) the creation process has been factored out to a static function in the `TantaWMFUtils` class in the *TantaCommon* library. The operation of this code has been extensively discussed earlier in this book and there is no real benefit to revisiting that topic here.

The next thing we do is get the Presentation Descriptor from the Media Source and use it to count the number of streams on offer.

```
// We now get a copy of the media source's presentation descriptor.
// Applications can use the presentation descriptor to select streams
hr = mediaSource.CreatePresentationDescriptor(out sourcePresentationDescriptor);

// Now we get the number of stream descriptors in the presentation.
hr = sourcePresentationDescriptor.GetStreamDescriptorCount(out sourceStreamCount);
```

Other than the error checking, both of the above operations are straight-forward. Once we have the number of streams in the Presentation, it is a simple matter to look at each stream in turn and find the one we want. This is the purpose of the `for` loop in the sample code block.

```

for (int i = 0; i < sourceStreamCount; i++)
{
    // we require the major type to be audio
    Guid guidMajorType = TantaWMFUtils.GetMajorMediaTypeFromPresentationDescriptor(
        sourcePresentationDescriptor, i);

    if (guidMajorType != MFMediaType.Audio) continue;

    // we also require the stream to be enabled
    hr = sourcePresentationDescriptor.GetStreamDescriptorByIndex(i, out
        streamIsSelected, out audioStreamDescriptor);

    // if the stream is selected, leave now we will release the
    // audioStream descriptor later
    if (streamIsSelected == true)
    {
        audioStreamIndex = i; // record this
        break;
    }

    // release the one we are not using
    if (audioStreamDescriptor != null)
    {
        Marshal.ReleaseComObject(audioStreamDescriptor);
        audioStreamDescriptor = null;
    }
    audioStreamIndex = -1;
}

```

With the error checking stripped out, the `for` loop is really just a simple series of tests. Ultimately, by the time the loop ends, we will have identified a Stream Descriptor for the first enabled (selected) audio stream in the Presentation. Note that the Stream Descriptors we do not use are released – any object we obtain from Windows Media Foundation must be released when you are done with it.

The next thing to do is to create the Media Sink.

```
mediaSink = OpenMediaFileSink(outputFileName);
```

Like the creation of the Media Source, the creation process for the Media Sink has been factored out into a separate function – in this case a call to `OpenMediaFileSink()` which is located in the applications `frmMain` class. The operation of this code has been extensively discussed earlier in the *Creating a Media Sink On a File* section of *The WMF Components* chapter and so we will not reproduce that information here.

We already have a Topology object and the next thing we need to do is create some nodes for it. Topology Nodes are the objects which will hold the WMF components (or references to them) until they are added to the Pipeline. The Topology Nodes also connect to each other and this enables the information flow through the branches in the Pipeline to be rigorously specified. Usually the input and/or output Media Types (as appropriate) for each step in the Pipeline is also specified on the Topology Nodes when they are created.

```

// Create a source node for this stream.
sourceAudioNode = TantaWMFUtils.CreateSourceNodeForStream(mediaSource,
    sourcePresentationDescriptor, audioStreamDescriptor);

```

Recall from our earlier discussion of Topology Nodes in *The WMF Components* chapter, that each Topology Node is explicitly either a source, sink or transform node. This mode is configured into them at creation time. The creation of Topology Nodes for a Media Source is common enough that the operation has been factored out in to a static `CreateSourceNodeForStream` function in the `TantaWMFUtils` class of the `TantaCommon` library. The operation of this code has also been discussed earlier in this book and will not be covered again here.

Note that that the source Topology Node requires both the Presentation Descriptor and the Stream Descriptor for its creation process. In addition, in the general case, the stream would have previously been “*selected*” in the Presentation Descriptor and a Media Type made “*current*” in the Stream Descriptor. This is how the Topology Node identifies the Stream and Media Type the Media Source represents.

The Topology Node for the Media Sink is also created with a call to the `CreateSinkNodeForStream` static function in the `TantaWMFUtils` class. All the creation process for the sink Topology Node requires is the ID of the stream on the Media Sink that will receive the data (some Media Sinks can have more than one). In this particular application the MP3 file sink can only have one stream and so the ID of 0 is assumed.

Once we have the Topology Nodes we add them to the Topology Object

```
// Add the nodes to the topology. First the source
hr = pTopology.AddNode(sourceAudioNode);

// then add the output
hr = pTopology.AddNode(outputSinkNode);
```

Since we have no other nodes in the Topology, we can directly connect the source and sink Topology Nodes.

```
// Connect the output stream from the source node to the input stream of the output node.
hr = sourceAudioNode.ConnectOutput(0, outputSinkNode, 0);
```

Once the Topology is built we can resolve it and build the Pipeline. This is done with a somewhat anti-climactic call to the `SetTopology()` function on the Media Session.

```
// Set the topology on the media session.
hr = mediaSession.SetTopology(0, pTopology);
```

That is it - at this point the Pipeline is created – but don’t expect the media data to immediately start moving. The Media Session exists to enable the application to control

the flow of the data. It is only logical that the Media Session might expect the application to tell it to start (and stop).

STARTING UP THE PIPELINE

The Media Session communicates with the application via its Callback Object. In the Tanta Sample programs this is always done via the `TantaAsyncCallbackHandler` class in the `TantaCommon` project. This class was discussed in detail in the *Callback Objects* section of *The WMF Components* chapter – you should review that part of the book again if you need some background on Callback Handlers.

The upshot is that the `TantaAsyncCallbackHandler` class intercepts the messages coming back from the Media Session and splits them into two categories: “events” and “errors”. These are passed back into the application via C# events and eventually the `HandleMediaSessionAsyncCallBackEvent` and `HandleMediaSessionAsyncCallBackErrors` functions will be (respectively) called. We will look at error handling in a moment, the code block below shows the contents of the `HandleMediaSessionAsyncCallBackEvent` function.

```

/// ++++++
/// <summary>
/// Handles events reported by the media session TantaAsyncCallbackHandler
/// </summary>
/// <param name="sender">the object sending the event</param>
/// <param name="mediaEvent">the event generated by the media session.
///     Do NOT release this here.</param>
/// <param name="mediaEventType">the eventType, this is just an enum</param>
/// <history>
///     01 Nov 18   Cynic - Originally Written
/// </history>
private void HandleMediaSessionAsyncCallBackEvent(object sender, IMFMediaEvent pEvent,
MediaEventType mediaEventType)
{
    LogMessage("Media Event Type " + mediaEventType.ToString());

    switch (mediaEventType)
    {
        case MediaEventType.MESessionTopologyStatus:
            // Raised by the Media Session when the status of a topology changes.
            // Get the topology changed status code. This is an enum in the event
            int i;
            HRESULT hr = pEvent.GetUINT32(MFAttributesClsid.MF_EVENT_TOPOLOGY_STATUS, out i);
            if (hr != HRESULT.S_OK)
            {
                throw new Exception("call to pEvent failed. Err=" + hr.ToString());
            }
            // the one the handler is probably most interested in is i == MFTopoStatus.Ready
            HandleTopologyStatusChanged(pEvent, mediaEventType, (MFTopoStatus)i);
            break;

        case MediaEventType.MESessionStarted:
            // Raised when the IMFMediaSession::Start method completes asynchronously.
            break;

        case MediaEventType.MESessionPaused:
            // Raised when the IMFMediaSession::Pause method completes asynchronously.
            break;

        case MediaEventType.MESessionStopped:
            // Raised when the IMFMediaSession::Stop method completes asynchronously.
            break;

        case MediaEventType.MESessionClosed:
    }
}

```

```

        // Raised when the IMFMediaSession::Close method completes asynchronously.
        break;

    case MediaEventType.MESessionCapabilitiesChanged:
        // Raised by the Media Session when the session capabilities change.
        // You can use IMFMediaEvent::GetValue to figure out what they are
        break;

    case MediaEventType.MESessionTopologySet:
        // Raised after the IMFMediaSession::SetTopology method completes asynchronously.
        // The Media Session raises this event after it resolves the topology
        // into a full topology and queues the topology for playback.
        break;

    case MediaEventType.MESessionNotifyPresentationTime:
        // Raised by the Media Session when a new presentation starts.
        // This event indicates when the presentation will start and
        // the offset between the presentation time and the source time.
        break;

    case MediaEventType.MEEndOfPresentation:
        // Raised by a media source when a presentation ends. This event
        // signals that all streams in the presentation are complete. The
        // Media Session forwards this event to the application.

        // we cannot successfully .Finalize_ on the SinkWriter
        // if we call CloseAllMediaDevices directly from this thread
        // so we use an asynchronous method
        Task taskA = Task.Run(() => CloseAllMediaDevices());
        // we have to be on the form thread to update the screen
        ThreadSafeScreenUpdate(this, false, "Done");
        break;

    case MediaEventType.MESessionRateChanged:
        // Raised by the Media Session when the playback rate changes.
        // This event is sent after the
        // IMFRateControl::SetRate method completes asynchronously.
        break;

    default:
        LogMessage("Unhandled Media Event Type " + mediaEventType.ToString());
        break;
}
}

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::HandleMediaSessionAsyncCallBackEvent

```

As you can see, the Media Session can send quite a few events – most of which this application quite indifferent to. We will not itemize and discuss each event here – the comments in the above code are pretty self-explanatory. The *TantaAudioFileCopyViaPipelineMP3Sink* application is only interested in two of the events: the *MESessionTopologyStatus* event and the *MEEndOfPresentation* event. The *MEEndOfPresentation* is important when shutting down the Pipeline and will be discussed below. It is the *MESessionTopologyStatus* event which is relevant to the start-up process of the Pipeline. It should be emphasized that just because this application ignores most of the events that doesn't mean they all do. For example, the *ctrlTantaEVRFilePlayer* control in the *TantaFilePlayBackAdvanced* Sample Project takes a great deal more interest in many of these other event types.

The *MESessionTopologyStatus* event is a funny one because it can contain a variety of meanings based on the contents of one of its Attributes.

```

// Raised by the Media Session when the status of a topology changes.
// Get the topology changed status code. This is an enum in the event
int i;
HRESULT hr = pEvent.GetUINT32(MFAttributesClsid.MF_EVENT_TOPOLOGY_STATUS, out i);

```

```
// the one the handler is probably most interested in is i == MFTopoStatus.Ready
HandleTopologyStatusChanged(pEvent, mediaEventType, (MFTopoStatus)i);
break;
```

The value of the Attribute identified by the `MF_EVENT_TOPOLOGY_STATUS` key is an enum of type `MFTopoStatus`. This value is passed in as a parameter on a call to the `HandleTopologyStatusChanged()` function. This function (we will not show the code – it is too trivial) just checks to see if the `topoStatus` parameter is equal to `MFTopoStatus.Ready` and if it is it calls the `HandleTopologyStatusChanged()` function.

```
private void MediaSessionTopologyNowReady(IMFMediaEvent mediaEvent)
{
    LogMessage("MediaSessionTopologyNowReady");

    try
    {
        StartFileCopy();
    }
    catch (Exception ex)
    {
        LogMessage("MediaSessionTopologyNowReady errored ex="+ex.Message);
        OISMessageBox(ex.Message);
    }
}

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::MediaSessionTopologyNowReady
```

The `StartFileCopy()` call simply tells the Media Session to begin.

```
// this is what starts the data moving through the pipeline
HRESULT hr = mediaSession.Start(Guid.Empty, new PropVariant());
if (hr != HRESULT.S_OK)
{
    throw new Exception("mediaSession.Start failed. Err=" + hr.ToString());
}
```

Yes, the logic here is possibly more circuitous than it really needs to be – probably a legacy of various experiments that never got fully backed out before the Tanta Sample Code was finalized (and this book was written). Ultimately the big takeaway is that the act of resolving the Topology triggered an event which caused the `mediaSession.Start()` call to be made. This call starts the data moving through the Pipeline. The parameters on the `mediaSession.Start()` call in this example are empty dummy values - but they can be used, as you will see later in the *TantaFilePlaybackAdvanced* Sample Project, to start the Pipeline at some arbitrary location in the Media Stream. This, of course, only works on file based Media Sources.

SHUTTING DOWN THE PIPELINE

Eventually, unless you are using a Media Source like a webcam, all of the media data will have been sent through the Pipeline and it will be time to close things down gracefully. When the last Media Sample has been sent, the Media Session will raise (among other things) an `MEEndOfPresentation` event. The appearance of this event can be used to begin the shutdown process on the Media Session and Pipeline.

However, there is a problem. Remember that none of the events which trigger calls to `HandleMediaSessionAsyncCallbackEvent` and `HandleMediaSessionAsyncCallbackErrors` will be executing on the form thread. We have to be very careful what we do – especially when we update the screen, which *always* has to be done from the forms main thread.

In addition, the shutdown should really be done from a separate thread so that the `HandleMediaSessionAsyncCallbackEvent` can return promptly otherwise lockups can happen.

```
... more code

case MediaEventType.MEEndOfPresentation:
    // Raised by a media source when a presentation ends. This event
    // signals that all streams in the presentation are complete. The
    // Media Session forwards this event to the application.

    // we cannot successfully .Finalize_ on the SinkWriter
    // if we call CloseAllMediaDevices directly from this thread
    // so we use an asynchronous method
    Task taskA = Task.Run(() => CloseAllMediaDevices());

    // we have to be on the form thread to update the screen
    ThreadSafeScreenUpdate(this, false, "Done");
    break;

... more code

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::HandleMediaSessionAsyncCallbackEvent
```

There are a variety of C# ways to run something as a separate thread and others which enable you to “*get back*” on the forms main thread. As you can see, the above sample code uses the C# `Task` mechanism to make a quick `CloseAllMediaDevices()` call followed by a call to the `ThreadSafeScreenUpdate()` function to handle the screen updating. Neither of these techniques will be discussed further here – they are pretty standard C# and you can easily find out more about them from online sources if you need to do so.

The `CloseAllMediaDevices()` function does deserve a bit of our attention though ...

```
private void CloseAllMediaDevices()
{
    HRESULT hr;
    LogMessage("CloseAllMediaDevices");

    // close and release our Callback Object
    if (mediaSessionAsyncCallbackHandler != null)
    {
        // stop any messaging or events in the Callback Object
        mediaSessionAsyncCallbackHandler.ShutDown();
        mediaSessionAsyncCallbackHandler = null;
    }

    // close the session (this is NOT the same as shutting it down)
    if (mediaSession != null)
    {
        hr = mediaSession.Close();
        if (hr != HRESULT.S_OK)
        {
            // just log it
            LogMessage("call to mediaSession.Close failed. Err=" + hr.ToString());
        }
    }
}
```

```
}

// Shut down the media source
if (mediaSource != null)
{
    hr = mediaSource.Shutdown();
    if (hr != HRESULT.S_OK)
    {
        // just log it
        LogMessage("call to mediaSource.Shutdown failed. Err=" + hr.ToString());
    }
    Marshal.ReleaseComObject(mediaSource);
    mediaSource = null;
}

// Shut down the media session (note we only closed it before).
if (mediaSession != null)
{
    hr = mediaSession.Shutdown();
    if (hr != HRESULT.S_OK)
    {
        // just log it
        LogMessage("call to mediaSession.Shutdown failed. Err=" + hr.ToString());
    }
    Marshal.ReleaseComObject(mediaSession);
    mediaSession = null;
}

// close the media sink
if (mediaSink != null)
{
    Marshal.ReleaseComObject(mediaSink);
    mediaSink = null;
}
}
```

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::HandleMediaSessionAsyncCallBackEvent

This is the only time we will document this particular procedure. All of the Tanta Sample Applications use it and, while each one is different, each is just releasing (and possibly finalizing) the Windows Media Foundation COM objects which are stored in class variables. As has been repeatedly stated - we must release everything WMF gives us. It should also be noted (bet you missed it in the above code) that closing down a Media Session is two-step process.

```
hr = mediaSession.Close();
... more code
hr = mediaSource.Shutdown();
```

First a call to `mediaSession.Close()` is made and then, sometime later, a call to `mediaSource.Shutdown()` happens. These calls must be made in that order and they are definitely *not* synonyms for each other. In order to properly terminate a Media Session, you must both close it and shut it down.

ERRORS IN THE PIPELINE

Errors happen, and any errors in the Pipeline will be routed back to the Media Session. This means if you are writing custom Pipeline components all you have to do to signal an error is throw a standard C# exception. In the Tanta Sample Projects, that error will

eventually appear as a call to the `HandleMediaSessionAsyncCallBackErrors` function – having been routed there by the `TantaAsyncCallbackHandler` Callback Object.

```

/// ++++++
/// <summary>
/// Handles errors reported by the media session TantaAsyncCallbackHandler
/// </summary>
/// <param name="sender">the object sending the event</param>
/// <param name="errMsg">the error message</param>
/// <param name="ex">the exception. Can be null</param>
/// <history>
///     01 Nov 18   Cynic - Originally Written
/// </history>
private void HandleMediaSessionAsyncCallBackErrors(object sender, string errMsg, Exception ex)
{
    if (errMsg == null) errMsg = "unknown error";
    LogMessage("HandleMediaSessionAsyncCallBackErrors Error" + errMsg);
    if (ex != null)
    {
        LogMessage("HandleMediaSessionAsyncCallBackErrors Exception trace = " + ex.StackTrace);
    }
    OISMessageBox("The media session reported an error\n\nPlease see the logfile.");
    // do everything to close all media devices
    CloseAllMediaDevices();
}

Source: TantaAudioFileCopyViaPipelineMP3Sink::frmMain::HandleMediaSessionAsyncCallBackErrors

```

The error handling code is impressively simple. This particular application makes no attempt to recover from an error and simply shuts everything down. Note, that as with the events coming back off the Media Session, we are not in the main form thread here. We don't have to be in this specific case, the `OISMessageBox()` call is designed to put itself back on the main form thread before it interacts with the screen and all of the `LogMessage()` type functionality is always thread safe as well.

DEALING WITH MULTIPLE STREAMS

So far, the discussion has revolved around a simple application which opens one Media Source, one Media Sink and has only one Media Stream. Obviously Pipelines get a bit more complex than that - and we will meet some in future chapters. For now, let's briefly consider the case where there is still only Media Source and one Media Sink but there are two streams – in other words we are copying an MP4 file which has both a video and audio stream.

The *TantaVideoFileCopyViaPipelineMP4Sink* Sample Project is designed to demonstrate this scenario. In the interests of space we will not discuss that code in detail. However, note that even though there are two streams, things are still fairly simple. The Media Types on each stream are different of course, but the output Media Type of the audio stream of the Media Source is still the same as the input Media Type on the audio stream of the Media Sink.

If you have been following the above discussion, you should be able to walk down the `PrepareSessionAndTopology` function in the `frmMain` class of the

TantaVideoFileCopyViaPipelineMP4Sink Sample Project without too much trouble. As you do this especially note how the audio and video streams are identified and separate input and output Topology nodes are prepared for each stream.

CREATING AN MP4 FILE SINK

The MP4 file sink is odd in that it wants both a video and/or audio Media Type to be specified at creation time. Since the *TantaVideoFileCopyViaPipelineMP4Sink* Sample Project just copies a file, we just give it the video and audio Media Types that the Media Source is producing. It does not necessarily have to be this way. We could, if we wished, specify different types – but then we would have to introduce conversion Transforms into the Topology. That sort of complication in the Pipeline is a topic for another chapter.

```

/// ++++++
/// <summary>
/// Opens the Media File Sink, both the video and audio types cannot be null
/// at the same time.
///
/// The caller must release the returned sink.
/// </summary>
/// <param name="outputFileName">the filename we write out to</param>
/// <param name="videoMediaType">the video media type - can be null</param>
/// <param name="audioMediaType">the audio media type - can be null</param>
/// <returns>an IMFMediaSink object or null for fail</returns>
/// <history>
/// 01 Nov 18 Cynic - Started
/// </history>
private IMFMediaSink OpenMediaFileSink(string outputFileName,
                                       IMFMediaType videoMediaType, IMFMediaType audioMediaType)
{
    HRESULT hr;
    IMFMediaSink workingSink = null;
    IMFByteStream outbyteStream = null;

    if ((outputFileName == null) || (outputFileName.Length == 0))
    {
        // we failed
        throw new Exception("OpenMediaFileSink: Invalid filename specified");
    }
    // either the video or audio type can be null but not both
    if ((videoMediaType == null) && (audioMediaType == null))
    {
        // we failed
        throw new Exception("OpenMediaFileSink: Both video and audio types are null");
    }

    try
    {
        // Create the media sink. We use the filename to create a byte stream and
        // then create the sink from that. The types configure the output

        // first we need a bytestream
        hr = MFExtern.MFCreateFile(MFFileAccessMode.ReadWrite,
                                   MFFileOpenMode.DeleteIfExists,
                                   MFFileFlags.None, outputFileName, out outbyteStream);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("Failed MFExtern.MFCreateFile, retVal=" + hr.ToString());
        }
    }
}

```

```

        if (outbyteStream == null)
        {
            // we failed
            throw new Exception("Failed to create Sink bytestream, Nothing will work.");
        }
        hr = MFExtern.MFCreateMPEG4MediaSink(outbyteStream, videoMediaType,
            audioMediaType, out workingSink);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("Failed MFCreateMPEG4MediaSink, retVal=" + hr.ToString());
        }
        if (workingSink == null)
        {
            // we failed
            throw new Exception("Failed to create media sink, Nothing will work.");
        }
    }
    catch (Exception ex)
    {
        // note this clean up is in the Catch block not the finally block.
        // if there are no errors we return it to the caller. The caller
        // is expected to clean up after itself
        if (workingSink != null)
        {
            // clean up. Nothing else has this yet
            Marshal.ReleaseComObject(workingSink);
            workingSink = null;
        }
        workingSink = null;
        throw ex;
    }

    return workingSink;
}

```

Source: TantaVideoFileCopyViaPipelineMP4Sink::frmMain::OpenMediaFileSink

As with the MP3 file sink, the MP4 file sink wants to operate on a byte stream – so we give it one. Unlike the MP3 file sink we also have to specify a video and audio Media Type as well.

```

hr = MFExtern.MFCreateMPEG4MediaSink(outbyteStream, videoMediaType,
    audioMediaType, out workingSink);

```

It is possible for one of these Media Types to be null but not both. Note that the act of creation here implicitly sets up separate video input and audio input streams on the MP4 file sink. We do not, at this point have any identifiers for them.

```

// Create the Video and Audio sink nodes. Note the use of the media type here. The
// MP4 File Sink creates two StreamSinks when it is created (video and audio) and we
// use the major media type in order to figure out which one is which. It is the
// StreamSink that gets added to the node, not the sink itself.
outputSinkNodeVideo = TantaWMFUtils.CreateSinkNodeForStream(mediaSink, MFMediaType.Video);

```

When time comes to create the output Topology Node, we call the Tanta CreateSinkNodeForStream function. This function accepts a Media Major Type value (video in the above code block) and uses it to identify the appropriate input stream on the MP4 file sink. It does this by obtaining the correct Stream Sink object (an IMFStreamSink) from the MP4 Media Sink. You ran into Stream Sinks in our earlier discussions. The relationship of a Stream Sink to a Media Sink is analogous to the relationship a Media Stream has to a Media Source. The Stream Sink will have a Media Major Type (video or audio) just like a Media Stream does and we can use this fact to get it from the MP4 file sink.

We will not reproduce the entire `CreateSinkNodeForStream()` function here – you can easily review it as you walk through the code. The major line of interest is the one which finds the Stream Sink.

```
// get the StreamSink
// streamSink = GetStreamSinkByMajorMediaType(mediaSink, majorMediaType);
```

The code for the `GetStreamSinkByMajorMediaType` function is below.

```
public static IMFStreamSink GetStreamSinkByMajorMediaType(IMFMediaSink mediaSink,
                                                         Guid majorMediaType)
{
    HRESULT hr;
    IMFStreamSink outStreamSink = null;
    IMFStreamSink workingStreamSink = null;
    IMFMediaTypeHandler workingMediaTypeHandler = null;
    IMFMediaType workingMediaType = null;
    Guid workingMajorType = Guid.Empty;

    if (mediaSink == null)
    {
        throw new Exception("GetStreamSinkByMajorMediaType No media sink object provided");
    }

    // we assume we will never have more streams sinks than this
    const int MAXSTREAMS = 10;

    // look at each possible stream sink on the media sink
    for (int streamIndex=0; streamIndex < MAXSTREAMS; streamIndex++)
    {
        try
        {
            // get the StreamSink
            hr = mediaSink.GetStreamSinkByIndex(streamIndex, out workingStreamSink);
            if (hr != HRESULT.S_OK)
            {
                throw new Exception("GetStreamSinkByIndex failed. Err=" + hr.ToString());
            }
            if (workingStreamSink == null)
            {
                throw new Exception("GetStreamSinkByIndex failed. workingStreamSink == null");
            }
            // get the media type handler from the steam sink
            hr = workingStreamSink.GetMediaTypeHandler(out workingMediaTypeHandler);
            if (hr != HRESULT.S_OK)
            {
                throw new Exception("GetMediaTypeHandler failed. Err=" + hr.ToString());
            }
            if (workingMediaTypeHandler == null)
            {
                throw new Exception("failed. workingMediaTypeHandler == null");
            }
            // get the current media type
            workingMediaTypeHandler.GetCurrentMediaType(out workingMediaType);
            if (hr != HRESULT.S_OK) continue;
            if (workingMediaType == null) continue;

            // get the major type
            hr = workingMediaType.GetMajorType(out workingMajorType);
            if (hr != HRESULT.S_OK) continue;
            if (workingMajorType == Guid.Empty) continue;

            if (workingMajorType == majorMediaType)
            {
                // make sure we do not release the workingStreamSink
                // which matches. The caller must do that
                outStreamSink = workingStreamSink;
                workingStreamSink = null;
                break;
            }
        }
        finally
        {
            if (workingStreamSink != null)
            {
                Marshal.ReleaseComObject(workingStreamSink);
            }
        }
    }
}
```

```

        workingStreamSink = null;
    }
    if (workingMediaTypeHandler != null)
    {
        Marshal.ReleaseComObject(workingMediaTypeHandler);
        workingMediaTypeHandler = null;
    }
    if (workingMediaType != null)
    {
        Marshal.ReleaseComObject(workingMediaType);
        workingMediaType = null;
    }
}

// by the time we get here the outStreamSink has either
// been set or it has not.
return outStreamSink;
}

Source: TantaCommon::TantaWMFUtils::GetStreamSinkByMajorMediaType

```

As you can see, the code looks similar to that used earlier in the `PrepareSessionAndTopology` call to identify the Media Streams in the Media Source. We literally had no other choice but to do it this way. We created the two Stream Sinks automatically in the MP4 file sink when we passed in the respective Media Types and we did not get back any Stream Sink objects or even stream ID values.

Once we have the Stream Sink, we can just give it directly to the Topology Node – as shown in the code below from the `CreateSinkNodeForStream` function. The Topology Node will be able to figure everything out from the Stream Sink – even the Media Sink itself.

```

// get the StreamSink
streamSink = GetStreamSinkByMajorMediaType(mediaSink, majorMediaType);

// Set the object pointer to the media stream sink
hr = outSinkNode.SetObject(streamSink);

```

Thus the output Topology Node is created and, once created, is added to the Topology. Once the nodes are all added, the Topology will contain four nodes. These are a source and sink node for the audio and a source and sink node for the video. The nodes are connected up in exactly the same way as the Topology nodes in the previous example: audio source node to audio sink node and video source node to video sink node.

You will note that the main difference between this example and the previous one is that we now have two parallel branches in the Topology. We have one Media Source with two streams feeding the branches and one Media Sink with two streams consuming the branches. This is where the power of the Media Session comes in. You do not have to do anything to move the data or synchronize its processing. The Media Session and Pipeline take care of all of that and you are sure the data will arrive at the sink at the proper time even though it is on two separate branches. The same is true when you have even more complicated Topologies with multiple Media Sources, Transforms, Tees and multiple Media Sinks

The remaining item which should be noted in regards to multiple streams in the Topology is that the connection process is not especially intuitive. The `PrepareSessionAndTopology` function connects them up with two statements as shown below.

```
hr = sourceVideoNode.ConnectOutput(0, outputSinkNodeVideo, 0);  
hr = sourceAudioNode.ConnectOutput(0, outputSinkNodeAudio, 0);
```

Both of the `ConnectOutput()` calls use an ID of 0 for the input stream and 0 for the output stream – yet the Media Source and Media Sink both have two streams each. How can this be? Well, the `ConnectOutput()` call on the Topology Node refers only to the stream the node “*knows about*”. In the above example, each node still has only one stream (stream ID 0 on the node) and so the above code works. On a “Tee” node you would have two output streams on the node and hence one of them would be the stream with an ID of 1.

Another thing to note is that we did not need to use any of this “Stream Sink” stuff when dealing with the MP3 file sink - although we probably could have done it that way if we wished. As was mentioned previously, each sink is somewhat different.

IMPLEMENTING THE READER-WRITER ARCHITECTURE

There are occasions where you need an easy way to read media data from a file. Of course it is trivial in C# to read a media file as binary data – but if you do all of the formatting and context is lost. For example, if you read the file as a binary object your code would need to understand the underlying file structure in considerable detail in order to present your application with a frame by frame sequence of video data. The Source Reader component solves this problem for reading media files and the Sink Writer component solves the similar problem when writing files.

Ultimately, the Source Reader presents your application with a sequence of Media Samples from a media file. The Sink Writer accepts a sequence of Media Samples from your application and writes them to a file.

Since the Source Reader provides Media Samples and the Sink Writer consumes them, they can make a nice binary pair with one feeding the other. Many of the examples you find on the Internet (and indeed some of the Tanta Samples) will structure the application that way. It should be emphasized that this Reader-Writer pairing is not a requirement. A Source Reader can be used independently as can a Sink Writer.

In this book, an application that uses either a Source Reader or a Sink Writer (or both) with no sign of a Media Session or Pipeline is considered to use the Reader-Writer Architecture. Applications that use a Pipeline to feed a Sink Writer are considered to use a Hybrid Architecture and you can read about them in the *Implementing a Hybrid Architecture* section below.

As discussed in detail in the *Synchronous vs Asynchronous Source Readers* section of *The WMF Components* chapter, there are two ways of obtaining data from a Source Reader. Either your application can sit in a loop and request each Media Sample in turn or it can set up a special Callback Object which provides a similar mechanism in a separate thread. In the section below, let's take a look at the first case – the simple “sit in a loop and consume the data” Reader-Writer Synchronous processing model.

THE SYNCHRONOUS READER-WRITER ARCHITECTURE

The *TantaAudioFileCopyViaReaderWriter* Sample Project implements a very simple Reader-Writer Synchronous Mode Architecture. It is the exact Reader-Writer analog of the Pipeline based *TantaAudioFileCopyViaPipelineMP3Sink* Sample Project - all the application does is copy an MP3 file.

Synchronous Mode processing also follows a general pattern. As a broad overview the steps are as follows...

1. Create the Source Reader by specifying the file name.
2. Create the Sink Writer by specifying the file name.
3. Look at each stream the Source Reader provides and find the one with the Media Major Type you are interested in.
4. Look in that stream and find the Media Type you are interested in. Make the chosen Media Type “current” on the stream.
5. Add a Stream to the Sink Writer and tell it the Media Type of the Media Samples it will be writing to the disk.
6. Tell the new stream on the Sink Writer the Media Type it will be receiving as input.
7. Tell the Sink Writer to begin writing – this initializes the file
8. Sit in a loop and call `ReadSample()` on the Source Reader and `WriteSample()` on the Sink Writer until there are no more Media Samples left.
9. Tell the Sink Writer to close the file.

Of course, if you are not using a Sink Writer, and are processing the data provided by the Source Reader yourself, then you can ignore the steps which reference the Sink Writer. Unlike the discussion of the Pipeline Architecture, we will not write out a second list with a more extensive sequence of actions - there isn't one. The Reader-Writer Architecture is designed to be simple. Instead let's look at some real code. The `CopyFile()` function in the `frmMain` class of the *TantaAudioFileCopyViaReaderWriter* Sample Project does all the work.

```
public void CopyFile(string sourceFileName, string outputFileName)
{
    HRESULT hr;

    int sinkWriterOutputAudioStreamId = -1;
    int audioSamplesProcessed = 0;
    bool audioStreamIsAtEOS = false;
    int sourceReaderAudioStreamId=-1;

    // not keen on endless loops. This is the maximum number
    // of streams we will check in the source reader.
    const int MAX_SOURCEREADER_STREAMS = 100;

    // create the SourceReader
    sourceReader = TantaWMFUtils.CreateSourceReaderSyncFromFile(
        sourceFileName, DEFAULT_ALLOW_HARDWARE_TRANSFORMS);
    if (sourceReader == null)
    {
        // we failed
        throw new Exception("CopyFile: Failed to create SourceReader, Nothing will work.");
    }
    // create the SinkWriter
    sinkWriter = TantaWMFUtils.CreateSinkWriterFromFile(
        outputFileName, DEFAULT_ALLOW_HARDWARE_TRANSFORMS);
    if (sinkWriter == null)
    {
        // we failed
        throw new Exception("CopyFile: Failed to create Sink Writer, Nothing will work.");
    }

    // find the first audio stream and identify the default Media Type
    // it is using. We could look into the stream and enumerate all of the
    // types on offer aand choose one from the list - but for a copy operation
    // the default will be quite suitable.

    sourceReaderNativeAudioMediaType = null;
    for (int streamIndex =0; streamIndex < MAX_SOURCEREADER_STREAMS; streamIndex++)
    {
        IMFMediaType workingType = null;
        Guid guidMajorType = Guid.Empty;

        // the the major media type - we are looking for audio
        hr = sourceReader.GetNativeMediaType(streamIndex, 0, out workingType);
        if (hr == HRESULT.MF_E_NO_MORE_TYPES) break;
        if (hr == HRESULT.MF_E_INVALIDSTREAMNUMBER) break;
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("failed GetNativeMediaType, retVal=" + hr.ToString());
        }
        if (workingType == null)
        {
            // we failed
            throw new Exception("failed on call to GetNativeMediaType, workingType == null");
        }

        // what major type does this stream have?
        hr = workingType.GetMajorType(out guidMajorType);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to GetMajorType failed. Err=" + hr.ToString());
        }
        if (guidMajorType == null)
        {
            throw new Exception("call to GetMajorType failed. guidMajorType == null");
        }
    }
}
```



```

    }

    // test for audio (there can be others)
    if ((guidMajorType == MFMediaType.Audio))
    {
        // this stream represents a audio type
        sourceReaderNativeAudioMediaType = workingType;
        sourceReaderAudioStreamId = streamIndex;
        // the sourceReaderNativeAudioMediaType will be released elsewhere
        break;
    }

    // if we get here release the type - we do not use it
    if (workingType != null)
    {
        Marshal.ReleaseComObject(workingType);
        workingType = null;
    }
}

// at this point we expect we can have a native video or a native audio media type
// or both, but not neither. if we don't we cannot carry on
if (sourceReaderNativeAudioMediaType == null)
{
    // we failed
    throw new Exception("failed sourceReaderNativeAudioMediaType == null");
}

// set the media type on the reader - this is the media type the source reader will output
// this does not have to match the media type in the file. If it does not the Source Reader
// will attempt to load a transform to perform the conversion. In this case we know it
// matches because the type we are using IS the same media type we got from the stream
hr = sourceReader.SetCurrentMediaType(
    sourceReaderAudioStreamId, null,
    sourceReaderNativeAudioMediaType);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("failed SetCurrentMediaType(a), retVal=" + hr.ToString());
}

// add a stream to the sink writer. The mediaType specifies the format of the
// samples that will be written to the file. Note that it does not necessarily
// need to match the format of the samples we provide to the sink writer. In
// this case, because we are copying a file, the media type
// we write to disk IS the media type the source reader reads from the disk.
hr = sinkWriter.AddStream(sourceReaderNativeAudioMediaType,
    out sinkWriterOutputAudioStreamId);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed adding the output stream, retVal=" + hr.ToString());
}

// Set the input format for a stream on the sink writer. Note the use of
// the stream index here. The input format does not have to match the output
// format that is written to the media sink. If the formats do not match, this call
// attempts to load an transform that can convert from the input format to the
// target format. If it cannot find one, and this is not a sure thing,
// it will throw an exception.
hr = sinkWriter.SetInputMediaType(sinkWriterOutputAudioStreamId,
    sourceReaderNativeAudioMediaType, null);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed SetInputMediaType(a), retVal=" + hr.ToString());
}

// begin writing on the sink writer
hr = sinkWriter.BeginWriting();
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("CopyFile: failed BeginWriting, retVal=" + hr.ToString());
}

// we sit in a loop here and get the sample from the source reader and write it out
// to the sink writer. An EOS (end of sample) value in the flags will signal the end.
// Note the application will appear to be locked up while we are in here. We are ok
// with this because it is quick and we want to keep things simple
while (true)

```

```

{
    int actualStreamIndex;
    MF_SOURCE_READER_FLAG actualStreamFlags;
    long timeStamp = 0;
    IMFSample workingMediaSample = null;

    // Request the next sample from the media source. Note that this could be
    // any type of media sample (video, audio, subtitles etc). We do not know
    // until we look at the stream ID. We saved the stream ID earlier when
    // we obtained the media types and so we can branch based on that.

    // In reality since we only set up one stream (audio) this will always be
    // the audio stream - but there is no need to assume this and the
    // TantaVideoFileCopyViaReaderWriter demonstrates an example with two
    // streams (audio and video)
    hr = sourceReader.ReadSample(
        TantaWMFUtils.MF_SOURCE_READER_ANY_STREAM,
        0,
        out actualStreamIndex,
        out actualStreamFlags,
        out timeStamp,
        out workingMediaSample
    );
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("Failed ReadSample on the reader, retVal=" + hr.ToString());
    }

    // the sample may be null if either end of stream or a stream tick is returned
    if (workingMediaSample == null)
    {
        // just ignore, the flags will have the information we need.
    }
    else
    {
        // the sample is not null
        if (actualStreamIndex == sourceReaderAudioStreamId)
        {
            // audio data
            // ensure discontinuity is set for the first sample in each stream
            if (audioSamplesProcessed == 0)
            {
                // audio data
                hr = workingMediaSample.SetUINT32(
                    MFAttributesClsid.MFSampleExtension_Discontinuity, 1);
                if (hr != HRESULT.S_OK)
                {
                    // we failed
                    throw new Exception("Failed SetUINT32, retVal=" + hr.ToString());
                }
                // remember this - we only do it once
                audioSamplesProcessed++;
            }
            hr = sinkWriter.WriteSample(sinkWriterOutputAudioStreamId, workingMediaSample);
            if (hr != HRESULT.S_OK)
            {
                // we failed
                throw new Exception("Failed WriteSample, retVal=" + hr.ToString());
            }
        }

        // release the sample
        if (workingMediaSample != null)
        {
            Marshal.ReleaseComObject(workingMediaSample);
            workingMediaSample = null;
        }
    }

    // do we have a stream tick event?
    if ((actualStreamFlags & MF_SOURCE_READER_FLAG.StreamTick) != 0)
    {
        if (actualStreamIndex == sourceReaderAudioStreamId)
        {
            // audio stream
            hr = sinkWriter.SendStreamTick(sinkWriterOutputAudioStreamId, timeStamp);
        }
        else
        {
            //
        }
    }
}

```

```

    }

    // is this stream at an END of Segment
    if ((actualStreamFlags & MF_SOURCE_READER_FLAG.EndOfStream) != 0)
    {
        // We have an EOS - but is it on the audio channel?
        if (actualStreamIndex == sourceReaderAudioStreamId)
        {
            // audio stream
            // have we seen this before?
            if (audioStreamIsAtEOS == false)
            {
                hr = sinkWriter.NotifyEndOfSegment(sinkWriterOutputAudioStreamId);
                if (hr != HRESULT.S_OK)
                {
                    // we failed
                    throw new Exception("NotifyEndOfSegment, retVal=" + hr.ToString());
                }
                audioStreamIsAtEOS = true;
            }
            // audio stream
        }
        else
        {
            {
            }

            // our exit condition depends on which streams are in use
            if (sourceReaderNativeAudioMediaType != null)
            {
                // only audio is active, if the audio stream is EOS we can leave
                if (audioStreamIsAtEOS == true) break;
            }
        }
    }
} // bottom of endless for loop

hr = sinkWriter.Finalize_();
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed sinkWriter.Finalize(), retVal=" + hr.ToString());
}
}

```

Source: TantaAudioFileCopyViaReaderWriter::frmMain::CopyFile

Rather lengthy isn't it? Well, yes it is, but if we review it section by section we can see there is a logical progression which mirrors the steps in the list above. The first thing we do is create the Source Reader and Sink Writer. Like the Pipeline Architecture discussion, both of these actions have been factored out in to static function calls in the TantaWMFUtils class. Unlike the Pipeline Architecture, both of these operations are pretty straight forward (even creating the Sink Writer).

```

sourceReader = TantaWMFUtils.CreateSourceReaderSyncFromFile(sourceFileName,
    DEFAULT_ALLOW_HARDWARE_TRANSFORMS);
sinkWriter = TantaWMFUtils.CreateSinkWriterFromFile(outputFileName,
    DEFAULT_ALLOW_HARDWARE_TRANSFORMS);

```

In the interests of saving space, the code for these two functions will not be discussed here. They were both previously reviewed in the *Source Reader and Sink Writer* section of *The WMF Components* chapter. One point to note though is the DEFAULT_ALLOW_HARDWARE_TRANSFORMS parameter – recall that both the Source Reader and Sink Writer will automatically load Transforms (if they can find them) to reconcile the Media Type and format they receive to the Media Type and format they are expected to output. Sometimes these Transforms can be hardware (as opposed to purely software) and this parameter controls that choice.

The next step is to sit in a loop and identify the stream on the output that we wish to use. Unlike with the Pipeline Architecture, all we will get here is an ID value of the stream rather than an `IMFMediaStream` object.

```
sourceReaderNativeAudioMediaType = null;
for (int streamIndex = 0; streamIndex < MAX_SOURCE_READER_STREAMS; streamIndex++)
{
    IMFMediaType workingType = null;
    Guid guidMajorType = Guid.Empty;

    // the the major media type - we are looking for audio
    hr = sourceReader.GetNativeMediaType(streamIndex, 0, out workingType);
    if (hr == HRESULT.MF_E_NO_MORE_TYPES) break;
    if (hr == HRESULT.MF_E_INVALID_STREAM_NUMBER) break;

    // what major type does this stream have?
    hr = workingType.GetMajorType(out guidMajorType);

    // test for audio (there can be others)
    if ((guidMajorType == MFMediaType.Audio))
    {
        // this stream represents a audio type
        sourceReaderNativeAudioMediaType = workingType;
        sourceReaderAudioStreamId = streamIndex;
        // the sourceReaderNativeAudioMediaType will be released elsewhere
        break;
    }

    // if we get here release the type - we do not use it
    if (workingType != null)
    {
        Marshal.ReleaseComObject(workingType);
        workingType = null;
    }
}
```

Once the loop has exited we will have the ID of the stream containing the audio data and also the default Media Type on that stream. It should be noted here that there are some things that are not done in the above code. For example, there may well be numerous Media Types on offer within the stream. We could enumerate the Media Types and make one of them “current” by using the `SetCurrentMediaType()` call. We do not do that here because we are doing a file copy operation. This means default output Media Type is probably good enough and the Source Reader or Sink Writer will automatically handle the conversions for us anyways if necessary. That, in essence, is why the Source Reader and Sink Writer are so popular – the programmer does not have to care as much about Media Type conversions. The only other place you get a free pass like that is with the Pipeline Architecture in file playback situations. Also observe that, in the above code section, any Media Types that are not used are released – using a Source Reader or Sink Writer does not relieve you of that responsibility.

So, we now have a Media Type and the ID of a suitable stream on the Source Reader. The next thing we do is use this Media Type to define the output Media Type of the Source Reader.

```
// set the media type on the reader - this is the media type the source reader will output
// this does not have to match the media type in the file. If it does not the Source Reader
// will attempt to load a transform to perform the conversion. In this case we know it
// matches because the type we are using IS the same media type we got from the stream
hr = sourceReader.SetCurrentMediaType(sourceReaderAudioStreamId, null,
```

```
sourceReaderNativeAudioMediaType);
```

Note the use of the `sourceReaderAudioStreamId` variable as a parameter. There is a shortcut you will sometimes see in use which is the use of a special hex stream ID value of `0xffffffffd`. This tells the Source Reader to “*just use the first audio stream you find*” and is commonly defined as the constant `MF_SOURCE_READER_FIRST_AUDIO_STREAM`. If you see that constant in use, you will probably also find the application did not enumerate the streams to find the Media Type. There is a similar `MF_SOURCE_READER_FIRST_VIDEO_STREAM` constant for video data.

Proceeding onwards, the code adds a stream to the Sink Writer. Note that a stream on a Sink Writer is called a Stream Sink and is of type `IMFStreamSink`. A Stream Sink is not a Media Stream (an `IMFMediaStream`) and the `IMFStreamSink` and `IMFMediaSink` interfaces do not inherit from each other.

The Media Type is also passed in when the Stream Sink is created and this defines the output format of the Media Samples which the Sink Writer will write.

```
// add a stream to the sink writer.
// The mediaType specifies the format of the samples that will be written
// to the file. Note that it does not necessarily need to match the format of the samples
// we provide to the sink writer. In this case, because we are copying a file, the media type
// we write to disk IS the media type the source reader reads from the disk.
hr = sinkWriter.AddStream(sourceReaderNativeAudioMediaType, out sinkWriterOutputAudioStreamId);
```

As can be seen, the above code simply uses the Media Type obtained from the stream on the Source Reader (the `sourceReaderNativeAudioMediaType`) to configure the output format. It does not have to be this way and if you specify a different Media Type then your copy operation will be a “*media conversion*” application.

At this point, the Sink Writer knows the Media Type in which it will write out the data – but what about the Media Type it will be receiving the data? This is configured in the next step.

```
// Set the input format for a stream on the sink writer. Note the use of the stream index here
// The input format does not have to match the output format that is written to the media sink
// If the formats do not match, this call attempts to load a transform that can convert from
// the input format to the target format. If it cannot find one, and this is not a sure thing,
// it will throw an exception.
hr = sinkWriter.SetInputMediaType(sinkWriterOutputAudioStreamId,
    sourceReaderNativeAudioMediaType, null);
```

Again, to keep things simple, we are again using the Media Type obtained from the stream on the Source Reader. This has to match the output Media Type of the Source Reader or we would have to perform some conversions on it ourselves in between reading the data and writing it. As mentioned previously, this input Media Type does not have to match the actual Media Type of the Media Samples being written – automatic conversions will be invoked if necessary.

Both the Source Reader and Sink Writer are now configured. The next step just initializes the output file on the Sink Writer.

```
// begin writing on the sink writer
hr = sinkWriter.BeginWriting();
```

Note that the above statement just initializes the Sink Writer and creates the output file. No data is being transferred yet. We have to do that task ourselves in a loop – as is shown in the next code block.

```
// we sit in a loop here and get the sample from the source reader and write it out
// to the sink writer. An EOS (end of sample) value in the flags will signal the end.
// Note the application will appear to be locked up while we are in here. We are ok
// with this because it is quick and we want to keep things simple
while (true)
{
    int actualStreamIndex;
    MF_SOURCE_READER_FLAG actualStreamFlags;
    long timeStamp = 0;
    IMFSample workingMediaSample = null;

    // Request the next sample from the media source. Note that this could be
    // any type of media sample (video, audio, subtitles etc). We do not know
    // until we look at the stream ID. We saved the stream ID earlier when
    // we obtained the media types and so we can branch based on that.

    // In reality since we only set up one stream (audio) this will always be
    // the audio stream - but there is no need to assume this and the
    // TantaVideoFileCopyViaReaderWriter demonstrates an example with two
    // streams (audio and video)
    hr = sourceReader.ReadSample(
        TantaWMFUtils.MF_SOURCE_READER_ANY_STREAM,
        0,
        out actualStreamIndex,
        out actualStreamFlags,
        out timeStamp,
        out workingMediaSample
    );

    // the sample may be null if either end of stream or a stream tick is returned
    if (workingMediaSample == null)
    {
        // just ignore, the flags will have the information we need.
    }
    else
    {
        // the sample is not null
        if (actualStreamIndex == sourceReaderAudioStreamId)
        {
            // audio data
            // ensure discontinuity is set for the first sample in each stream
            if (audioSamplesProcessed == 0)
            {
                // audio data
                hr = workingMediaSample.SetUINT32(
                    MFAttributesClsid.MFSampleExtension_Discontinuity, 1);
                if (hr != HRESULT.S_OK)
                {
                    // we failed
                    throw new Exception("CopyFile: SetUINT32, retVal=" + hr.ToString());
                }
                // remember this - we only do it once
                audioSamplesProcessed++;
            }
            hr = sinkWriter.WriteSample(sinkWriterOutputAudioStreamId, workingMediaSample);
        }

        // release the sample
        if (workingMediaSample != null)
        {
            Marshal.ReleaseComObject(workingMediaSample);
            workingMediaSample = null;
        }
    }
}
```

```
... code to detect the end of the stream removed
} // bottom of endless for loop
```

The code which detects the end of the data on the stream has been removed. If we ignore that code for the moment we can see that the loop really is very simple. The process runs as follows: we loop endlessly, and call `ReadSample()` on the Source Reader. If the Media Sample is the first one that we have seen, a `MFSampleExtension_Discontinuity` flag is set in it's Attributes. Either way, the next step is to call `WriteSample()` on the Sink Writer and then release the Media Sample object. Read, write and release – that is all we are doing until we get the signal that the stream has ended.

Reading a null Media Sample from the Source Reader does not necessarily imply that the stream is at an end. The Source Reader will use it to send signals of many types to the object processing the data (i.e. the application). The flags value received on the `ReadSample()` call will allow us to interpret the messages. The sample code block below shows how this process works.

```
// do we have a stream tick event?
if ((actualStreamFlags & MF_SOURCE_READER_FLAG.StreamTick) != 0)
{
    if (actualStreamIndex == sourceReaderAudioStreamId)
    {
        // audio stream
        hr = sinkWriter.SendStreamTick(sinkWriterOutputAudioStreamId, timeStamp);
    }
    else
    {
    }
}

// is this stream at an END of Segment
if ((actualStreamFlags & MF_SOURCE_READER_FLAG.EndOfStream) != 0)
{
    // We have an EOS - but is it on the audio channel?
    if (actualStreamIndex == sourceReaderAudioStreamId)
    {
        // audio stream
        // have we seen this before?
        if (audioStreamIsAtEOS == false)
        {
            hr = sinkWriter.NotifyEndOfSegment(sinkWriterOutputAudioStreamId);
            if (hr != HRESULT.S_OK)
            {
                // we failed
                throw new Exception("CopyFile: NotifyEndOfSegment, retVal=" + hr.ToString());
            }
            audioStreamIsAtEOS = true;
        }
        // audio stream
    }
    else
    {
    }
}

// our exit condition depends on which streams are in use
if (sourceReaderNativeAudioMediaType != null)
{
    // only audio is active, if the audio stream is EOS we can leave
    if (audioStreamIsAtEOS == true) break;
}
}
```

If the `MF_SOURCE_READER_FLAG.StreamTick` flag is present, all we need to do is send that message on to the Sink Writer along with the current timestamp. The Sink Writer will know how to handle it.

If we have a `MF_SOURCE_READER_FLAG.EndOfStream` flag, we simply tell the Sink Writer to close down with a `NotifyEndOfSegment()` call and then break out of the loop;

```
hr = sinkWriter.NotifyEndOfSegment(sinkWriterOutputAudioStreamId);
audioStreamIsAtEOS = true;

... some code

if (audioStreamIsAtEOS == true) break;
```

After the loop exits we have to ensure the Sink Writer closes down the output file properly. The `NotifyEndOfSegment()` call does not do this.

```
hr = sinkWriter.Finalize_();
```

That is pretty much it as far as the Synchronous Reader-Writer Architecture goes. We set up the Source Reader and Sink Writer and then sit in a loop reading, writing and passing on any messages we see. If we get an end of stream signal we tell the Sink Writer about it and exit the loop. All-in-all it really is a pretty simple process.

DEALING WITH MULTIPLE STREAMS

You may have gotten the impression that because the Reader-Writer Architecture is simpler to use than the Pipeline that it cannot handle multiple streams. This is definitely not true. The *TantaVideoFileCopyViaReaderWriter* Sample project demonstrates the process of copying an MP4 file and these have both video and audio streams.

We will not reproduce the entire two stream code from that project – if you are interested have a look at the `CopyFile` function in the `frmMain` class of that project. Let's talk about the main differences though.

1. A single Source Reader and Sink Writer are used, as before.
2. The stream ID and Media Type of the first video stream in the Source Reader are found.
3. The stream ID and Media Type of the first audio stream in the Source Reader are found.
4. In exactly the same way that we did in the single stream example, we configure the output Media Type on each Media Stream of the Source Reader and create and set the Media Type on each Sink Stream of the Sink Writer.

5. We enter a loop, and read the Media Sample. This Media Sample will contain either audio or video data. We detect this and take care to write it to the appropriate stream on the Media Sink.
6. The close down process is slightly more complicated because we have to get an `MF_SOURCE_READER_FLAG.EndOfStream` flag on both streams before we can quit the loop.

Let us have a closer look at the loop which processes multiple streams.

```
// we sit in a loop here and get the sample from the source reader and write it out
// to the sink writer. An EOS (end of sample) value in the flags will signal the end.
while (true)
{
    int actualStreamIndex;
    MF_SOURCE_READER_FLAG actualStreamFlags;
    long timeStamp = 0;
    IMFSample workingMediaSample = null;

    // Request the next sample from the media source. Note that this could be
    // any type of media sample (video, audio, subtitles etc). We do not know
    // until we look at the stream ID. We saved the stream ID earlier when
    // we obtained the media types and so we can branch based on that.
    hr = sourceReader.ReadSample(
        TantaWMFUtils.MF_SOURCE_READER_ANY_STREAM,
        0,
        out actualStreamIndex,
        out actualStreamFlags,
        out timeStamp,
        out workingMediaSample
    );
    if (hr != HRESULT.S_OK)
    {
        // we failed
        throw new Exception("CopyFile: Failed ReadSample, retVal=" + hr.ToString());
    }

    // the sample may be null if either end of stream or a stream tick is returned
    if (workingMediaSample == null)
    {
        // just ignore, the flags will have the information we need.
    }
    else
    {
        // the sample is not null
        if (actualStreamIndex == sourceReader.AudioStreamId)
        {
            // audio data
            // ensure discontinuity is set for the first sample in each stream
            if (audioSamplesProcessed == 0)
            {
                // audio data
                hr = workingMediaSample.SetUINT32(
                    MFAttributesClsid.MFSampleExtension_Discontinuity, 1);
                if (hr != HRESULT.S_OK)
                {
                    // we failed
                    throw new Exception("CopyFile: Failed SetUINT32, retVal=" + hr.ToString());
                }
                // remember this - we only do it once
                audioSamplesProcessed++;
            }
            hr = sinkWriter.WriteSample(sinkWriter.OutputAudioStreamId, workingMediaSample);
            if (hr != HRESULT.S_OK)
            {
                // we failed
                throw new Exception("CopyFile: WriteSample, retVal=" + hr.ToString());
            }
        }
        else if (actualStreamIndex == sourceReader.VideoStreamId)
        {
            // video data
            // ensure discontinuity is set for the first sample in each stream
            if (videoSamplesProcessed == 0)
```

```

        {
            // video data
            hr = workingMediaSample.SetUINT32(
                MFAttributesClsid.MFSampleExtension_Discontinuity, 1);
            if (hr != HRESULT.S_OK)
            {
                // we failed
                throw new Exception("Failed SetUINT32, retVal=" + hr.ToString());
            }
            // remember this - we only do it once
            videoSamplesProcessed++;
        }
        hr = sinkWriter.WriteSample(sinkWriterOutputVideoStreamId, workingMediaSample);
        if (hr != HRESULT.S_OK)
        {
            // we failed
            throw new Exception("Failed WriteSample, retVal=" + hr.ToString());
        }
    }

    // release the sample
    if (workingMediaSample != null)
    {
        Marshal.ReleaseComObject(workingMediaSample);
        workingMediaSample = null;
    }
}

// do we have a stream tick event?
if ((actualStreamFlags & MF_SOURCE_READER_FLAG.StreamTick) != 0)
{
    if (actualStreamIndex == sourceReaderVideoStreamId)
    {
        // video stream
        hr = sinkWriter.SendStreamTick(sinkWriterOutputVideoStreamId, timeStamp);
    }
    else if (actualStreamIndex == sourceReaderAudioStreamId)
    {
        // audio stream
        hr = sinkWriter.SendStreamTick(sinkWriterOutputAudioStreamId, timeStamp);
    }
    else
    {
    }
}

// is this stream at an END of Segment
if ((actualStreamFlags & MF_SOURCE_READER_FLAG.EndOfStream) != 0)
{
    // We have an EOS - but is it on the video or audio channel?
    // we have to get it on both
    if (actualStreamIndex == sourceReaderVideoStreamId)
    {
        // video stream
        // have we seen this before?
        if (videoStreamIsAtEOS == false)
        {
            hr = sinkWriter.NotifyEndOfSegment(sinkWriterOutputVideoStreamId);
            if (hr != HRESULT.S_OK)
            {
                // we failed
                throw new Exception("Failed NotifyEndOfSegment, retVal=" + hr.ToString());
            }
            videoStreamIsAtEOS = true;
        }
    }
    else if (actualStreamIndex == sourceReaderAudioStreamId)
    {
        // audio stream
        // have we seen this before?
        if (audioStreamIsAtEOS == false)
        {
            hr = sinkWriter.NotifyEndOfSegment(sinkWriterOutputAudioStreamId);
            if (hr != HRESULT.S_OK)
            {
                // we failed
                throw new Exception("Failed NotifyEndOfSegment, retVal=" + hr.ToString());
            }
            audioStreamIsAtEOS = true;
        }
    }
    // audio stream
}

```

```

    }
    else
    {
    }
    // our exit condition depends on which streams are in use
    if ((sourceReaderNativeVideoMediaType != null) &&
        (sourceReaderNativeAudioMediaType != null))
    {
        // if both streams are at EOS we can leave
        if ((videoStreamIsAtEOS == true) && (audioStreamIsAtEOS == true)) break;
    }
    else if (sourceReaderNativeVideoMediaType != null)
    {
        // only video is active, if the video stream is EOS we can leave
        if (videoStreamIsAtEOS == true) break;
    }
    else if (sourceReaderNativeAudioMediaType != null)
    {
        // only audio is active, if the audio stream is EOS we can leave
        if (audioStreamIsAtEOS == true) break;
    }
    }
} // bottom of endless for loop
Source: TantaVideoFileCopyViaPipelineMP4Sink::frmMain::CopyFile

```

As mentioned previously, we will not discuss the multiple-stream case in detail here. However, if you read down the above code block it becomes apparent that the presence of multiple streams presents little problem for the Sink Writer. Note however, how the code carefully checks for an end of stream signal on both streams before it breaks out of the loop.

THE ASYNCHRONOUS READER-WRITER ARCHITECTURE

As discussed in the *Synchronous vs Asynchronous Source Readers* section of *The WMF Components* chapter, the Asynchronous Reader-Writer Architecture was designed to enable the loop which processes the `ReadSample()`, `WriteSample()` sequence to be performed in a separate thread. If you think back to the previous section – both of the sample projects used there will appear to be frozen or “locked-up” until the loop ends. Even in those simple examples, it would probably have been better to spin up the processing loop in a separate C# thread.

The ability of C# to easily send processing to separate threads has rendered the Asynchronous Reader-Writer Architecture somewhat redundant in WMF.net. Nonetheless you will see it used in various code examples around the Internet and it is worthwhile understanding how it functions. The *TantaCaptureToFileViaReaderWriter* Sample Project uses a Source Reader in Asynchronous Mode to process the video data obtained from a webcam. A Sink Writer is used to save this information to the disk as an MP4 file. In order to keep things simple, no audio data is processed – although a separate Source Reader could easily be set up to obtain an audio stream from a microphone which is then recorded alongside the video in the MP4 file.

Here is a broad overview of Asynchronous Reader-Writer Architecture.

1. The Source Reader and Sink Writer are set up in the same way as is done in Synchronous Mode.
2. A Callback Object implementing the `IMFSourceReaderCallback` interface is given to the Source Reader when it is created.
3. The Source Reader returned will be an `IMFSourceReaderAsync` object not an `IMFSourceReader` as in the Synchronous Mode.
4. The first `ReadSample()` call is made on the Source Reader
5. The Media Sample appears in a function of the Callback Object and that function calls `WriteSample()` on the Media Sink and then requests the next Media Sample with a `ReadSample()` call.
6. The loop proceeds inside the Callback Object via a sequence of `WriteSample()` and `ReadSample()` calls until the stream runs out of data.

All of the code of interest is in the `TantaCaptureToFileViaReaderWriter` Sample Project - either in the `CaptureToFile` function of the `frmMain` class or in the `TantaSourceReaderCallbackHandler` class itself. Let's join the code in the `CaptureToFile` function where we create the Asynchronous Source Reader.

```
... more code

// create a new Callback Object. This, once we get it all wired up, will act
// as a pump to move the data from the source to the sink
workingSourceReaderCallBackHandler = new TantaSourceReaderCallbackHandler();

// create the source reader
workingSourceReader = TantaWMFUtils.CreateSourceReaderAsyncFromDevice(currentDevice,
    workingSourceReaderCallBackHandler);

... more code

Source: TantaCaptureToFileViaReaderWriter::frmMain::CaptureToFile
```

The operation of the `Tanta CreateSourceReaderAsyncFromDevice()` function was discussed in the *Creating a Source Reader on a Device* section of *The WMF Components* chapter and will not be reproduced here. The main point to remember is that, at this point, we now have an `IMFSourceReaderAsync` object and it knows about the Callback Object. The next thing we do is give the Callback Object the information it needs.

```
... more code

// now set the source and the sink in the Callback Object. It needs to know these
// in order to operate
workingSourceReaderCallBackHandler.SourceReader = workingSourceReader;
workingSourceReaderCallBackHandler.SinkWriter = workingSinkWriter;
workingSourceReaderCallBackHandler.InitForFirstSample();
workingSourceReaderCallBackHandler.SourceReaderAsyncCallbackError =
    HandleSourceReaderAsyncCallbackErrors;

... more code

Source: TantaCaptureToFileViaReaderWriter::frmMain::CaptureToFile
```

As we usually do with Callback Objects in the Tanta samples, we set up a Delegate/Event Mechanism to pass information back to the application. In this particular case we only have an error handler, any notification type events sent by the Source Reader will be handled within the Callback Object itself.

The data will be entirely processed within the `OnReadSample()` function of the Callback Object but before we get to that let us look at the action that triggers the first appearance of the Media Sample in that location.

```
... more code

// Request the first video frame from the media source. The TantaSourceReaderCallbackHandler
// set up earlier will be invoked and it will continue requesting and processing video
// frames after that.
hr = workingSourceReader.ReadSample(
    TantaWMFUtils.MF_SOURCE_READER_FIRST_VIDEO_STREAM,
    0,
    IntPtr.Zero,
    IntPtr.Zero,
    IntPtr.Zero,
    IntPtr.Zero
);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed first ReadSample on the reader, retVal=" + hr.ToString());
}

... more code

Source: TantaCaptureToFileViaReaderWriter::frmMain::CaptureToFile
```

Yes, it is our old friend the `ReadSample()` call. If you read on past that point you will see that, other than a few lines of admin code and some `ReleaseComObject()` calls to clean up, the code exits the `CaptureToFile` function and the entire copy process proceeds independently in its own thread within in the Callback Object.

The `OnReadSample` function in the Callback Object is similarly straightforward. We will reproduce it here in its entirety so you can see the flow of control.

```
/// ++++++
/// <summary>
/// This gets called when a Called IMFSourceReader.ReadSample method completes
/// (assuming the SourceReader has been given this class during setup with
/// an attribute of MFAttributesClsid.MF_SOURCE_READER_ASYNC_CALLBACK).
///
/// The first ReadSample triggers it after that it continues by itself
/// </summary>
/// <param name="hrStatus">The status code. If an error occurred while
///     processing the next sample, this parameter contains the error code.</param>
/// <param name="streamIndex">The zero-based index of the stream that
///     delivered the sample.</param>
/// <param name="streamFlags">A bitwise OR of zero or more flags from the
///     MF_SOURCE_READER_FLAG enumeration.</param>
/// <param name="sampleTimeStamp">The time stamp of the sample, or the time
///     of the stream event indicated in streamFlags. The time is given
///     in 100-nanosecond units. </param>
/// <param name="mediaSample">A pointer to the IMFSample interface of a media sample.
///     This parameter might be NULL.</param>
/// <returns>Returns an HRESULT value. Reputedly, the source reader
///     ignores the return value.</returns>
/// <history>
///     01 Nov 18 Cynic - Started
/// </history>
public HRESULT OnReadSample(HRESULT hrStatus, int streamIndex,
```

```

        MF_SOURCE_READER_FLAG streamFlags, long sampleTimeStamp,
        IMFSample mediaSample)
{
    HRESULT hr = HRESULT.S_OK;
    try
    {
        lock (this)
        {
            // are we capturing? if not leave
            if (IsCapturing() == false)
            {
                return HRESULT.S_OK;
            }

            // have we got an error?
            if (Failed(hrStatus))
            {
                string errMsg = "OnReadSample, Error on call =" + hrStatus.ToString();
                SourceReaderAsyncCallbackError(this, errMsg, null);
                return hrStatus;
            }

            // have we got a sample? It seems this can be null on the first sample
            // in after the ReadSample that triggered this. So we just ignore it
            // and request the next to get things rolling
            if (mediaSample != null)
            {
                // we have a sample, if so is it the first non null one?
                if (isFirstSample)
                {
                    // yes it is set up our timestamp
                    firstSampleBaseTime = sampleTimeStamp;
                    isFirstSample = false;
                }

                // write the sample out
                hr = sinkWriter.WriteSample(0, mediaSample);
                if (Failed(hr))
                {
                    string errMsg = "OnReadSample, Error on WriteSample =" + hr.ToString();
                    SourceReaderAsyncCallbackError(this, errMsg, null);
                    return hr;
                }
            }

            // Read another sample.
            hr = (sourceReader as IMFSOURCE_READER_ASYNC).ReadSample(
                TantaWMFUtils.MF_SOURCE_READER_FIRST_VIDEO_STREAM,
                0,
                IntPtr.Zero, // actual
                IntPtr.Zero, // flags
                IntPtr.Zero, // timestamp
                IntPtr.Zero // sample
            );
            if (Failed(hr))
            {
                string errMsg = "OnReadSample, Error on ReadSample =" + hr.ToString();
                SourceReaderAsyncCallbackError(this, errMsg, null);
                return hr;
            }
        }
    }
    catch (Exception ex)
    {
        if (SourceReaderAsyncCallbackError != null)
        {
            SourceReaderAsyncCallbackError(this, ex.Message, ex);
        }
    }
    finally
    {
        SafeRelease(mediaSample);
    }

    return hr;
}

```

Source: TantaCaptureToFileViaReaderWriter::TantaSourceReaderCallbackHandler::OnReadSample

As you can see, the above code is not much more than the simple Synchronous Mode `ReadSample()`, `WriteSample()` loop except that in this particular case there is no need for any code to detect the end of the stream. The reason for this is simple, since the media data is coming off of a webcam there *is* no end to the stream. The capturing process ends when the user presses the “Stop Capture” button on the GUI. Since the `OnReadSample` code executes in a thread provided by the Source Reader the users screen remains unaffected and is continuously available.

Other things to note in the `OnReadSample` function are that the parameters passed in are pretty much identical to the ones you would get back from a `ReadSample()` call on the Source Reader operating in Synchronous Mode.

In addition, particularly note that that the initial `ReadSample()` in the main application form triggered the first call to the `OnReadSample` function and the `ReadSample()` call at the bottom of that function ensures the next Media Sample will arrive in the same way. In other words, the `OnReadSample` function effectively triggers itself once it has been set running. Note also that everything happens inside a `Lock()` construct. The Source Reader shouldn’t send a new Media Sample until the currently executing `OnReadSample()` call returns - but it doesn’t hurt to be sure.

The remainder of the *TantaCaptureToFileViaReaderWriter* Sample Project will be discussed in more detail in the *Capture with a Reader-Writer Architecture* section of the *Capturing Camera Data* chapter.

IMPLEMENTING A HYBRID ARCHITECTURE

Now that you know about the Pipeline Architecture and the Reader-Writer Architecture the following statement should make sense to you.

A Hybrid Architecture is simply a Pipeline Architecture in which one or more of the components of the Pipeline intercepts the Media Samples and feeds copies of them to a Sink Writer.

Thus, in a Hybrid Architecture, the data still proceeds from the Media Source to the Media Sink under the control of a Media Session. The Topology can be as complex as any other standard Pipeline you might wish to make.

The components that intercept the Media Samples as they pass through the Pipeline and pass them to a Sink Writer are either a specialized Media Sink designed for that purpose or a custom user written Transform.

The name of the specialized Media Sink is called the Sample Grabber Sink and both the *TantaAudioFileCopyViaPipelineAndWriter* and the *TantaVideoFileCopyViaPipelineAndWriter* Sample Projects implement it. The difference between the two is, as usual, that the *TantaVideoFileCopyViaPipelineAndWriter* project implements the process with two streams (audio and video).

The usage of a Transform to intercept the samples and feed them to a Sink Writer is not documented and there do not seem to be any other examples of that technique available on the Internet at the time this book was written. The *MFTTantaSampleGrabber_Sync* class in the *TantaCaptureToScreenAndFile* Sample Project implements this methodology. This technique was chosen both for demonstration purposes, and because it really simplified the requirement to repeatedly turn video capturing on and off on a video pipeline while still recording the full stream of data to a file. Normally this sort of thing would be done by introducing a Tee node into the Topology and the resulting two branches would feed both an MP4 File Sink and a Sample Grabber Sink.

We will leave a discussion of the *MFTTantaSampleGrabber_Sync* class in the *TantaCaptureToScreenAndFile* Sample Project to the *Capture with a Hybrid Architecture* section of the *Capturing Camera Data* chapter since it is more relevant there. For now, let's take a look at the Sample Grabber Sink and see how it is configured and how it passes the media data to a Sink Writer. The code section below is taken from the *TantaAudioFileCopyViaPipelineAndWriter* Sample Project.

The process of setting up the Topology is completely standard (as discussed in the *Implementing the Pipeline Architecture* section above) until we get to the part where the Media Sink is created.

```
... more code

// Create the sample grabber sink Callback Object.
sampleGrabberSinkCallback = new TantaSampleGrabberSinkCallback();

// create the activator for the sample grabber sink
hr = MFExtern.MFCreateSampleGrabberSinkActivate(currentAudioMediaType,
                                                sampleGrabberSinkCallback, out sampleGrabberSinkActivate);
if (sampleGrabberSinkActivate == null)
{
    throw new Exception("call to sampleGrabberSinkActivate == null");
}

// To run as fast as possible, set this attribute (requires Windows 7):
hr = sampleGrabberSinkActivate.SetUINT32(
```



```
MFAttributesClsid.MF_SAMPLEGRABBERSINK_IGNORE_CLOCK, 1)
... more code
Source: TantaAudioFileCopyViaPipelineAndWriter::frmMain::PrepareSessionAndTopology
```

The Sample Grabber Sink is a Microsoft supplied WMF component. The only way to create one is through the use of an Activator and we get that Activator by calling the static `MFCreatSampleGrabberSinkActivate` function. As you have no doubt noticed, the Sample Grabber Sink requires the use of a Callback Object. The mechanism by which the Media Samples are fed to the Sink Writer is analogous to the way the Asynchronous Source Reader works. The Callback Object type in the above code is `TantaSampleGrabberSinkCallback` but that class just directly inherits from the Windows Media Foundation `IMFSampleGrabberSinkCallback2` interface.

Also note the use of the `MF_SAMPLEGRABBERSINK_IGNORE_CLOCK` GUID as an Attribute key. The Sample Grabber Sink requires this Attribute to be set to true (1) or the Media Session will simply send the media data through the Pipeline at normal playback speeds. Without this flag, the copy would take as long as the file takes to play. Neither the MP3 or MP4 file sink seem to require this flag – probably it is implemented by default.

The Sample Grabber Sink functions exactly like the MP3 (or MP4) file sink except that it just discards every Media Sample it obtains. However, prior to calling `ReleaseComObject()` on the Media Sample, the Sample Grabber Sink also calls the `OnProcessSample` function in the Callback Object (or `OnProcessSampleEx` if you are using the more advanced `IMFSampleGrabberSinkCallback2` interface). Other than the Callback Object, the Sample Grabber Sink is totally opaque to your program – you really have no idea what is going on in there.

As with all Callback Objects, it is up to you to write them. The `TantaSampleGrabberSinkCallback` is one example but you can structure them however you wish – as long as you correctly support the required Callback interface.

The Sink Writer was also created in the `PrepareSessionAndTopology` function and, after it was created, it was passed into the `TantaSampleGrabberSinkCallback` as a class variable. Note that, as with the Asynchronous Source Reader Callback Object, only an error handling event is needed. The Media Session still controls the entire Pipeline and the Sink Writer and the Callback Object are not a part of it.

```
... more code
// open up the Sink Writer
workingSinkWriter = OpenSinkWriter(outputFileName);
if (workingSinkWriter == null)
{
    MessageBox.Show("OpenSinkWriter did not return a media sink. Cannot continue.");
    return;
}
```

```
// now set the the sink in the Callback Object. It needs to know this
// in order to operate
sampleGrabbersSinkCallback.SinkWriter = workingSinkWriter;
sampleGrabbersSinkCallback.InitForFirstSample();
sampleGrabbersSinkCallback.SampleGrabberAsyncCallbackError =
    HandleSampleGrabberAsyncCallBackErrors;

... more code
```

Source: TantaAudioFileCopyViaPipelineAndWriter::frmMain::PrepareSessionAndTopology

The Media Session is started up in the standard way, the Topology is resolved, and once the Media Session is started the data begins to arrive in the `OnProcessSampleEx` function of the Callback Object. The code for this function is listed below. In particular, note that no Media Sample object is passed in on any of the parameters to the call.

```
// +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+  
/// <summary>  
/// Called when the sample grabber sink processes a sample. We can use this  
/// to do what we want with the media data  
/// </summary>  
/// <param name="guidMajorMediaType">the media type</param>  
/// <param name="sampleFlags">the sample flags</param>  
/// <param name="sampleSize">the sample size</param>  
/// <param name="sampleDuration">the sample duration</param>  
/// <param name="sampleTimeStamp">the sample time</param>  
/// <param name="sampleBuffer">the sample buffer</param>  
/// <param name="sampleAttributes">the attributes for the sample</param>  
/// <history>  
///      01 Nov 18   Cynic - Ported in  
/// </history>  
public HRESULT OnProcessSampleEx(Guid guidMajorMediaType, int sampleFlags,  
                                long sampleTimeStamp, long sampleDuration,  
                                IntPtr sampleBuffer, int sampleSize,  
                                IMFAttributes sampleAttributes)  
{  
    IMFSample outputSample = null;  
    HRESULT hr;  
  
    try  
    {  
        if(sinkWriter==null)  
        {  
            string errMsg = "OnProcessSample, Error sinkWriter==null";  
            SampleGrabberAsyncCallbackError(this, errMsg, null);  
            return HRESULT_E_FAIL;  
        }  
  
        // we have all the information we need to create a new output sample  
        outputSample = TantaWMFUtils.CreateMediaSampleFromIntPtr(  
            sampleFlags, sampleTimeStamp,  
            sampleDuration, sampleBuffer, sampleSize, null);  
        if (outputSample == null)  
        {  
            string errMsg = "outputSample == null";  
            SampleGrabberAsyncCallbackError(this, errMsg, null);  
            return HRESULT_E_FAIL;  
        }  
  
        // write the sample out  
        hr = sinkWriter.WriteSample(sinkWriterMediaStreamId, outputSample);  
        if (Failed(hr))  
        {  
            string errMsg = "Error on WriteSample =" + hr.ToString();  
            SampleGrabberAsyncCallbackError(this, errMsg, null);  
            return hr;  
        }  
    }  
    finally  
    {  
        if(outputSample!=null)  
        {  
            Marshal.ReleaseComObject(outputSample);  
            outputSample = null;  
        }  
    }  
}
```

```

    }

    return HRESULT.S_OK;
}
Source: TantaAudioFileCopyViaPipelineAndWriter::TantaSampleGrabberSinkCallback::OnProcessSampleEx

```

The media data arrives in the `OnProcessSampleEx` function as an `IntPtr` to an `IMFMediaBuffer` object. The Media Sink definitely does **not** want its data in that format, so we wrap the Media Buffer in a Media Sample before handing it over. Since this requirement is likely to come up reasonably often the conversion process has been factored out into a static function call in the `TantaWMFUtils` library.

```

// we have all the information we need to create a new output sample
outputSample = TantaWMFUtils.CreateMediaSampleFromIntPtr(sampleFlags,
    sampleTimeStamp, sampleDuration, sampleBuffer, sampleSize, null);

```

Of course, a Media Sample has a lot more information in it other than a Media Buffer and that is why we feed things like the timestamp and duration into the `CreateMediaSampleFromIntPtr()` call. The authors of the Sample Grabber Sink provide everything to make a Media Sample if needed – they just do not provide the Media Sample itself. The conversion and wrapping process used by the `CreateMediaSampleFromIntPtr()` call have been documented in the *Creating a Media Buffer* section of *The WMF Components* chapter and will not be discussed here.

Once you have a Media Sample, it can be given to the Sink Writer with a standard `WriteSample()` call and the Sink Writer will know what to do with it. Note that the newly created Media Sample is immediately released after it is given to the Sink Writer. If the Sink Writer wishes to keep it around it will make a copy or set a new reference and release that.

```

// write the sample out
hr = sinkWriter.WriteSample(sinkWriterMediaStreamId, outputSample);

```

Errors have to be treated somewhat differently. If we throw an exception in the `OnProcessSample` function it will not get transmitted back to the application via the Media Session. Apparently the Sample Grabber Sink traps and ignores such exceptions.

```

if (outputSample == null)
{
    string errMsg = " Error CreateMediaSampleFromBuffer outputSample == null";
    SampleGrabberAsyncCallbackError(this, errMsg, null);
    return HRESULT.E_FAIL;
}

```

As you can see in the above code, any errors explicitly call the `SampleGrabberAsyncCallbackError` event with an informational message and other details. The error handler in the application main form will receive this and deal with it. Remember, the message will not arrive on the forms main thread so you will have to get back on that thread before interacting with any forms or controls.

Another thing to note in the above example of a Hybrid Architecture is that the closing of the Media Session will also trigger the proper shutdown and `Finalize_` on the Sink Writer. This is done from within the context of the standard `CloseAllMediaDevices()` call. There is no need to trap an end of stream message in the Callback Object because as far as that object is concerned, the stream of media data never ends.

As mentioned previously, see the discussion in the *Capture with a Hybrid Architecture* section of the *Capturing Camera Data* chapter regarding the *TantaCaptureToScreenAndFile* Sample Project for information on how to use a Transform to feed a Sink Writer.

One last thing that should be mentioned is that you never see a Hybrid Architecture which uses a Source Reader which feeds into a Pipeline. Theoretically this is possible but there is no “*Sample Grabber Media Source*” component supplied by Microsoft which you could use to stuff the Media Samples from the Source Reader into the Pipeline. You could write your own of course – it would be an interesting challenge – but until you (or someone else) does that, the Sample Grabber Sink is all there is.

Windows Media Foundation: Getting Started in C#

Chapter 8

RENDERING AUDIO AND VIDEO

One of the most common Windows Media Foundation tasks is to display video on a screen. Similarly, the playing of audio via the computer speakers or headphones is also often required. Since these are such common requirements, the implementers of Windows Media Foundation have built custom components to satisfy them.

The Enhanced Video Renderer (EVR) is a component which simplifies the process of rendering a video stream (or streams) on the display. In WMF, the EVR is implemented as a Media Sink – an endpoint of a branch in a Topology.

The Streaming Audio Renderer (SAR) is designed to render audio. Like the EVR, the SAR is a Media Sink however it is single stream only.

It should be noted that neither the EVR nor the SAR are absolutely required in order to render video and audio data in Windows Media Foundation. If you wish, it is quite possible to write your own video or audio Media Sink and have that object interact

directly with the appropriate Windows sub-systems. The reason this is not more commonly done is that both the EVR and SAR are quite sophisticated components and most of the functionality you might require is already contained within them. The writing of custom Media Sinks is not covered in this book, nor do any of the Tanta Sample Projects provide an example. The MF.Net example code does, however, provide an example of just such a homebrew video renderer in its *EVRPresenter* sample.

As mentioned above, the EVR and SAR are Media Sinks. This means you need to choose the Pipeline Architecture in order to use them. In other words, the requirement to use an EVR or SAR explicitly defines the choice of architecture so you can forget any notions you may have about using a Source Reader – it will be of no use to you here. Review the *Implementing the Pipeline Architecture* section in the *Practical WMF Architectures* chapter if your memory is hazy.

On the plus side note that both the EVR and SAR are Renderers, thus any Topology that implements them is a “playback” Pipeline and so Transforms can be automatically added to resolve Media Type differences when the Topology is resolved.

AN OVERVIEW OF THE SAR

To begin, we will discuss the Streaming Audio Renderer. Since it only has one stream, and the defaults are good enough for most uses, we might as well get it out of the way before we pursue the much more complex topic of the EVR.

Earlier versions of Windows Media Foundation forced you to choose an Audio Endpoint Device when you configured the SAR. This meant that you could choose to play through the speakers, headphones or some other audio device. In the newer Windows Operating systems (Windows 7, 8 and 10) the concept of a defined Audio Endpoint Device has largely been abstracted away and the target audio device is under the control of the operating system. This is what makes it possible to plug in a set of USB headphones while you are playing music and have the sound be automatically re-routed to them.

The Audio Endpoint Device was found via an enumeration mechanism practically identical to the one used to identify the Video Capture Devices on the system (except it produces sink, not source devices). You can still do this if you wish – but if you do not specify any Audio Endpoint Device, your Media Samples will be routed by the Streaming Audio Renderer in the standard Windows way. This means the sound behaves the way the user expects: via headphones if they are plugged in and to the speakers if they are not. If you dig about on the Internet you will still come across example code that

enumerates the Audio Endpoint Devices prior to setting up the SAR. Just be aware that most of the time, unless you have specific reasons to the contrary, this is probably not what you will want to do.

Let's look at the creation process for the Streaming Audio Renderer, you can find the code below in use in the *TantaFilePlaybackSimple* Sample Project.

```

/// ++++++
/// <summary>
/// Create a topology node for SAR Audio Renderer sink. The caller must
/// release the returned node.
/// </summary>
/// <history>
/// 01 Nov 18 Cynic - Originally Written
/// </history>
public static IMFTopologyNode CreateSARRendererOutputNodeForStream()
{
    HRESULT hr;
    IMFTopologyNode outputNode = null;
    IMFActivate pRendererActivate = null;

    try
    {
        // Create a downstream node.
        hr = MFExtern.MFCreateTopologyNode(MFTTopologyType.OutputNode, out outputNode);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to MFCreateTopologyNode failed. Err=" + hr.ToString());
        }
        if (outputNode == null)
        {
            throw new Exception("failed. outputNode == null");
        }

        // There are two ways to initialize an output node
        // 1) From a pointer to the stream sink.
        // 2) From a pointer to an activation object for the media sink.
        // since we do not have a stream sink at this point we are going to go the
        // activation object route. This is what we are doing below.

        // Create an activation object for the streamin audio renderer (SAR) media sink.
        hr = MFExtern.MFCreateAudioRendererActivate(out pRendererActivate);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("MFCreateAudioRendererActivate failed. Err=" + hr.ToString());
        }
        if (pRendererActivate == null)
        {
            throw new Exception("pRendererActivate == null");
        }

        // Set the IActivate object on the output node. Note that not all node types use
        // this object. On transform nodes this is IMFTransform or IMFActivate interface
        // and on output nodes it is a IMFStreamSink or IMFActivate interface. Not used
        // on source or tee nodes.
        hr = outputNode.SetObject(pRendererActivate);

        // Return the IMFTopologyNode pointer to the caller.
        return outputNode;
    }
    catch
    {
        // If we failed, release the pNode
        if (outputNode != null)
        {
            Marshal.ReleaseComObject(outputNode);
        }
        throw;
    }
    finally
    {
        // Clean up.
        if (pRendererActivate != null)
        {
            Marshal.ReleaseComObject(pRendererActivate);
        }
    }
}

```

```
    }  
    }  
}  
  
Source: TantaCommon::TantaWMFUtils::CreateSARRendererOutputNodeForStream
```

The above code block creates the Streaming Audio Renderer as part of the creation process for an output Topology Node. There are two ways of creating a SAR, the first (as shown below) is to supply the Topology Node with an Activator which then creates the SAR when the Topology is resolved.

```
// Create an activation object for the streamin audio renderer (SAR) media sink.  
hr = MFExtern.MFCreateAudioRendererActivate(out pRendererActivate);
```

The second method is to use a call to the static `MFCreateAudioRenderer` function and create it directly. Either way, the Activator or the SAR object is given to the Topology Node via the `SetObject()` call. If you want your application to ever be able to play Protected Media Content (PMP) you have to go the Activator route – otherwise the method you use does not matter. PMP is not discussed in this book.

```
// Set the IActivate object on the output node.  
hr = outputNode.SetObject(pRendererActivate);
```

Once you have created the SAR Topology Node you can connect it up to other nodes in the audio branch of the Topology and, when the Pipeline begins running, the sound will play out the appropriate device. The Media Session will control the data rate of the sound so that, if there are video branches in the Pipeline, the sound and video play in a synchronized manner. There are also some specialized volume control facilities available on the Media Session and, of course, the user can always change the volume in the usual way themselves.

AUDIO VOLUME AND MUTING

Changing the audio volume and muting can be easily done via the `IMFSimpleAudioVolume` interface object we obtain from the Media Session. This interface provides functions which will both `get` and `set` the current volume and apply muting.

There are multiple volume controls on a Windows system. The PC itself has a Master Volume control and this level is set via software. Ultimately, though, there are speakers which render the sound and these too often have independent amplification and volume controls.

Windows Media Foundation takes a pragmatic approach to volume management. It treats any volume it might output as a level between 0 and 1. Level 0 means no sound and level 1 means full Master Volume – in other words, the full level of whatever the PC is currently configured to output. The default is level 1. Thus in MF.Net you can only

adjust the effective volume from the full Master Volume to down to zero. It is not possible, from within MF.Net, to make the volume louder than the Windows System is currently configured to output.

The Windows Media Foundation volume is expressed as an attenuation level between 0 and 1. So, for example, an attenuation level of 0.5 is half of the PC's full Master Volume. The code below, which applies an incremental attenuation level to the existing volume, shows how the volume can be adjusted using simple calls to `GetMasterVolume()` and `SetMasterVolume()` on the `IMFSimpleAudioVolume` service.

```
HResult hr;
IMFSimpleAudioVolume simpleAudioService = null;
object rcServiceObj = null;

// sanity check
if (mediaSession == null) return false;

// We get the audio volume service from the Media Session.
hr = MFExtern.MFGetService(
    mediaSession,
    MFServices.MR_POLICY_VOLUME_SERVICE,
    typeof(IMFSimpleAudioVolume).GUID,
    out rcServiceObj
);
if (hr != HResult.S_OK)
{
    throw new Exception("call to MFExtern.MFGetService failed. Err=" + hr.ToString());
}
if (rcServiceObj == null)
{
    throw new Exception("call to MFExtern.MFGetService failed. rcServiceObj == null");
}
// set the rate control service now for later use
simpleAudioService = (rcServiceObj as IMFSimpleAudioVolume);

// now get the current attenuation level on the audio service
float attenuationLevel = 1.0f;
hr = simpleAudioService.GetMasterVolume(out attenuationLevel);
if (hr != HResult.S_OK)
{
    throw new Exception("call to simpleAudioService.GetMasterVolume failed.");
}

// now perform the increment
attenuationLevel += attenuationIncrement;

// anything below 0 or above 1 will throw an error. Volume is expressed
// as an attenuation level, where 0.0 indicates silence and 1.0 indicates
// full volume (no attenuation). The actual full volume level is controlled
// by the PC - perhaps even a knob on the speakers.

if (attenuationLevel < 0f) attenuationLevel = 0f;
if (attenuationLevel > 1.0f) attenuationLevel = 1.0f;

// now set the attenuation level on the audio service
hr = simpleAudioService.SetMasterVolume(attenuationLevel);
if (hr != HResult.S_OK)
{
    throw new Exception("call to simpleAudioService.SetMasterVolume failed.");
}

Source: TantaCommon::TantaWMFUtils::IncrementAudioVolumeOnSession
```

Muting and un-muting the sound is accomplished by similar calls to `GetMute()` and `SetMute()` on the `IMFSimpleAudioVolume` service. The sample code below, which toggles the mute state, illustrates this procedure.

```
HResult hr;
```

Rendering Audio and Video

```
IMFSimpleAudioVolume simpleAudioService = null;
object rcServiceObj = null;

// sanity check
if (mediaSession == null) return false;

try
{
    // We get the audio volume service from the Media Session.
    hr = MFExtern.MFGetService(
        mediaSession,
        MFServices.MR_POLICY_VOLUME_SERVICE,
        typeof(IMFSimpleAudioVolume).GUID,
        out rcServiceObj
    );
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to MFExtern.MFGetService failed. Err=" + hr.ToString());
    }
    if (rcServiceObj == null)
    {
        throw new Exception("call to MFExtern.MFGetService failed. rcServiceObj == null");
    }
    // set the rate control service now for later use
    simpleAudioService = (rcServiceObj as IMFSimpleAudioVolume);

    // now get the mute state on the audio service
    bool muteState = false;
    hr = simpleAudioService.GetService(out muteState);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to audioVolumeService.GetService failed. Err=" + hr.ToString());
    }

    // toggle the state
    if (muteState == true) muteState = false;
    else muteState = true;

    // now set the mute state on the audio service
    hr = simpleAudioService.SetMute(muteState);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to audioVolumeService.SetMute failed. Err=" + hr.ToString());
    }
    return true;
}
finally
{
    // release the audio service interface
    if (simpleAudioService != null)
    {
        Marshal.ReleaseComObject(simpleAudioService);
        simpleAudioService = null;
    }
}

Source: TantaCommon::TantaWMFUtils::ToggleAudioMuteStateOnSession
```

It seems the MF.Net system, by mechanisms unknown, remembers the mute state and the volume attenuation level. This is true even if the application is closed down and restarted. Thus it is important to either turn off the mute state and reset the attenuation level to 1 before you shut down or set these on startup from some saved level.

IMPORTANT NOTE: Remember to turn off the mute state and reset the attenuation level to 1 before you shut down or to set these on startup from some saved level.

Windows Media Foundation remembers the last levels set and the next application starting will inherit them.

AN OVERVIEW OF THE EVR

The Enhanced Video Renderer is very a sophisticated tool. Remember how, in the discussion of *The Pipeline* in *The WMF Components* chapter, it was mentioned that the COM objects forming the Topology do not really know anything about each other? This is especially true for the EVR. As long as the EVR object is given the correct information in the correct way it does not care if a WMF Media Session is calling it or if something else entirely is feeding it. The EVR can be called from completely different architectures and, in fact, the same EVR component you get in Windows Media Foundation is also used to display video in DirectShow graphs.

OLDER VERSIONS OF THE EVR

Actually, the EVR was originally developed for DirectShow and has proceeded through numerous iterations and improvements. When digging around on the Internet you will sometimes come across acronyms like VMR-7 and VMR-9. The acronym VMR stands for Video Media Renderer. These are the older video renderers which were the default on Windows XP. They are still available in later versions of Windows and many older DirectShow based software products will use these renderers. It is conceivable that, instead of using the newer WMF EVR, you could look around in the registry and somehow use VMR-7 or VMR-9 for your display needs – however, there is nothing in MF.Net that supports this. VMR-7 or VMR-9 or any other earlier iteration of the Enhanced Video Renderer will not be discussed further here.

THE EVR AND DIRECT3D

Ultimately, the Enhanced Video Renderer has to display a moving image on the screen. Of course, there is much more to it than that – but before we proceed to the various capabilities of the EVR, let's discuss how the image is displayed.

If you dig right into it, technically, the EVR has no ability to actually light up a pixel on the screen. This functionality is abstracted away in Windows. What the Enhanced Video Renderer does is create a Direct3D device (effectively a buffer in memory) and then a Windows sub-system called the Desktop Window Manager (DWM) handles the presentation on the screen. In older windows operating system versions, (XP and

earlier), this sub-system was known as the Desktop Compositing Engine or DCE and it was much less capable than the DWM.

Compositing, in case you are not familiar with the term, is the act of combining various visual images into a single image. In the case of the Desktop Window Manager, everything that wants to draw anything on the screen simply writes to its own video buffer and the DWM handles the build of the final image on the desktop. It is this technology which makes things like the Aero interface possible and, rather more importantly, provides the Enhanced Video Renderer with some advanced capabilities.

The EVR can even share the Direct3D device on which it draws with other software running external to the EVR and because the device is so abstract an entity, the DWM can automatically invoke hardware acceleration, de-interlacing and various other useful performance enhancing features.

MULTIPLE STREAMS IN THE EVR

The Enhanced Video Renderer is a Media Sink (an `IMFMediaSink`) but it does much more than display a single stream of video – it can also act as a mixer (or compositor if you wish) for up to 16 different streams. As you might imagine, there are all sorts of rules regarding what can and cannot be done with the inbound streams. Not the least of which is synchronization between the streams, color correction and dealing with the various formats and interlacing modes they might use. Multiple streams, window-in-window and various other cool visual effects are a pretty advanced topic and this book will not discuss them. For now, just realize that the EVR is much more than a simple *“take an image and paint it on the screen”* type object.

A SUMMARY OF THE CAPABILITIES OF EVR

We have all seen a video play on the screen in any number of products and have a good idea of the functionality we expect such an application to provide. Fortunately the Enhanced Video Renderer provides pretty much all of these capabilities so, if we ever need to implement those sort of features, a lot of the hard work has already been done for us. Listed below are some of the operations supported in the EVR.

The Enhanced Video Renderer can...

- Display a video on the surface of a form or control.
- Display up to 16 streams simultaneously and handle the overlaying and transparencies.
- Control the aspect ratio of the video on display.

- Provide software magnification.
- Handle display window size change events from external sources and dynamically adjust.
- Handle video format change events and dynamically adjust.
- Start, pause and stop the video presentation at arbitrary points.
- Seek forwards and backwards in the video stream and restart from a point in time.
- Fast forward and reverse and also implement slow motion speeds.
- Take a bitmap snapshot of the image currently on display at any time.

There is a section further on in this chapter devoted to each topic and that section will explain the concept and associated requirements in detail. In addition, the Tanta library code contains a control named `ctlTantaEVRFilePlayer` which, along with the `TantaFilePlayerAdvanced` sample solution, provides a reasonably straight forward reference implementation of each technique.

THE TANTAFILEPLAYBACK SAMPLES

There are two Tanta Sample Projects which use the EVR renderer and both of these contain controls which, to a greater or lesser extent, provide a plug-in module applications can use to display video data. These applications are...

TantaFilePlaybackAdvanced – this application uses a control from the *TantaCommon* library named `ctlTantaEVRFilePlayer`. This control is designed to provide a simple drop in mechanism which applications can use to display file based video data. The `ctlTantaEVRFilePlayer` control contains all of the functionality necessary to manipulate the stream of data (pause, stop, restart, fast forward & etc.). In this sample code, the Media Session and Pipeline are located inside the `ctlTantaEVRFilePlayer` control itself and the application does little more than feed it with a filename to play. Since the entire Pipeline is contained within the `ctlTantaEVRFilePlayer` control, it also adds a Streaming Audio Renderer to the Topology so that audio can be played.

TantaFilePlaybackSimple – this application uses the more conventional Media Session and Pipeline structure as discussed in the *Implementing the Pipeline Architecture* section of the *Practical WMF Architectures* chapter. The control used in this application is named `ctlTantaEVRStreamDisplay` and it is designed to provide a limited functionality control which applications can use to display a stream of video data. Unlike the `ctlTantaEVRFilePlayer` control, the `ctlTantaEVRStreamDisplay` control has no functionality to stop or pause the media data. The control is pretty much just limited to

providing a screen on which to display video data and handling the screen size changes if the user adjusts the form size. There is no SAR component implemented within the `ctlTantaEVRStreamDisplay` control – the application must provide that itself if it needs it.

THE CTLTANTA EVRSTREAMDISPLAY CONTROL

To start off our discussion of the Enhanced Video Renderer, let's consider the

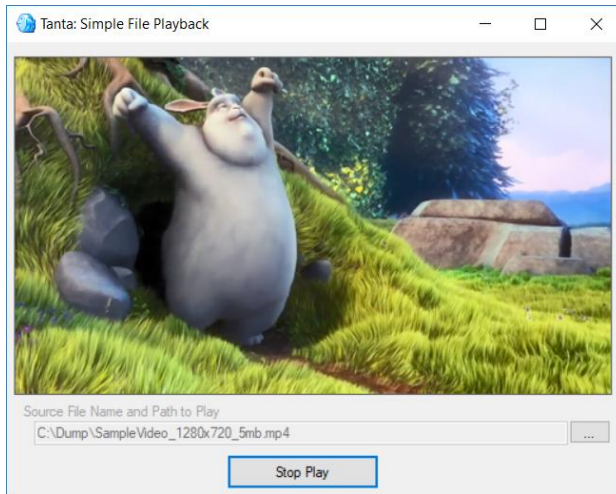


Figure 8.1: The *TantaFilePlaybackSimple* Application

TantaFilePlaybackSimple application and its use of the `ctlTantaEVRStreamDisplay` control. Knowing what you now know about the Pipeline Architecture, it can be said that the *TantaFilePlaybackSimple* application is really just another standard implementation of it.

Really, it is quite straight forward. In fact, the entire *TantaFilePlaybackSimple* application is just a blatant cut-and-paste rework

of the *TantaVideoFileCopyViaPipelineMP4Sink* application you are already familiar with. The only major difference is that instead of using the MP4 File sink with two Stream Sinks we use an EVR Sink and a SAR Sink with one Stream Sink each. We have also dropped the `ctlTantaEVRStreamDisplay` control onto the main form display.

Let's join the *TantaFilePlaybackSimple* application in the `PrepareSessionAndTopology` function of the `frmMain` class as the output Topology Nodes for the Media Sinks are created.

```
... more code

// Create the Video sink node.
outputSinkNodeVideo = TantaWMFUtils.CreateEVRRendererOutputNodeForStream(
    this.ctlTantaEVRStreamDisplay1.DisplayPanelHandle);
if (outputSinkNodeVideo == null)
{
    throw new Exception("MFCreatetopologyNode(v) failed. outputSinkNodeVideo == null");
}

// Create the Audio sink node.
outputSinkNodeAudio = TantaWMFUtils.CreateSARRendererOutputNodeForStream();
if (outputSinkNodeAudio == null)
{
    throw new Exception("MFCreatetopologyNode(a) failed. outputSinkNodeAudio == null");
}

... more code

Source: TantaFilePlaybackSimple::frmMain::PrepareSessionAndTopology
```

Both of the above calls create the required output Topology Node using a static function in the `TantaWMFUtils` class of the `TantaCommon` library. The operation of the `CreateSARRendererOutputNodeForStream` function was covered earlier in the *An Overview of the SAR* section. The `CreateEVRRendererOutputNodeForStream()` routine for the EVR is pretty much identical to the SAR version except for some obvious changes – but we will document it here for the sake of completeness.

```
// +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+  
/// <summary>  
/// Create a topology node for EVR Video Renderer sink. The caller must  
/// release the returned node.  
/// </summary>  
/// <param name="videoWindowHandle">the handle to the window on which  
/// video streams will display</param>  
/// <returns>the output stream node</returns>  
/// <history>  
///    01 Nov 18 Cynic - Originally Written  
/// </history>  
public static IMFTopologyNode CreateEVRRendererOutputNodeForStream(IntPtr videoWindowHandle)  
{  
  
    HRESULT hr;  
    IMFTopologyNode outputNode = null;  
    IMFActivate pRendererActivate = null;  
  
    try  
    {  
        // Create a downstream node.  
        hr = MFExtern.MFCreateTopologyNode(MFTopologyType.OutputNode, out outputNode);  
        if (hr != HRESULT.S_OK)  
        {  
            throw new Exception("call to MFCreateTopologyNode failed. Err=" + hr.ToString());  
        }  
        if (outputNode == null)  
        {  
            throw new Exception("call to MFCreateTopologyNode failed. outputNode == null");  
        }  
  
        // Create an activation object for the enhanced video renderer (EVR) media sink.  
        hr = MFExtern.MFCreatVideoRendererActivate(videoWindowHandle, out pRendererActivate);  
        if (hr != HRESULT.S_OK)  
        {  
            throw new Exception("MFCreatVideoRendererActivate failed. Err=" + hr.ToString());  
        }  
        if (pRendererActivate == null)  
        {  
            throw new Exception("failed. pRendererActivate == null");  
        }  
  
        // Set the IActivate object on the output node. Note that not all node types use  
        // this object. On transform nodes this is IMFTransform or IMFActivate interface  
        // and on output nodes it is a IMFStreamSink or IMFActivate interface. Not used  
        // on source or tee nodes.  
        hr = outputNode.SetObject(pRendererActivate);  
  
        // Return the IMFTopologyNode pointer to the caller.  
        return outputNode;  
    }  
    catch  
    {  
        // If we failed, release the outputNode  
        if (outputNode != null)  
        {  
            Marshal.ReleaseComObject(outputNode);  
        }  
        throw;  
    }  
    finally  
    {  
        // Clean up.  
        if (pRendererActivate != null)  
        {  
            Marshal.ReleaseComObject(pRendererActivate);  
        }  
    }  
}
```

Rendering Audio and Video

```
}  
Source: TantaCommon::TantaWMFUtils::CreateEVRRendererOutputNodeForStream
```

It should be noted that the EVR needs a window handle so that it knows where to draw its output. The SAR needs no such information - interacting as it does with the generic Windows audio sub-system. A standard Windows Panel control is implemented in the `ctlTantaEVRStreamDisplay` control for the EVR to draw on and a window handle is readily obtained from it.

```
public IntPtr DisplayPanelHandle  
{  
    get  
    {  
        return this.panelDisplayPanel.Handle;  
    }  
}  
Source: TantaCommon::ctlTantaEVRStreamDisplay::DisplayPanelHandle
```

Once the Topology Nodes are created, we simply connect them up as usual: audio source node to audio output node and video source node to video output node.

```
... more code  
// hr = sourceVideoNode.ConnectOutput(0, outputSinkNodeVideo, 0);  
if (hr != HRESULT.S_OK)  
{  
    throw new Exception("call to ConnectOutput(v) failed. Err=" + hr.ToString());  
}  
hr = sourceAudioNode.ConnectOutput(0, outputSinkNodeAudio, 0);  
if (hr != HRESULT.S_OK)  
{  
    throw new Exception("call to ConnectOutput(a) failed. Err=" + hr.ToString());  
}  
... more code  
Source: TantaFilePlaybackSimple::frmMain::PrepareSessionAndTopology
```

No surprises there – by now this is just pretty standard stuff for you. There is only one other oddity – because we used an Activator to create the EVR (and SAR) our application does not have knowledge of those objects. We will need the EVR object though, so we dig it out of the Media Session when we get the `TopologyNowReady` event after the Topology is resolved. The procedure is to ask the Media Session for the EVR object via a `GetService()` call.

```
private void MediaSessionTopologyNowReady(IMFMediaEvent mediaEvent)  
{  
    HRESULT hr;  
    object evrVideoService;  
  
    LogMessage("MediaSessionTopologyNowReady");  
  
    // we need to obtain a reference to the EVR Video Display Control.  
    // We used an Activator to configure this in the Topology and so  
    // there is no reference to it at this point. However the media session  
    // knows about it and so we can get it from that.  
  
    try  
    {  
        // we need to get the active IMFVideoDisplayControl. The EVR  
        // presenter implements this interface  
        hr = MFExtern.MFGetService(  
            mediaSession,
```



```

        MFServices.MR_VIDEO_RENDER_SERVICE,
        typeof(IMFVideoDisplayControl).GUID,
        out evrVideoService
    );
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to MFGetService failed. Err=" + hr.ToString());
    }
    if (evrVideoService == null)
    {
        throw new Exception("evrVideoService == null");
    }

    // set the video display now for later use
    evrVideoDisplay = evrVideoService as IMFVideoDisplayControl;
    // also give this to the display control
    ctlTantaEVRStreamDisplay1.EVRVideoDisplay = evrVideoDisplay;
}
catch (Exception ex)
{
    evrVideoDisplay = null;
    ctlTantaEVRStreamDisplay1.EVRVideoDisplay = evrVideoDisplay;
    LogMessage("Error: " + ex.Message);
}

try
{
    StartFilePlay();
}
catch (Exception ex)
{
    LogMessage("MediaSessionTopologyNowReady errored ex="+ex.Message);
    OISMessageBox(ex.Message);
}
}

Source: TantaFilePlaybackSimple.frmMain:MediaSessionTopologyNowReady

```

Note how the EVR object is given to the `ctlTantaEVRStreamDisplay` control. Be aware that this object, like all of the objects you get from WMF, will need to be released when playback is over. Once we have the EVR object, the above code calls the `StartFilePlay()` function and that is just a wrapper for the `Start()` call on the Media Session that we have discussed many times before.

```

// this is what starts the data moving through the pipeline
HRESULT hr = mediaSession.Start(Guid.Empty, new PropVariant());

```

The `Start()` call on the Media Session starts the data moving through the Pipeline and both the video and audio streams will reach their respective renderers in a synchronized manner. In other words, the Media Session takes care of making sure the sound matches the picture on display and the whole process is throttled so that they both play at normal speeds. As has been mentioned previously – the Pipeline is a powerful, sophisticated technology and the functionality you get from it is well worth the small trouble of setting it up in the first place.

The only other functionality of interest is the way in which the `ctlTantaEVRStreamDisplay` control handles screen size changes. If the user adjusts the size of the screen, they will expect the video image to adjust appropriately. It is worth a look to see this process in action, however, we will delay the discussion of this procedure until the *Handling Size Change Events* section further on in this chapter. The

technique used involves such concepts as Source Windows and Destination Rectangles and you will have a better understanding of that background material by that time. Still, if you are interested, have a look at the `ctlTantaEVRStreamDisplay_SizeChanged` function in the `ctlTantaEVRStreamDisplay` control – the comments there are pretty explanatory.

It should be noted that several other Tanta Sample Projects (*TantaCaptureToScreenAndFile*, *TantaTransformDirect*, *TantaTransformInDLLClient*) also use the `ctlTantaEVRStreamDisplay` control as part of their operation. You will meet this control every time a Tanta Sample Application needs a simple way to display video data.

THE CTLTANTA EVRFILEPLAYER CONTROL

The philosophy behind the `ctlTantaEVRFilePlayer` control is somewhat different than that of the `ctlTantaEVRStreamDisplay` control. The `ctlTantaEVRFilePlayer` control is designed to present the user with a tool that can be dropped onto a form and which handles all of the mechanisms necessary for a file playback operation. In other words, it is `ctlTantaEVRFilePlayer` control itself which contains all of the buttons and sliders which start, stop and control the flow of the video data. This eliminates the need for the application to implement that functionality itself. The control is a self-contained display engine – give it a file and it will let the user play it.

The *TantaFilePlaybackAdvanced* Sample Application demonstrates the use of the

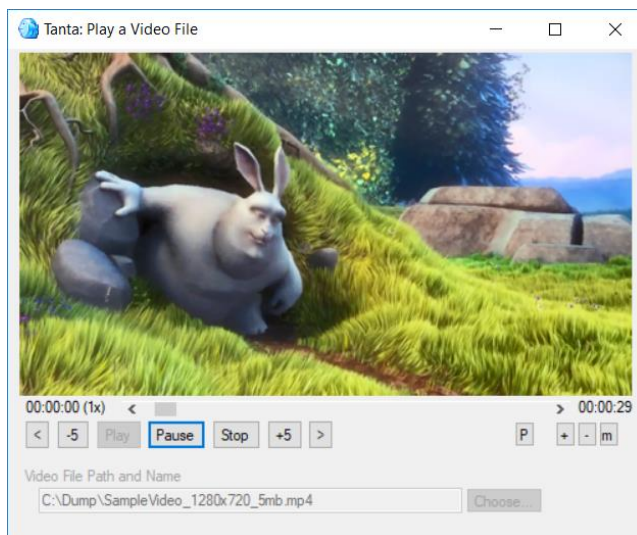


Figure 8.2: The TantaFilePlaybackAdvanced Application

`ctlTantaEVRFilePlayer` control. Remember how it was mentioned previously that the `ctlTantaEVRFilePlayer` control contains all of the GUI functionality which can be used to start, stop and control the flow of the video data? Well in order to do that, the control needs to interact with the Media Session and Pipeline. In other words, the `ctlTantaEVRFilePlayer` control also implements the Media Session and Topology functionality which

creates the Pipeline. It should be noted that, in this case, the function which sets up the Pipeline is named `OpenVideoFileAndPrepareSessionAndPlay`. If you look at the code

for this function, you will see that it is mostly identical to the `PrepareMediaSessionAndTopology` function used in the *TantaFilePlaybackSimple* Sample Application. The `frmMain` of the *TantaFilePlaybackAdvanced* application has nothing to do with the Pipeline other than supply the `ctlTantaEVRFilePlayer` control with the name of the file the user has chosen to play.

The `OpenVideoFileAndPrepareSessionAndPlay` function is launched out of the `buttonPlay_Click` handler when the user clicks on the controls *Play* button. The vast majority of this code is the standard Tanta Pipeline setup (it just happens to be in a control rather than a form) and it functions in exactly the same way: the Pipeline is created and the SAR and EVR Media Sinks are attached to an MP4 Media Source. The only real difference is that this code also contains the ability to add a Transform to the video branch of the Pipeline. You have not met Transforms in any detail yet and, in this application, there is no Transform to add so the resulting Pipeline will look identical to the one in the previous example.

In the interests of space (it is really quite lengthy) we will not reproduce the code for this Pipeline setup here. You have probably seen more than enough Pipeline source code by now anyways and are probably able to read the sample source as if it was a book. Here is some pseudo code which illustrates the process.

```
public void OpenVideoFileAndPrepareSessionAndPlay()
{
    Create Media Session

    Give the Media Session a Callback Object so it can report events and errors

    Create a Topology

    Create the Media Source from a file

    Get a Presentation Descriptor from the Media Source

    Find the StreamDescriptor of the first enabled video stream from the Presentation

    Find the StreamDescriptor of the first enabled audio stream from the Presentation

    Create the source video and audio Topology Nodes

    Create the output Topology Node for the EVR

    Create the output Topology Node for the SAR

    Add all the nodes to the Topology

    Do we have a Transform?
    If yes
        Create a Transform Topology Node and add to the topology
    If no
        Proceed as normal.

    Connect the source audio Topology Node to the output audio Topology Node

    Do we have a Transform?
    If yes
        Connect the source video Topology Node to the Transform Topology Node
        Connect the Transform Topology Node to the output video Topology Node
    If no
        Connect the source video Topology Node to the output video Topology Node
}
```

```
    Resolve the Topology  
    Get the playback duration from the Media Session  
}  
Source: TantaCommon::ctlTantaEVRFilePlayer::OpenVideoFileAndPrepareSessionAndPlay (pseudo code)
```

Ultimately, after the `OpenVideoFileAndPrepareSessionAndPlay` function completes, the media data is moving through the Pipeline to the appropriate sink (EVR and SAR). The user will see the video display on the screen and hear the synchronized audio.

Besides the Transform Topology Node code (you will meet that again in the *Working With Transforms* chapter), the only real difference is that the duration of the file playback is obtained from the Media Session.

```
// get the duration from the presentation descriptor now. This is nothing to do  
// with the creation of the topology. We will eventually need this so  
// we can tell the user how long the video is. We have to get it from the  
// presentation descriptor and so might as well just get it here  
VideoDuration = TantaWMFUtils.GetDurationFromPresentationDescriptor(  
    sourcePresentationDescriptor); ();
```

As the comments in the above code block state – the duration is not really a part of the build of the Pipeline. We just need a Presentation Descriptor to obtain it and, since we happen to have one available at that point, we take the opportunity to extract and store it in a class variable for later use.

It should also be noted that we have to obtain the Enhanced Video Renderer object after the Topology has been resolved. As usual, we dig it out of the Media Session when we get the `TopologyNowReady` event after the Topology is resolved. See the `MediaSessionTopologyNowReady` function for more information.

Since the `MediaSessionTopologyNowReady` function records the Enhanced Video Renderer object in an instance variable in the `ctlTantaEVRFilePlayer` class, the EVR object is available to manipulate the image stream while the video is playing. Actions such as resizing the screen, stopping, pausing, fast forwarding and taking snapshots from the video stream are all possible because of this.

THE VIDEO WINDOW DRAWING SURFACE

Let's take a small digression from our discussion of the EVR in order to provide some background details on the drawing process. The region on which the Enhanced Video Renderer places its output is called the Drawing Surface. Most of the EVR examples you might see on the Internet simply use the entire form area inside the top panel and header bars as the display area. This works but has complications. Not the least of which is that your application will have no room for anything else on the form and you will find

that the form controls are rendered on top of the video being displayed. In other words, unless you take care to work around it with clever positioning and resizing, you might have a button sitting on top of your running video.

The video surface does not have to be the entire area of a form. The surface on which the Enhanced Video Renderer draws is determined by a Window Handle you provide in the EVR's `SetVideoWindow` call. If you choose to create the EVR via an Activator (which is what the `ctlTantaEVRFilePlayer` control in the Tanta sample source does) you can also supply the window handle in the `MFCreatVideoRendererActivate` call.

You can use any surface for which you can find a window handle as the EVR's drawing surface. Since each Windows form and control comes equipped with a `Control.Handle` property you have a wide range of choices. Some, obviously, are better than others. For example, you would probably not want to draw on a button control unless you really wanted your video to be clickable (and even then there are much better ways of doing that).

In the Tanta `ctlTantaEVRFilePlayer` control sample, the actual handle used for a display surface is of a standard Windows Panel control placed on the parent control. The Panel control provides a nice, uncomplicated, borderless area on which to draw and also allows the use the entire Panel control surface for the display area. The parent control contains the Panel control and places any other necessary controls (such as buttons) around it. If the parent control also builds and configures Media Session and Pipeline, then all interactions with the Media Session can be handled within the parent control.

Using a Panel control as the display surface also makes the math nice and easy and you will appreciate that when we come to implement support for window resizing and software magnification.

The example below is clipped from the `ctlTantaEVRFilePlayer` control in the *TantaCommon* Sample Project.

```
// Create the output node for the video renderer.
outputSinkNodeVideo = TantaWMFUtils.CreateEVRRendererOutputNodeForStream(
    this.panelDisplayPanel.Handle);
if (outputSinkNodeVideo == null)
{
    throw new Exception("call failed. outputSinkNodeVideo == null");
}

// Create the output node for the audio renderer.
outputSinkNodeAudio = TantaWMFUtils.CreateSARRendererOutputNodeForStream();
if (outputSinkNodeAudio == null)
{
    throw new Exception("call failed. outputSinkNodeAudio == null");
}

Source: TantaCommon::ctlTantaEVRFilePlayer::OpenVideoFileAndPrepareSessionAndPlay
```

As you can see in the above code section, the window handle of the Panel control is passed in on the `MFCCreateVideoRendererActivate` call. All of the usual functions of the Panel control continue operate correctly and so, for example, we can use it's `Anchor` property and the `SizeChanged` event to automatically detect window resizes of the parent form.

THE VIDEO WINDOW APPEARANCE

There are a number of factors that will affect the appearance of the video on the drawing surface, the parts of the drawing surface used, and the portion of the frames in the video stream which are rendered onto the screen.

ABOUT ASPECT RATIOS

The relative proportions of the height and width of an image is called the Aspect Ratio. This is true even for still images. The Aspect Ratio is usually expressed as a width and height number separated by a colon (for example 4:3). It is important to realize that this is a ratio, not an absolute size, and that an Aspect Ratio does not have a unit of measurement value such as centimeters, inches or pixels.

An Aspect Ratio of 1.33:1 (4:3) is the most common one you will run across as this is the standard TV video image Aspect Ratio. Another common Aspect Ratio is 1.85:1 (roughly 16:9) which is the usual widescreen TV Aspect Ratio. Interestingly, some newer DVD's contain the video data in both 4:3 and 16:9 formats and will play one or the other depending on the type of TV on which the image is being rendered. PC monitor Aspect Ratios vary considerably with 8:5 being common.

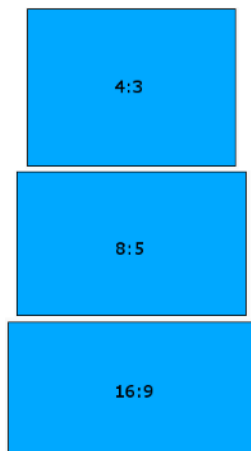


Figure 8.3: Example Aspect Ratios

So, what happens if the display surface on which the image is being rendered is not in the same proportions as the video frames? In other words, what happens if the Aspect Ratio of the frames in the video stream is not the same as the Aspect Ratio of the display device? This issue is extremely common when using the Enhanced Video Renderer. After all it does draw on a display surface (a form or control) which can be arbitrarily re-shaped according to the whim of the user.

In the event of differing Aspect Ratios, there are three options of which only the first two are generally acceptable. If you have differing Aspect Ratios you can...

1. Display the entire video to the maximum in whichever direction is the constraining dimension and leave empty space (usually black) in the displayable areas of the display surface.
2. Fill the drawing surface to the maximum and remove the areas which exceed the bounds of the video display area.
3. Stretch the video frames to fit. This usually involves duplicating rows or columns, depending on the dimension being stretched to insert filler.

Option 1 is known as “letterboxing” or “pillarboxing” depending on whether the image leaves displayable areas at the top and bottom or left and right respectively. Letter boxing is commonly seen when displaying a 16:9 widescreen video on an older TV that is designed for a 4:3 aspect ratio.

This method of coping with differing Aspect Ratios has the advantage that it displays the entire contents of the frames in the video stream but that there is a lot of empty unused space on two of the edges.

Option 2 is known as clipping. One dimension is maximized and the remainder is simply not displayed. Typically it is assumed that the most interesting part of the frame is in the center and so equal parts of two edges, usually the left and right edges, are removed. This has the advantage that the video on display is much larger however, any events on the periphery are not shown to the user. You may sometimes see this method of adjusting an aspect ratio referred to as “pan-and-scan”.

Option 3 is known as stretching and it is generally considered to be the least acceptable of the three options. However it is sometimes seen on video rendered in PC windows since users can simply resize the window and its proportions to stop any stretching they find unacceptable. It does have the advantage that the entire contents of the frames in the video stream are always visible – if somewhat distorted.

There is another usage of the term Aspect Ratio which has not been discussed here. Some devices do not have “square” pixels. They have rectangular pixels in which one dimension, usually the height, is longer than the width. This is known as the Pixel Aspect Ratio. The discussion above about the video window Aspect Ratio is known as the Picture Aspect Ratio. As you might imagine, having different Pixel Aspect Ratios on different devices can cause considerable problems in the rendering. This issue can be coped with in code but the EVR has no intrinsic support for non-square Pixel Aspect Ratios and so the topic will not be discussed further here.

THE EVR AND ASPECT RATIOS

The Enhanced Video Renderer supports various Aspect Ratio treatments. There are a number of these, of which only Letterboxing and Stretching are built-in modes. The remainder of the built-in Aspect Ratio modes are either obscure, deprecated or unsupported and so will not be discussed here. Clipping is also possible – but it is not a mode per-se. You can easily set up clipping, and much more besides, but that is the topic of the following section entitled *The EVR Video Window Size and Position*.

It appears, at least in some very general tests, that the default Aspect Ratio Mode in the Enhanced Video Renderer is letterboxing. However, this is easy to change. The Aspect Ratio is set in the EVR through a call to the `SetAspectRatioMode()` function and this takes one parameter which is an enum of type `MFVideoAspectRatioMode`. In reference to the above discussion of the default Aspect Ratio Mode, the letterboxing default is actually `MFVideoAspectRatioMode.PreservePicture` and stretching mode is `MFVideoAspectRatioMode.None`.

The effects of changing the EVR's Aspect Ratio Mode are easy to test. Setting the mode should be done after it is fully configured and the topology is built – possibly just before the Media Session is started. Just compile up the *TantaFilePlaybackAdvanced* sample and find the `StartPlayback()` function in the *TantaCommon.ctlTantaEVRFilePlayer* control. Inserting either of the two lines below will demonstrate the various aspect ratio modes as you resize the screen.

```
... more code
evrVideoDisplay.SetAspectRatioMode(MFVideoAspectRatioMode.None);
// evrVideoDisplay.SetAspectRatioMode(MFVideoAspectRatioMode.PreservePicture);

HRESULT hr = mediaSession.Start(Guid.Empty, new PropVariant());
... more code
Source: TantaCommon::ctlTantaEVRFilePlayer::StartFilePlay
```

THE EVR VIDEO WINDOW SIZE AND POSITION

The Enhanced Video Renderer displays its video on the surface of whichever window it is configured with. There is an extensive discussion of this in *The Video Window Drawing Surface* section.

However, having a display surface to draw on does not mean the entire content of the video stream is necessarily always completely visible on that surface. One reason why all, or part, of the drawing surface might not be used is that the Aspect Ratios of the surface and the display area might be different. The previous section on *The EVR and*

Aspect Ratios discusses this concept in some detail and so it will not be discussed further here.

The part of the video stream which is rendered on the drawing surface is controlled by two numerical windows internal to the Enhanced Video Renderer. These windows are called the Source Window and the Destination Window. These two windows are specified in a call to the EVR's *SetVideoPosition* member. This call can be made at any time after the EVR is created and either or both of the windows can be changed at any time.

The Source Window determines which portion of the video is displayed. It is specified in normalized coordinates. In other words, the Source Window is configured with four values - each between 0 and 1. To display the entire video image we would set the Source Window rectangle to {0, 0, 1, 1}. To display the bottom right quarter we would set it to {0.75f, 0.75f, 1, 1}. The default Source Window value is {0, 0, 1, 1}.

The Destination Rectangle defines a rectangle within the clipping window (the video surface) where the video appears. This value is specified in pixels, relative to the display surface. In order to fill the entire display area, set the destination rectangle to {0, 0, width, height} where the width and height values are the dimensions of the window or control client area.

In this way all, or part, of the source video stream will be presented on the display surface within the area specified by the Destination Window.

If the Source and Destination Windows are set up in such a manner that the entire display surface is not useable due to incompatible Aspect Ratios, then the Enhanced Video Renderer will automatically adjust the displayable area in a way which is appropriate to the currently set Aspect Ratio Mode. This will either be stretching, clipping or letterboxing. Please see the previous discussion of the EVR's treatment of Aspect Ratio issues for more details.

In the *ctlTantaEVRFilePlayer* control, the display surface is the entire area of a child Panel control rather than the surface of the main control itself. This makes it possible to place buttons and other controls on the parent control without having to adjust the Destination Window size to cope with their presence. It is, of course, possible to do it the other way – draw directly on the main control client area and then use the Destination Window to ensure the region containing the buttons and other controls are not overwritten. In other words, we can use the Destination Window to arbitrarily write

on a specific sub-section of a display surface even though the entire area is theoretically writeable.

SOFTWARE MAGNIFICATION

Creative use of the Source Window can provide software magnification. If we leave the destination window the same and then use a Source Window of {0.25f, 0.25f, 0.75f, 0.75f} we will get a 2x magnification effect as the Enhanced Video renderer expands the displayable part of the video frames to fit the area. Quite large magnifications are possible as is illustrated in the Figures 8.4 and 8.5 below.



Figure 8.4: No Magnification



Figure 8.5: Large Magnification

Software magnification does not provide the underlying video stream with additional resolution – it can only make the details already present somewhat larger. Of course, if the resolution of the video stream is already much greater than the content in the video window, it will be possible to see extra detail not previously displayed.

HANDLING SIZE CHANGE EVENTS

Unless you specifically enforce a static screen size, it is rare on PCs for the user to avoid resizing the display area. Thus, if you are drawing on a client window area, you will need to change the Destination Window in response to that windows size change events. If you are drawing on the surface of a control, your application can get the same events simply by ensuring your control is appropriately anchored in all four directions.

Your code will receive size changed notices by setting up and handling the `SizeChanged()` event. Once inside this event, the Enhanced Video Renderers `SetVideoPosition` member can be called in order to adjust the display area. As mentioned previously, the `ctlTantaEVRFilePlayer` control in the `TantaCommon` Sample Project draws on the entire surface of a child Panel control. As the code below shows, it is a relatively trivial procedure to adjust the EVR's Destination window to match that of the new size of the panel control. The Panel control (`panelDisplayPanel` in this example)

is anchored to the Top, Left, Right and Bottom of the *ctlTantaEVRFilePlayer* controls client area.

```
HResult hr;
if (evrVideoDisplay == null) return;

try
{
    MFRect destinationRect = new MFRect();
    MFVideoNormalizedRect sourceRect = new MFVideoNormalizedRect();

    // populate a MFVideoNormalizedRect structure that specifies the source rectangle.
    // This parameter can be NULL. If this parameter is NULL, the source rectangle
    // does not change.
    sourceRect.left = 0;
    sourceRect.right = 1;
    sourceRect.top = 0;
    sourceRect.bottom = 1;

    // populate the destination rectangle. This parameter can be NULL. If this parameter
    // is NULL, the destination rectangle does not change.
    destinationRect.left = 0;
    destinationRect.top = 0;
    destinationRect.right = panelDisplayPanel.Width;
    destinationRect.bottom = panelDisplayPanel.Height;

    // now set the video display coordinates
    hr = evrVideoDisplay.SetVideoPosition(sourceRect, destinationRect);
    if (hr != HResult.S_OK)
    {
        throw new Exception("ctlTantaEVRFilePlayer SizeChanged failed. Err=" + hr.ToString());
    }
}
catch (Exception ex)
{
    LogMessage("Size change failed exception happened. ex=" + ex.Message);
    NotifyPlayerErrored(ex.Message, ex);
}

Source: TantaCommon::ctlTantaEVRFilePlayer::ctlTantaEVRFilePlayer_SizeChanged
```

PLAYBACK CONTROL

After years of dealing with DVD players and online videos, any viewer of a video stream will have a pretty good idea of what they think they should be able to do in the way of control. This means stopping, restarting, pausing, fast forward, rewind and generally skipping about in the video stream. If your application does not provide these controls you will probably get complaints. Fortunately the Enhanced Video Renderer supports all of these features – although you do have to use a variety of techniques to implement them.

The following section provides a detailed discussion of each option and you can find a working example to review in the *ctlTantaEVRFilePlayer* control in the *TantaFilePlayerAdvanced* Sample Project.

STARTING, PAUSING AND STOPPING THE EVR

In general, the Media Session manages the stream of media data moving from the sources to the sinks. This is true in Pipelines used for either the playback or the creation

of media files. The Media Session is all about the control of Media Streams and it is the `IMFMediaSession` interface on the Media Session that enables you to Start, Pause, Restart and Stop the video playback. When you perform operations like this you are not interacting with the EVR – you are simply preventing the media data from reaching it.

If you look carefully in the `OpenVideoFileAndPrepareSessionAndPlay` function of the `ctlTantaEVRFilePlayer` control you will see that the Media Session object is stored as a class variable as part of its creation process.

```
// Create the media session.  
hr = MFExtern.MFCreateMediaSession(null, out mediaSession);
```

Since the `mediaSession` object is a class variable it is available to any function in the `ctlTantaEVRFilePlayer` control. Once you have a Media Session object, controlling the actual flow of information is actually quite straight forward.

Recall how the Pipeline was originally launched by issuing the `Start()` function call on the Media Sessions `IMFMediaSession` interface. This function can also be used to restart the Pipeline after it has been stopped and the parameters to this function can be used to determine the start position within the media stream. In order to start a media stream at the beginning of a file, all that is necessary is to feed in dummy values to the `Start()` command as shown below

```
HResult hr = mediaSession.Start(Guid.Empty, new PropVariant());  
if (hr != HResult.S_OK)  
{  
    throw new Exception("Scall to mediaSession.Start failed. Err=" + hr.ToString());  
}  
  
Source: TantaFilePlaybackAdvanced::TantaCommon::ctlTantaEVRFilePlayer::StartPlayback
```

A Media Session is paused by issuing the `Pause()` call on its `IMFMediaSession` interface. There are no parameters to this call. What is really happening in this call is that the Presentation Clock is stopped. This clock controls the rate of display - if the clock is not moving forward then the Media Session does not move data and the video and audio appear to be paused.

Be aware that a second call to the `Pause()` function does not toggle the Pipeline back on. That must be done with a second call to the `Start()` function – again with the dummy parameters. In this case, since the session has been paused, the `Start()` just restarts the current Presentation Clock and hence resumes the media playback at the previous stop point.

The code below demonstrates an example of how pause functionality might be toggled on and off.

```
if (PlayerState == TantaEVRPlayerStateEnum.Paused)  
{
```

```
// we are already paused - we restart
hr = mediaSession.Start(Guid.Empty, new PropVariant());
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to mediaSession.Start failed. Err=" + hr.ToString());
}
}
else
{
    hr = mediaSession.Pause();
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to mediaSession.Pause() failed. Err=" + hr.ToString());
    }
    PlayerState = TantaEVRPlayerStateEnum.PausePending;
}
}

Source: TantaCommon::ctlTantaEVRFilePlayer::buttonPause_Click
```

In the above code it is not obvious why the act of pausing the video session sets the `PlayerState` variable to a value of `PausePending` and yet we test on a value of `Paused`. We won't discuss this in detail here since it is not a WMF concern and you can look it up for yourself in the `ctlTantaEVRFilePlayer` source. What is actually happening is that a call to the `Pause()` function triggers an event in the Media Sessions Callback Object which, when processed, will set the state of the `PlayerState` variable to a value of `Paused`. This can be seen in the `ctlTantaEVRFilePlayer` code in the `HandleMediaSessionAsyncCallBackEvent` function.

A Media Session can be stopped by issuing the `Stop()` function call. As with the `Pause()` function call, there are no parameters. A stopped Media Session will remove the video from the display and replace the drawing area with the default background color (which is usually black). A Media Session which has been stopped cannot be restarted.

GETTING THE DURATION AND THE CURRENT PROGRESS

It is quite common to want to know the duration of a media file being played and also where in the stream the current frame on display is positioned. Having this information makes it possible to do interesting things like updating a Progress Bar which displays the current state. It should be noted that it is possible to jump forward and backwards through a file based stream – even if the media data has not already been rendered. As we shall see in a later section, this makes it possible to set up a Scroll Bar control so that a mouse drag on its handle can make the Media Session move immediately to that relative point in the stream.

Although they are grouped together in this chapter, in reality, the duration and current progress are really two separate issues since their values are derived from two different components.

Rendering Audio and Video

The duration of a file based stream can be found by digging around in the Presentation Descriptor of the Media Source. Recall that we previously obtained the duration as the last action in the `OpenVideoFileAndPrepareSessionAndPlay` function after setting up the Pipeline.

```
// get the duration from the presentation descriptor now. This is nothing to do
// with the creation of the topology. We will eventually need this so
// we can tell the user how long the video is. We have to get it from the
// presentation descriptor and so might as well just get it here
VideoDuration = TantaWMFUtils.GetDurationFromPresentationDescriptor(
    sourcePresentationDescriptor); ());

Source: TantaCommon::ctlTantaEVRFilePlayer::OpenVideoFileAndPrepareSessionAndPlay
```

The actual method of deriving the duration from the Presentation Descriptor is quite straight forward. The code section below details the operation of the `GetDurationFromPresentationDescriptor` function in the `TantaWMFUtils` library.

```
public static UInt64 GetDurationFromPresentationDescriptor(
    IMFPresentationDescriptor sourcePresentationDescriptor)
{
    Int64 presentationDuration = 0;

    if (sourcePresentationDescriptor == null)
    {
        throw new Exception("No presentation descriptor provided");
    }

    // Ask the presentation descriptor for the duration. This should never be negative even
    // though it is stored as an Int64
    sourcePresentationDescriptor.GetUINT64(
        MFAttributesClsid.MF_PD_DURATION,
        out presentationDuration);

    return (UInt64)presentationDuration;
}

Source: TantaFilePlayback::TantaCommon::TantaWMFUtils
```

As can be seen above, obtaining the duration is just a matter of requesting it from Attribute collection of the Presentation Descriptor using the `MF_PD_DURATION` GUID key.

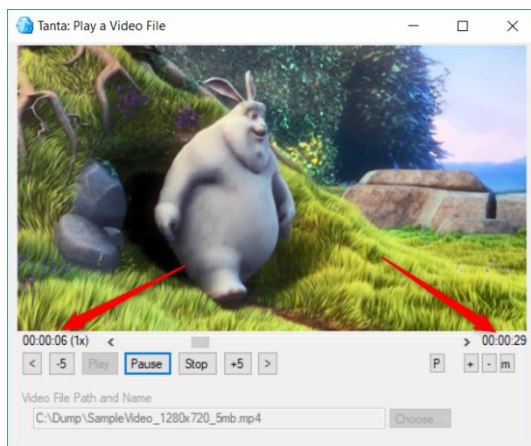


Figure 8.6: Current Position and Duration in the TantaFilePlayerAdvanced Application

Interestingly, this is one of those cases when dealing with Attributes that it is not necessary to get an `IMFAttributes` container from the object. The Presentation Descriptor directly implements the `IMFAttribute` interface – it *is* an `IMFAttributes` object. Thus the duration can just be directly requested.

Both the duration and current progress point are times. This is why you will sometimes see the current progress point referred to as the Presentation Clock. Unlike a normal clock, the Presentation Clock can be stopped and

re-started according to the current playing state of the Media Session. Even more unlike a normal clock, the Presentation Clock can also be speeded up or reversed in order to provide a fast forward or rewind capability.

Both duration and current Presentation Clock are stored in units of 100 nanoseconds (sometimes called “ticks”). They are stored internally as a 64 bit integer (an `Int64`). However this value will never be negative and the Tanta Sample Projects typically cast it to a `UInt64` whenever they store it.

Converting these clock values to seconds is as easy as dividing by 10000000. The *TantaWMFUtils* code provides a handy conversion tool which will take a Presentation Clock value and convert it to a string in the format HH:MM:SS. This output can be seen near the bottom right and left hand corners of the *ctlTantaEVRFilePlayer* display in the *TantaFilePlayerAdvanced* Sample

The current progress of the playing file (the value of the current Presentation Clock) is obtained from the Media Session. As with many things WMF, getting access to it is a two-step process, the static `GetPresentationTimeFromSession` function in the *TantaWMFUtils* library below shows how it is done.

```
public static UInt64 GetPresentationTimeFromSession(IMFMediaSession mediaSession)
{
    HRESULT hr;
    IMFClock clockObject = null;
    Int64 presentationClock = 0;

    if (mediaSession == null)
    {
        throw new Exception("No mediaSession provided");
    }

    try
    {
        // get our presentation clock, this needs to be released.
        hr = mediaSession.GetClock(out clockObject);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to MediaSession.GetClock failed");
        }
        if (clockObject == null)
        {
            throw new Exception("call to MediaSession.GetClock failed. clockObject == null");
        }
        // we have the clock object, but we actually need an IMFPresentationClock
        // the clock object returned above is both so we cast it.
        if ((clockObject is IMFPresentationClock) == true)
        {
            // the cast is valid, so get the time from it, this comes back as an Int64
            // even though it will never be negative
            (clockObject as IMFPresentationClock).GetTime(out presentationClock);
        }
    }
    finally
    {
        // release the clock object we just obtained
        if (clockObject != null)
        {
            Marshal.ReleaseComObject(clockObject);
        }
    }
    // return what we got
    return (UInt64)presentationClock;
}
```

```
    }  
  }  
}  
  
Source: TantaCommon::TantaWMFUtils::GetPresentationTimeFromSession
```

The first thing we have to do is get the `IMFClock` object from the Media Session.

```
// get our presentation clock, this needs to be released.  
hr = mediaSession.GetClock(out clockObject);
```

What we actually want is an object that implements the `IMFPresentationClock` interface but there is no direct way to get that from the Media Session. It is not obvious but the `IMFClock` object we get back from the `GetClock()` call is actually an `IMFPresentationClock` as well so a quick cast resolves that problem for us.

```
// the cast is valid, so get the time from it, this comes back as an Int64  
// even though it will never be negative  
(clockObject as IMFPresentationClock).GetTime(out presentationClock);
```

After that, we can just issue a simple `GetTime()` call on the `IMFPresentationClock` object and we will get the current Presentation Clock as an `Int64` – and the value of this will be the number of 100ns units since the start of play.

As with the duration, once we have the time value it is trivial to convert it into seconds or some human readable format.

SEEKING FORWARD AND BACK IN THE STREAM

Once you have the value of the current Presentation Clock, seeking to a new position in the video stream is fairly simple. In order to seek to a new position all that is required is to call the Media Session `Start()` command with a new Presentation Clock value as an input parameter. The code below, clipped from the *TantaFilePlayerAdvanced* sample, demonstrates how to set up a jump 5 seconds forward in the stream.

```
// we only permit this action if we are started or paused  
if (((PlayerState == TantaEVRPlayerStateEnum.Started) ||  
    (PlayerState == TantaEVRPlayerStateEnum.Paused)) == false)  
{  
    LogMessage("not started quitting now");  
    return;  
}  
  
// In order to seek forward 5sec really all we do here is add  
// 5 Sec (in 100ns chunks) to the current presentation clock  
// and call the session start function again  
  
// get the current time  
UInt64 presentationTime = TantaWMFUtils.GetPresentationTimeFromSession(mediaSession);  
  
// calc the new time, this call checks so as not to make the new time go out of bounds  
presentationTime = TantaWMFUtils.AddSecondsTo100nsTime(5, presentationTime, VideoDuration);  
  
// perform the seek, note we have to convert the new presentation time to an Int64  
// this is the way that WMF uses it even though it will never go negative.  
HRESULT hr = mediaSession.Start(Guid.Empty, new PropVariant((Int64)presentationTime));  
if (hr != HRESULT.S_OK)  
{  
    throw new Exception("call to mediaSession.Start failed. Err=" + hr.ToString());  
}
```



```
// were we paused? Make sure we stay paused
if(PlayerState == TantaEVRPlayerStateEnum.Paused)
{
    hr = mediaSession.Pause();
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to mediaSession.Pause() failed. Err=" + hr.ToString());
    }
    PlayerState = TantaEVRPlayerStateEnum.PausePending;
}
```

Source: `TantaCommon::ctlTantaEVRFilePlayer::button5SecPlus_Click`

The procedure is quite straight forward, first the current presentation time must be obtained. The call to the helper function *GetPresentationTimeFromSession* (discussed in the previous *Getting the Duration and the Current Progress* section) takes care of that. The presentation time value is recorded in 100 nanosecond units and the subsequent call to the *AddSecondsTo100nsTime* helper function takes an input value in seconds and adjusts presentation time appropriately.

```
// get the current time
UInt64 presentationTime = TantaWMFUtils.GetPresentationTimeFromSession(mediaSession);

// calc the new time, this call checks so as not to make the new time go out of bounds
presentationTime = TantaWMFUtils.AddSecondsTo100nsTime(5, presentationTime, VideoDuration);
```

Actually, there is more going on in this call than is first obvious. The presentation time must never be allowed to go below zero. This will cause an error when it is applied. Equally, the adjusted presentation time cannot be allowed to exceed the total duration or a similar error will occur. The *AddSecondsTo100nsTime* helper function handles all of this – in particular, note that the *VideoDuration* property is used to pass in the previously obtained total video duration for comparison purposes.

The new Presentation Clock value is passed in using the Media Session *Start()* command.

```
HRESULT hr = mediaSession.Start(Guid.Empty, new PropVariant((Int64)presentationTime));
```

It is the usage of the `new PropVariant((Int64)presentationTime)` parameter that tells the Media Session where to start. Note the cast of the presentation time to an *Int64*. As mentioned previously, the Presentation Clock is treated as an *Int64* in Windows Media Foundation and the Tanta sample code treats the presentation clock as a *UInt64* for manipulation and storage purposes. In addition, particularly note the discussion in the *PropVariant* section of the *MF.Net Programming Fundamentals* chapter. There are some significant differences in the C# implementation of *PropVariants*. In C#, a *UInt64* *PropVariant* is not the functional equivalent of an *Int64* *PropVariant* like it is in C++.

To start at the beginning of the media stream, an empty value can be passed in on the Media Session *Start()* command. Indeed, if you look at other calls to the *Start()*

Rendering Audio and Video

command on the Media Session in the *TantaFilePlayer* sample you will see an empty *PropVariant* value is passed in.

```
HResult hr = mediaSession.Start(Guid.Empty, new PropVariant());
if (hr != HResult.S_OK)
{
    throw new Exception("Call to mediaSession.Start failed. Err=" + hr.ToString());
}

Source: TantaFilePlayback::TantaCommon::ctlTantaEVRFilePlayer::StartPlayback()
```

In order to seek backward simply subtract the appropriate amount of 100 nano-second units from the Presentation Clock and call the Media Session *Start()* command as before. As mentioned previously, take care to not let the Presentation Clock go below zero or you will get an error when you apply it.

IMPORTANT NOTE: *The seeking technique described above works well for a simple button press. If you are generating a flood of seek actions then you need to make sure you do not overwhelm the session with seek commands.*

As the above note indicates, there you have to take great care to avoid flooding the Media Session with seek requests. Scroll Bars or Track Bars, in particular, are notorious for this. For example, it is common to tie the position of the Thumb control in a horizontal Scroll Bar control to the position in the image stream being viewed. As the user drags the Thumb control forward and backwards, a constant barrage of seek requests could be generated – far more than the Media Session can handle.

In such cases the seek requests must be throttled. The section of code below, clipped from the *TantaFilePlayer* sample, shows one method of how this can be done. The *PlayerState* variable acts as a semaphore to indicate if the Media Session is in the act of changing things. In particular, note how the *PlayerState* is set to a value of *TantaEVRPlayerStateEnum.StartPending* immediately after the session is started.

```
// we only permit this action if we are started or paused, We do NOT want to flood the
// session with seek requests. The act of seeking will set PlayerState to something else
// and it will be reset in the media sessions Callback Object.
if ((PlayerState == TantaEVRPlayerStateEnum.Started) ||
    (PlayerState == TantaEVRPlayerStateEnum.Paused)) == false)
{
    return;
}

switch(e.Type)
{
    case ScrollEventType.EndScroll:
        // all scrolling actions have ended
        return;

    case ScrollEventType.LargeIncrement:
        // the user clicked in the scroll bar to the right of the Thumb
        // convert a value in the range of 0 - 1000 to a video position. In this
        // case it is a delta offset
```

```

        deltaTime = TantaWMFUtils.ConvertRangeValueToVideoPosition(
            VideoDuration,
            TantaWMFUtils.DEFAULT_LARGE_INCREMENT_FOR_DURATIONRANGE);
        // get the current presentation time
        presentationTime = TantaWMFUtils.GetPresentationTimeFromSession(mediaSession);
        // calc a new time
        presentationTime += deltaTime;
        if (presentationTime > VideoDuration) presentationTime = VideoDuration;
        // start the session with the new time
        hr = mediaSession.Start(Guid.Empty, new PropVariant((Int64)presentationTime));
        // flag this it will inhibit future calls until the session has completely started
        PlayerState = TantaEVRPlayerStateEnum.StartPending;
        return;
    ... more code
Source: TantaCommon::ctlTantaEVRFilePlayer::scrollBarVideoPosition_Scroll

```

The `PlayerState` semaphore is reset to a value of `TantaEVRPlayerStateEnum.Started` in the Media Session's `HandleMediaSessionAsyncCallbackEvent` Callback Object.

```

case MediaEventType.MESessionStarted:
    // Raised when the IMFMediaSession::Start method completes asynchronously.
    PlayerState = TantaEVRPlayerStateEnum.Started;
    break;
Source: TantaCommon::ctlTantaEVRFilePlayer::HandleMediaSessionAsyncCallbackEvent

```

Callback Objects are the Media Session's mechanism for transmitting event state changes back to the owner code. In this case we can rely on the `MESessionStarted` event to let us know that the Media Session really has started and we set the `PlayerState` semaphore to a value of `TantaEVRPlayerStateEnum.StartPending` and this is then used in the event handler of the `scrollBarVideoPosition` control to enable it to send another seek request through the system.

FAST FORWARDING AND REWINDING THE EVR

A Media Session provides dedicated playback controls for the fast forwarding and rewinding of the media stream. However, a Media Session cannot provide all possible forward and reverse speed values - there are minimum and a maximum values you can use. Requests outside the minimum or maximum speeds will throw an error - as will requesting a negative rate (rewind) if reverse speeds are not supported.

If rewind or fast forward speeds suitable for your requirements are not supported, you can usually implement both operations via a clever seeking mechanism. This, of course, is not a true rewind or fast forward – but from the users perspective it functions much the same. There is much discussion of the techniques involved with this in the *Seeking Forward and Back in the Stream* section above.

To change the playback rate, the `IMFRateControl` interface is used. The `IMFRateSupport` interface is used to determine the range of playback rates (including reverse playback) that are supported. Both the `IMFRateControl` and the

`IMFRateSupport` interface are services which are obtained from the Media Session using an `MFGetService` call.

In order to adjust the playback speed, the rate at which the stream of images appear in the display window must either be increased or slowed down. This speed is normalized to a value of 1 and all variations are specified relative to that. A playback rate of 1 is the normal, real time, forward play. A playback rate of 2 is fast forward at 2x speed and, similarly, a playback rate of -1 is reverse play at normal speed. Partial values are also allowed so it is possible, for example, to set a fast forward speed of 1.5 which is a 1.5x forward rate. Perhaps not so obviously, a value of 0.5 could be used to configure half-speed slow motion. A playback rate of 0 will step you forward one frame – after that, to get another frame, you would have to seek to a new position using the techniques described in the *Seeking Forward and Back in the Stream* section.

The playback rate cannot be adjusted while the Media Session is playing. It is possible to make a `SetRate` call to change it while the session is playing – but the request will just be ignored and no error will be returned. In order to change the playback rate the Media Session must be stopped, or in some cases, paused. Stopping the Media Session, changing the rate and restarting it always works. However, the user might see a brief flash on the screen. Pausing the Media Session works well when transitioning, either up or down, from one positive rate to another. Pausing and restarting the Media Session does not seem to exhibit a flash but, as described below, pausing and restarting is not always possible. The permitted transitions are listed below and these are also documented on the `IMFRateControl::SetRate` method help page.

Playback State	Forward/Reverse	Forward/Zero	Reverse/Zero
Running	No	No	No
Paused	No	Yes	No
Stopped	Yes	Yes	Yes

Permitted Playback Rate Transitions

Be aware that when the Media Session is paused, or stopped, in order to adjust the playback rate you have to provide a Presentation Clock value when restarting. Not providing any value at all (`null`) or an empty `PropVariant` class (`new PropVariant()`) will simply restart the video stream from the very beginning. This is unlikely to be the desired effect when fast forwarding. Here is some example code illustrating the fast forwarding process.

```
// we have previously checked that the current rate is > 0 here otherwise we would
// have to Stop() not Pause() here.
UInt64 presentationTime = TantaWMFUtils.GetPresentationTimeFromSession(mediaSession);
```

```
mediaSession.Pause();

float rateRequested = 1.2f;
bool wantThinned = true;
outBool = TantaWMFUtils.SetCurrentPlaybackRateOnSession(mediaSession,
    wantThinned, rateRequested);

// start the session back up
mediaSession.Start(Guid.Empty, new PropVariant((Int64)presentationTime));
PlayerState = TantaEVRPlayerStateEnum.StartPending;
```

Source: Not in Tanta Samples

The above code uses calls into the *TantaWMFUtils* library to simplify the process of acquiring the presentation time and setting the rate. A quick look at the *TantaCommon* source code will soon show that these calls are really just wrappers for commonly used, but complex, multi-line operations.

A call to the `SetRate()` method of the `IMFRateControl` interface is used to adjust the presentation rate. The process first obtains the `IMFRateControl` interface from the Media Session and then uses that interface to adjust the rate. The code below demonstrates this process – in particular, note how the `IMFRateControl` interface is released at the `finally` block at the end.

```
HResult hr;
IMFRateControl rateControlService = null;
object rcServiceObj = null;

// sanity check
if (mediaSession == null) return false;

try
{
    // We get the rate control service from the Media Session.
    hr = MFExtern.MFGetService(
        mediaSession,
        MFServices.MF_RATE_CONTROL_SERVICE,
        typeof(IMFRateControl).GUID,
        out rcServiceObj
    );
    if (hr != HResult.S_OK)
    {
        throw new Exception("call to MFExtern.MFGetService failed. Err=" + hr.ToString());
    }
    if (rcServiceObj == null)
    {
        throw new Exception("call to MFExtern.MFGetService failed. rcServiceObj == null");
    }
    // set the rate control service now for later use
    rateControlService = (rcServiceObj as IMFRateControl);

    // now set the current rate on the rate control interface
    hr = rateControlService.SetRate(wantThinned, newRate);
    if (hr != HResult.S_OK)
    {
        throw new Exception("call to rateControlService.SetRate failed. Err=" + hr.ToString());
    }
    return true;
}
finally
{
    // release the rate control interface
    if (rateControlService != null)
    {
        Marshal.ReleaseComObject(rateControlService);
        rateControlService = null;
    }
}


```

Source: TantaCommon:TantaWMFUtils:SetCurrentPlaybackRateOnSession

As discussed previously, note that the `newRate` variable is a float. This value is always relative to 1 so a value of `0.5f` would be used to configure half-speed slow motion playback and a value of `1.8f` would be used to configure a 1.8x fast forward speed. Also note that, although it is not shown, the Media Session would be stopped or paused prior to the above code being called and it would be restarted immediately afterwards.

Also note that in the above code fragment, a Boolean variable named `wantThinned` is also passed in and so it would seem that now is a good time to talk a bit more about what Thinning is and why you might or might not want to use it.

If you are presenting the video to the user at a normal speed you are, by definition presenting every frame the video stream possesses on the screen. When fast forwarding, you can similarly choose to present every frame at a faster rate or you can choose to skip various frames and present only a selection of intermediate ones. The second method is called Thinning.

Clearly there are limits to how fast Windows Media Foundation can render frames to the video display and if you permit Thinning in a fast forward (or rewind) mode you will have much faster image rates available to you. This, it would seem, is an ideal situation – the user will not miss a few frames so there is no need to display each and every one and thus extremely fast rates are possible.

However, as always, things are not so simple. Most storage formats, for example, do not store every frame as a complete picture each of which is a standalone displayable image. Instead, most video compression techniques use a key-frame and intermediate frame mechanism. Thus the first frame on display is the complete image and then a sequence of following frames form what are basically deltas in which only the changes to the key frame are recorded. This achieves some spectacular compression ratios since, for example, a scene with an unchanging background would not have to contain any information about the background in most frames. However, there is the disadvantage that you cannot just display any random intermediate frame and expect to get the image. Up, and until, the next key frame is present in the stream the video image on the display is a composite of the key frame and *all* subsequent intermediate delta frames.

Thus the problem with Thinning arises as a side effect of video compression. When fast forwarding, for example, it is usually not possible to simply display every fourth frame in order to get a 4x speed increase. When dealing with compressed video streams, the data on the fourth frame probably isn't sufficient to display the entire intended image. Instead, in order to display any arbitrary frame, it would be necessary to go back to the previous key frame and then apply every delta frame up to the actual frame you wish to

display. Of course, if every frame is processed then, by definition, Thinning is not really being applied and all you are doing is saving a bit of time when drawing on the screen.

Typically, MP4 video streams only insert a key frame every 5 to 10 seconds. This value is configurable by the software creating the MP4 file. DVD compressed video usually inserts a key frame every 15 frames or so. If you use Thinning on a compressed stream like MP4 you may well find that your fast forward or rewind operation jumps about quite a bit and you see an annoying amount of shudder and jitter in the displayed video. When the Enhanced Video Renderer is dealing with a compressed stream with plenty of key frames, such as that used in DVDs which typically display 25 frames a second, Thinning can be used without much problem. Typically, in such situations, the fast forward progress looks smooth because there is less than a second between each display image.

It is possible to detect the maximum speed which can be used in un-Thinned mode and then switch in to Thinned mode for speeds above that. It all depends on the application – jumping forward 5 or 10 seconds when really forwarding fast may not matter in an hour long MP4. It is likely to matter a great deal if the MP4 video is only 30 seconds long.

The maximum and minimum display rates in both Thinned and un-Thinned mode can be determined by querying the `IMFRateSupport` interface. The code below shows the process.

```
public static bool GetFastestRate(
    IMFMediaSession mediaSession,
    MFRateDirection rateDirection,
    bool wantThinned,
    out float supportedRate)
{
    HRESULT hr;
    IMFRateSupport rateSupportService = null;
    object rsServiceObj = null;
    supportedRate = 0;

    // sanity check
    if (mediaSession == null) return false;
    try
    {
        // We get the rate support service from the Media Session.
        hr = MFExtern.MFGetService(
            mediaSession,
            MFServices.MF_RATE_CONTROL_SERVICE,
            typeof(IMFRateSupport).GUID,
            out rsServiceObj
        );
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to MFExtern.MFGetService failed. Err=" + hr.ToString());
        }
        if (rsServiceObj == null)
        {
            throw new Exception("call to MFExtern.MFGetService failed. rsServiceObj == null");
        }

        // set the rate control service now for later use
        rateSupportService = (rsServiceObj as IMFRateSupport);
        // now get the slowest rate
    }
}
```

Rendering Audio and Video

```
        hr = rateSupportService.GetFastestRate(rateDirection,
            wantThinned,
            out supportedRate);
        if (hr == HRESULT.MF_E_REVERSE_UNSUPPORTED)
        {
            // just let the user know the rate is not supported
            return false;
        }
        else if (hr == HRESULT.MF_E_THINNING_UNSUPPORTED)
        {
            // just let the user know the rate is not supported
            return false;
        }
        else if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to rateSupportService.GetFastestRate failed.");
        }
        // rate is supported
        return true;
    }
    finally
    {
        // release the rate control interface
        if (rateSupportService != null)
        {
            Marshal.ReleaseComObject(rateSupportService);
            rateSupportService = null;
        }
    }
}
```

Source: TantaCommon:TantaWMFUtils:GetFastestRate

The procedure to obtain the fastest rate probably seems quite familiar by now. The `IMFRateSupport` interface is obtained from the session via an `MFGetService` call. Then the `GetFastestRate()` method is called on that interface in order to obtain the fastest rate. There is a similar `GetSlowestRate()` method. In particular, note that the Thinning status is specified in the `wantThinned` variable. Of course, the fastest (or slowest) display rate is highly dependent on whether Thinning is permitted to be used. The other thing to note in the above code is the `rateDirection` parameter. This is an enum named `MFRateDirection` and it specifies the direction (forward or reverse) for which the fastest speed is being requested.

There are a total of eight options for which speeds may be collected (forward, reverse), (fastest, slowest) and (Thinned, un-Thinned). This is a bit of a pain to acquire and the Tanta library provides an automated way of collecting all of this information. The *TantaCommon* project contains a class named `TantaMediaSessionPlaybackRateCapabilities` which acts as a container for this information and can automatically acquire it. A sample call would be as follows...

```
TantaMediaSessionPlaybackRateCapabilities msRates = new
    TantaMediaSessionPlaybackRateCapabilities(mediaSession);
if (msRates != null) msRates.DumpPlaybackRatesToLog();
```

Source: Not in Tanta Samples

The act of creating the `TantaMediaSessionPlaybackRateCapabilities` class with a valid Media Session will obtain all the required data. The contents can subsequently be accessed with calls to properties like `FastestForwardSpeedThinned` and

`FastestReverseSpeedNonThinned`. Note that reverse speeds are returned as positive numbers and must be multiplied by -1 before use in a `SetRate()` call. A negative return value indicates that the operation is not supported at all. Thus a call to `FastestReverseSpeedNonThinned` returning a value of -1 means that reverse play is not supported in un-Thinned mode.

The above call to the `DumpPlaybackRatesToLog()` method will write the contents of the `TantaMediaSessionPlaybackRateCapabilities` class to the log file. An example of the output from this call is shown below.

```
ReverseSpeedIsSupportedNonThinned=False
FastestForwardSpeedNonThinned=2
SlowestForwardSpeedNonThinned=0.125
FastestReverseSpeedNonThinned=-1
SlowestReverseSpeedNonThinned=-1
ReverseSpeedIsSupportedThinned=True
FastestForwardSpeedThinned=8
SlowestForwardSpeedThinned=0.125
FastestReverseSpeedThinned=8
SlowestReverseSpeedThinned=0.125
```

Source: Not in Tanta Samples

There are a variety of other useful routines in the `TantaWMFUtils` class. Two items which may be of particular use are the `IsPlaybackRateSupported` function which is a wrapper for an `IsRateSupported()` call on the `IMFRateSupport` interface. This function will accept a proposed rate and indicate if that value is supported or not. The `IsRewindSupported` function provides a similar yes or no answer to detect if it is possible to invoke reverse play with negative speeds.

It should be noted that it seems that the Enhanced Video Renderer does not support reverse play very well on most stream types. This is not too surprising, if you recall the previous discussion regarding the mechanics of Thinning. The operation is very difficult – the EVR rendering engine would have to look much further back in the stream to find the key frame and then build forward to the display frame – for each frame as it stepped back through the sequence. This would be extremely slow and problematic. You will probably find, for most versions of the Enhanced Video Renderer, that reverse play is not supported in un-Thinned mode as it is just too much work. In Thinned mode on things like MP4 streams you may well find the stream plays in reverse but that it is extremely jerky as the displayed image just jumps backwards between key frames.

TAKING A SNAPSHOT OF THE VIDEO DISPLAY

There are many situations in which the user of an application might wish to take a still snapshot of the video image on display. True, this can be done on a Windows system by pressing the `Alt-PrintScreen` key and then cleaning up the resulting screen shot in

some sort of image processing software. This method is somewhat tedious however, and there is a better way. The Enhanced Video Renderer contains a member function named `GetCurrentImage()` which will convert the image on display, at the moment it is called, into a bitmap image (.bmp) file.

It should be noted that the `GetCurrentImage()` call does not actually supply all of the contents of the bitmap file. It supplies only the important bits and you have to add your own file header and bolt them all together. This is an easy enough thing to do, however, the really annoying part is that the documentation does not bother to mention that you need to do this and so you are left wondering why your saved bitmaps are always unreadable. Quite, why the implementers of the `GetCurrentImage()` function didn't see fit to just sort it all out internally and return a fully formed bitmap image is one of the mysteries of the universe which will probably never be explained.

Below is some sample code from the Tanta `ctlTantaEVRFilePlayer` control example which demonstrates how to take a snapshot of the display area of the Enhanced Video Renderer. Note that the EVR must be running for this code to be functional.

```
private void buttonTakeSnapShot_Click(object sender, EventArgs e)
{
    HRESULT hr;
    BinaryWriter bitmapWriter = null;
    BitmapInfoHeader workingBitmapInfoHeader = new BitmapInfoHeader();
    IntPtr bitmapData = IntPtr.Zero;
    int bitmapDataSize = 0;
    long bitmapTimestamp = 0;

    // we have to be playing or paused
    if ((PlayerState == TantaEVRPlayerStateEnum.Started)
        || (PlayerState == TantaEVRPlayerStateEnum.Paused)) == false)
    {
        // just send a warning beep
        System.Media.SystemSounds.Beep.Play();
        return;
    }

    try
    {
        // set the size here. the docs briefly state you have to do this in a one
        // liner towards the bottom. However, they REALLY mean it - nothing will
        // work without this being done
        workingBitmapInfoHeader.Size = Marshal.SizeOf(typeof(BitmapInfoHeader));

        // get the image on the screen now. This will give us the image data and the
        // bitmap info header. However, be aware that there are two headers associated
        // with every .bmp file. The first is a file header (which we have to build
        // ourselves) and the second is an info header which we are given in the call below.
        // Also note that the memory for the bitmapData variable we receive
        // here needs to be freed
        hr = evrVideoDisplay.GetCurrentImage(
            workingBitmapInfoHeader,
            out bitmapData,
            out bitmapDataSize,
            out bitmapTimestamp);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("buttonTakeSnapShot Click failed. Err=" + hr.ToString());
        }

        // bitmapData is an IntPtr. Use Marshal to copy the video data out into a byte array
        // bitmapDataSize is the length of bitmapData
        byte[] managedArray = new byte[bitmapDataSize];
        Marshal.Copy(bitmapData, managedArray, 0, bitmapDataSize);
    }
}
```

```

// build the output filename. By default this is the directory
// of the playing video file
string outputBitmapFile = Path.Combine(
    SnapshotDirectory,
    TantaWMFUtils.BuildFilenameWithTimeStamp(BITMAP_PREFIX, BITMAP_EXTENSION));

// now we have to build and populate the bitmap fileheader. None of the
// documentation tells you that you have to this - but the bitmap file will
// not be readable if you do not
TantaBitMapFileHeader fileHeader = new TantaBitMapFileHeader();
fileHeader.bfOffBits = (uint)(Marshal.SizeOf(fileHeader) +
    Marshal.SizeOf(workingBitmapInfoHeader));
fileHeader.bfReserved1 = 0;
fileHeader.bfReserved2 = 0;
fileHeader.bfSize = (uint)(Marshal.SizeOf(fileHeader) +
    Marshal.SizeOf(workingBitmapInfoHeader) + bitmapDataSize);
fileHeader.bfType = 0x4d42;

// Create a binary writer to output the file. We will just populate the file
// with the newly created fileheader, the info header we got from the
// GetCurrentImage call and the actual image data itself. We just write these
// sequentially one after the other.
bitmapWriter = new BinaryWriter(File.OpenWrite(outputBitmapFile));

// convert the file header to a byte[]. This is surprisingly complex in C#
byte[] fileBufferAsBytes = TantaWMFUtils.ConvertStructureToByteArray(fileHeader);
// write the file header out to the new bitmap file
bitmapWriter.Write(fileBufferAsBytes, 0, Marshal.SizeOf(fileHeader));

// convert the info header to a byte[]
byte[] infoBufferAsBytes =
    TantaWMFUtils.ConvertStructureToByteArray(workingBitmapInfoHeader);
// write the info header out to the new bitmap file
bitmapWriter.Write(infoBufferAsBytes, 0, workingBitmapInfoHeader.Size);

// write the actual data of the bitmap
bitmapWriter.Write(managedArray);

// close up
bitmapWriter.Flush();
bitmapWriter.Close();
bitmapWriter = null;

// tell the user audibly that things went well
System.Media.SystemSounds.Hand.Play();
}
catch
{
    // we did not succeed
    System.Media.SystemSounds.Beep.Play();
}
finally
{
    // clean up
    if(bitmapData != null)
    {
        Marshal.FreeCoTaskMem(bitmapData);
    }
    if(bitmapWriter != null)
    {
        bitmapWriter.Close();
        bitmapWriter = null;
    }
}
}
}

Source: TantaCommon::ctlTantaEVRFilePlayer::buttonTakeSnapshot_Click

```

The `GetCurrentImage()` call takes four parameters. The first is a `BitmapInfoHeader` struct. These are treated just like classes in MF.Net and so can just be passed in normally. However, do realize that you need to set information in this struct prior to passing it in (discussed below) and that you will also get information back in it which you will later need to use to build the bitmap.

The remaining parameters are defined as `out` values. They are the bitmap data itself (an `IntPtr`), the size of the bitmap data (an `int`) and a timestamp (a `long`). The fact that the bitmap data is an `IntPtr` means that accessing the actual data requires a special call to Marshal the information into .NET's managed memory space - this too is discussed below. The timestamp is presented as the number of 100ns units since the start of play. You may wish to review the discussion of the Presentation Clock in the *Getting the Duration and the Current Progress* section for more information on this topic. We could use this timestamp to build a file name if we wished, but the code above prefers to use the current system date and time for that purpose.

The `BitmapInfoHeader` structure is passed into the `GetCurrentImage()` call and its `Size` property needs to be initialized first. The documentation does state this - but because the requirement is not emphasized it is easy to miss. If you do not set the size properly, the `GetCurrentImage()` call will fail. Since we are dealing with C# it is not possible to directly discover the size of the `BitmapInfoHeader` structure as we would in C or C++. Instead, as can be seen in the sample code below, the `Marshal.SizeOf(typeof(BitmapInfoHeader))` call is used to perform this operation.

```
// set the size here.
workingBitmapInfoHeader.Size = Marshal.SizeOf(typeof(BitmapInfoHeader));

// get the image on the screen now.
hr = evrVideoDisplay.GetCurrentImage(
    workingBitmapInfoHeader,
    out bitmapData,
    out bitmapDataSize,
    out bitmapTimestamp);
```

After the `GetCurrentImage()` call successfully completes, we will have a fully populated `BitmapInfoHeader` structure, the bitmap data `IntPtr` will point at the data for the bitmap and the bitmap size variable will be populated. Note that the data returned for the bitmap needs to be released once you have copied it elsewhere otherwise you will get a memory leak.

In order to use the bitmap data, we need to convert it to a more accessible format in managed memory. A `byte[]` array is the typical container. The section of code below (reproduced from the full sample above) uses the bitmap data size and a `Marshal.Copy()` call to perform the copy and conversion of an `IntPtr` to a `byte[]` array.

```
// bitmapData is an IntPtr. Use Marshal to copy the video data out into a byte array
// bitmapDataSize is the length of bitmapData
byte[] managedArray = new byte[bitmapDataSize];
Marshal.Copy(bitmapData, managedArray, 0, bitmapDataSize);
```

As mentioned above, another essential thing to realize when creating a bitmap snapshot is that there are actually two header components associated with every bitmap. These headers are the `BitmapFileHeader` and the `BitmapInfoHeader` and it is important to

be aware that these are not just two names for the same thing. They are separate, distinct and you need them both. Moreover, the `GetCurrentImage()` function does not give you the `BitmapFileHeader` and you will need to build and populate this object yourself prior to saving the bitmap image. Neither C# nor MF.Net contain a definition for the `BitmapFileHeader` and so the *TantaCommon* library contains a definition named *TantaBitMapFileHeader*. MF.Net does, however, contain a definition for the `BitmapInfoHeader` struct in the `WindowsMediaFoundation.Misc` library and that is the definition used in the above sample code.

The population of the `BitMapFileHeader` and struct is shown below (reproduced from the full sample code above).

```
// now we have to build and populate the bitmap fileheader. None of the
// documentation tells you that you have to this - but the bitmap file will
// not be readable if you do not
TantaBitMapFileHeader fileHeader = new TantaBitMapFileHeader();
fileHeader.bfOffBits = (uint)(Marshal.SizeOf(fileHeader) +
    Marshal.SizeOf(workingBitmapInfoHeader));
fileHeader.bfReserved1 = 0;
fileHeader.bfReserved2 = 0;
fileHeader.bfSize = (uint)(Marshal.SizeOf(fileHeader) +
    Marshal.SizeOf(workingBitmapInfoHeader) + bitmapDataSize);
fileHeader.bfType = 0x4d42;

// convert the file header to a byte[]. This is surprisingly complex in C#
byte[] fileBufferAsBytes = TantaWMFUtils.ConvertStructureToByteArray(fileHeader);
```

As can be seen, the fields to populate and the calculation of the correct values for them are not especially obvious. However, the procedure is not all that difficult if you have an example. Note the conversion of the `fileHeader` value to a `byte[]` format at the end.

Once you have a populated `BitmapFileHeader`, you also need to get the `BitmapInfoHeader` in a `byte[]` format.

```
// convert the info header to a byte[]
byte[] infoBufferAsBytes =
    TantaWMFUtils.ConvertStructureToByteArray(workingBitmapInfoHeader);
```

Once you have all three parts, the creation of the bitmap file is simply a matter of writing them out one immediately after the other to a file with a `.bmp` extension. The order of the output is always the `BitMapFileHeader` first, then the `BitmapInfoHeader` and finally the bitmap data.

```
// write the file header out to the new bitmap file
bitmapWriter.Write(fileBufferAsBytes, 0, Marshal.SizeOf(fileHeader));

// write the info header out to the new bitmap file
bitmapWriter.Write(infoBufferAsBytes, 0, workingBitmapInfoHeader.Size);

// write the actual data of the bitmap
bitmapWriter.Write(managedArray);

// close up
bitmapWriter.Flush();
bitmapWriter.Close();
```

Rendering Audio and Video

This can be seen in the above sample code where a standard C# `BinaryWriter` object is used to output and concatenate all the data. In particular, note how data written by the `BinaryWriter` is cleared from memory with a `Flush()` call before it is closed. Once this is done, the bitmap file is on the disk and should be readable by any software which can operate on bitmaps.

It is important to free up the bitmap data memory returned by the `GetCurrentImage()` function. This operation is performed in the sample code with a call to `FreeCoTaskMem()` in the `finally{}` block.

```
// clean up
if(bitmapData != null)
{
    Marshal.FreeCoTaskMem(bitmapData);
}
```

Windows Media Foundation: Getting Started in C#

Chapter 9

WORKING WITH TRANSFORMS

The *Transforms* section in *The WMF Components* chapter of this book provided an overview of Windows Media Transforms (WMT's) and if you are unfamiliar with the contents of that section and of the discussion of Pipelines in the *Windows Media Foundation Architecture* chapter, you will probably find it very useful to go back and review those two sections now. An understanding of those concepts will greatly assist you in absorbing the following information.

A Transform is an object that processes the data in the Pipeline as that data traverses from a Media Source to a Media Sink. As such, a Transform has both inputs and outputs – if it only had outputs it would be a Media Source and if it only had inputs it would be a Media Sink.

Transforms are used for all sorts of purposes and the requirement to support a wide variety of functions has had an effect on the design of its architecture. For example, say it was desirable to present the video data and audio data from a camera in two locations – on the screen in a real time window and streamed over the Internet for a permanent record. A Transform designed as a splitter (a demultiplexer or Tee) could be used to copy the incoming video data and place it on two different outputs. In such a situation, it would be said that there is a branch in the Topology at that point. Similarly, a case can

be made for a requirement to have multiple inputs multiplexed onto a single output – perhaps two video feeds with one displaying as a video window in a larger video.

Thus it is necessary that the architecture of WMF Transforms has the capability of supporting one or more inputs and also one or more outputs. The actual number of inputs and outputs depends on the purpose of the Transform and unless a Transform actually supports multiple outputs you cannot make it do so. In other words, the design supports multiple inputs and outputs but any one Transform is only required to have one input and one output at minimum. Many, if not most, Transforms have only a single input and a single output.

The data moving through each input and output in a Transform is called a stream. If, as in the above example, the Transform splits the input into multiple outputs there would be one stream going into the Transform and multiple streams coming out of it. There may well be other Transforms in each of the two resulting branches and the Topology of each branch need not be the same.

Note that the previous example mentioned Video and Audio data as outputs from a Camera and Microphone. These are two examples of a Media Major Types (there are others). The Media Source will present these two Media Major Types as two Media Streams – thus the first branch in the Topology would start at the Media Source. In general, a Transform is expected to handle only one Media Major Type. In other words, there can be video transforms and audio transforms – but any one Transform is not both. This makes sense – since the Topology has branched long before the stream

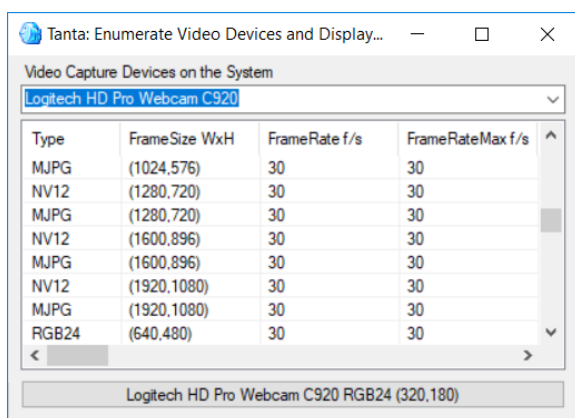


Figure 9.1: The TantaVideoFormats Sample Application

reaches the Transform there is no need for a Transform to be able to deal with both Media Major Types.

This is not to say that simply because the Transform only has to deal with one Media Major Type that the situation is necessarily simple for it. For any one Media Major Type there can be a multitude of format standards, frame sizes and compression methods available.

Taking a USB camera on a PC as an example (in Figure 9.1 a Logitech C920 Webcam), we can see there are a number of formats and frame sizes on offer. The image below does not show all of the available formats – there are over 60 of them for that camera and those are just the ones for uncompressed video.

The Media Source must be told the Media Sub-Type it will use when it obtains data from the camera. Which Media Sub-Type actually gets selected from the many on offer usually depends on the application and its interaction with the Media Source. Perhaps you, as the programmer, picked one for it and effectively just said “*Use NV12 format in 1920x1080*” or perhaps there was some other algorithmic mechanism to choose one. In many cases, however, the Media Source implemented in the application just uses the cameras default Media Sub-Type – which goes a long way towards explaining why many applications display webcam video at a resolution of 640x480 even though just about every camera is capable of much better these days.

Speaking of speed, it is probably worth mentioning that, if the camera is sending the data over a relatively slow link (USB or the Internet for example), it would probably be preferable to have the Media Source choose a highly compressed Media Sub-Type such as H.264 from the video source. This would mean that the data traversing the slowest part of the transmission path would already be compressed by the camera and thus a much larger frame size might be usable. Of course, a suitable decompression Transform might have to be added to the Topology in order to convert the data into a format acceptable to the Media Sink if the Media Sink cannot accept that format directly.

The presence of compression and decompression transforms (collectively known as Codecs) in the Pipeline brings up an important point. Although a decompression transform will probably only have one input and one output (and hence one input stream and one output stream) the amount of data leaving the Transform will be much greater than the amount of data entering it. The data entering the Transform will essentially be a densely compressed binary stream and the output will be a series of blocks of data - each of which contains a collection of meaningful information. For a video stream, each block will probably be one video frame – the image to display on the output at that time. Since the media data leaving the Transform is in a different format than the media data that entered it, the output stream of the Transform will have a different Media Sub-Type than the input stream.

The fact that a Transform can actually generate more data than it receives means that the Transform will have to get the storage space from somewhere. Since the quantity of memory which might be required is entirely arbitrary, the Transform architecture must be able to cope with the fact that the Transform might need to request additional data buffers for the output and that this data storage must be released (or reused) when their purpose is served or the system will quickly run out of memory. The converse is also true in the case of a compression Transform. There will be much more data

entering a compression Transform than leaves it and the various memory allocation and de-allocation requirements will need to be sorted out there as well.

THE TANTA TRANSFORM SAMPLE PROJECTS

The Tanta Sample Projects contain four applications which demonstrate the use of Transforms. The discussion in the following sections is going to reference these samples reasonably frequently. It would, at this point, be useful to undertake a slight digression so that you have a useful reference to the demonstration Transforms which are available and are aware of their purpose.

It must be noted that none of the Transforms in the Tanta Samples have been written from the ground up. They are all derived from base classes (called the Tanta Transform Base classes) which provide much of the functionality. *The Tanta Transform Base Classes* section below will provide more detail on the base classes – for now just realize that these classes automate a lot of the routine work which must be performed in order to implement a Transform. This does mean though, that if you compare the code in these Transforms to other sample code you may find on the Internet, that the structure will look a bit different. You may have to dive into the base classes to follow the operation of many of the `IMFTransform` interface functions.

Tanta Sample Projects implement a total of six Transforms - only one of which uses the Asynchronous Mode. The other five are all Synchronous Mode. In reality, Asynchronous Mode Transforms are a pretty specialized and advanced topic. They will not be discussed in any detail in this book.

Four of the Transforms are implemented in the `TantaTransformDirect` Sample Project. These Transforms are instantiated using the C# `new` operator and are added into the Topology using the techniques discussed in the *Adding Transforms To a Topology* section of *The WMF Components* chapter. Another is a very specialized Transform located in the `TantaCaptureToScreenAndFile` Sample Project and it is similarly loaded. The remaining Transform is implemented as a DLL in order to demonstrate that technology. It can be found in the `TantaTransformInDLL` Sample Project. The companion `TantaTransformInDLLClient` Sample Project dynamically loads this Transform and interacts with it.

The Tanta Sample Transforms are as follows...

`MFTTantaFrameCounter_Sync` – Located in the `TantaTransformDirect` Sample Project, this Synchronous Mode Transform is intended to be the simplest

possible demonstration of a Transform. It just counts the Media Samples as they pass through. It performs in place processing and hands back as output the same Media Sample it received as input. Because the Transform does not perform any processing on the data (other than counting it) this Transform is capable of processing data of any Media Sub-Type. This example Transform also demonstrates the ability of an application to interact with a Transform and retrieve information (the frame count) from it.

`MFTTantaGrayscale_Sync` – Located in the *TantaTransformDirect* Sample Project, this Synchronous Mode Transform is intended to convert the image it receives into grayscale. For purposes of demonstration, this Transform does not perform in-place processing and returns a Media Sample containing a copy of the input buffer to the Media Session. Since the process of converting an image to grayscale is so intimately tied to the format of the data, this Transform only supports the YUY2, UYVY and NV12 Media Sub-Types.

`MFTTantaGrayscale_Async` – Located in the *TantaTransformDirect* Sample Project, this Synchronous Mode Transform is an Asynchronous Mode version of the `MFTTantaGrayscale_Sync` Transform and is intended as a demonstration of that technology. This Transform also supports only the YUY2, UYVY and NV12 Media Sub-Types.

`MFTTantaWriteText_Sync` – Located in the *TantaTransformDirect* Sample Project this Synchronous Mode Transform draws user defined text on the image. This Transform performs in-place processing and returns a Media Sample containing the updated version of the original input buffer. Both full over-write and semi-transparent writing modes are demonstrated. Since the process of writing on an image to grayscale is so intimately tied to the format of the data, this Transform only supports the RGB32 Media Sub-Type.

`MFTTantaVideoRotator_Sync` – Located in the *TantaTransformInDLL* Sample Project this Synchronous Mode Transform is implemented as a DLL and is able to rotate the video image. This Transform performs in-place processing and returns a Media Sample containing the updated version of the original input buffer. Since the process of rotating an image is so intimately tied to the format of the data, this Transform only supports the RGB32 Media Sub-Type. The rotation mode of the image can be set dynamically while the Media Session is running. The *TantaTransformInDLLClient* Sample Project demonstrates how the DLL containing this Transform is dynamically loaded and various methods of interacting with it to exchange information.

`MFTTantaSampleGrabber_Sync` – This is a very specialized Synchronous Mode Transform located in the *TantaCaptureToScreenAndFile* Sample Project. This Transform copies the Media Samples that are passing through and gives them to a Sink Writer so they can be recorded to disk. The application can communicate with the Transform to turn the recording on and off as required. Because this Transform does not perform any processing on the data (other than copying it), this Transform is structured to process data of any Media Sub-Type.

It must be recognized that the Tanta Transforms (other than the `MFTTantaSampleGrabber_Sync`) are not original to the Tanta Library. These Transforms are re-writes of the open source sample versions which ship with the MF.Net library. Those examples, in turn, appear to be C# ports of the original C++ Transform examples. The techniques used in these Transforms are quite sophisticated and their presence as examples is much appreciated.

BASIC TRANSFORM OPERATION

We have seen in a previous section how a Transform always has at least one input and at least one output. In addition, the Media Sub-Type on the input can be different than the Media Sub-Type on the output. This was mentioned in the context of a decompression Transform but there are also many other Transforms dedicated specifically to the conversion of data between various Media Sub-Types. These types of conversion Transforms are known as Digital Signal Processors (DSP's).

It was also mentioned that it is possible that the amount of data leaving the Transform might be different than the amount which entered it. In order to preserve clarity, we will not cover that aspect of Transforms in this section. Let's leave that discussion for the *Data Processing in the Transform* section further on and, for now, simply assume the Transform can easily acquire or release all the memory it wishes (because that is exactly what it does).

It should be mentioned that the content below is going to get kind of technical. Unless you are going to write your own Transforms most of the information below will be largely academic. Transforms that you just use (rather than write yourself) are typically just "black-boxes" and your only interaction with them is setting up the streams and Media Types they use as you build your Topology.

If you are writing your own Transform then, for the most part, what you are really doing is implementing an `IMFTTransform` interface wrapped around a relatively small portion

of your own custom code. In order to make the discussion of the complex `IMFTransform` interface somewhat simpler, the functions it requires have been broken up (in this book) into three functional groups. These groups are

1. The `IMFTransform` functions related to stream management.
2. The `IMFTransform` functions related to Media Type negotiations.
3. The `IMFTransform` functions related to the processing of the data.

Each of these functional groups will be discussed in one of the sections below.

Don't be too worried about implementing the `IMFTransform` interface, the Tanta Library provides some base classes which make it much easier to write Transforms for certain common situations. This book will not discuss the general case of writing the entire Transform from the ground up. That subject could easily fill a book this size by itself and is not an especially introductory topic. We will, however, cover usage of the Tanta Transform base classes to build a Transform. This, and the various Tanta Transform Sample Projects, will make it possible for you to create the infrastructure of a Transform relatively quickly and permit you to focus on the meaningful parts of the code such as the processing of the media data.

THE TANTA TRANSFORM BASE CLASSES

There are two Transform Base classes in the Tanta Library. These are `TantaMFTBase_Sync` and `TantaMFTBase_Async`. As the names suggest, one (`TantaMFTBase_Sync`) is a base class for Synchronous Mode Transforms and the other (`TantaMFTBase_Async`) is a base class for Asynchronous Mode Transforms. The *TantaTransformInDLL* Sample Project also directly incorporates its own copy of the `TantaMFTBase_Sync` class (`MFTBaseStandalone_Sync`). This is done so that the resulting DLL does not require the presence of the *TantaCommon* library.

These base classes make it easy to implement Transforms in C# - but there are some caveats. The Tanta Transform Base classes are only usable in certain circumstances.

If your Transform inherits from one of the Tanta Transform Base Classes (`TantaMFTBase_Sync` or `TantaMFTBase_Async`) most of the work of supporting the `IMFTransform` interface is done for you. However, these classes only support a simple Transform model. With the Tanta Transform Base classes you can only have one input

and one output stream, the Media Type on the input stream must be identical to the Media Type on the output stream and there can only be one output sample for every input sample.

The Tanta Transform Base classes work by directly implementing the `IMFTransform` interface. These functions, besides performing a bit of admin, do either of two things. Many of the base class functions simply handle the work themselves and any class that inherits from them does not even need to know they exist. Other functions simply call a `protected virtual` or `abstract` function in the base class which has the same name but which uses the prefix “*On*”. Thus, if the Media Session calls the `SetOutputType()` function on the `IMFTransform` interface, the `OnSetOutputType()` function will be called in the base class. If your Transform overrides the “*On*” function, then it can handle the processing. The functions in the base class marked “virtual” are the ones you may or may not need to override in your Transform and the ones marked “abstract” do, of course, need to be implemented.

Classes derived from the Tanta Transform Base Classes need only override a few abstract or virtual functions in order to easily implement a fully operational Synchronous or multi-threaded Asynchronous Transform.

It is important to recognize that the Transform Base Classes did not originate with the Tanta Library. The authors of MF.Net (note, not WMF itself) designed these two base classes and incorporated them into the open source sample code which ships with that library. In the MF.Net sample code, the two base classes are named `SyncMFTBase` and `ASyncMFTBase`. The Tanta Transform Base classes are directly derived from these `SyncMFTBase` and `ASyncMFTBase` classes with few changes - mostly just more detailed comments and reformatting. The structure and concept of the `SyncMFTBase` and `ASyncMFTBase` classes really is some beautiful work.

We will now return to a discussion of basic Transform operations with a view to providing you with the information you might need in order to implement your own Transforms. Since this is intended to be an introductory book, we will simplify things by mostly only discussing Synchronous Mode Transforms which are derived from the `TantaMFTBase_Sync` base class.

TRANSFORMS AND MULTIPLE STREAMS

Transforms have at least one input stream and one output stream but they can have more of each. It is entirely up to the Transform if it wishes to support more than one stream on its input or output – some will support a variable number. This implies there is a mechanism to find out things like how many input streams exist, add a new output stream, discover the maximum number of output streams – and so on. Pretty much anything you think you might want to do regarding the creation and configuration of streams on a Transform can be done through the `IMFTransform` interface.

Does this mean that, when you write your own Transform, that you have to support the creation of multiple streams and all the other stream related function calls? Well, the answer is: **Yes** and **no** and **maybe no**.

- **Yes**, your Transform has to implement a function for each one specified in the `IMFTransform` interface.
- **No**, if your Transform only supports one input and one output stream you can just return a `HRESULT.E_NOTIMPL` or some other sensible default value and so the implementation of many of the stream related functions are not too much trouble.
- **Maybe No**, you may not have to implement each `IMFTransform` interface function at all. If your Transform can just inherit from one of the Tanta Transform Base classes most of this work will be done for you.

It should be noted that nearly all of the time, the stream configuration group of functions of the `IMFTransform` interface will only ever be called during the resolution of the Topology. Other than that, even though the specification supports the dynamic addition and removal of streams, it typically never happens.

Since there is only one input and one output stream in Transforms derived from the Tanta Transform Base classes, most of the functions in the stream configuration group are handled for you behind the scenes. However, there are some things you do need to implement. Your code will be expected to handle the `GetInputStreamInfo` and the `GetOutputStreamInfo` function calls in order to provide Transform specific information. As per the naming convention, this means that you have to override the `OnGetInputStreamInfo` and the `OnGetOutputStreamInfo` functions in the Tanta Transform Base Classes. It is the job of these functions to provide the Media Session with information on how the Transform handles the Media Samples. Listed below is a code section from the `MFTTantaVideoRotator_Sync` class in the `TantaTransformInDLL` Sample Project.

Working With Transforms

```
//+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+  
/// <summary>  
/// Return settings to describe input stream. This should get the buffer  
/// requirements and other information for an input stream.  
/// (see IMFTTransform::GetInputStreamInfo).  
///  
/// An override of the abstract version in TantaMFTBase_Sync.  
///</summary>  
///<param name="pStreamInfo">The struct where the parameters get set.</param>  
///<history>  
///    01 Nov 18 Cynic - Ported In  
///</history>  
override protected void OnGetInputStreamInfo(ref MFTInputStreamInfo pStreamInfo)  
{  
  
    // return the image size  
    pStreamInfo.cbSize = m cbImageSize;  
  
    // MFT_INPUT_STREAM_WHOLE_SAMPLES - Each media sample(IMFSample interface) of  
    //     input data from the MFT contains complete, unbroken units of data.  
    // MFT_INPUT_STREAM_SINGLE_SAMPLE_PER_BUFFER - Each input sample contains  
    //     exactly one unit of data  
    // MFT_INPUT_STREAM_FIXED_SAMPLE_SIZE - All input samples are the same size.  
    // MFT_INPUT_STREAM_PROCESSES_IN_PLACE - The MFT can perform in-place processing.  
    //     In this mode, the MFT directly modifies the input buffer. When the client calls  
    //     ProcessOutput, the same sample that was delivered to this stream is returned in  
    //     the output stream that has a matching stream identifier. This flag implies that  
    //     the MFT holds onto the input buffer, so this flag cannot be combined with the  
    //     MFT_INPUT_STREAM_DOES_NOT_ADDREF flag. If this flag is present, the MFT must  
    //     set the MFT_OUTPUT_STREAM_PROVIDES_SAMPLES or MFT_OUTPUT_STREAM_CAN_PROVIDE_SAMPLES  
    //     flag for the output stream that corresponds to this input stream.  
    pStreamInfo.dwFlags = MFTInputStreamInfoFlags.WholeSamples |  
        MFTInputStreamInfoFlags.FixedSampleSize |  
        MFTInputStreamInfoFlags.SingleSamplePerBuffer |  
        MFTInputStreamInfoFlags.ProcessesInPlace;  
}  
}}
```

Source: TantaTransformInDLL::MFTTantaVideoRotator Sync::OnGetInputStreamInfo

The `OnGetInputStreamInfo` function returns an `MFTInputStreamInfo` struct with the image size and some processing flags. The `m_cbImageSize` value will have been set earlier in a call to `OnSetInputType` which we have not discussed yet. The comments regarding the flags are pretty self-explanatory, however, the use of the `MFTInputStreamInfoFlags.ProcessesInPlace` flag is quite important.

If your Transform returns the `ProcessesInPlace` flag when asked for the input stream info then it is saying it will make changes directly into the Media Buffer the Media Session provides on input and return that same buffer when asked for output. If it does not implement this flag, then the Media Session will expect the Transform to build and return a new buffer.

The `ProcessesInPlace` flag fundamentally affects the operation of your Transform. Not all Transforms need the overhead of creating a new buffer for each Media Sample they output and the “*processing-in-place*” mode is much more efficient. This topic is discussed in more detail in the *Processing in the Transform* section below.

TRANSFORM STREAMS AND MEDIA TYPES

Every input stream in the Transform has a Media Sub-Type as does every output stream. Many Transforms can accept and process data in any one of several Media Sub-Types. As you might imagine there are quite a number of functions on the `IMFTransform` interface devoted to the setting, getting and negotiation of the Media Sub-Type on each stream.

It should be noted, in the unlikely event you were in any doubt, that each Transform that accepts multiple Media Sub-Types is explicitly coded to be able to process those media formats. A Transform always “knows” what Media Sub-Types it can deal with because some programmer somewhere went to a great deal of trouble to make it happen.

The Media Sub-Type related functions of the `IMFTransform` interface are a bit more complex in that they have to support the caller enumerating the various format possibilities offered by the Transform and then selecting the one it prefers.

There are a variety of reference implementations of the various Media Sub-Type related functions of the `IMFTransform` interface available on the Internet and we will not undertake a detailed discussion of the general case here. So, if you write your own Transform, do you have to support the negotiation of Media Types on your streams? Well, the answer is: **yes**, **yes** and **no** and **some**.

- **Yes**, your Transform has to implement a function for each one specified in the `IMFTransform` interface – no getting away from that.
- **Yes**, you still have to some support Media Type negotiation even if your Transform is hardcoded to only use one Media Type. The process of resolving the Topology will make calls to these functions and they cannot all return “*sorry can’t do it*”
- **No**, in a common case, you don’t really have to implement each `IMFTransform` interface function. Your Transform can just inherit from one of the Tanta Transform Base classes and much of this work will be done for you.
- **Some**, if you do inherit from the Tanta Transform base classes you still have to implement some of the code that negotiates the Media Sub-Types on the Transform – the base classes cannot hard code support for a specific Media Type. However, the things you have to write are simple overrides and much of rest the work is taken care of for you.

Similar to the discussion in the section above on stream configuration, the Media Type group of functions on the `IMFTTransform` interface will only ever be called during the resolution of the Topology. Other than that, even though the interface supports the dynamic modification of Media Sub-Types, it typically never happens. In other words, once the Pipeline is up and rolling pretty much nobody is crazy enough to want to change the media format on a Transform stream even though the standard says the Transform should support that. Doing so is just asking for trouble.

So if you do configure your Transform to inherit from the Tanta Transform Base classes, what do you need to do in order to properly support the Media Type negotiation? Basically there are three functions in this group that you need to concern yourself with.

To be more accurate, there are three functions for the input stream and three for the output stream, but since the Tanta Transform Base classes require the output Media Type to be identical to the Media Type on the input stream, you really only need to worry about the ones relevant to the input.

The three most common functions are: `OnCheckInputType`, `OnEnumInputTypes` and `OnSetInputType`. Of course there are others, but you will rarely need to override the base classes handling of them. In the Tanta Transform Samples, the processing in each function each varies considerably and they can get rather lengthy. Rather than reproduce each example of the code here, we will just provide a summary of the behaviors and you can inspect the source yourself. See *The Tanta Transform Sample Projects* section above for a more detailed overview of the function of each Transform.

`MFTTantaFrameCounter_Sync`

`OnCheckInputType` – will accept any media type.

`OnEnumInputTypes` – just returns `HResult.MF_E_NO_MORE_TYPES`.

`OnSetInputType` – leaves it to the base class to record the input Media Type.

`MFTTantaGrayscale_Sync` and `MFTTantaGrayScale_Async`

`OnCheckInputType` – Performs a series of checks on the supported list of Media Types.

`OnEnumInputTypes` – just returns the Media Type from the supported list at the specified index.

`OnSetInputType` – the base class records the input Media Type and the override extracts a variety of information from the input Media Type depending on the Media Sub-Type chosen.

`MFTTantaWriteText_Sync` and `MFTTantaVideoRotator_Sync`

`OnCheckInputType` – Performs a series of checks on the single Media Type it supports

`OnEnumInputTypes` – just returns the Media Type from the supported list at the specified index.

`OnSetInputType` – the base class records the input Media Type and the override extracts a variety of information from the input Media Type.

`OnSetOutputType` – if the output Media Type is not already set, this call makes the output Media Type equal to the Input Media type.

In general the `OnCheckInputType` and `OnEnumInputTypes` calls are only used by the Media Session to figure out a preferred Media type. To support these functions all you really need to do is provide a list of acceptable Media Sub-Types in the form of an array of GUIDs. The `OnSetInputType` function is usually custom written for each Transform. When the input Media Type is set, the user written override in the Transform usually takes that opportunity to dig various things like the frame size and interlace mode out of the Attributes of the incoming Media Type. If the Transform needs this information in order to later manipulate the media data, it is useful to get it in the `OnSetInputType` function and save it in a class variable for later use.

PROCESSING IN THE TRANSFORM

The Media Session controls the transfer of the data to the Transform. It picks the data up off one component in the Pipeline and gives it to the next object in the branch. Leaving aside all the complexities the Media Session has around the synchronization of the movement of the data (the Presentation Clock and all that), from the point of view of the Transform, the operation is actually pretty simple. It works like this...

1. The Media Session has some data
2. The Media Session asks the Transform if it can accept the data
3. If yes, the Media Session gives the data to the Transform.
4. The Media Session asks the Transform if it has any data
5. If yes, the Media Session gets the data from the Transform.
6. And repeat with the next component in the branch.

Of course, it really is a bit more complex than that as steps 2-3 and 4-5 are both operating simultaneously in different threads. This is what makes it possible to pull far more data off an output stream than was input (and vice-versa). However, as a sort of broad understanding, the above sequence is not too far off.

As you might imagine, this means that the `IMFTransform` interface is designed to support the above types of requests. The four functions on the `IMFTransform` interface which assist with the processing of the media data are: `GetInputStatus`, `ProcessInput`, `GetOutputStatus`, `ProcessOutput`.

Due to the fact that the Tanta Transform Base classes only permit one output Media Sample for each input Media Sample, the two functions `GetInputStatus` and `GetOutputStatus` are entirely implemented in the base classes. The Transform will simply block while it is processing data. This means that if either of those two functions can execute, the Transform, by definition, is ready to accept a Media Sample if it does not already have one or return one if it does. The base classes can detect that state and respond appropriately.

The `ProcessInput` function is similarly handled by the base classes. It just stores the incoming Media Sample in a class variable for later processing. You could, of course, override `OnProcessInput` in your Transform and implement your own behavior.

It is in the `OnProcessOutput` function that most of the work happens. By the time this function gets called, the Media Sample has been given to the Transform in a previous call to the `ProcessInput` function. The job of the `OnProcessOutput` function is to process the data according to its requirements. For example, the `MFTTantaFrameCounter_Sync` Transform just counts the Media Sample and hands it back, the `MFTTantaGrayscale_Sync` Transform copies the image to another buffer, converts it to grayscale and hands that over to the Media Session - it does not perform in-place processing.

The above discussion is the basic mechanics of how the media data is presented to the Transform and processed within it. Basically, if you are using the Tanta Transform Base classes, everything is all done in the `OnProcessOutput` call and the `OnProcessInput` call just provides the sample to the base class.

How then does the data actually get processed? Well, the processing that is done is highly specific to the work required and the input Media Sub-Type it is operating on. There is not much point in reproducing the source code from each Tanta Example Transforms here. The code is extremely lengthy, highly detailed and specific to the Transform involved. The *Raw Data Handling in the Transform* section below summarizes the operation of a representative Tanta example Transform and you can easily look at the `OnProcessOutput` call in the others to review how they perform their operation. Considerable care has been taken to make the comments in those functions as explanatory as possible.

EVENTS AND MESSAGES

There are two functions related to the sending of events and messages on the `IMFTTransform` interface. These are `ProcessEvent` and `ProcessMessage`. Both of these functions are handled internally within the Tanta Transform Sync Base class and there is no way to override this without changing the base class code.

The processing is simple. `ProcessEvent` function just returns `HResult.E_NOTIMPL` and ignores any events that are sent. The `ProcessMessage()` call just resets the Transform if a `NotifyStartOfStream` or `CommandFlush` message is received and ignores any other types. The events and message passing mechanism appears not to be very well used (or necessary) in Synchronous Mode Transforms.

The situation is considerably different in Asynchronous Mode Transforms and the Tanta Transform Asynchronous Base class necessarily makes extensive use of both events and messages. Asynchronous Mode Transforms are a pretty advanced concept and will not be discussed in this book. If you are interested, the `MFTTantaGrayscale_Async` Transform in the `TantaTransformsDirect` Sample Project does demonstrate an implementation of Asynchronous Transforms though.

RAW DATA HANDLING IN THE TRANSFORM

Ultimately the point of most Transforms is to modify or manipulate the incoming data.

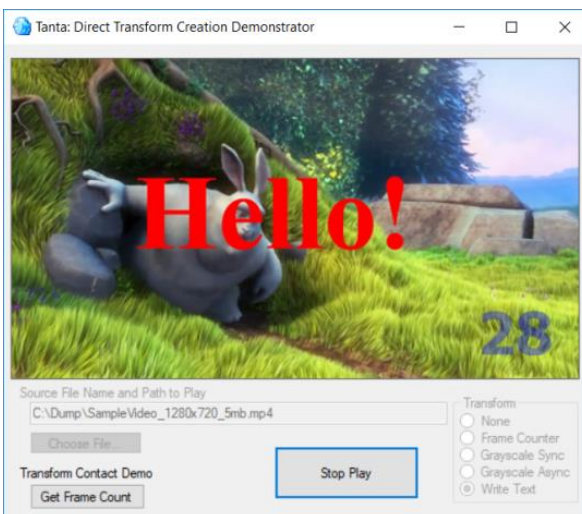


Figure 9.2: The TantaTransformDirect Application

However, it should be noted that there can be situations such in which the input data remains unchanged and the content is just examined.

There is insufficient space in this document to discuss the variety of methods used to process the raw media data in each of the Tanta Sample Transforms. We will however cover one particular case as a reference example and that, hopefully, will enable you to better interpret the other samples (and examples you may find on the Internet).

The Transform we will look at (`MFTTantaWriteText_Sync`), is designed to overwrite the video frame with two items of text. The first item of text, a simple string, is opaque and

Working With Transforms

the second, the frame count, is a steadily incrementing integer rendered on the display as a semi-transparent string (see Figure 9.2). The `MFTTantaWriteText_Sync` Transform can be seen in operation in the `TantaTransformDirect` Sample Project. Note that this Transform only accepts one Media Sub-type of RGB32, but the height and width of the format can be whatever the sender wishes.

We will join this Transform in the `OnProcessOutput()` call and the Media Sample object will have been previously set in the `InputSample` property by the Media Session in a previous `OnProcessInput()` call.

```
/// ++++++
/// <summary>
/// This is the routine that performs the transform. Assumes InputSample is set.
///
/// An override of the abstract version in TantaMFTBase_Sync.
/// </summary>
/// <param name="outputSampleDataStruct">The structure to populate with output data.</param>
/// <returns>S_OK unless error.</returns>
/// <history>
///     01 Nov 18   Cynic - Ported In
/// </history>
protected override HRESULT OnProcessOutput(ref MFTOutputDataBuffer outputSampleDataStruct)
{
    HRESULT hr = HRESULT.S_OK;
    IMFMediaBuffer outputMediaBuffer = null;

    // we are processing in place, the input sample is the output sample,
    // the media buffer of the input sample is the media buffer of the output sample.

    try
    {
        // Get the data buffer from the input sample.
        hr = InputSample.ConvertToContiguousBuffer(out outputMediaBuffer);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("ConvertToContiguousBuffer failed. Err=" + hr.ToString());
        }

        // now that we have an output buffer, do the work to write text on them.
        WriteTextOnBuffer(outputMediaBuffer);

        // Set status flags.
        outputSampleDataStruct.dwStatus = MFTOutputDataBufferFlags.None;

        // The output sample is the input sample. We get a new IUnknown for the Input
        // sample since we are going to release it below. The client will release this
        // new IUnknown
        outputSampleDataStruct.pSample = Marshal.GetIUnknownForObject(InputSample);
    }
    finally
    {
        // clean up
        SafeRelease(outputMediaBuffer);

        // Release the current input sample so we can get another one.
        // the act of setting it to null releases it because the property
        // is coded that way
        InputSample = null;
    }

    return HRESULT.S_OK;
}

Source: TantaTransformDirect::MFTTantaWriteText_Sync::OnProcessOutput
```

This `MFTTantaWriteText_Sync` Transform uses “in-place” processing. This means that the Media Buffer of the input sample is the Media Buffer of the output sample. As you

may recall from previous discussions this mode (if enabled) is set as a flag in the `OnGetInputStreamInfo()` call. The first action is to get the output Media Buffer.

```
// Get the data buffer from the input sample.
hr = InputSample.ConvertToContiguousBuffer(out outputMediaBuffer);
```

It is possible for a Media Sample to contain more than one Media Buffer if it does, the `ConvertToContiguousBuffer()` call copies the data from the original buffers into one new buffer. In typical use, most samples do not contain multiple buffers so the function does nothing and what we get is what we are really after – an `IMFMediaBuffer` object returned in the `outputMediaBuffer` variable.

We immediately call a function to perform the write of the text on the screen.

```
// now that we have an output buffer, do the work to write text on them.
WriteTextOnBuffer(outputMediaBuffer);
```

The `WriteTextOnBuffer()` function will be discussed below – but for now assume the operation has happened and let's look at the return and cleanup in the `OnProcessOutput()` call. Since the input Media Sample was modified in-place, simply assigning the input Media Sample to the output return structure is sufficient.

```
// The output sample is the input sample. We get a new IUnknown for the Input
// sample since we are going to release it below. The client will release this
// new IUnknown
outputSampleDataStruct.pSample = Marshal.GetIUnknownForObject(InputSample);
```

Note that the base class is going to release the `InputSample` object no matter what we do (this operation is hard coded into it) and the caller is also going to want to release this Media Sample. We carefully acquire another reference (the `Marshal.GetIUnknownForObject()` wrapper) and pass that back in order to make sure everything that expects to release the object can do so with no complications. Not doing this will cause a lot of hard to debug problems and errors in the Pipeline. Once we get past that, the final steps in the `OnProcessOutput()` call are pretty standard operations.

```
// clean up
SafeRelease(outputMediaBuffer);

// Release the current input sample so we can get another one.
// the act of setting it to null releases it because the property
// is coded that way
InputSample = null;
```

In particular, note that since we obtained the Media Buffer from WMF we have to release it. The act of setting the `InputSample` to null will also release it since the property is hard coded to do that.

Now let's turn our attention to the operations in the `WriteTextOnBuffer` function. This function receives a Media Buffer as a parameter.

Working With Transforms

```
/// ++++++
/// <summary>
/// Write the text on the output buffer
/// </summary>
/// <param name="outputMediaBuffer">Output buffer</param>
/// <history>
/// 01 Nov 18 Cynic - Ported In
/// </history>
private void WriteTextOnBuffer(IMFMediaBuffer outputMediaBuffer)
{
    IntPtr destRawDataPtr = IntPtr.Zero; //Destination buffer.
    int destStride=0; // Destination stride.
    bool destIs2D = false;

    try
    {
        // Lock the output buffer. Use the IMF2DBuffer interface
        // (if available) as it is faster
        if ((outputMediaBuffer is IMF2DBuffer) == false)
        {
            // not an IMF2DBuffer - get the raw data from the IMFMediaBuffer
            int maxLen =0;
            int currentLen = 0;
            TantaWMFUtils.LockIMFMediaBufferAndGetRawData(outputMediaBuffer,
                out destRawDataPtr, out maxLen, out currentLen);
            // the stride is always this. The Lock function does not return it
            destStride = m_lStrideIfContiguous;
        }
        else
        {
            // we are an IMF2DBuffer, we get the stride here as well
            TantaWMFUtils.LockIMF2DBufferAndGetRawData((outputMediaBuffer as IMF2DBuffer),
                out destRawDataPtr, out destStride);
            destIs2D = true;
        }

        // count this now. We only use this to write it on the screen
        m_FrameCount++;

        // We could eventually offer the ability to write on other formats depending on the
        // current media type. We have this hardcoded to ARGB for now
        WriteImageOfTypeARGB( destRawDataPtr,
            destStride,
            m_imageWidthInPixels,
            m_imageHeightInPixels);

        // Set the data size on the output buffer. It probably is already there
        // since the output buffer is the input buffer
        HRESULT hr = outputMediaBuffer.SetCurrentLength(m_cbImageSize);
        if (hr != HRESULT.S_OK)
        {
            throw new Exception("call to SetCurrentLength failed. Err=" + hr.ToString());
        }
    }
    finally
    {
        // we MUST unlock
        if(destIs2D == false) TantaWMFUtils.UnlockIMFMediaBuffer(outputMediaBuffer);
        else TantaWMFUtils.UnlockIMF2DBuffer((outputMediaBuffer as IMF2DBuffer));
    }
}

Source: TantaTransformDirect:MFTTantaWriteText_Sync::WriteTextOnBuffer
```

As you will recall from previous sections, a Media Buffer can be either an IMFMediaBuffer or an IMF2DBuffer. We need to get access to the raw data (an IntPtr) and the only way to do that is to Lock() the buffer. The call is slightly different depending on the buffer type. Let's take a look at the IMFMediaBuffer case.

```
int maxLen =0;
int currentLen = 0;
TantaWMFUtils.LockIMFMediaBufferAndGetRawData(outputMediaBuffer,
    out destRawDataPtr, out maxLen, out currentLen);
// the stride is always this. The Lock function does not return it
destStride = m_lStrideIfContiguous;
```


The main thing we are interested in here is the `destRawDataPtr` value. The other variables we will use to process the video frame are all derived from the input `Media Type` value originally set in the `OnSetInputType()` call way back when the `Topology` was resolved. The `IMF2DBuffer` case is similar except that it will return a stride value to us.

WRITING TEXT ON A VIDEO FRAME

Continuing on from the discussion in the previous section, a call to the `WriteImageOfTypeARGB()` function is next. This function takes the `IntPtr` of the data and the width and height of the frame. These too were set in the `OnSetInputType()` call.

```
WriteImageOfTypeARGB( destRawDataPtr,
                      destStride,
                      m_imageWidthInPixels,
                      m_imageHeightInPixels);
```

When the `WriteImageOfTypeARGB()` call returns we have to set the length in the output `Media Buffer` – although this step is probably redundant. We are doing in-place processing, the value is probably already there, and it will not have changed.

```
// Set the data size on the output buffer.
HRESULT hr = outputMediaBuffer.SetCurrentLength(m_cbImageSize);
```

The only remaining operation in the `WriteTextOnBuffer` function is to perform the unlock of the `Media Buffer`.

```
finally
{
    // we MUST unlock
    if(destIs2D == false) TantaWMFUtils.UnlockIMFMediaBuffer(outputMediaBuffer);
    else TantaWMFUtils.UnlockIMF2DBuffer((outputMediaBuffer as IMF2DBuffer));
}
```

The unlock of the Media Buffer after you have locked it is critically important. You must do this.

Also note that the type of unlock call differs depending on whether the `Media Buffer` was originally locked as an `IMFMediaBuffer` or `IMF2DBuffer`.

Now we get to the heart of the matter – the write of the text on the video frame. It actually is surprisingly simple because we use a bit of C# trickery. The call to `WriteImageOfTypeARGB` performs this operation.

```
/// ++++++
/// <summary>
/// Write Text on an ARGB formatted image
/// </summary>
/// <param name="pDest">Pointer to the destination buffer.</param>
/// <param name="lDestStride">Stride of the destination buffer, in bytes.</param>
/// <param name="dwWidthInPixels">Frame width in pixels.</param>
/// <param name="dwHeightInPixels">Frame height, in pixels.</param>
/// <history>
/// 01 Nov 18  Cynic - Ported In
```

Working With Transforms

```
/// </history>
private void WriteImageOfTypeARGB(
    IntPtr pDest,
    int lDestStride,
    int dwWidthInPixels,
    int dwHeightInPixels
)
{
    // Although the actual data is down in unmanaged memory
    // we do not need to use "unsafe" access to get at it.
    // The new Bitmap call does this for us. This is probably
    // only useful in this sort of rare circumstance. Normally
    // you have to copy it about. See the MFTTantaGrayscale_Sync code.

    // The strings to display.
    string sString1 = "Hello!";
    string sString2 = m_FrameCount.ToString();

    // A wrapper around the video data.
    using (Bitmap v = new Bitmap(m_imageWidthInPixels,
        m_imageHeightInPixels, m_lStrideIfContiguous, PixelFormat.Format32bppRgb, pDest))
    {
        using (Graphics g = Graphics.FromImage(v))
        {
            float sLeft;
            float sTop;
            SizeF d;

            // String1 goes right in the middle of the video using
            // the overlay font created earlier
            d = g.MeasureString(sString1, m_fontOverlay);

            sLeft = (m_imageWidthInPixels - d.Width) / 2.0f;
            sTop = (m_imageHeightInPixels - d.Height) / 2.0f;

            g.DrawString(sString1, m_fontOverlay, System.Drawing.Brushes.Red,
                sLeft, sTop, StringFormat.GenericTypographic);

            // Add a frame number in the bottom right using the
            // transparent font created earlier
            d = g.MeasureString(sString2, m_transparentFont);

            sLeft = (m_imageWidthInPixels - d.Width) - 10.0f;
            sTop = (m_imageHeightInPixels - d.Height) - 10.0f;

            g.DrawString(sString2, m_transparentFont, m_transparentBrush,
                sLeft, sTop, StringFormat.GenericTypographic);
        }
    }
}

Source: TantaTransformDirect::MFTTantaWriteText_Sync::WriteImageOfTypeARGB
```

There are a couple of time (and code) saving things happening in the above function. We create a C# bitmap from the incoming IntPtr to the raw data. This incorporates its own Marshal operation and brings the information nicely up into .NET's managed memory space.

```
using (Bitmap v = new Bitmap(m_imageWidthInPixels,
    m_imageHeightInPixels, m_lStrideIfContiguous, PixelFormat.Format32bppRgb, pDest))
```

Once we have the data as a bitmap we have a wonderful array of C# tools at our disposal. For example, we obtain a `Graphics` object from it.

```
using (Graphics g = Graphics.FromImage(v))
```

Now that we have the buffer data in the form of a `Graphics` object we can draw on it as if it were any form or control display area. This is very useful since, ultimately as we write the text, we would prefer not to be fiddling around turning pixels on and off. We

are also going to want to be able select the Typeface and size of the text we write – this means using fonts.

Font object creation has a certain amount of overhead and since they are re-useable there is no need to create the fonts each time we process a frame. Accordingly we also created the fonts during the much earlier call to the `OnSetInputType()` function. We will take a bit of a digression to show this creation.

```
// create the font
m_fontOverlay = new Font(
    "Times New Roman",
    fSize,
    System.Drawing.FontStyle.Bold,
    System.Drawing.GraphicsUnit.Point);

// now the transparent font for the frame count in the
// bottom right hand corner
// scale the font size in some portion to the video image
fSize = 5;
fSize *= (m_imageWidthInPixels / 64.0f);

if (m_transparentFont != null) m_transparentFont.Dispose();

m_transparentFont = new Font(
    "Tahoma",
    fSize,
    System.Drawing.FontStyle.Bold,
    System.Drawing.GraphicsUnit.Point);

Source: TantaTransformDirect::MFTTantaWriteText_Sync::OnSetInputType
```

In particular, note that the size and other font characteristics (but not the transparency or color) are set at this time.

Returning to the `WriteImageOfTypeARGB` function, we can see that the actual write of the text on the screen is somewhat of an anti-climax.

```
// String1 goes right in the middle of the video using the overlay font created earlier
d = g.MeasureString(sString1, m_fontOverlay);

sLeft = (m_imageWidthInPixels - d.Width) / 2.0f;
sTop = (m_imageHeightInPixels - d.Height) / 2.0f;

g.DrawString(sString1, m_fontOverlay, System.Drawing.Brushes.Red,
    sLeft, sTop, StringFormat.GenericTypographic);
```

Besides a bit of math to set the position of the write operation the `DrawString()` function of the `Graphics` object does all the work for us. The draw of the transparent string is similarly simple – note the use of the transparent font and a transparent brush created earlier.

```
// Add a frame number in the bottom right using the transparent font created earlier
d = g.MeasureString(sString2, m_transparentFont);

sLeft = (m_imageWidthInPixels - d.Width) - 10.0f;
sTop = (m_imageHeightInPixels - d.Height) - 10.0f;

g.DrawString(sString2, m_transparentFont, m_transparentBrush, sLeft,
    sTop, StringFormat.GenericTypographic);
```

Both the bitmap and the graphics object were created with the `C# using` construct. This means that they both get properly disposed by the time the function exits.

It should also be noted that the first string written was hardcoded to a value of “Hello!”. This value could of course be dynamically set in the Transform using any of the techniques described in the *Passing Information In and Out of a Transform* section of this chapter. The second string, rendered in a transparent font, is just the frame count of the particular video image on the screen. The `OnProcessOutput()` call also collects these.

ADDING TRANSFORMS TO THE PIPELINE

The Topology is a representation (a map) of the way the data will flow from the Media Sources to the Media Sinks. A Pipeline is the instantiated version of the Topology.

Transforms can be automatically added to a Topology in certain cases. This typically happens in playback applications where a WMF entity called a Topology Loader will be invoked to automatically look for and add various decompression and format conversion Transforms to the Pipeline in order to make it possible to successfully render video and sound. This automatic Topology resolution mechanism only operates on Pipelines implementing a renderer sink (the EVR or SAR) and is not available for other applications, such as saving video to a file on disk. In such cases you, as the programmer, will have to provide the code to find and include the required Transforms.

It should be noted that the use of the Topology Loader is not exclusive. It is possible to add your own Transforms to the Pipeline even if the Topology is automatically resolved. This is demonstrated in the *TantaTransformDirect* and *TantaTransformInDLLClient* Sample Projects. In such situations, you can just kind of throw your Transform into the Topology along with the Media Source and Media Sink, roughly join them all up in the order you wish and the Topology Loader will figure out what needs to be done in order to make it all work. More formally, this means that the Topology Loader will examine the Media Types offered by the source and sinks, the supported Media Types on the Transforms you added and the conversion and decompression Transforms available on the system. Once the Topology Loader has an idea of what resources are available, it will connect everything up, adding new Transforms as necessary, and will make a functioning Pipeline out of them - if it can (it is not omnipotent).

The *Transforms* section in *The WMF Components* chapter of this book has an expanded discussion of the Topology Loader and we will only consider the general case in the discussions that follow.

It is often necessary to explicitly specify and manually add Transforms to a Topology. As you have by now come to expect with WMF, there are a number of equivalent ways to

do this. Fundamentally though, the process of adding a Transform to your Pipeline divides into two situations: either you have the Transform source code contained within the C# application you are writing or you have a Transform installed in the system registry and you have to find and load it via COM and Windows Media Foundation Tools. If the Transform is located in the registry it may or may not be written in a managed .NET language - C++ is the typical language for most Transforms. Of course, if the Transform is directly included in the C# solution (either as class in the application project or in a separate project entirely) it will have been written in C#.

Actually, though, it does not matter if the Transform you wish to use is written in a different language. The `IMFTransform` interface will always behave the same and the MF.Net library will provide the tools to find and load it into the Pipeline.

It also does not matter if the Transform is Synchronous or Asynchronous and the two types of Transform can be mixed in a Pipeline. The Media Session will automatically deal with the different interactions required and you, as the programmer of the application, need not concern yourself with that issue. Of far more interest as you build the Topology are the Media Type configuration (both Media Major and Media Sub-Types), the Media Sources and Media Sinks and the number of streams and branches the Topology will have to accommodate.

Each section below summarizes a method of creating a Topology Node which represents a Transform and the final section demonstrates the procedure for connecting that node into the Topology. To set the stage, in all examples below, the Topology object and the Media Session will have been created, the Media Source will have been chosen and a Topology Node created for it. Since the examples are a video playback application and we are placing our Transform in the video branch of the Pipeline, the EVR video renderer will be used as the Media Sink. A Topology Node will have been created for the Media Sink as well. It is useful to keep in mind that all of these configurations are being applied on the Topology branch associated with the `MFMediaType.Video` Media Major Type stream – audio would be handled on a separate branch.

ADDING A TRANSFORM WHEN YOU HAVE THE SOURCE CODE

If you have the source code for the Transform included within the application source, the procedure for creating a Topology Node for it is simple. All that is necessary is to compile the Transform object up, instantiate it with the C# `new` operator and give it to the topology as a binary via a `SetObject()` call on a Transform Node. In the code below the `VideoTransform` variable holds the Transform object and this is known not to be null.

Working With Transforms

```
IMFTopologyNode videoTransformNode = null;

// Create a video Transform
hr = MFExtern.MFCreateTopologyNode(MFTopologyType.TransformNode, out videoTransformNode);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to MFCreateTopologyNode failed. Err=" + hr.ToString());
}
if (videoTransformNode == null)
{
    throw new Exception("call to MFCreateTopologyNode(t) failed. videoTransformNode == null");
}

// set the transform object (it is an IMFTransform) as an object
// on the transform node.
hr = videoTransformNode.SetObject(VideoTransform);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to videoTransformNode.SetObject failed. Err=" + hr.ToString());
}

Source: TantaTransformDirect::frmMain::PrepareSessionAndTopology
```

In particular, note that the `MFTopologyType.TransformNode` parameter was specified in the call to the static `MFCreateTopologyNode()` function which creates the Topology Node. There are four types of Topology Node and if you are creating one for Transforms you will need to take care to get this right. The *Topologies* section in *The WMF Components* chapter has more details on this topic.

ADDING A TRANSFORM WHEN YOU HAVE AN ACTIVATOR

If you have the activator object for the Transform (perhaps because you enumerated the available Transforms on the system) you can simply give the Activator to the Topology by creating a Transform Node and calling `SetObject()` with the activator object as a parameter.

```
IMFTopologyNode pTransformNode = null;

// We have an activator object supplied from some other source
IMFActivator pTransformActivator = <supplied from elsewhere>;

// Create the transform node.
hr = MFExtern.MFCreateTopologyNode(MFTopologyType.TransformNode, out pTransformNode);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to MFExtern.MFCreateTopologyNode failed. Err=" + hr.ToString());
}

// set the transform activator on the transform node. The topology will see
// that it is an activator and will create the transform object from it
hr = pTransformNode.SetObject(pTransformActivator);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to pTransformNode.SetObject failed. Err=" + hr.ToString());
}

// Add the transform node to the topology.
hr = pTopology.AddNode(pTransformNode);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to pTopology.AddNode failed. Err=" + hr.ToString());
}

Source: Not in Tanta Sample Source
```

The source code is exactly the same as the previous direct mode example code except that an `IMFActivator` object is passed in on the `SetObject()` call instead of an `IMFTransform`.

ADDING A TRANSFORM WHEN YOU HAVE A KNOWN GUID

If you know the GUID of a Transform registered on your system you can create a Transform Node for it and configure it with a call to `SetGUID()`. The Transform object will be instantiated when the Topology is resolved.

```
IMFTopologyNode tmpTransformNode = null;

// Create the transform node.
hr = MFExtern.MFCreateTopologyNode(MFTopologyType.TransformNode, out tmpTransformNode);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to MFExtern.MFCreateTopologyNode failed. Err=" + hr.ToString());
}

// set the transform Guid on the transform node. Since this is an attribute we also
// have to tell it what the guid means - hence the MF_TOPONODE_TRANSFORM_OBJECTID as a key
hr = tmpTransformNode.SetGUID(MFAttributesClsid.MF_TOPONODE_TRANSFORM_OBJECTID, transformGuid);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to tmpTransformNode.SetGUID failed. Err=" + hr.ToString());
}

// Add the transform node to the topology.
hr = pTopology.AddNode(tmpTransformNode);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to pTopology.AddNode failed. Err=" + hr.ToString());
}

// also save the node here
VideoTransformNode = tmpTransformNode;

Source: TantaCommon::ctlTantaEVRFilePlayer::AddBranchToPartialTopology
```

Note that the Transform GUID supplied to the Transform Node is the value of an Attribute. This means that we also have to tell it what that particular GUID means. The `MF_TOPONODE_TRANSFORM_OBJECTID` value (itself a GUID) is the expected key to use for this purpose.

ADDING A TRANSFORM BY CREATING IT FROM A GUID

If you know the GUID of a Transform registered on your system. You can also instantiate it yourself via COM with a call to `CoCreateInstance()` and then give the object to the Topology by creating a Topology Node for it and calling `SetObject()` with the newly instantiated Transform as the parameter.

```
/// ++++++
/// <summary>
/// Gets a transform object from a Guid
///
/// </summary>
/// <param name="transformGuid">the guid of the transform</param>
/// <param name="wantLocalServer">if true use CLSCTX_LOCAL_SERVER otherwise
/// CLSCTX_INPROC_SERVER</param>
/// <returns>the transform object - this must be released</returns>
/// <history>
```

Working With Transforms

```
/// 01 Nov 18 Cynic - Originally Written
/// </history>
public static IMFTransform GetTransformFromGuid(Guid transformGuid, bool wantLocalServer)
{
    object retInstance = null;
    IMFTransform transformObj = null;

    try
    {
        // set up for INPROC or LOCAL server
        uint serverType = CLSCTX_INPROC_SERVER;
        if (wantLocalServer == true) serverType = CLSCTX_LOCAL_SERVER;

        // call COM and create and instance from the Guid
        UInt32 hResult = CoCreateInstance(transformGuid,
                                          IntPtr.Zero,
                                          serverType,
                                          typeof(IMFTransform).GUID,
                                          out retInstance);

        if (hResult != 0) return null;
        if (retInstance == null) return null;
        transformObj = (IMFTransform)retInstance;
    }
    catch
    {
    }
    finally
    {
    }
    return transformObj;
}

Source: TantaCommon::TantaWMFUtils::GetTransformFromGuid
```

Once you have the Transform object, the process of adding it to the Topology is identical to that in the first example in this section (*Adding a Transform When You have the Source Code*) where the Transform was created with the C# `new` operator. All that is necessary is to use the `SetObject()` call on the Transform Node with the instantiated Transform. Accordingly, the above code shows the much more interesting process of getting an instantiated Transform from the GUID via a `CoCreateInstance()` call and the *Adding a Transform When You have the Source Code* section can be referred to for an example of how to add it to the Topology. You can find the `GetTransformFromGuid` function in the `TantaWMFUtils` class of the `TantaCommon` sample code.

CONNECTING TRANSFORM NODES

In the above sections, a Topology Node representing the Transform was created. This node will have either the Transform object itself, an Activator for the Transform or a GUID of the Transform in the registry. We now discuss how the newly created Transform Nodes can be connected inside the Topology – the procedure is the same no matter how the Topology Node was created. In the code below, the Topology Node for the Media Source (`sourceVideoNode`) and the Topology Node for the Media Sink (`outputSinkNodeVideo`) have already been created and added to the Topology.

```
// Do we have a Transform Node?
if (VideoTransform != null)
{
    // yes we do, inject the transform node into the topology
    // first source node to transform node
    hr = sourceVideoNode.ConnectOutput(0, videoTransformNode, 0);
}
```



```

    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to ConnectOutput(t1) failed. Err=" + hr.ToString());
    }
    // now transform node to sink node
    hr = videoTransformNode.ConnectOutput(0, outputSinkNodeVideo, 0);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to ConnectOutput(t2) failed. Err=" + hr.ToString());
    }
}

```

Source: *TantaTransformDirect::frmMain::PrepareSessionAndTopology*

In general, the Source Node is connected to the Transform node and the Transform Node is connected to the Output Node. In reality, the process is just a logical extension of how the Topology Nodes would be connected in the absence of a Transform Node as is shown below.

```

else
{
    // no we do not have a Transform Node, just connect the source to the sink directly
    hr = sourceVideoNode.ConnectOutput(0, outputSinkNodeVideo, 0);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("call to ConnectOutput failed. Err=" + hr.ToString());
    }
}

```

Source: *TantaTransformDirect::frmMain::PrepareSessionAndTopology*

As noted previously, in playback Pipelines, the Topology Loader may well insert other Transforms in between these nodes in order to make the Media Types used on the inputs and outputs of each component in the Pipeline work.

MAKING A TRANSFORM AVAILABLE

It is possible to include a Transform directly in your code (see the *TantaTransformDirect* Sample Project) - however, this method is not the one typically used. The more common method of distributing a Transform and making it available is to create it as a separate standalone entity (a DLL) and configure it in the registry so that other applications can find and use it.

A globally unique GUID key value created by the implementer of the Transform is used to specify the Transform in the registry. Any applications which know the GUID key can then find the Transform and use it. Of course, this presents a bit of a problem to applications that might need the functionality the Transform provides, but which do not know the specific GUID value the Transform is registered under. The idea that it could be very useful to have the ability to find all of the Transforms in the registry by category and function was not lost on the implementers of Windows Media Foundation. In fact, there is a special static function in WMF named *MFTEnumEx* specifically intended for this purpose and the *TantaTransformPicker* Sample Project demonstrates its use.

There are actually two registrations that need to happen in order to make an MFT both available in the registry and also discoverable by other WMF applications. The Transform must first be registered as COM object to make it available and it can also, optionally, be configured in the registry to make it discoverable by other WMF applications.

You do not necessarily need to make the MFT discoverable if the applications which will use the Transform already know the GUID of the DLL. This is the approach taken in the *TantaTransformInDLL* and *TantaTransformInDLLClient* pair of samples. Having said that, for demonstration purposes, the *TantaTransformInDLL* code actually does auto-register the Transform DLL so that it can be discovered on the system. The *TantaTransformInDLLClient* example does not need to perform the discovery operation since it already knows the GUID to use because that value is hardcoded into it at compile time.

COM INTEROP DECORATIONS

If a Transform is to be generally available to Windows Media Foundation applications on a system, it must be compiled as a DLL and that DLL must be registered as a COM object. In Visual Studio there is nothing particularly special about the structure of the source code of a Transform used as a DLL. However, there are some options which must be applied at the project level and some which must be applied to the source code in order for the DLL to be useable as a Transform.

Besides adhering to the *IMFTransform* interface (it *has* to do that), the source code of a Transform to be used as a DLL should have a number of C# decorations applied to the class and optionally to some of the functions and properties.

An example of a C# function decoration is shown below...

```
/// ++++++
/// <summary>
/// Get the current frame count and prepend a user supplied string. The
/// output is a string in a ref variable.
///
/// Note how this function is ComVisible. This
/// function can be used by a .NET client via Reflection and Late Binding
/// to interact with the transform.
/// </summary>
/// <param name="frameCountLeadingText">the leading text to prepend. Cannot be null</param>
/// <param name="outString">the string with the framecount is returned here</param>
/// <returns>true the operation was successful, false it was not</returns>
/// <history>
/// 01 Nov 18 Cynic - Originally Written
/// </history>
[ComVisible(true)]
public bool FrameCountAsFunctionDemonstrator(string frameCountLeadingText, ref string outString)
```

```

{
    // we say the leading text cannot be null
    if (frameCountLeadingText == null)
    {
        outString = "";
        return false;
    }
    // set up the string
    outString = frameCountLeadingText + m_FrameCount.ToString();
    return true;
}

```

Source: TantaTransformInDLL:MFTTantaVideoRotator_Sync::FrameCountAsFunctionDemonstrator

In the above source code, the `[ComVisible(true)]` decoration ensures the function is visible to other applications which have loaded it via COM. It must be noted that this is only true provided that some other things (discussed below) also happen. Normally only the functions which will be called by the application using the Transform will need to be made COM visible.

It should be noted that you will also (in other sources) sometimes see these decorations referred to as *“attributes”*. The use of the label *“attributes”* in this situation does not refer to Windows Media Foundation Attributes – although there will be plenty of those in use in a Transform as well. The dual use is a rather unfortunate naming collision - the meaning of which you have to just figure out from the context. This book will always refer to C# class and function *“attributes”* as *“decorations”* in order to avoid confusion with WMF Attribute key-value pairs.

The primary thing you need to do in order to make your Transform (or any DLL) source code suitable for registration in COM Interop is to apply `ComVisible` and `Guid` decorations to the beginning of your class.

```

/// These class decorations are important. This is the GUID under which the MFT
/// will be registered. If you copy this code you should change this.
/// You should also probably change the class name. This will appear in the
/// registry as well. Use the TantaTransformPicker sample application to view
/// the MFT's in the registry.
[ComVisible(true),
Guid("F1E67619-FB5B-470B-9306-EBF40D54985E"),
ClassInterface(ClassInterfaceType.None)]
public sealed class MFTTantaVideoRotator_Sync : TantaMFTBaseStandalone_Sync
{
    // Format information
    private int m_imageWidthInPixels;
    ....
}

```

Source: TantaTransformInDLL:MFTTantaVideoRotator_Sync

In the above code block, there are three decorations on the `MFTTantaVideoRotator_Sync` class. The first, `[ComVisible(true)]` makes the class visible when loaded via COM Interop. The second, `Guid("F1E67619-FB5B-470B-9306-EBF40D54985E")`, is the unique GUID value by which this Transform will be known in the registry. The third option, `[ClassInterface(ClassInterfaceType.None)]`, is a decoration which indicates the type of class interface generated for the class when it is registered for COM Interop.

A decoration similar to ...

```
[ComVisible(true),  
Guid("F1E67619-FB5B-470B-9306-EBF40D54985E"),  
ClassInterface(ClassInterfaceType.None)]
```

... on your class source code is necessary if your Transform is to be registered for COM Interop.

You must create a new GUID for your Transform – this is **very** important. It is also a good idea to give the Transform a new class name. This class name, among other things, will appear in the enumeration list if the object is also configured to be discoverable by other Windows Media Foundation applications.

If you are implementing your own Transform, make sure to generate a new GUID for it - this is easily done using the Create GUID option in the Tools Menu of Visual Studio.

The C# class decorations discussed above just make it possible to register the Transform for COM Interop – the fact that they exist does not automatically register it. The actual registration can be performed in one of several ways.

In order to register a Transform for COM Interop you can...

1. You can register the DLL manually through the use of the Windows `RegAsm.exe` utility.
2. You can tick the "Register for COM Interop" option on the Build tab of the properties of your Visual Studio project and it will auto-register each time you compile.
3. If you wrap the DLL in an installer you can use the tools provided by that installation package.

REGISTERING A TRANSFORM MANUALLY

Manual registration of a Transform DLL for COM Interop is performed by running the `RegAsm.exe` utility on the command line against the assembly DLL. It should be noted that, in order to update the registry, you will have to start the command line object as Administrator or the account you use will have to already have such privileges.

The RegAsm utility is a .NET utility (it is not part of the standard Windows distribution) and so the location of the `RegAsm.exe` file will vary depending on the .NET version you are using. You will probably have to look for it.

The actual command line for `RegAsm.exe` is not complex. The code below will unregister the `MFTTantaVideoRotator_Sync` Transform - if it is present. Remember the path to the RegAsm utility is specific to the .NET version – you will probably need to adjust this.

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\regasm
C:\Projects\Tanta\TantaTransformInDLL\bin\Debug\MFTTantaVideoRotator_Sync.dll /u
Source: TantaTransformInDLL:ManualRegister.txt
```

Registering the Transform is similarly simple. The code below registers the `MFTTantaVideoRotator_Sync` DLL and uses the `/tlb` and `/codebase` options. You can look these options up in the online help to find out what they do.

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\regasm
C:\Projects\Tanta\TantaTransformInDLL\bin\Debug\MFTTantaVideoRotator_Sync.dll
/tlb:MFTTantaVideoRotator_Sync.tlb /codebase
Source: TantaTransformInDLL:ManualRegister.txt
```

REGISTERING A TRANSFORM DURING COMPILATION

It is possible to ensure your Transform DLL is automatically registered each time you compile it. Obviously this is only useful during development - if you were shipping the Transform assembly as a binary either you, or the end user, would have to use some other method of registration.

Automatic COM registration at compile time is achieved by ticking the "*Register for COM Interop*" option on the Build tab of the properties of the project. As with the manual registration option, you will need Administrator privileges in order to register the DLL. This is why you need to start the *TantaTransformInDLL* solution as Administrator - otherwise you will not have permission to update the registry. A screenshot of the "*Register for COM Interop*" option is shown in Figure 9.3.

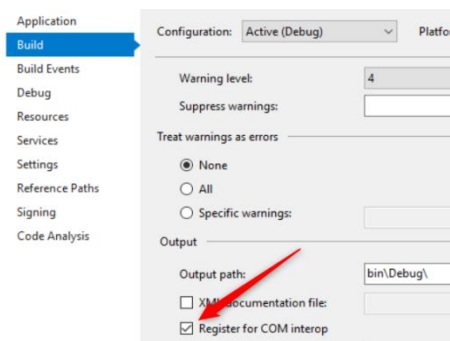


Figure 9.3: The Build Property of the *TantaTransformInDLL* Sample Application

Note that if you untick the "*Register for COM Interop*" option you will not need to be an Administrator and you will still get a DLL but it will not be registered for COM.

It should be noted that all Visual Studio is really doing here is calling `RegAsm.exe` on your behalf. It has exactly the same effect as a manual registration with `RegAsm.exe` using the `/tlb` and `/codebase` options.

REGISTERING A TRANSFORM VIA AN INSTALLER APPLICATION

There are many installation applications available and a discussion of them is outside the scope of this book. The registration of a Transform DLL for COM Interop via this method will not be discussed here. However, it should be noted that many people frown on the idea of shipping a DLL without an installer – there is a strong feeling that in order to provide simple user access and to get a consistent registration (and un-registration) a DLL should always be wrapped in installation software of some form.

MAKING A TRANSFORM DISCOVERABLE

Previously in the *Making a Transform Available* section, it was noted that two registrations must be performed in order to make a Transform available via COM Interop and also discoverable by other Windows Media Foundation applications. COM Interop registration has been discussed in the sections above. We will now turn our attention to the mechanism by which a Transform DLL can be configured in the registry in order to make it discoverable (or *Enumerable* – to use the technical term) by other WMF applications on the system.

In reality, the act of making a Transform assembly DLL Enumerable is also intimately tied to the use of the `RegAsm` utility. The designers of the WMF architecture noted that a Transform is always written for a specific purpose. In other words, any one Transform is a Decoder, an Encoder, a Video Effect or something else – but never really more than one thing. Since the purpose of the Transform is always known at compile time by the developer of the Transform there is no reason not to provide the ability for the Transform self-register itself for Enumeration. The thinking also went that since any Transform has to be registered for COM Interop, why not make the self-registration happen at that time as well? Thus it turns out that, if certain things are appropriately configured by the developer in the Transform DLL, a Transform can automatically make itself discoverable by other WMF applications at the same time as it is registered for COM Interop. Similarly it can make itself un-enumerable when un-registered for COM Interop.

Theoretically, how might a Transform render itself enumerable? Well, there are two possible ways – either the `RegAsm` utility could read some sort of standard information inside the DLL and configure things on behalf of the DLL or the `RegAsm` utility can

actually call a function inside the DLL and then the DLL itself can take whatever actions it needs to in order to configure itself. The designers of the Windows Media Foundation architecture decided to take the latter path – presumably because that sort of “*call-into-the-dll*” functionality already existed as part of COM Interop and they would not have to implement anything new.

It turns out that if you have a function in the DLL which returns `void` and accepts one parameter of type `Type` and you also decorate that function with the C# `[ComRegisterFunctionAttribute]` tag, then the RegAsm utility will execute that function when it registers the DLL for COM Interop. A similar function, decorated with the C# `[ComUnregisterFunctionAttribute]` tag, will be called when the DLL is unregistered for COM Interop. Note this is true execution – RegAsm brings the DLL into memory and executes those functions as part of its process. The code in those functions will be called exactly as if you had opened the DLL in a C# application and directly called the function. This also means that the contents of those functions will execute with whatever permissions the RegAsm exe is using – presumably Administrator rights. System access in these functions is not strictly limited to modifying the registry – it is entirely possible to manipulate files and perform other actions.

In the Tanta Sample Code and in most other Transforms, the register and un-register functions are given the standard names of `DllRegisterServer` and `DllUnregisterServer` – but that is just convention, in reality these names could be anything. The contents of the `DllRegisterServer` and `DllUnregisterServer` functions from the *TantaTransformInDLL* sample are shown in the code section below. In particular, note the use of the `[ComRegisterFunctionAttribute]` and `[ComUnregisterFunctionAttribute]` decorations.

```
//+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+  
/// <summary>  
/// Set up the function which will be automatically called by COM when this  
/// DLL is registered for COM interop. We use this call to register the MFT  
/// and make it available for discovery by other MFT applications.  
/// </summary>  
/// <history>  
///     01 Nov 18   Cynic - Ported In  
/// </history>  
[ComRegisterFunctionAttribute]  
static private void DllRegisterServer(Type t)  
{  
  
    HRESULT hr = MFExtern.MFTRegister(  
        t.GUID,  
        MFTTransformCategory.MFT_CATEGORY_VIDEO_EFFECT,  
        t.Name,  
        MFT_EnumFlag.SyncMFT,  
        0,  
        null,  
        0,  
        null,  
        null  
    );  
  
    MFEError.ThrowExceptionForHR(hr);  
}
```

Working With Transforms

```
/// ++++++
/// <summary>
/// Set up the function which will be automatically called by COM when this
/// DLL is registered for COM interop. We use this call to deregister the MFT
/// and make it unavailable for discovery by other MFT applications.
/// </summary>
/// <history>
///     01 Nov 18   Cynic - Ported In
/// </history>
[ComUnregisterFunctionAttribute]
static private void DllUnregisterServer(Type t)
{
    HRESULT hr = MFExtern.MFTUnregister(t.GUID);
}
Source: TantaTransformInDLL: :MFTTantaVideoRotator_Sync: :DLLRegisterServer
```

Ultimately, it is the `MFTRegister()` and `MFTUnregister()` calls doing the actual work and these static functions are part of the Windows Media Foundation library. Note how the GUID, Transform Category and type of Transform (Synchronous and Asynchronous) are among the items of information passed in on the `MFTRegister()` call. These parameters will be associated with the Transform in the registry and can be used by MFT applications to dynamically sift suitable Transforms from the sometimes rather lengthy list they enumerate.

Once it is complete, a properly registered Transform should be discoverable by other MFT applications. You can use the *TantaTransformPicker* sample application to test this. If you compile the *TantaTransformInDLL* Sample Project (remember you have to be Administrator) you should be able to see the `MFTTantaVideoRotator_Sync` Transform listed under the Video Effect category.

Remember there is also an unregister mode as well. If you compile *TantaTransformPicker* sample application (as Administrator) the `DLLUnregisterServer` function will be called first and, when that returns, only then will the `DLLRegisterServer()` call will be made. It should also be noted that successful registration of the DLL either for COM Interop or for WMF Enumeration does not mean that the Transform is without errors or even properly coded. Neither process checks to see if the class being registered even bothers to implement the `IMFTTransform` interface.

LOCAL DISCOVERABILITY

The `MFTRegister` function in the previous section makes the Transform enumerable to any application on the system. What about cases in which you want the Transform to only be “discoverable” by the current process? This is supported, and more on that in a moment, but first let’s discuss why you might want a Transform to only be visible and enumerable by the current process.

There are occasions, where you want the Transform discovery process to function normally but do not wish to make any changes which will affect the operation of other

applications on the system. One example of this is on Windows 7. Windows 7 provides quite a number of conversion Transforms (codecs) of which, very few are actually enumerable by default. In other words, your application can use these codecs all it wishes, but it has to specifically request them. This works well in situations like the Pipeline Architecture where you can specify a Transform by GUID in order to add it into the Topology. This can never work in the Reader-Writer Architecture for components like the Source Reader and Sink Writer which automatically load Transforms behind the scenes. There is no direct way for your application to influence the Transforms those components choose to use. All you can do is make sure that the Transforms you wish them to use are available.

Of course, you will probably not want to make such application specific Transforms available globally to the entire system – doing so could have completely unknown effects on other software. The designers of Windows Media Foundation realized this and have implemented functions called `MFTRegisterLocal` and `MFTRegisterLocalByCLSID`. These two functions work exactly the same as the `MFTRegister` function except the Transform they register is only visible to the current process. In other words, in the current process, the Transform you register locally can be enumerated and available along with the global ones in a way which is totally transparent to WMF. Other processes, however, simply will not see your locally registered Transform.

The sample code below shows the local registration of the Microsoft Supplied `ColorConverterDMO` Transform in order to make it available for use to a Sink Writer object.

```
// Windows 10, by default, provides an adequate set of codecs which the Sink Writer can
// find to write out the MP4 file. This is not true on Windows 7.

// If we are not on Windows 10 we register (locally) a codec
// the Sink Writer can find and use. The ColorConverterDMO is supplied by
// microsoft it is just not available to enumerate on Win7 etc.
// Making it available locally does not require administrator privs
// but only this process can see it and it disappears when the process
// closes

OperatingSystem os = Environment.OSVersion;
int versionID = ((os.Version.Major * 10) + os.Version.Minor);
if (versionID < 62)
{
    Guid ColorConverterDMOGUID = new Guid("98230571-0087-4204-b020-3282538e57d3");

    // Register the color converter DSP for this process, in the video
    // processor category. This will enable the sink writer to enumerate
    // the color converter when the sink writer attempts to match the
    // media types.
    hr = MFExtern.MFTRegisterLocalByCLSID(
        ColorConverterDMOGUID,
        MFTTransformCategory.MFT_CATEGORY_VIDEO_PROCESSOR,
        "",
        MFT_EnumFlag.SyncMFT,
        0,
        null,
        0,
        null
    );
}
```

```
);
}
Source: TantaCaptureToScreenAndFile::StartRecording::MFTTantaSampleGrabber_Sync
```

The above operation is not necessary on Windows 10 and so the local registration is skipped for Windows versions 10 and above. Once the `ColorConverterDMO` has been registered locally, it will be available to the Sink Writer like any other registered Transform.

ENUMERATING THE TRANSFORMS ON THE SYSTEM

The previous section (*Making a Transform Available*) mentioned that there were really two problems involved in making a DLL based Transform usable by Windows Media Foundation Client Applications. Firstly, the Transform has to be made generally available via the COM system and secondly, the Transform can (optionally) be made “discoverable” by other WMF objects. If a Transform is not discoverable, then any WMF Client that wishes to use it must know beforehand the GUID under which the Transform is registered for COM Interop.

The process of discovering suitable transforms is called Enumeration. The following sections will document the process of discovering all of the Transforms on the system and, as a bonus, also show how an application can find out what additional capabilities those Transforms offer.

THE TANTATransformPICKER SAMPLE APPLICATION

In the discussion below, the `TantaTransformPicker` Sample Application will be used to

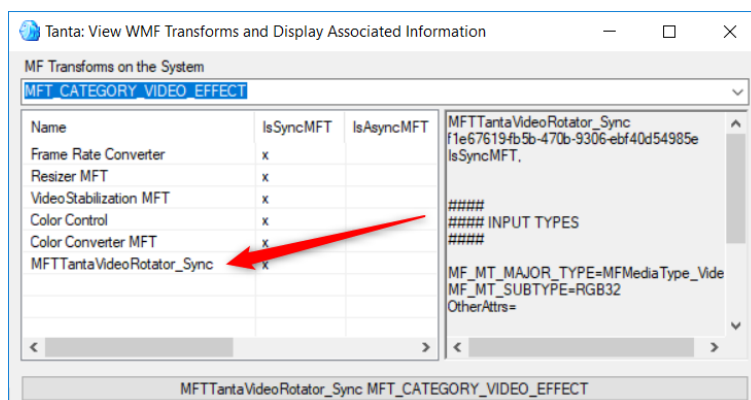


Figure 9.4: The TantaTransformPicker Sample Application

illustrate the process of discovering the Transforms available on a system. For purposes of demonstration, the code for the Transform enumeration operation has been embedded into a control in the `TantaCommon` library. This has the nice side effect of making that functionality available for

re-use by other applications.

The `ctlTantaTransformPicker` control is designed to be easily placed on the screen in order to provide a “pick-list” of options and information for the user. That is pretty much all, besides a bit of supporting code, that the `TantaTransformPicker` sample application does. The application does not do anything with a picked Transform besides displaying the known properties. This is just fine for our purposes in this section, since we are much more interested in how to obtain this information rather than how to use it once we do have it.

Before we proceed into the following sections, it should be noted that some of the details on view in the `ctlTantaTransformPicker` control display are based on information located in the registry outside of the Transform DLL. Other information on display is obtained by temporarily instantiating the DLL and interrogating it. This provides a nice way of dividing up the discussion and the following sections reflect this.

REGISTRY BASED TRANSFORM INFORMATION

As you might imagine, on any particular system there can be a large number of Transforms to sift through and so each Transform is associated with a major category based on function. As we shall see in subsequent sections, the enumeration process also has the ability to find out the Media Sub-Types the Transform supports on input and output. This is all designed to assist the WMF Client Application (and Topology Loader) in making a decision.

The Categories any under which one Transform can be listed are defined by the `MFTTransformCategory` class in the MF.Net `MediaFoundation` library. For reference, these categories are listed in the code section below.

```
MFT_CATEGORY_VIDEO_DECODER
MFT_CATEGORY_VIDEO_ENCODER
MFT_CATEGORY_VIDEO_EFFECT
MFT_CATEGORY_MULTIPLEXER
MFT_CATEGORY_DEMULTIPLEXER
MFT_CATEGORY_AUDIO_DECODER
MFT_CATEGORY_AUDIO_ENCODER
MFT_CATEGORY_AUDIO_EFFECT
MFT_CATEGORY_VIDEO_PROCESSOR
MFT_CATEGORY_OTHER

Source: MediaFoundation::MFTTransformCategory
```

The ComboBox at the top of the `ctlTantaTransformPicker` control defines the current category being enumerated. It can be seen in the previous screenshot of the `TantaTransformPicker` sample application, that the `MFT_CATEGORY_VIDEO_EFFECT` category is currently selected.

Once a category has been chosen, all Transforms available on the system in that category are displayed in the ListView below. It can be seen in the previous screenshot

that there are six Transforms in the `MFT_CATEGORY_VIDEO_EFFECT` category. Five have been placed there by other entities and one is the `MFTTantaVideoRotator_Sync` Transform output by the *TantaTransformInDLL* Sample Project. We will briefly reference the code in that sample application in order to demonstrate how the configuration of the source code is reflected in the enumeration information on display.

Besides the name, each Transform also has other associated information displayed in the columns to the right. If the `IsSyncMFT` column is marked then the Transform is Synchronous, if the `IsAsyncMFT` column is flagged then the Transform runs in Asynchronous Mode. The `IsHardware` takes care of the special case in which a Transform is actually a kind of driver mapped onto a physical device. This is often seen in Transforms that use hardware decoding and encoding based on the capabilities of a video card. A Transform which is hardware based can also be either Synchronous or Asynchronous – the two flags are not related. Lastly, the GUID of the Transform is presented in the `Guid` column.

All of the previous values such as name, mode (Synchronous or Asynchronous) and the GUID are stored in the registry when the `MFTExtern.MFTRegister` function is called in the DLL. The code for this from the `MFTTantaVideoRotator_Sync` Transform output by the *TantaTransformInDLL* sample application is shown below. There is a great deal of discussion about how this code is activated in the *Making a Transform Discoverable* section and so that information will not be reproduced here. Ultimately though, the end result is that the following code in the Transform is executed to perform the registration.

```
/// ++++++
/// <summary>
/// Set up the function which will be automatically called by COM when this
/// DLL is registered for COM interop. We use this call to register the MFT
/// and make it available for discovery by other MFT applications.
/// </summary>
/// <history>
/// 01 Nov 18 Cynic - Ported In
/// </history>
[ComRegisterFunctionAttribute]
static private void DllRegisterServer(Type t)
{
    HRESULT hr = MFTExtern.MFTRegister(
        t.GUID,
        MFTTransformCategory.MFT_CATEGORY_VIDEO_EFFECT,
        t.Name,
        MFT_EnumFlag.SyncMFT,
        0,
        null,
        0,
        null,
        null
    );
    MFTError.ThrowExceptionForHR(hr);
}

Source: TantaTransformInDLL:MFTTantaVideoRotator_Sync::DLLRegisterServer
```

The category under which the Transform is listed is entirely derived from the flag in the second parameter of the `MFTRegister()` call. In this case it is `MFT_CATEGORY_VIDEO_EFFECT`. The mode of the transform is hardcoded in the fourth parameter as `SyncMFT`. This makes sense, the mode of operation of a Transform is inherent in the design of the Transform and it is always known to the implementer. A Transform does not change from Synchronous to Asynchronous without a great deal of re-writing of code.

The first and third parameters are the name of the class of the Transform and the GUID of the Transform. These are derived from the C# Type of the object and this Type is passed in when the DLL is registered for COM Interop. The name is just the Type Name of the class of the Transform and the GUID value is actually the class `Guid` decoration on that Type. This is reproduced below – in particular observe the way the GUID is associated with the class. It can also be seen that the name of the class, `MFTTantaVideoRotator_Sync`, is the same name on display in the *TantaTransformPicker* sample application.

```
/// These class decorations are important. This is the GUID under which the MFT
/// will be registered. If you copy this code you should change this.
/// You should also probably change the class name. This will appear in the
/// registry as well. Use the TantaTransformPicker sample application to view
/// the MFT's in the registry.
[ComVisible(true),
Guid("F1E67619-FB5B-470B-9306-EBF40D54985E"),
ClassInterface(ClassInterfaceType.None)]
public sealed class MFTTantaVideoRotator_Sync : TantaMFTBaseStandalone_Sync
{
    // Format information
    private int m_imageWidthInPixels;
    ....
}

Source: TantaTransformInDLL:MFTTantaVideoRotator_Sync
```

It is interesting to observe that there is nothing enforcing the content of any of the parameters on this call. If you, as the programmer, choose to label it with wildly

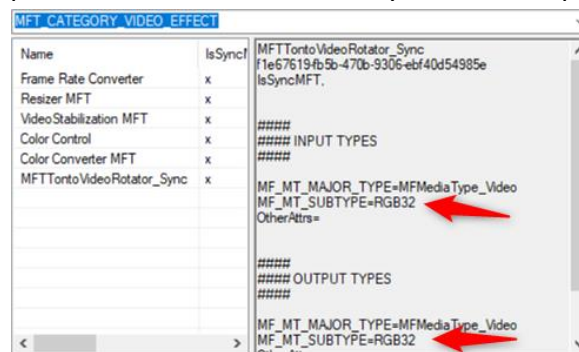


Figure 9.5: The TantaTransformPicker Sample Application

inappropriate categories, duplicate GUIDs and erroneous modes of operation there is nothing to stop you. All that will happen is that the Transform will not be properly discoverable by other WMF Client Applications and may well be rendered unusable if it is. You could, of course, also supply your own name and a GUID from other sources. You don't have

to use the information derived from the type.

The screenshot of the *TantaTransformPicker* sample application as shown in Figure 9.5 also contains a panel on the extreme right hand side of the display. This contains

detailed information on the Input and Output Media Sub-Types of the selected Transform. This information is supposed to be stored in the registry with the other details and should be fed into the `MFTRegister()` call using parameters 7 and 9. As you can see in the previous code section, the `DllRegisterServer()` call in the `MFTTantaVideoRotator_Sync` Transform does not do this. This is quite common – and so the `ctlTantaTransformPicker` control actually interrogates the selected Transform for the Media Sub-Type information. This is far more reliable since all Transforms, in order to be useable, *must* enable a caller to enumerate their input and output Media Sub-Types. Of course, as we shall see, this requires the Transform to be instantiated (albeit temporarily) but it does also provide the opportunity to demonstrate that technique.

ENUMERATING THE TRANSFORMS

Fundamentally, enumerating the Transforms is very easy – in theory all you need to do is make a call to `MFExtern.MFTEnumEx()` with the appropriate parameters and you will be returned a list of the Activator objects for each Transform. However, the process is made more difficult by some odd quirks in the way the `MFExtern.MFTEnumEx()` function works. The code below shows the Transform enumeration code from the `DisplayTransformsForCurrentCategory` function in the `ctlTantaTransformPicker` control.

[illegible]

```

// including the MFT_ENUM_FLAG_SYNCMFT flag
// which messes us up. This also appears to be true for the
// FieldOfUse and transcode only flags so we
// do not include them
if (flagVal == MFT_EnumFlag.LocalMFT) continue;
if (flagVal == MFT_EnumFlag.FieldOfUse) continue;
if (flagVal == MFT_EnumFlag.TranscodeOnly) continue;
// some of the higher flags are just for sorting the return results
if (flagVal >= MFT_EnumFlag.All) break;

hr = MFExtern.MFTEnumEx(currentCategory.GuidValue, flagVal,
    null, null, out activatorArray, out numResults);
if (hr != HRESULT.S_OK)
{
    throw new Exception("call to MFExtern.MFTEnumEx failed. HR=" + hr.ToString());
}

// now loop through the returned activators
for (int i = 0; i < numResults; i++)
{
    // extract the friendlyName and symbolicLinkName
    Guid outGuid = TantaWMFUtils.GetGuidForKeyFromActivator(
        activatorArray[i], MFAttributesClsid.MFT_TRANSFORM_CLSID_Attribute);
    string friendlyName =
        TantaWMFUtils.GetStringForKeyFromActivator(activatorArray[i],
            MFAttributesClsid.MFT_FRIENDLY_NAME_Attribute);

    // create a new TantaMFTCapabilityContainer for it
    TantaMFTCapabilityContainer workingMFTContainer = new
        TantaMFTCapabilityContainer(friendlyName, outGuid, currentCategory);
    // do we have this in our list yet
    int index = transformList.FindIndex(x => x.TransformGuidValue ==
        workingMFTContainer.TransformGuidValue);
    if (index >= 0)
    {
        // yes, it does contain this transform, just record the new sub-category
        transformList[index].EnumFlags |= flagVal;
    }
    else
    {
        // no, it does not contain this transform yet, set the sub-category
        workingMFTContainer.EnumFlags = flagVal;
        // and add it
        transformList.Add(workingMFTContainer);

        if ((activatorArray[i] is IMFAttributes)==true)
        {
            StringBuilder outSb = null;
            List<string> attributesToIgnore = new List<string>();
            attributesToIgnore.Add("MFT_FRIENDLY_NAME_Attribute");
            attributesToIgnore.Add("MFT_TRANSFORM_CLSID_Attribute");
            attributesToIgnore.Add("MF_TRANSFORM_FLAGS_Attribute");
            attributesToIgnore.Add("MF_TRANSFORM_CATEGORY_Attribute");
            hr = TantaWMFUtils.EnumerateAllAttributesAsText((activatorArray[i] as
                IMFAttributes), attributesToIgnore, 100, out outSb);
        }
    }

    // clean up our activator
    Marshal.ReleaseComObject(activatorArray[i]);
}

// now display the transforms
foreach (TantaMFTCapabilityContainer mftCapability in transformList)
{
    ListViewItem lvi = new ListViewItem(new[] {
        mftCapability.TransformFriendlyName,
        mftCapability.IsSyncMFT,
        mftCapability.IsAsyncMFT, mftCapability.IsHardware,
        mftCapability.TransformGuidValueAsString});
    lvi.Tag = mftCapability;
    listViewAvailableTransforms.Items.Add(lvi);
}

listViewAvailableTransforms.Columns.Add("Name", 250);
listViewAvailableTransforms.Columns.Add("IsSyncMFT", 70);
listViewAvailableTransforms.Columns.Add("IsAsyncMFT", 90);
listViewAvailableTransforms.Columns.Add("IsHardware", 90);
listViewAvailableTransforms.Columns.Add("Guid", 200);

```

Working With Transforms

```
    }  
    finally  
    {  
        if (tmpSourceReader != null)  
        {  
            // close and release  
            Marshal.ReleaseComObject(tmpSourceReader);  
            tmpSourceReader = null;  
        }  
    }  
}  
  
Source: TantaCommon::ctlTantaTransformPicker::DisplayTransformsForCurrentCategory
```

The `MFTExtern.MFTEnumEx()` function provides a list of Transforms when given the current Category and various flags as defined by the `MFT_EnumFlag` enum. The problem is that the `MFT_EnumFlag` enum is a bitwise enum and thus the various flags can be combined with a C# `or` operator. This makes it easy for a client application (which may have a list of features it wants) to specify things. It makes it rather harder for software, which wants to interrogate the list, to see what flags are on a particular Transform. For example, a client application could combine the flags of `MFT_EnumFlag.AsyncMFT | MFT_EnumFlag.SyncMFT | MFT_EnumFlag.Hardware` if the actual type of Transform required was not relevant. In that event, a list containing all three types of Transform would be returned. However, once we have the list, there is no way to interrogate any one Transform to find out what it is – the Transform does not return this information.

The way we address this situation is to loop through the `MFT_EnumFlag` enum and only apply one flag at a time.

```
foreach (MFT_EnumFlag flagVal in Enum.GetValues(typeof(MFT_EnumFlag)))  
{  
    ... more code  
  
    hr = MFTExtern.MFTEnumEx(currentCategory.GuidValue, flagVal,  
                             null, null, out activatorArray, out numResults);  
  
    ... more code  
}
```

Of course we will probably see the same Transform more than once and so we track the returned information to a store (called the `transformList` which is just a C# List of `TantaMFTCapabilityContainer` objects).

```
// do we have this in our list yet  
int index = transformList.FindIndex(x => x.TransformGuidValue ==  
    workingMFTContainer.TransformGuidValue);  
if (index >= 0)  
{  
    // yes, it does contain this transform, just record the new sub-category  
    ... more code  
}  
else  
{  
    // no, it does not contain this transform yet, set the sub-category  
    ... more code  
}
```

Every time we get a Transform we check to see if we have already seen that Transform. If we have not seen it, we add it to the store in a `TantaMFTCapabilityContainer` object. If we have seen the Transform before we simply update the records for the

existing value in the `TantaMFTCapabilityContainer` of that object in the store. In this round-about way we can build up a list of all of the flags applicable to any one Transform for later display.

It should be noted at this time that for some flag options (`MFT_EnumFlag.LocalMFT` for example), it is not possible to figure out if they are set using the above mechanism. Setting this flag is the equivalent of also setting `MFT_EnumFlag.SyncMFT` and you will get all Synchronous Transforms in a list - the local ones will be ordered first though. Accordingly, the sample code just ignores the `MFT_EnumFlag` values that cannot be determined.

Note that the enumeration operation does not return to us a list of instantiated Transform objects. Instead what we get is a list of Activators which could, if you wished, create the Transform for you. These Activators can tell us certain things about the Transform such as the GUID it is registered under and the user readable “*Friendly Name*”.

```
... more code

// extract the friendlyName and symbolicLinkName
Guid outGuid = TantaWMFUtils.GetGuidForKeyFromActivator(
    activatorArray[i], MFAttributesClsid.MFT_TRANSFORM_CLSID_Attribute);

string friendlyName =
    TantaWMFUtils.GetStringForKeyFromActivator(activatorArray[i],
        MFAttributesClsid.MFT_FRIENDLY_NAME_Attribute);
... more code
```

This is useful because it means that we do not necessarily need to instantiate the Transform to get that information. At this point we are able to populate a nice display list of the names of all the Transforms and the information we have on them (such as the GUID and `MFT_EnumFlags`). Eventually, when the user clicks on a particular Transform object, we will instantiate the Transform to get further information from it. This saves us the performance overhead of instantiating every Transform when the user is not likely to be interested 99% of them.

Activators, however, are COM objects – they are expected to be released and they are strictly one use items. We could store the Activator object in our `TantaMFTCapabilityContainer` object for each Transform and release them when the program closed. However, if we ever used that Activator to instantiate a Transform, we could never use it again and the whole list would have to be rebuilt. Instead, the `ctlTantaTransformPicker` control just stores the GUID and Friendly Name and instantiates the Transform from that information if it needs to do so. This also allows us to demonstrate the technique of creating a Transform if all you have is the GUID.

```
// clean up our activator
Marshal.ReleaseComObject(activatorArray[i]);
```

You will note in the code section above, that each Activator is released with a call to `Marshal.ReleaseComObject(activatorArray[i])` immediately after the GUID and Friendly Names are obtained from it.

TRANSFORM BASED INFORMATION

If a user clicks on the Friendly Name of the Transform, then more information on that Transform will be displayed in the right hand panel. These details are mostly the Media Sub-Types which the Transform is prepared to accept as Input and Output, however, the names of other Attribute information stored within the Transform is also presented. As mentioned in the *Making a Transform Discoverable* section, the Media Sub-Type information is supposed to be recorded in the Registry and hence available for general access if you know the GUID. However, many Transforms do not make this information available at registration time and so interrogating the Transform for it is really the only reliable mechanism.

The act of clicking on the Friendly Name of the Transform eventually causes the `SetTransformInfoPanel` function of the `ctlTantaTransformPicker` control to be called. That code is fairly lengthy and is reasonably standard so it will not be reproduced here – you can easily look it up and review it. The really interesting operations happen in the static `GetInputMediaTypesFromTransformByGuid` function of the `TantaWMFUtils` class called from within the `SetTransformInfoPanel` function. This is shown in the code section below.

```
/// ++++++
/// <summary>
/// Gets a list of input Media Types object from a Transform represented
/// by a guid.
///
/// NOTE: the media types returned here must be released
///
/// </summary>
/// <param name="transformGuid">the guid of the transform</param>
/// <param name="wantLocalServer">if true use CLSCTX LOCAL SERVER
/// otherwise CLSCTX INPROC SERVER</param>
/// <returns>a list of Media Types - these must be released</returns>
/// <history>
/// 01 Nov 18 Cynic - Originally Written
/// </history>
public static List<IMFMediaType> GetInputMediaTypesFromTransformByGuid(
    Guid transformGuid,
    bool wantLocalServer)
{
    IMFTransform transformObj = null;
    IMFMediaType mediaType = null;
    HRESULT hr;
    List<IMFMediaType> outList = new List<IMFMediaType>();

    try
    {
        // get the transform object
        transformObj = GetTransformFromGuid(transformGuid, wantLocalServer);
        if (transformObj == null) return outList;

        // get all of the media types this transform can handle
        // I do not like endless loops. So we cap this with
```

```

        // a hardcoded limit
        for (int typeCounter = 0; typeCounter < MAX_TYPES_TESTED_PER_TRANSFORM; typeCounter++)
        {
            try
            {
                // get the available input type for the current typeCounter
                hr = transformObj.GetInputAvailableType(0, typeCounter, out mediaType);
                // not found, we are done
                if (hr != HRESULT.S_OK) break;
                if (mediaType == null) break;

                // add it now
                outList.Add(mediaType);
            }
            catch
            {
            } // bottom of try...catch
        } // bottom of for (int typeCounter = 0; ...
    }
    catch
    {
        // do nothing
    }
    finally
    {
        // make sure this is released
        if (transformObj != null)
        {
            // close and release
            if (Marshal.IsComObject(transformObj) == true)
                Marshal.ReleaseComObject(transformObj);
            transformObj = null;
        }
    }

    return outList;
}

```

Source: TantaCommon::TantaWMFUtils::GetInputMediaTypesFromTransformByGuid

The first thing the `GetInputMediaTypesFromTransformByGuid` function needs to do is to instantiate the Transform. This is done with a call to `GetTransformFromGuid()` which takes the GUID value of the Transform as a parameter. The operation of this call was discussed in the previous *Adding a Transform By Creating it from a GUID* section and so that information will not be reproduced here. Suffice it to say that the `GetTransformFromGuid()` call returns the instantiated Transform object which, you will also note, is carefully released in the `finally` block in the above code section.

Once the Transform has been instantiated, a call to the `GetInputAvailableType()` function will return an `IMFMediaType` object describing the one of the Input media types supported by the Transform. The `GetInputAvailableType()` function is itself an enumerator and we call it repeatedly with an incrementing index in order to get all of the available types.

```

// get the available input type for the current typeCounter
hr = transformObj.GetInputAvailableType(0, typeCounter, out mediaType);

```

It should be stated that we know the `GetInputAvailableType()` function will be present in the Transform (and what its behavior will be) because its presence is mandated as part of the `IMFTransform` interface which all Transforms must support. When we call the `GetInputAvailableType()` function we are talking directly to the

code of the Transform and the actions of this this can be observed in any of the Transforms in the Tanta Sample Projects.

The `IMFMediaType` object returned by the `GetInputMediaTypesFromTransformByGuid()` call also needs to be released. You can see this being done in the `SetTransformInfoPanel` function after the Media Sub-Type details have been extracted from it. It cannot be emphasized often enough that objects returned via COM Interop always need to be released when you are finished with them - not doing so will generate memory leaks.

In closing, it can be seen that by enumerating the Transforms on the system and then by instantiating a specific Transform we can interrogate the standard functions of the `IMFTransform` interface to find out more information on the Transform. For the currently selected Transform, the `ctlTantaTransformPicker` control displays both the information derived from the registry and the “extra” information derived directly from the Transform itself in a panel on the right hand side of the display.

PASSING INFORMATION IN AND OUT OF A TRANSFORM

In the majority of cases Transforms are standalone entities. Standalone in the sense that they know what they need to do and they do not require any exchange of information from the client application in order to do it. In the `TantaTransformDirect` sample code, the `MFTTantaGrayscale_Sync` and `MFTTantaWriteText_Sync` Transforms are examples of this. Sometimes, however, this is not the case and the client either needs to receive information from the Transform or the Transform needs information from the client – or both. The `MFTTantaFrameCounter_Sync` transform in the `TantaTransformDirect` Sample application and the `MFTTantaVideoRotator_Sync` Transform in the `TantaTransformInDLL` Sample Project are examples of such requirements.

The methods which can be used to exchange information between the WMF Client Application and the Transform depend on the method used to load the Transform into the Pipeline (Direct or COM) and also on the type of information to be transferred. Listed below are several methods of client-transform information exchange – there may well be others so don’t treat these examples as a definitive list.

1. If the Transform is present in the same solution as the client then the Client Application can just directly call functions and properties in it.

2. If the Transform was loaded via COM, or directly, information can be exchanged by setting and/or getting one of the Transforms Attributes. This can be done in either direction.
3. If the Transform was loaded via COM, or directly, and the Transform is coded in C#, information can be exchanged via Reflection and Late Binding.
4. If the Transform was loaded via COM and the Transform is not coded in C# information can be exchanged via Marshal and other standard COM methods.

Each of the above methods (except #4) will be discussed below in more detail.

INFORMATION EXCHANGE VIA DIRECT CALLS

This particular method is trivial. If the Transform source code is present in your solution – either directly in the same project as the client or in a different project within that solution - then you can just call public functions and properties in that class as you would any other C# object.

Of course, you have to have the instantiated Transform object which was added to the Pipeline. This object, however, is fairly simple to record at the time it is added to the Pipeline – after all it is your code which created the Transform with the C# `new` operator. As a reference, this can be seen to be happening in the *TantaTransformDirect* sample application when the Transform is added in the EVR Renderer control. The relevant section of code is reproduced below.

```

/// ++++++
/// <summary>
/// Handle a checked changed on the transform radio button
/// </summary>
/// <history>
///     01 Nov 18  Cynic - Originally Written
/// </history>
private void radioButtonMFTFrameCounter_CheckedChanged(object sender, EventArgs e)
{
    if (radioButtonMFTFrameCounter.Checked == false) return;
    // give it the transform
    MFTTantaFrameCounter_Sync fCounter = new MFTTantaFrameCounter_Sync();
    SetTransformOnEVRControl(fCounter);
}

/// ++++++
/// <summary>
/// Set the transform on the EVR control. Can only be done when the
/// video is not playing.
/// </summary>
/// <param name="transformObj">the transform object to use, can be null
/// for no transform</param>
/// <history>
///     01 Nov 18  Cynic - Originally Written
/// </history>
private void SetTransformOnEVRControl(IMFTTransform transformObj)
{
    // we only permit this action if we are not playing
    if (ctlTantaEVRFilePlayer1.PlayerState != TantaEVRPlayerStateEnum.Ready)

```

Working With Transforms

```
{
    OISMessageBox("A video is currently playing");
    return;
}

// give it to the EVR player
ctlTantaEVRFilePlayer1.VideoTransform = transformObj;
}
```

Source: TantaTransformDirect::frmMain::SetTransformOnEVRControl

The call to `SetTransformOnEVRControl()` is activated when the state of the radio buttons on the screen is changed.

The presence of the actual instantiated object and its source code is used to good effect when retrieving the `FrameCount` value from the `MFTTantaFrameCounter_Sync` Transform. Although, the call to fetch the frame count is trivial, it will be reproduced below for completeness.

```
/// ++++++
/// <summary>
/// Displays the frame count from the Transform (if it is in use)
/// </summary>
/// <history>
///     01 Nov 18 Cynic - Originally Written
/// </history>
private void buttonGetFrameCount_Click(object sender, EventArgs e)
{
    // we have to have one
    if(ctlTantaEVRFilePlayer1.VideoTransform==null)
    {
        OISMessageBox("No transform found.");
        return;
    }
    // it has to be the frame counter transform
    if ((ctlTantaEVRFilePlayer1.VideoTransform is MFTTantaFrameCounter_Sync) == false)
    {
        OISMessageBox("The Transform in use is not the Frame Counter.");
        return;
    }
    // just display a dialog box
    OISMessageBox("The current frame count is " +
        (ctlTantaEVRFilePlayer1.VideoTransform as
            MFTTantaFrameCounter_Sync).FrameCount.ToString());
}
```

Source: TantaTransformDirect::frmMain::buttonGetFrameCount_Click

It is, of course, also possible to set up a C# Delegate and Event mechanism in direct mode (option 4 in the list above) to trigger a call back from the Transform into the Client Application.

INFORMATION EXCHANGE VIA ATTRIBUTES

Transforms can have Windows Media Foundation Attributes. In fact, Asynchronous Transforms require the use of these as part of their configuration mechanism. Synchronous Transforms do not require the use of Attributes. However, an Attribute Container (`IMFAttributes`) can still be set up on a Synchronous Transform and the Attributes it contains can be used as an information transfer mechanism between the Client Application and the Transform.

The Attribute Container is set up for you in both of the Transform base classes (TantaMFTBase_Sync and TantaMFTBase_Async) in the *TantaCommon* sample library and also on the standalone Transform base class (TantaMFTBaseStandalone_Sync) used in the *TantaTransformInDLL* sample.

In order to explore this topic further, we will look at the usage of the Attribute transfer mechanism between the transform in the *TantaTransformDLL* sample code and the Client Application in the *TantaTransformInDLLClient* example. The Transform in the *TantaTransformDLL* sample code DLL is called *MFTTantaVideoRotator_Sync*, and its function is to rotate the video on display through a variety of flips and orientations. The “RotateFlipType” mode used by the Transform can be changed by the client application while the video is actually running. Clearly there has to be some way for the client application to communicate this change to the Transform and, in this example, the Attribute method is used. Note that this mechanism is used primarily for demonstration purposes and some of the other methods discussed in this section would also work as well (and possibly be more efficient).

The Attribute Container object is created and managed by the *TantaMFTBase_Sync* base class and most of it is pretty standard. One thing to particularly note is the mechanism by which the Attribute Container object is returned from within the Transform.

```

/// ++++++
/// <summary>
/// Gets the global attribute store for this Transform
///
/// </summary>
/// <param name="pAttributes">Receives a pointer to the IMFAttributes interface.
/// The caller must release the interface.</param>
/// <returns>S_OK or other for fail</returns>
/// <history>
/// 01 Nov 18 Cynic - Ported In
/// </history>
public HRESULT GetAttributes(out IMFAttributes pAttributes)
{
    pAttributes = null;
    HRESULT hr = HRESULT.S_OK;

    try
    {
        lock (m_TransformLockObject)
        {
            // Using GetUniqueRCW means the caller can do
            // ReleaseComObject without trashing our copy. We *don't*
            // want to return a clone because we *do* want them to be
            // able to change our attributes.
            pAttributes =
                TantaWMFUtils.GetUniqueRCW(m_TransformAttributeCollection) as IMFAttributes;
        }
    }
    catch (Exception e)
    {
        hr = (HRESULT)Marshal.GetHRForException(e);
    }
    return hr;
}

```

Source: TantaCommon::TantaMFTBase_Sync::GetAttributes

The Transform is a COM object and the information it returns is expected to adhere to the COM standard. This means the caller (the client application in this case) will expect to release the Attribute Container object. In order to give the caller something to release via COM, and yet make sure that the object is not garbage collected by the .NET runtime we create something known as a Runtime Callable Wrapper (RCW). This effectively wraps the object in some code that can be safely released by the caller. We do not want to just create a clone of the Attribute Object and hand that over. That would suffice for passing information out of the Transform to the Client Application, but any changes made to a cloned object by the Client Application would be lost and hence unavailable to the Transform. We definitely do not want to just return the Attribute Object without the RCW. The `GetAttributes` function is part of the `IMFTransform` interface and the obligation of the caller to safely release the returned Attributes object is mandatory.

The client application (the `frmMain` class in the `TantaTransformInDLLClient` sample) responds to user option changes on the screen by sending the new `RotateFlipType` value to the Transform. It does this by retrieving the Attribute Container from the Transform. Actually it calls the EVR Renderer control for this container because the EVR Renderer stored the Topology Node of the Transform when it added the Transform to the Pipeline. Let's look at the code in the `frmMain` class first and then we will explore what is going on in the EVR Renderer control

```
// this Guid is the key we use to set the FlipMode on the attributes
// of the transform we inserted into the PipeLine. The FlipMode is
// retrieved by the Transform so it also needs to know this Guid.
// Other than that, there is nothing special about this value.
private Guid clsidFlipMode = new Guid("EF5FB03A-23B5-4250-9AA6-0E70907F8B4B");

private void SetFlipModeOnTransform(RotateFlipType flipType)
{
    // get the attribute container from the transform in the EVR player
    IMFAttributes attributeContainer = ctlTantaEVRFilePlayer1.GetTransformAttributes();
    if (attributeContainer == null) return;

    // set the flipType as an int32. Attributes cannot contain enums
    HRESULT hr = attributeContainer.SetUINT32(clsidFlipMode, (int)flipType);

    // release it
    System.Runtime.InteropServices.Marshal.ReleaseComObject(attributeContainer);
}

Source: TantaTransformInDLLClient::frmMain::SetFlipModeOnTransform
```

In the above code, the `attributeContainer` object is obtained from the `ctlTantaEVRFilePlayer` control.

```
// get the attribute container from the transform in the EVR player
IMFAttributes attributeContainer = ctlTantaEVRFilePlayer1.GetTransformAttributes();
```

Note, however, that the `RotateFlipMode` is an Enum. An Attribute is simply a key value pair where the key is a GUID and the value is a `PropVariant` (see the *About Attributes* and *PropVariant* sections for more information). A `PropVariant` cannot contain complex

objects – just simple things like an `Int32` or `string` or `Guid`). Thus, the code above converts the `RotateFlipMode` Enum into an `Int32` before storing it in the Attribute.

```
// release it
System.Runtime.InteropServices.Marshal.ReleaseComObject(attributeContainer);
```

The Transform is expected to know this has been done and is expected to cast it back when it retrieves it. Another thing to realize is that the Attribute requires a GUID as a key. The Transform is also expected to know this GUID value and that it should use it to access the Attribute information. In particular, note the call to `ReleaseComObject()` in the last line.

```
// set the flipType as an int32. Attributes cannot contain enums
HResult hr = attributeContainer.SetUINT32(clsidFlipMode, (int)flipType);
```

If we had not earlier wrapped the `attributeContainer` object in an RCW we would be generating a great deal of hard to debug problems here.

The Attribute method of Client/Transform communication is limited to simple chunks of information which can be represented as a PropVariant. Also, both Client and Transform must know the GUID key value under which this information is stored.

The EVR Renderer control retrieves the Attribute Container by querying the Transforms Topology Node. From this it can get the instantiated Transform object itself. It will know that the Transform Object implements the `IMFTransform` interface and the `GetAttributes` function is a required part of that interface.

[illegible]

Working With Transforms

```
// will be obtained in such a way so that it is safe for the
// caller to release it.
hr = (transformObject as IMFTransform).GetAttributes(out attributeContainer);
if (hr != HRESULT.S_OK) return null;
if (attributeContainer == null) return null;
// return the attribute container
return attributeContainer;
}
```

Source: TantaCommon::ctrlTantaEVRFilePlayer::GetTransformAttributes

It should be noted that the only reason the EVR Renderer control knows about the Transform Topology Node is because that object was recorded by the control when it created the Topology and added the Transform to the Pipeline. If you have the Topology Node of a Transform, getting the Transform object itself is a simple call on the `GetObjectFunction()` of the `IMFTransform` interface.

```
// get the transform object from the node
hr = VideoTransformNode.GetObject(out transformObject);
```

Once the required information is stored in the Attribute, the Transform has immediate access to it. Remember the Attribute Container on which this Attribute is set in the client application is the same one that the Transform is using – this is how the information is passed. For completeness, let's turn to the Transform and see how it accesses the new `RotateFlipType` value – this is pretty standard stuff though.

```
//+=====  
/// <summary>  
/// Get the FlipMode from this object. It located as an Attribute there. The  
/// client knows the topology node when it adds this MFT to the pipeline  
/// and so it can get access to the attributes of this class even though  
/// it was dynamically created via COM and adjust this attribute to control things.  
/// </summary>  
/// <history>  
///      01 Nov 18   Cynic - Originally Written  
/// </history>  
private RotateFlipType FlipMode  
{  
    get  
    {  
        IMFAttributes attributeContainer = null;  
  
        // get the attribute container  
        HRESULT hr = this.GetAttributes(out attributeContainer);  
        if (hr != HRESULT.S_OK) return RotateFlipType.RotateNoneFlipNone;  
        if (attributeContainer == null) return RotateFlipType.RotateNoneFlipNone;  
  
        // we expect this to be a RotateFlipType enum. However, attributes  
        // cannot have enums (just things like strings or ints or doubles  
        // so the enum will have been casted to an int32 by the client.  
        int enumAsInt = MFExtern.MFGetAttributeUINT32(  
            attributeContainer,  
            clsidFlipMode,  
            (int)RotateFlipType.RotateNoneFlipNone);  
  
        // return the int as an enum  
        return (RotateFlipType)enumAsInt;  
    }  
}
```

Source: TantaTransformInDLL::MFTTantaVideoRotator Sync::FlipMode

In particular, notice that the Transform knows the GUID key (`clsidFlipMode`) under which to find the current flip mode value. As mentioned previously, this GUID must be the same as the one used by the caller to store it. It should also be noted how the `Int32`

of the `Attribute` value is converted back to a `RotateFlipType` enum as part of the `return`.

It is also interesting to see where the new flip mode data is used - the `TantaTransformInDLL` source code can be reviewed for this purpose. As a helpful get-started hint, notice how each time the `MFTTantaVideoRotator_Sync` Transform processes a new frame (ultimately out of the `OnProcessOutput()` call), it will get the current `RotateFlipType` value. This is why the rotation mode can be dynamically changed while the video is running – the processing for each video frame is independent of any other.

INFORMATION EXCHANGE VIA REFLECTION AND LATE BINDING

.NET assemblies always present the types and signatures of the public classes, functions and properties of the objects they define. Much of this information is also publically available in an instantiated object. Thus if you have an instantiated C# object it is possible, if you know how, to call functions and properties within it. This is true even if the source code for that object is unknown and you obtained the object from elsewhere (perhaps via COM Interop).

The process of calling a function or property in a .NET object for which your program does not have compile time knowledge of that objects source code is called *Late Binding*. The alternative is a call which is hardcoded into the resulting binary at compile time. The `System.Reflection` collection of classes provides all the tools needed to access the public properties and functions of a .NET object and you will see this process referred to as *Reflection and Late Binding* or perhaps just *Late Binding*.

The technique of *Late Binding* can be used to exchange information with a Transform which has been written in a .NET language such as C#. As has been discussed previously (see the *Information Exchange Via Attributes* section), it is not too difficult to get access to the Transform object itself even if that object is in an unknown DLL and originally provided by COM Interop. The problem is that it is not possible, at compile time, to setup calls into some unknown object which will only later be valid. Instead, *Late Binding* must be used to call into the object if the name and signature (return value and parameters) of the function or property being called is known.

The `TantaTransformInDLLClient` and the `MFTTantaVideoRotator_Sync` Transform in the `TantaTransformDLL` sample code are designed to demonstrate the concept of Client-Transform communication via *Late Binding*. Although the primary purpose of the `MFTTantaVideoRotator_Sync` Transform is to demonstrate a dynamically loaded (via

COM Interop) C# Transform which rotates the video on the screen, that Transform also counts the frames as they pass through the system. The `frmMain` class of the

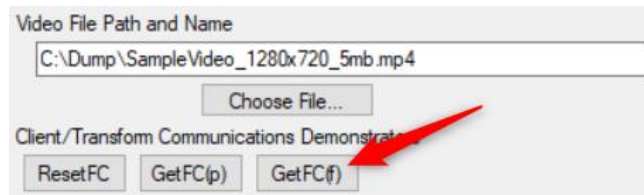


Figure 9.6: A Section of the TantaTransformInDLLClient Main Form

`TantaTransformInDLLClient` solution contains a series of buttons on the screen which, when pressed, will locate the instantiated `MFTTantaVideoRotator_Sync`

Transform object and call various properties and procedures in it to retrieve or reset the current frame count. Since the buttons are only there to demonstrate the technique, the frame count information is just displayed in a pop-up message box and nothing further is done with it.

There are three examples of the Late Binding presented as buttons on the `TantaTransformInDLLClient` application main form. The first two (`ResetFC` and `GetFC(p)`) reset the Frame Count (FC) via a property and get the Frame Count via a property (p). The code behind these operations is simple and can readily be observed in the `buttonResetFrameCount_Click` and `buttonGetFCViaProperty_Click` of the `frmMain` class. The code below demonstrates how to call a property in an object if you know the name of the property and the type that property uses.

```
private void buttonGetFCViaProperty_Click(object sender, EventArgs e)
{
    LogMessage("buttonGetFCViaProperty_Click called");

    // get the transform
    IMFTransform transformObject = ctlTantaEVRFilePlayer1.GetTransform();
    if (transformObject == null)
    {
        LogMessage("buttonGetFCViaProperty: transformObject == null");
        OISMessageBox("No transform object. Is the video running?");
        return;
    }

    // get the real type of the transform. This assumes it is a .NET
    // based transform - otherwise it will probably just be a generic
    // _ComObject and the code below will fail.
    Type transformObjectType = transformObject.GetType();

    // set up to invoke the FrameCountAsPropertyDemonstrator. Note that
    // we have to know the name of the property we are calling and the
    // type it takes.
    try
    {
        object frameCount = transformObjectType.InvokeMember(
            "FrameCountAsPropertyDemonstrator",
            BindingFlags.GetProperty,
            null, transformObject, null);
        if ((frameCount is int) == true)
        {
            LogMessage("The frame count is " + frameCount.ToString());
            OISMessageBox("FrameCount=" + frameCount.ToString());
        }
    }
    catch (Exception ex)
    {
        OISMessageBox("An error occurred please see the logfile");
        LogMessage(ex.Message);
        LogMessage(ex.StackTrace);
    }
}
```

Source: `TantaTransformInDLLClient::frmMain::buttonGetFCViaProperty_Click`

Over in the `MFTTantaVideoRotator_Sync` Transform object, there is absolutely nothing distinctive about the `FrameCountAsPropertyDemonstrator` property – it is just a standard `get/set` public property whose value is an `int`. Note that the `InvokeMember()` call is on the *Type* of the Transform object, not the object itself, and that the Transform object itself is actually one of the parameters to that call.

Rather more interesting is the `FrameCountAsFunctionDemonstrator` function of the `MFTTantaVideoRotator_Sync` Transform. This function returns a `bool` and accepts two strings as parameters - one of which is a `ref` variable. Since this code is designed for purposes of demonstration - all it really does is accept the input string, append the Frame Count to it and then returns the result in the `ref` variable. This demonstrates a two way communication mechanism between the client application and the Transform which can exchange an object of any type.

As with the `FrameCountAsPropertyDemonstrator` property, there is nothing special about the `FrameCountAsFunctionDemonstrator` function. For completeness it will, however, be reproduced below.

```

/// 
/// Get the current frame count and prepend a user supplied string. The
/// output is a string in a ref variable.
/// 
/// Note how this function is ComVisible This is not necessary to make it
/// accessible to a .NET client via Reflection and Late Binding
/// to interact with the transform but will make it visible to non-.NET
/// clients via standard COM calls.
/// 
/// the leading text to prepend. Cannot be null
/// the string with the framecount is returned here
/// true the operation was successful, false it was not
/// 
///      01 Nov 18   Cynic - Originally Written
/// 
[ComVisible(true)]
public bool FrameCountAsFunctionDemonstrator(
    string frameCountLeadingText,
    ref string outString)
{
    // we say the leading text cannot be null
    if (frameCountLeadingText == null)
    {
        outString = "";
        return false;
    }
    // set up the string
    outString = frameCountLeadingText + m_FrameCount.ToString();
    return true;
}

```

Source: TantaTransformInDLL:MFTTantaVideoRotator Sync::FrameCountAsFunctionDemonstrator

In the above text the `[ComVisible]` decoration is not necessary in order to make the function visible to a C# client via Late Binding. It will, however, make the function visible via standard COM Interop methods so it is included for thoroughness.

The call to this function in the main form of the Client Application is much more interesting.

Working With Transforms

```
/// ++++++
/// <summary>
/// Demonstrates the client/transform communications. Displays the
/// frame count in the rotator transform by calling the a function.
/// The function requires two parameters a leading string and a ref string
/// which is the output. A boolean is returned to indicate success. The
/// frame count is appended to the user supplied leading string.
///
/// This function uses late binding and expects the rotator transform
/// to be instantiated.
/// </summary>
/// <history>
/// 01 Nov 18 Cynic - Originally Written
/// </history>
private void buttonGetFCViaFunction_Click(object sender, EventArgs e)
{
    LogMessage("buttonGetFCViaFunction_Click called");

    // get the transform
    IMFTransform transformObject = ctlTantaEVRFilePlayer1.GetTransform();
    if (transformObject == null)
    {
        LogMessage("buttonGetFCViaFunction: transformObject == null");
        OISMessageBox("No transform object. Is the video running?");
        return;
    }

    // get the real type of the transform. This assumes it is a .NET
    // based transform - otherwise it will probably just be a generic
    // ComObject and the code below will fail.
    Type transformObjectType = transformObject.GetType();

    // set up our parameters. both are strings, the second is ref string
    object[] parameters = new object[2];
    string outText = "Unknown FrameCount";
    parameters[0] = "I just checked, the frame count is ";
    parameters[1] = outText;

    // set up our parameter modifiers. This is how we tell the InvokeMember
    // call that one of our parameters is a ref
    ParameterModifier paramMods = new ParameterModifier(2);
    paramMods[1] = true;
    ParameterModifier[] paramModifierArray = { paramMods };

    try
    {
        // set up to invoke the FrameCountAsFunctionDemonstrator. Note that
        // we have to know the name of the function we are calling, the return
        // type and its parameter types
        object retVal = transformObjectType.InvokeMember(
            "FrameCountAsFunctionDemonstrator",
            BindingFlags.InvokeMethod,
            null,
            transformObject,
            parameters,
            paramModifierArray,
            null, null);
        if ((retVal is bool) == false)
        {
            LogMessage("call to FrameCountAsFunctionDemonstrator failed.");
            OISMessageBox("call to FrameCountAsFunctionDemonstrator failed.");
            return;
        }
    }
    catch (Exception ex)
    {
        OISMessageBox("An error occured please see the logfile");
        LogMessage(ex.Message);
        LogMessage(ex.StackTrace);
    }

    if (parameters[1] == null)
    {
        LogMessage("buttonGetFCViaFunction_Click: Null value returned for ref parameter.");
        OISMessageBox("Null value returned for ref parameter.");
        return;
    }
    if ((parameters[1] is string) == false)
    {
        LogMessage("buttonGetFCViaFunction_Click: Reference value is not a string");
    }
}
```

```
OISMessageBox("Reference value is not a string.");
return;
}

LogMessage("buttonGetFCViaFunction Click: " + (parameters[1] as string));
OISMessageBox((parameters[1] as string));
}

Source: TantaTransformInDLLClient::frmMain::buttonGetFCViaFunction_Click
```

Detailed discussions of how a function can be called via Reflection and Late Binding can be found in any number of online resources and so that content will not be reproduced here. However we will discuss the major steps in the process.

The instantiated Transform object is retrieved via the `ctlTantaEVRFilePlayer1.GetTransform()` call. The Tanta `ctlTantaEVRFilePlayer` control offers this facility and that call will work if the Transform was loaded in direct mode or via a GUID and COM. Previous sections (*Getting The WMF Object From a Topology Node* and *Information Exchange Via Attributes*) discussed the mechanisms used to obtain this object and that information will not be reproduced here.

```
// get the transform
IMFTransform transformObject = ctlTantaEVRFilePlayer1.GetTransform();
```

Once we have the instantiated object, we can get the C# Type from it. This is done via the standard `GetType()` call which is available on every .NET object.

```
// get the real type of the transform. This assumes it is a .NET
// based transform - otherwise it will probably just be a generic
// ComObject and the code below will fail.
Type transformObjectType = transformObject.GetType();
```

The next step is to set up our parameters. Both parameters are strings - the fact that the second parameter is a `ref` string does not matter at this point. We use an array of objects for this purpose.

```
// set up our parameters. both are strings, the second is ref string
object[] parameters = new object[2];
string outText = "Unknown FrameCount";
parameters[0] = "I just checked, the frame count is ";
parameters[1] = outText;
```

After the parameters are set up, we configure the parameter modifiers. The only reason we need these is because the second parameter is a `ref` value. If there were no `ref` or `out` values we could ignore this step and just pass in a null value for the parameter modifier array.

```
// set up our parameter modifiers. This is how we tell the InvokeMember
// call that one of our parameters is a ref
ParameterModifier paramMods = new ParameterModifier(2);
paramMods[1] = true;
ParameterModifier[] paramModifierArray = { paramMods };
```

At this point we can make the call into the instantiated object and, once it completes, examine both the return value and the contents of the `ref` parameter. This is done through Reflection calling `InvokeMember()` on the Type object of the Transform.

Working With Transforms

```
// set up to invoke the FrameCountAsFunctionDemonstrator. Note that
// we have to know the name of the function we are calling, the return
// type and its parameter types
object retVal = transformObjectType.InvokeMember(
    "FrameCountAsFunctionDemonstrator",
    BindingFlags.InvokeMethod,
    null,
    transformObject,
    parameters,
    paramModifierArray,
    null, null);
```

The `InvokeMember()` call takes as parameters the name of the member being called ("FrameCountAsFunctionDemonstrator"), a flag that indicates a method and not a property is being called, the instantiated object itself, the parameter array and the parameter modifiers.

The return value from the `InvokeMember()` call is always an object but we can cast it to whatever form we wish. The contents of the parameter array have to be examined to obtain the value of the output `ref` parameter. This too is an object – in the above code section, note the careful Type testing of that parameter value before it is used.

Of course, implicit in the above discussion is the fact that the name and signature of the method being called is known. Usually you will know this, (just not in a way that can be used at compile time), otherwise why would you be making the call in the first place?

INFORMATION EXCHANGE VIA COM AND MARSHAL

COM is designed to enable communication between two objects loaded at runtime. If the C# based Transform was loaded via COM, and the client application is not coded in C#, information can still be exchanged between the two via Marshal and other standard COM methods. A discussion of how a non C# client might exchange information with a C# Transform is beyond the scope of this book and will not be discussed further. If you need this information, it should be possible to find on-line sources which explain the process of connecting to a .NET assembly via COM Interop.

Windows Media Foundation: Getting Started in C#

Chapter 10

CAPTURING CAMERA DATA

One of the many useful things you may wish to do with Windows Media Foundation is capture video data (and possibly audio data) to a screen or a file. As with most things in WMF, there are multiple approaches which can be taken. This section will document two methods, one which uses the Reader-Writer Architecture and a second which uses a Hybrid Architecture. It is certainly possible to capture video data to a file using a pure Pipeline Architecture, but there is no specific Tanta Sample Project illustrating that. The Pipeline part of the *TantaCaptureToScreenAndFile* Hybrid Architecture demonstrates the process of capturing video data and rendering it to a screen and the Hybrid part of the architecture can also save those images to disk using a Sink Writer.

It should also be noted, in the interests of simplicity, that none of the Tanta Sample Projects capture audio data along with the video – although this feature could certainly be added to either of the two examples above. See any of the “Video File Copy” Tanta Sample Projects for insights into how multiple streams can be added to any of the architectures.

TIMESTAMP REBASING

Before we have a look at the nuts-and-bolts of video data capture, let's undertake a brief digression to discuss timestamps. As you will recall from previous sections, Media Samples are the containers used to transport the media data around the system – no matter which architecture is in use. Every Media Sample contains a timestamp that indicates when, relative to the start of play, the sample should be rendered. The Media Sample also contains a duration which indicates how long the Media Sample should be rendered for.

Typically, when video devices such as webcams provide Media Samples, the timestamp they use will probably just be a number indicating the current system date and time when the frame was composed. File based media data (such as MP4 files) usually expect the timestamp of the first frame they contain to start at zero and the timestamp of every subsequent frame to be incremented upwards based on that.

Do you see where we are going with this? If you are capturing data, ideally, the first Media Sample written to a file should be given a timestamp of 0 and every subsequent Media Sample should have its timestamp adjusted relative to that. In other words, the algorithm looks like...

```
Is this the first Media Sample being written to the file?
if Yes
    save the timestamp of the first Media Sample in a class variable
if No
    do nothing

// adjust the timestamp of all Media Samples
newTimestamp = existingTimestamp-firstMediaTimestamp

Write out the Media Sample with the adjusted timestamp.
```

This is called Timestamp Rebasing; all Media Samples will be rebased relative to zero - no matter what value the webcam used to start the sequence.

Timestamp Rebasing seems to have been more of an issue in the past than it is today. The MP4 File Sink (which is the Media Sink most often used to record video data) now appears to perform Timestamp Rebasing automatically and the above procedure is redundant. Both of the Tanta Sample Projects which write webcam data to MP4 files, implement Timestamp Rebasing for demonstration purposes and then either leave it commented out or as an option. This does not seem to cause the MP4 File Sink any problems. It should be noted that the various Tanta Sample Projects which copy media data from one file to another have no need for Timestamp Rebasing – presumably the first sample out of the input file already has a timestamp of 0 and so that is what gets written to the output file.

CAPTURE WITH A READER-WRITER ARCHITECTURE

Video capture with the Reader-Writer Architecture is a fairly straightforward process. A

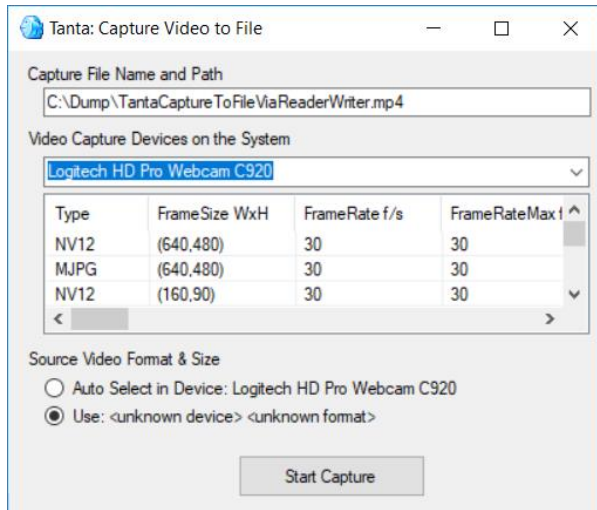


Figure 10.1: The *TantaCaptureToFileViaReaderWriter* Sample Project

Source Reader is setup to supply the video data from a webcam and a Sink Writer is configured to write it to disk and, as usual, you write a loop to pull the data off the source and give it to the sink.

The *TantaCaptureToFileViaReaderWriter* demonstrates this method.

Since it is desirable to keep the screen GUI operational during the capture process, the Asynchronous Mode Reader-Writer Architecture was used in the *TantaCaptureToFileViaReaderWriter*

Sample Project. In reality, Synchronous Mode could also have been used (and would have been simpler) if the Media Sample processing loop was just spun up in a separate C# thread. However, doing so would have eliminated the opportunity to demonstrate the usage of the Asynchronous Mode Reader-Writer Architecture in a Tanta Sample.

As can be seen in the screen shot above, the *TantaCaptureToFileViaReaderWriter* project uses the *ctlTantaVideoPicker* control from the *TantaCommon* library to provide the user with the means to choose the webcam to capture and the format to store. The output file name is specified at the top of the form and a simple button begins the capture from that camera.

SETTING THE OUTPUT MEDIA TYPE ON THE SOURCE READER

Let's join the capture process in the *CaptureToFile()* function of the *frmMain* class. Most of this code is the pretty standard Reader-Writer Architecture set-up and so we will not reproduce that discussion here. See the *Implementing the Reader-Writer Architecture* section of the *Practical WMF Architectures* chapter for more details. However the code does get interesting at the point where the output format of the Source Reader object is configured. As can be seen in the above image, the user has the option of selecting a specific Media Type and format or having a default one chosen for it from the available Media Types.

Capturing Camera Data

```
if (radioButtonUseSpecified.Checked == true)
{
    // we saved the video format container here - this is just the last one that came in
    if ((radioButtonUseSpecified.Tag == null)
        || ((radioButtonUseSpecified.Tag is TantaMFVideoFormatContainer)==false))
    {
        MessageBox.Show("No source video device and format selected. Cannot continue.");
        return;
    }
    // get the container
    TantaMFVideoFormatContainer videoFormatCont =
        (radioButtonUseSpecified.Tag as TantaMFVideoFormatContainer);

    // configure the Source Reader to use this format
    hr = TantaWMFUtils.ConfigureSourceReaderWithVideoFormat(
        workingSourceReader,
        videoFormatCont);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        MessageBox.Show("Failed on Configure VideoFormat (a), retVal=" + hr.ToString());
        return;
    }
}
else
{
    // prepare a list of subtypes we are prepared to accept from the video source
    // device. These will be tested in order - the first match will be used.
    List<Guid> subTypes = new List<Guid>();
    subTypes.Add(MFMediaType.NV12);
    subTypes.Add(MFMediaType.YUY2);
    subTypes.Add(MFMediaType.UYVY);
    subTypes.Add(MFMediaType.RGB32);
    subTypes.Add(MFMediaType.RGB24);
    subTypes.Add(MFMediaType.IYUV);

    // make sure the default Media Type is one of the above video formats
    hr = TantaWMFUtils.ConfigureSourceAsyncReaderWithVideoFormat (
        workingSourceReader, subTypes, false);
    if (hr != HRESULT.S_OK)
    {
        // we failed
        MessageBox.Show("Failed on test VideoFormat (b), retVal=" + hr.ToString());
        return;
    }
}

// if we get here we know the source reader now has a configured format but we might not
// know which one it is. So we ask it. It will return a video type
// we will use this later on to configure our sink writer. Note, we have to properly dispose
// of the videoType object after we use it.
hr = workingSourceReader.GetCurrentMediaType(
    TantaWMFUtils.MF_SOURCE_READER_FIRST_VIDEO_STREAM, out videoType);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed on call to GetCurrentMediaType, retVal=" + hr.ToString());
}

Source: TantaCaptureToFileViaReaderWriter::frmMain::CaptureToFile
```

If the user chooses a specific format we will obtain this information from the radioButton in a TantaMFVideoFormatContainer.

```
if (radioButtonUseSpecified.Checked == true)
{
    // we saved the video format container here - this is just the last one that came in
    if ((radioButtonUseSpecified.Tag == null)
        || ((radioButtonUseSpecified.Tag is TantaMFVideoFormatContainer)==false))
    {
        ... more code
    }
}
```

This container does not contain the Media Type object itself – just a variety of information on the Media Type and format. We pass this into the

ConfigureSourceReaderWithVideoFormat static function located in the TantaWMFUtils class.

```
// configure the Source Reader to use this format
hr = TantaWMFUtils.ConfigureSourceReaderWithVideoFormat(workingSourceReader, videoFormatCont);
```

A method of configuring a Source Reader with a specified set of media type details should probably be documented for you and so it is reproduced below.

```
public static HRESULT ConfigureSourceReaderWithVideoFormat(
    IMFSourceReaderAsync sourceReader,
    TantaMFVideoFormatContainer videoFormatContainer)
{
    HRESULT hr = HRESULT.S_OK;
    IMFMediaType mediaTypeObj = null;
    // this seems a reasonable maximum
    int maxFormatsToTestFor = 100;

    if (sourceReader == null)
    {
        // we failed
        throw new Exception("Cno reader supplied");
    }

    if (videoFormatContainer == null)
    {
        // we failed
        throw new Exception("no video format container supplied");
    }

    try
    {
        // the code below loops through all media types in the sourceReader.
        // It converts them to a temporary TantaMFVideoFormatContainer. Once
        // we have that we compare it against the input container. If we get
        // a match we set the source reader to use tha media type.

        // loop through a reasonable number of mediaTypes looking for a match
        for (int typeIndex = 0; typeIndex < maxFormatsToTestFor; typeIndex++)
        {
            // get the next media type
            hr = sourceReader.GetNativeMediaType(
                TantaWMFUtils.MF_SOURCE_READER_FIRST_VIDEO_STREAM,
                typeIndex, out mediaTypeObj);
            if (hr == HRESULT.MF_E_NO_MORE_TYPES)
            {
                // we are all done. The outSb container has been populated
                return HRESULT.S_OK;
            }
            else if (hr != HRESULT.S_OK)
            {
                // we failed
                throw new Exception("failed GetNativeMediaType, retVal=" + hr.ToString());
            }

            // get a temporary format container from the media type
            TantaMFVideoFormatContainer tmpContainer =
                TantaMediaTypeInfo.GetVideoFormatContainerFromMediaTypeObject(
                    mediaTypeObj, videoFormatContainer.VideoDevice);
            if (tmpContainer == null)
            {
                // we failed
                throw new Exception("failed GetVideoFormatContainerFromMediaTypeObject");
            }

            // does this container match the one that was passed in?
            if (videoFormatContainer.CompareTo(tmpContainer) == 0)
            {
                // yes it matches. This is the format that was specified.
                // We can configure with this. Sset the media type on the reader
                hr = sourceReader.SetCurrentMediaType(
                    TantaWMFUtils.MF_SOURCE_READER_FIRST_VIDEO_STREAM, null, mediaTypeObj);
                if (hr != HRESULT.S_OK)
                {
                    // we failed
                    throw new Exception("failed SetCurrentMediaType, retVal=" + hr.ToString());
                }
            }
        }
    }
}
```

Capturing Camera Data

```
    }

    // release the media type
    if (mediaTypeObj != null)
    {
        Marshal.ReleaseComObject(mediaTypeObj);
        mediaTypeObj = null;
    }
    // we are done
    return HRESULT.S_OK;
}

// if we get here we failed, we could not match the input format
return HRESULT.E_FAIL;
}
finally
{
}
}
```

Source: TantaCommon::TantaWMFUtils::ConfigureSourceReaderWithVideoFormat

In the above code we are greatly assisted by the fact that the TantaMFVideoFormatContainer container was created (by the ctlTantaVideoPicker control) from a list of Media Types the webcam said it supports. We will not discuss every detail of the above code block – the operation should be fairly clear. All we need to do is look at each Media Type the Source Reader supports

```
// loop through a reasonable number of mediaTypes looking for a match
for (int typeIndex = 0; typeIndex < maxFormatsToTestFor; typeIndex++)
{
    ... more code
}
```

For every Media Type we find we create a new TantaMFVideoFormatContainer from it and compare the two.

```
TantaMFVideoFormatContainer tmpContainer =
    TantaMediaTypeInfo.GetVideoFormatContainerFromMediaTypeInfo(
        mediaTypeObj, videoFormatContainer.VideoDevice);

// does this container match the one that was passed in?
if (videoFormatContainer.CompareTo(tmpContainer) == 0)
{
    ... more code
}
```

When we find a match we simply set it “current” on the stream as shown below.

```
// yes it matches. This is the format that was specified.
// We can configure with this. Set the media type on the reader
hr = sourceReader.SetCurrentMediaType(
    TantaWMFUtils.MF_SOURCE_READER_FIRST_VIDEO_STREAM, null, mediaTypeObj);
... more code
```

Once the above code completes, we have chosen the output format of the Source Reader object. It should be noted that we are using the MF_SOURCE_READER_FIRST_VIDEO_STREAM constant and are only considering the first video stream the Source Reader offers. This may not be desirable - some Webcams offer more than one stream. For example, the Logitech C920 webcam offers a second (disabled by default) stream containing H.264 compressed video streams. The above code would totally ignore that.

Returning to the `CaptureFile()` function we turn our attention to the sequence of operations which unfold when the user simply choses to use the default Media Type the webcam provides. In this case we decide (for demonstration purposes) that only a few Media Sub-Types will be acceptable and we test to see if the default Media Type matches any one of that supplied list. If the default Media Type on the stream does not match any of the types on the list, then we throw an error. Here is the setup in the `CaptureFile()` function just before the test happens.

```
// prepare a list of subtypes we are prepared to accept from the video source
// device. These will be tested in order - the first match will be used.
List<Guid> subTypes = new List<Guid>();
subTypes.Add(MFMediaType.NV12);
subTypes.Add(MFMediaType.YUY2);
subTypes.Add(MFMediaType.UYVY);
subTypes.Add(MFMediaType.RGB32);
subTypes.Add(MFMediaType.RGB24);
subTypes.Add(MFMediaType.IYUV);

// make sure the default Media Type is one of the above video formats
hr = TantaWMFUtils.ConfigureSourceAsyncReaderWithVideoFormat(
    workingSourceReader, subTypes, false);

... more code
```

The checking is also done with a call to a `ConfigureSourceAsyncReaderWithVideoFormat` static function (it uses different parameters than the previous one) located in the `TantaWMFUtils`. In the interests of space we will not reproduce all of the details of this function – just the interesting bits.

```
hr = sourceReader.GetNativeMediaType(
    TantaWMFUtils.MF_SOURCE_READER_FIRST_VIDEO_STREAM, 0, out mediaTypeObj);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("failed on call to GetNativeMediaType, retVal=" + hr.ToString());
}

// Get the GUID value associated with a MF_MT_SUBTYPE key.
hr = mediaTypeObj.GetGUID(MFAttributes.Clsid.MF_MT_SUBTYPE, out subtype);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("failed on call to mediaTypeObj.GetGUID, retVal=" + hr.ToString());
}

// now loop through and check, does this match any of the ones we want?
foreach (Guid guidValue in subTypes)
{
    if (subtype == guidValue)
    {
        // set the media type on the reader
        hr = sourceReader.SetCurrentMediaType(
            TantaWMFUtils.MF_SOURCE_READER_FIRST_VIDEO_STREAM, null, mediaTypeObj);
        // flag it
        matchedNativeType = true;
        break;
    }
}
if (matchedNativeType == false)
{
    // if we get here we failed
    return HRESULT.E_FAIL;
}

return HRESULT.S_OK;

Source: TantaCommon::TantaWMFUtils::TestSourceReaderStreamUsesFormat
```

Each item in the list is the GUID of a Media Sub-type. We get the native Media Type from the Source Reader and extract its Media Sub-Type as a GUID.

```
hr = sourceReader.GetNativeMediaType(  
    TantaWMFUtils.MF_SOURCE_READER_FIRST_VIDEO_STREAM, 0, out mediaTypeObj);  
  
// Get the GUID value associated with a MF_MT_SUBTYPE key.  
hr = mediaTypeObj.GetGUID(MFAttributesClsid.MF_MT_SUBTYPE, out subtype);  
  
... more code
```

Once we have that it is a simple matter to check for a match against our list and then set the current Media Type on the Source Reader with a call to `SetCurrentMediaType()`.

```
hr = sourceReader.SetCurrentMediaType(  
    TantaWMFUtils.MF_SOURCE_READER_FIRST_VIDEO_STREAM, null, mediaTypeObj);  
  
... more code
```

As before, once the above code completes we have chosen the output format of the Source Reader object. It should be noted that here too we are using the `MF_SOURCE_READER_FIRST_VIDEO_STREAM` constant and are only considering the first video stream the Source Reader offers.

It should also be mentioned that, since we might not explicitly know the Media Type that has been set, we simply ask the Source reader for it again in the `CaptureFile()` function. This is done with the statement below

```
// if we get here we know the source reader now has a configured format but we might not  
// know which one it is. So we ask it. It will return a video type  
// we will use this later on to configure our sink writer. Note, we have to properly dispose  
// of the videoType object after we use it.  
hr = workingSourceReader.GetCurrentMediaType(  
    TantaWMFUtils.MF_SOURCE_READER_FIRST_VIDEO_STREAM, out videoType);
```

CONFIGURING AN MP4 SINK WRITER

Proceeding on down the `CaptureFile()` function in the `frmMain` class of the *TantaCaptureToFileViaReaderWriter* Sample Project, the next step is to properly configure the Media Type the Sink Writer will write to disk. We are explicitly specifying the output format here and the code below provides a nice example of a Media Type being created from scratch. Well, mostly from scratch – some parts, such as the frame size and interlace mode, are obtained from the output Media Type of the source stream. We definitely do not want to change those things. This code section will run a bit long – but you should probably have an example of how this kind of Media Type object build is done somewhere in this book.

```
// now we configure the encoder. This sets up the sink writer so that it knows what format  
// the output data should be written in. The format we give the writer does not  
// need to be the same as the format it outputs to disk - however to make life  
// easier for ourselves we will copy a lot of the settings from the videoType retrieved above  
  
// create a new empty media type for us to populate  
hr = MFExtern.MFCreateMediaType(out encoderType);
```



```

if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed on call to MFCreateMediaType, retVal=" + hr.ToString());
}

// The major type defines the overall category of the media data. Major types
// include video, audio, script & etc.
hr = encoderType.SetGUID(MFAttributesClsid.MF_MT_MAJOR_TYPE, MFMediaType.Video);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed setting the MF_MT_MAJOR_TYPE, retVal=" + hr.ToString());
}

// The subtype GUID defines a specific media format type within a major type.
// For example, within video, the subtypes include MFMediaType.H264 (MP4),
// MFMediaType.WMV3 (WMV), MJPEG & etc. Within audio, the
// subtypes include PCM audio, Windows Media Audio 9, & etc.
hr = encoderType.SetGUID(MFAttributesClsid.MF_MT_SUBTYPE, MEDIA_TYPERETO_WRITE);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed setting the MF MT SUBTYPE, retVal=" + hr.ToString());
}

// this is the approximate data rate of the video stream, in bits per second,
// for a video media type in the MF.Net sample code this is 240000 but I
// found 2000000 to be much better. I am not sure,
// at this time, how this value is derived or what the tradeoffs are.
hr = encoderType.SetUINT32(MFAttributesClsid.MF_MT_AVG_BITRATE, TARGET_BIT_RATE);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed setting the MF_MT_AVG_BITRATE, retVal=" + hr.ToString());
}

// populate our new encoding type with the frame size of the videoType selected earlier
hr = TantaWMFUtils.CopyAttributeData(videoType, encoderType,
    MFAttributesClsid.MF_MT_FRAME_SIZE);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed copying the MF MT FRAME SIZE, retVal=" + hr.ToString());
}

// populate our new encoding type with the frame rate of the video type selected earlier
hr = TantaWMFUtils.CopyAttributeData(videoType, encoderType,
    MFAttributesClsid.MF_MT_FRAME_RATE);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed copying the MF_MT_FRAME_RATE, retVal=" + hr.ToString());
}

// populate our new encoding type with the pixel aspect ratio of
// the video type selected earlier
hr = TantaWMFUtils.CopyAttributeData(videoType, encoderType,
    MFAttributesClsid.MF_MT_PIXEL_ASPECT_RATIO);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed MF MT PIXEL ASPECT_RATIO, retVal=" + hr.ToString());
}

// populate our new encoding type with the interlace mode of the video type selected earlier
hr = TantaWMFUtils.CopyAttributeData(videoType, encoderType,
    MFAttributesClsid.MF_MT_INTERLACE_MODE);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed copying the MF_MT_INTERLACE_MODE, retVal=" + hr.ToString());
}

// add a stream to the sink writer. The encoderType specifies the format
// of the samples that will be written to the file. Note that it does not necessarily need to
// match the input format.
int sinkStream = 0;
hr = workingSinkWriter.AddStream(encoderType, out sinkStream);
if (hr != HRESULT.S_OK)
{

```

Capturing Camera Data

```
// we failed
throw new Exception("Failed adding the output stream, retVal=" + hr.ToString());
}

Source: TantaCaptureToFileViaReaderWriter::frmMain::CaptureToFile
```

The code in the above section can be basically be broken down in to three parts: the creation of the Media Type Object, the population of the Media Type object with new items and the population of the Media Type Object with media information derived from the output Media Type of the Source Reader. Let's take each one in turn.

The Media Type is easily created with a call to the `MFCreatMediaType` static function.

```
// create a new empty media type for us to populate
hr = MFExtern.MFCreatMediaType(out encoderType);
```

We set the Media Major Type, the Media Sub-Type and the Target Bit Rate in the output media type.

```
// The major type is video
hr = encoderType.SetGUID(MFAttributesClsid.MF_MT_MAJOR_TYPE, MFMediaType.Video);

// The subtype GUID defines a specific media format type within a major type.
hr = encoderType.SetGUID(MFAttributesClsid.MF_MT_SUBTYPE, MEDIA_TYPTO_WRITE);

// this is the approximate data rate of the video stream, in bits per second
hr = encoderType.SetUINT32(MFAttributesClsid.MF_MT_AVG_BITRATE, TARGET_BIT_RATE);
```

The `MEDIA_TYPTO_WRITE` value is a constant which is equal to `MFMediaType.H264` in this application. The H.264 format (which you can find out about online) is a nice, standard, compressed format for MP4 files. You will note that the Media Sub-Type of the samples being output by the Source Reader is almost certainly not H.264 and so some converting will have to be done. As we will shortly see, the Sink Writer will handle all of this sort of thing for us.

We copy very specialized things like the frame size, frame rate, interlace mode and pixel aspect ratio from the Source Readers Media Type. There is, unless we really want to do some serious converting, no need to change these in the resulting MP4 file.

```
// populate our new encoding type with the frame size of the videoType selected earlier
hr = TantaWMFUtils.CopyAttributeData(videoType, encoderType,
    MFAttributesClsid.MF_MT_FRAME_SIZE);

// populate our new encoding type with the frame rate of the video type selected earlier
hr = TantaWMFUtils.CopyAttributeData(videoType, encoderType,
    MFAttributesClsid.MF_MT_FRAME_RATE);

// populate our new encoding type with the pixel aspect ratio of
// the video type selected earlier
hr = TantaWMFUtils.CopyAttributeData(videoType, encoderType,
    MFAttributesClsid.MF_MT_PIXEL_ASPECT_RATIO);

// populate our new encoding type with the interlace mode of the video type selected earlier
hr = TantaWMFUtils.CopyAttributeData(videoType, encoderType,
    MFAttributesClsid.MF_MT_INTERLACE_MODE);
if (hr != HRESULT.S_OK)
```

There are likely to be a lot more Attributes in the Source Reader Media Type that we could copy across but the ones specified above seem to do the job. It should be noted

that there is an alternate way this procedure could be done. The `IMFMediaType` interface does contain a `CopyAllItems()` function and this is wrapped up nicely in the `CloneMediaType()` function of the `TantaWMFUtils` class. We could have just cloned the Source Reader Media Type and replaced the Media Sub-Type and Frame Rate Attributes in the cloned value.

Once we have the new Media Type built, we tell the Sink Writer to use it as an output. This is done as part of the act of the output stream.

```
// add a stream to the sink writer. The encoderType specifies the format
// of the samples that will be written to the file. Note that it does not necessarily need to
// match the input format.
int sink_stream = 0;
hr = workingSinkWriter.AddStream(encoderType, out sink_stream);
```

Note that we carefully record the ID of the output stream in the above call – we will need it in the next step.

At this point the Sink Writer has an output stream and that stream has a specified Media Type. We now have to tell the Sink Writer the format of the data it will be receiving on that stream. The Media Type of output by the Source Reader (the `videoType` variable) is used for this purpose.

```
// Set the input format for a stream on the sink writer.
hr = workingSinkWriter.SetInputMediaType(sink_stream, videoType, null);
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed SetInputMediaType on the writer, retVal=" + hr.ToString());
}

Source: TantaCaptureToFileViaReaderWriter::frmMain::CaptureToFile
```

Clearly the input Media Type on the Sink Writers stream is not identical to that being written to the file. The Sink Writer will automatically invoke a conversion Transform to make the appropriate changes. This is all automatic and is probably the primary reason people like the Sink Writer so much.

The last steps in the `CaptureToFile()` function initialize the Sink Writer and request the first sample from the Source Reader

```
// now we initialize the sink writer for writing.
hr = workingSinkWriter.BeginWriting();
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed BeginWriting on the writer, retVal=" + hr.ToString());
}

// Request the first video frame from the media source. The TantaSourceReaderCallbackHandler
// set up earlier will be invoked and it will continue requesting and processing video
// frames after that.
hr = workingSourceReader.ReadSample(
    TantaWMFUtils.MF_SOURCE_READER_FIRST_VIDEO_STREAM,
    0,
    IntPtr.Zero,
    IntPtr.Zero,
    IntPtr.Zero,
```

```
IntPtr.Zero
});
if (hr != HRESULT.S_OK)
{
    // we failed
    throw new Exception("Failed first ReadSample on the reader, retVal=" + hr.ToString());
}
}
Source: TantaCaptureToFileViaReaderWriter::frmMain::CaptureToFile
```

You have seen the `BeginWriting()` command before in other chapters – it just initializes the Sink Writer and opens the output file. The first `ReadSample()` call has also been discussed. Since the Source Reader in this example is operating in Asynchronous Mode, this call will trigger a Media Sample to be sent to the `OnReadSample()` function in the Source Readers Callback Object. Once the first Media Sample arrives, the read process will be self-sustaining in a separate thread. The Media Samples will be read from the Source Reader (the webcam) and written to the Sink Writer (the MP4 file) until the user stops the process. See *The Asynchronous Reader-Writer Architecture* section in the *Practical WMF Architectures* chapter for an extensive discussion of this process.

CAPTURE WITH A HYBRID ARCHITECTURE

The act of capturing data with a Hybrid Architecture implies that you have a Pipeline and that some object in that Pipeline is intercepting the Media Samples they pass through. In a capture application, copies of the intercepted Media Samples are then handed off to a Sink Writer while the Pipeline proceeds to render the video data.

The *TantaCaptureToScreenAndFile* Sample Project demonstrates the display and capture process. The application displays the stream of images from a webcam on a screen and also, optionally, allows the user to activate a “save to disk” operation which simultaneously records the video data on display to a file.

The capture process in *TantaCaptureToScreenAndFile* Sample Project is performed by introducing a special Transform into the Pipeline. The Transform, named `MFTTantaSampleGrabber_Sync` operates in an in-place Synchronous Mode Transform that makes no changes to the data as it passes through. The `OnProcessOutput()` simply hands the Media Sample off to a previously configured Sink Writer (it will only make a copy if it is rebasing the timestamp). When that call returns, the Transform gives the Media Sample back to the Media Session. Essentially, the `MFTTantaSampleGrabber_Sync` is just the example Frame Counter Transform with a few added extras so that the client can communicate with it to enable and disable the write of the Media Sample. Other than the addition of the `MFTTantaSampleGrabber_Sync` Transform (and that in itself is a pretty standard process for you by now), there is nothing at all unusual about the Pipeline in this

application. It is just a Pipeline from a webcam to an EVR Renderer which has a Transform in the video stream.

The `StartRecording()` function in the `MFTTantaSampleGrabber_Sync` Transform is called by the application out of a button press event. All it does is set up a Sink Writer in exactly the same way as you previously saw in the *Configuring an MP4 Sink Writer* section during the discussion of the *Capture with a Reader-Writer Architecture* operation. We will not duplicate that code here – its operation has already been extensively discussed. It should be noted that the presence or absence of a Sink Writer object in the Transform is the only thing that enables or disables the write to disk. If the Sink Writer object exists the Media Sample is given to it, if not the Transform just returns it as normal. The creation of the Sink Writer and the write to the Sink Writer necessarily happen in different threads so a standard C# `lock` is obtained to make sure the Sink Writer object does not disappear while the `OnProcessOutput()` function is using it.

The code in the `OnProcessOutput()` function is also quite simple and is reproduced below.

```

/// +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=
/// <summary>
/// This is the routine that performs the transform. Unless the sinkWriter object
/// is set all we do is pass the sample on. If the sink writer object is set
/// we give the sample to it for writing. There are two modes - one where we just
/// give the sinkwriter the input sample and the other where we clone the input
/// sample and rebase the timestamps.
///
/// An override of the abstract version in TantaMFTBase_Sync.
/// </summary>
/// <param name="pOutputSamples">The structure to populate with output values.</param>
/// <returns>S_Ok unless error.</returns>
/// <history>
///     01 Nov 18 Cynic - Originally written
/// </history>
protected override HRESULT OnProcessOutput(ref MFTOutputDataBuffer outputSampleDataStruct)
{
    HRESULT hr = HRESULT.S_OK;
    IMFMediaBuffer inputMediaBuffer = null;
    IMFSample sinkWriterSample = null;
    IMFAttributes sampleAttributes = null;
    long sampleDuration = 0;
    int sampleSize = 0;
    long sampleTimeStamp = 0;
    int sampleFlags = 0;

    // in this MFT we are processing in place, the input sample is the output sample,
    // the media buffer of the input sample is the media buffer of the output sample.
    // That's for the pipeline. If a sink writer exists we also write the sample data
    // out to the sink writer. This provides the effect of displaying on the
    // screen and simultaneously recording.

    // There are two ways the sink writer can be given the media sample data. It can
    // just be given the input sample directly or a copy of the sample can be made
    // and that copy given to the sink writer.

    // There is also an additional complication - the sample has a timestamp and
    // video cameras tend to just use the current date and time as a timestamp.
    // There are several reports that MP4 files need to have their first frame
    // starting at zero and then every subsequent frame adjusted to that
    // new base time. Certainly the Microsoft supplied example code (and see the
    // TantaCaptureToFileViaReaderWriter also) take great care to do this. This
    // requirement does not seem to exist - my tests indicate it is not necessary

```

Capturing Camera Data

```
// to start from 0 in the MP4 file. Maybe the Sink Writer has been improved
// and now does this automatically. For demonstration purposes the timebase-rebase
// functionality has been included and choosing that mode copies the sample
// and resets the time. If the user does not rebase the time we simply
// send the input sample to the Sink Writer as-is.

try
{
    // Set status flags.
    outputSampleDataStruct.dwStatus = MFTOutputDataBufferFlags.None;

    // The output sample is the input sample. We get a new IUnknown for the Input
    // sample since we are going to release it below. The client will release this
    // new IUnknown
    outputSampleDataStruct.pSample = Marshal.GetIUnknownForObject(InputSample);

    // are we recording?
    if (workingSinkWriter != null)
    {
        // we do everything in a lock
        lock (sinkWriterLockObject)
        {
            // are we in timebase rebase mode?
            if (wantTimebaseRebase == false )
            {
                // we are not. Just give the input sample to the Sink Writer which will
                // write it out.
                hr = workingSinkWriter.WriteSample(sinkWriterVideoStreamId, InputSample);
                if (hr != HRESULT.S_OK)
                {
                    throw new Exception("WriteSample(a) failed. Err=" + hr.ToString());
                }
            }
            else
            {
                // the timebase rebase option has been chosen. We need to
                // create a copy of the input sample so we can adjust the time on it.

                // Get the data buffer from the input sample.
                hr = InputSample.ConvertToContiguousBuffer(out inputMediaBuffer);
                if (hr != HRESULT.S_OK)
                {
                    throw new Exception("Convert failed. Err=" + hr.ToString());
                }

                // get some other things from the input sample
                hr = InputSample.GetSampleDuration(out sampleDuration);
                if (hr != HRESULT.S_OK)
                {
                    throw new Exception("GetSampleDuration failed. Err=" + hr.ToString());
                }
                hr = InputSample.GetTotalLength(out sampleSize);
                if (hr != HRESULT.S_OK)
                {
                    throw new Exception("GetTotalLength failed. Err=" + hr.ToString());
                }
                hr = InputSample.GetSampleTime(out sampleTimeStamp);
                if (hr != HRESULT.S_OK)
                {
                    throw new Exception("GetSampleTime failed. Err=" + hr.ToString());
                }

                // get the attributes from the input sample
                if (InputSample is IMFAttributes)
                    sampleAttributes = (InputSample as IMFAttributes);
                else sampleAttributes = null;

                // we have all the information we need to create a new output sample
                sinkWriterSample = TantaWMFUtils.CreateMediaSampleFromBuffer(
                    sampleFlags, sampleTimeStamp,
                    sampleDuration, inputMediaBuffer,
                    sampleSize, sampleAttributes);
                if (sinkWriterSample == null)
                {
                    throw new Exception("sinkWriterSample == null");
                }

                // we have a sample, if so is it the first non null one?
                if (isFirstSample)
                {

```

```

        // yes it is set up our timestamp
        firstSampleBaseTime = sampleTimeStamp;
        isFirstSample = false;
    }

    // rebase the time stamp
    sampleTimeStamp -= firstSampleBaseTime;

    hr = sinkWriterSample.SetSampleTime(sampleTimeStamp);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("SetSampleTime failed. Err=" + hr.ToString());
    }

    // write the sample out
    hr = workingSinkWriter.WriteSample(
        sinkWriterVideoStreamId, sinkWriterSample);
    if (hr != HRESULT.S_OK)
    {
        throw new Exception("WriteSample(b) failed. Err=" + hr.ToString());
    }
}

}
}
finally
{
    // clean up
    if (inputMediaBuffer != null)
    {
        Marshal.ReleaseComObject(inputMediaBuffer);
        inputMediaBuffer = null;
    }

    if (sinkWriterSample != null)
    {
        Marshal.ReleaseComObject(sinkWriterSample);
        sinkWriterSample = null;
    }

    // Release the current input sample so we can get another one.
    // the act of setting it to null releases it because the property
    // is coded that way
    InputSample = null;
}

return HRESULT.S_OK;
}

```

Source: `TantaCaptureToScreenAndFile:MFTTantaSampleGrabber_Sync::OnProcessOutput`

The lead in to the `OnProcessOutput` function simply performs the standards actions required for the return – the flags are set and the `InputSample` value is given a new reference and placed in the output structure so it can be returned.

```

// Set status flags.
outputSampleDataStruct.dwStatus = MFTOutputDataBufferFlags.None;

// The output sample is the input sample. We get a new IUnknown for the Input
// sample since we are going to release it below. The client will release this
// new IUnknown
outputSampleDataStruct.pSample = Marshal.GetIUnknownForObject(InputSample);

... more code

```

If you are unfamiliar with the reasons behind the above two lines of code you should review the *Raw Data Handling in the Transform* section in the *Working With Transforms* chapter.

Next, if the Sink Writer exists, we write out the data. Let's take the simple case in which there is no timebase rebasing needed.

```
// give the input sample to the Sink Writer which will write it out.  
hr = workingSinkWriter.WriteSample(sinkWriterVideoStreamId, InputSample);
```

Impressively simple isn't it. We have a Sink Writer and we simply give the `InputSample` to it and it does the work of writing the data to the MP4 file.

If the user does implement the time-base rebase option, then we have a little more work to do. We cannot just update the timestamp in the existing `InputSample` sample because that would totally mess up the Enhanced Video Renderer at the end of the Pipeline. The EVR needs a consistent sequence of timestamps in the Media Samples it displays – remember the user can turn the “Record to Disk” option on and off but the video continuously displays on the form.

Since the `InputSample` Media Sample must be given back to the Media Session unchanged, if we are going to give one to the Media Sink with a different timestamp then we shall have to make a copy. This process is documented in the section below.

CREATING A COPY OF A MEDIA SAMPLE

Continuing on from the previous discussion, the first act of building a new Media Sample is to get the underlying Media Buffer.

```
// Get the data buffer from the input sample.  
hr = InputSample.ConvertToContiguousBuffer(out inputMediaBuffer);
```

Nothing too amazing there – we have seen this before. Next we collect some of the important characteristics of the `InputSample` object into temporary variables.

```
// get some other things from the input sample  
hr = InputSample.GetSampleDuration(out sampleDuration);  
hr = InputSample.GetTotalLength(out sampleSize);  
hr = InputSample.GetSampleTime(out sampleTimeStamp);  
hr = InputSample.ConvertToContiguousBuffer(out inputMediaBuffer);
```

We also get the Attributes. Since the `IMFSample` interface inherits from `IMFAttributes` `Input Sample` is an Attribute Container. We store this value too...

```
// get the attributes from the input sample  
if (InputSample is IMFAttributes) sampleAttributes = (InputSample as IMFAttributes);  
else sampleAttributes = null;;
```

... and then we create the Media Sample with a call to the static `CreateMediaSampleFromBuffer` function in the `TantaWMFUtils` class.

```
// we have all the information we need to create a new output sample  
sinkWriterSample = TantaWMFUtils.CreateMediaSampleFromBuffer(sampleFlags, sampleTimeStamp,  
    sampleDuration, inputMediaBuffer,  
    sampleSize, sampleAttributes);
```

The operation of the `CreateMediaSampleFromBuffer` function was discussed in the *Creating a New Media Sample* section of *The WMF Components* chapter and we will

not reproduce that information here. Ultimately the information provided just gives us a new Media Sample and we can adjust the sample time...

```
// rebase the time stamp
sampleTimeStamp -= firstSampleBaseTime;

hr = sinkWriterSample.SetSampleTime(sampleTimeStamp);
```

... and write the copy of the InputSample Media Sample out to the Sink Writer as before.

```
// write the sample out
hr = workingSinkWriter.WriteSample(sinkWriterVideoStreamId, sinkWriterSample);
```

Other than the copy of the Media Sample (which we only had to do because we were rebasing the time), the entire process is not very difficult to follow.

The various WMF entities that we accumulated during our processing need to be released.

```
// clean up
if (inputMediaBuffer != null)
{
    Marshal.ReleaseComObject(inputMediaBuffer);
    inputMediaBuffer = null;
}

if (sinkWriterSample != null)
{
    Marshal.ReleaseComObject(sinkWriterSample);
    sinkWriterSample = null;
}

// Release the current input sample so we can get another one.
// the act of setting it to null releases it because the property
// is coded that way
InputSample = null;
```

As usual, we also null out the InputSample, if a Media Sample was present the base class will release it at that time.

It should be noted that the above method using a Transform would probably not be the typical method of simultaneously displaying video data and writing it to disk. A more standard method would be to implement a “tee” Transform and have two branches coming off of that – each with identical Media Samples. The EVR would sit at the end of one branch and a Sample Grabber Sink at the end of the other. However, this architecture would make it hard to turn the recording functionality on and off as the *TantaCaptureToScreenAndFile* application does. Also, you have already seen the Sample Grabber Sink in action in the Tanta File Copy Sample Projects and so there is not much more to new information that can be provided there.

Windows Media Foundation: Getting Started in C#

Appendix I

THE TANTA SAMPLE CODE

The MF.Net library provides a collection of C# sample projects – these are, in general, a pretty faithful port of the WMF C++ samples and they do demonstrate most of the fundamental techniques. The major problem with them is that they are structured in the C++ way and that way is not super intuitive or familiar to C# users. The MF.Net sample code is, fairly complex, re-entrant with multiple classes per file and some of them even use “old school” C++ mechanisms which put messages on the Windows Message Pump to send notifications and events. None of this, including the traditional C++ sparsity of comments, really serves to assist a person new to the technology in learning Windows Media Foundation in C#.

In order to provide demonstrations of various WMF techniques for this book, a new set of sample programs have been written. Some, such as the *TantaCaptureToFileViaReaderWriter*, are just re-factored and commented versions of the MF.Net examples. Others, such as the *TantaCaptureToScreenAndFile* contain techniques not demonstrated anywhere.

There are 15 Tanta Sample Projects and each serves to illustrate a WMF concept. In addition, there are many more comments in the code, there is only one class per file and any event mechanisms use the far simpler C# Delegate/Event system. This chapter

will provide a summary description of each Tanta Sample – the comments and `readme.txt` file associated with each project will provide much more information.

It should be noted that the much of the useful, common code has been factored out into a utility library named *TantaCommon*. This library is intended to provide a resource which can be included in a C# WMF.net application in order to provide supporting functionality. This library also contains useful controls which can do things like pick a video format or act as a self-contained video display engine.

As you read through the various sections in this book you will see code blocks (in a blue background) which demonstrate various concepts. The C# source in these code blocks are all taken from the Tanta Sample Applications and the bottom of each code block will contain a reference to the project and file from which the code was derived.

```
Source: TantaCommon::ctlTantaEVRFilePlayer::CreateOutputNode
```

DOWNLOADING THE TANTA SAMPLE PROJECTS

The Tanta Sample Projects are open source and released under the MIT License. It should be noted that some parts of the code in the Tanta Sample Projects are based on the MF.Net samples and that code, in turn, is derived from the original Microsoft samples. These have been placed in the public domain without copyright.

You can download, clone or fork the Tanta Sample Projects at the following address:

<https://github.com/OfItselfSo/Tanta>

A selection of sample MP3 and MP4 files are available at the root of the Tanta Sample Application repository. All of the Tanta Sample Projects expect, by default, to read and write to a directory named "C:\Dump". You can save yourself a bit of typing if you create that directory and copy the sample files there.

TANTA SAMPLE APPLICATIONS

TantaAudioFileCopyViaPipelineAndWriter – Demonstrates the Hybrid Architecture by copying a single stream (audio) MP3 file. This application may not work on Windows 7 due to codec unavailability.

TantaAudioFileCopyViaPipelineMP3Sink – Demonstrates the Pipeline Architecture by copying a single stream (audio) MP3 file.

TantaAudioFileCopyViaReaderWriter – Demonstrates the Reader-Writer Architecture by copying a single stream (audio) MP3 file. This application may not work on Windows 7 due to codec unavailability.

The Tanta Sample Code

TantaCaptureToFileViaReaderWriter – Uses a Reader-Writer Architecture to capture video directly to a file.

TantaCaptureToScreenAndFile – Uses a Hybrid Architecture to display video on the screen and, optionally, capture it to a file.

TantaFilePlaybackAdvanced – Uses the Pipeline Architecture to play a media file containing audio and video tracks. This application uses the `ctlTantaEVRFilePlayer` control from the *TantaCommon* library and demonstrates various Pipeline control mechanisms such as Pause, Fast-Forward, Jump Scrolling and volume control etc.

TantaFilePlaybackSimple - Uses the Pipeline Architecture to play a media file containing audio and video tracks. This application uses the `ctlTantaEVRStreamDisplay` control from the *TantaCommon* library to demonstrate simple audio and video playback with multiple streams.

TantaTransformDirect - Uses the Pipeline Architecture to demonstrate how to use the Tanta Transform Base classes to build and add Transforms to a Topology. This application contains Transforms which count the video frames, convert the image to grayscale or write text on the video display – both Synchronous and Asynchronous Mode Transforms are demonstrated.

TantaTransformInDLLClient – A Pipeline Architecture client which uses a DLL based Transform. This application also demonstrates various methods the client application can use to communicate with DLL based Transforms.

TantaTransformInDLL – A project which creates a Transform as a DLL and also, optionally, registers it on the system. The Transform in this application rotates or mirrors the video on display.

TantaTransformPicker – A project which uses the `ctlTantaTransformPicker` control in the *TantaCommon* library to enumerate and display the capabilities of the Transforms registered on the system.

TantaVideoFileCopyViaPipelineAndWriter – Demonstrates the Hybrid Architecture by copying a two stream (audio and video) MP4 file. This application may not work on Windows 7 due to codec unavailability.

TantaVideoFileCopyViaPipelineMP4Sink – Demonstrates the Pipeline Architecture by copying a two stream (audio and video) MP4 file.

TantaVideoFileCopyViaReaderWriter – Demonstrates the Reader-Writer Architecture by copying a two stream (audio and video) MP4 file. This application may not work on Windows 7 due to codec unavailability.

TantaVideoFormats – Uses the `ctlTantaVideoPicker` control from the *TantaCommon* library to show the video formats offered by the video capture devices (webcams) on the PC.

Windows Media Foundation: Getting Started in C#

Appendix II

CONVERTING BETWEEN C++ AND C# CODE

EXAMPLES

You will find help and sample code online for Windows Media Foundation in C++ and almost nothing for MF.Net and C#. Do not despair! It is usually quite possible to convert between the two fairly easily and pretty much any technique you might find which works in WMF with C++ will also work in MF.Net and C#. You just have to know how to translate.

In general if you see a C++ pointer call with an arrow operator (\rightarrow) like the one below...

```
float rate = 0;
BOOL bThin;
hr = pRateControl->GetRate(&bThin, &rate);
```

...you can rather easily replace it with an equivalent MF.Net call using a dot operator (.) like the one below...

```
float currentRate = 0;
bool isThinned = false;
hr = rateControlService.GetRate(ref isThinned, out currentRate);
```

Since all object variables are essentially references in C#, the translation from C++ pointer to C# object name works rather well. This is true too of the C++ & reference operator. In general, when you see the & reference operator you can just use a variable name for a value type and apply the `out` key word. You can see an example of this in the usage of the `currentRate` parameter above. It is interesting to note that, in the above example, the `isThinned` parameter is implemented as a `ref` and not an `out`. This is because even though it is a `bool` and a value type the same as the previous `float` variable, it is actually an in-out parameter on the WMF call. Admittedly though, it is not terribly obvious from looking at the C++ code that you have to treat it that way. Fortunately, there are not too many of these and a bit of playing about or looking at the source (one of the great things about open source) will soon put you right.

Here is a list of some of the things you might run across.

CODE CONVERSIONS IN GENERAL FUNCTION CALLS

GETTING A BOOL

C++ Prototype:

```
HRESULT GetStreamDescriptorByIndex(
    [in]  DWORD          dwIndex,
    [out] BOOL           *pfSelected,
    [out] IMFStreamDescriptor **ppDescriptor);
```

C++ Example:

```
BOOL fSelected = FALSE;
HRESULT hr = pPD->GetStreamDescriptorByIndex(iStream, &fSelected, &pSD);
```

MF.Net Prototype:

```
HResult GetStreamDescriptorByIndex(
    int dwIndex,
    out bool pfSelected,
    out IMFStreamDescriptor ppDescriptor);
```

MF.Net C# Example:

```
bool fSelected = false;
hr = sourcePD.GetStreamDescriptorByIndex(iStream, out fSelected, out pSourceSD);
```

SETTING A BOOL

C++ Prototype:

```
HRESULT SetMute([in] const BOOL bMute);
```

C++ Example:

```
BOOL wantMuted = FALSE;
pSimpleAudioService->SetMute(wantMuted);
```

Converting Between C++ and C# Code Examples

MF.Net Prototype:

```
HResult SetMute(bool bMute);
```

MF.Net C# Example:

```
bool wantMuted = true;  
hr = simpleAudioService.SetMute(wantMuted);
```

GETTING AN ENUM

C++ Prototype:

```
HRESULT GetUINT32([in] REFGUID guidKey, [out] UINT32 *punValue));  
  
HRESULT CreateObjectFromURL(  
    [in] LPCWSTR      pwszURL,  
    [in] DWORD        dwFlags,  
    [in] IPropertyStore *pProps,  
    [out] MF_OBJECT_TYPE *pObjectType,  
    [out] IUnknown     **ppObject  
);
```

C++ Example:

```
HRESULT hr = pEvent->GetUINT32(MF_EVENT_TOPOLOGY_STATUS, &status);  
  
HRESULT hr = pSourceResolver->CreateObjectFromURL(  
    sURL,                // URL of the source.  
    MF_RESOLUTION_MEDIASOURCE, // Create a source object.  
    NULL,                // Optional property store.  
    &ObjectType,          // Receives the created object type.  
    &pSource              // Receives a pointer to the media source.  
);
```

MF.Net Prototype:

```
HResult GetUINT32(Guid guidKey, out int punValue);  
  
HResult CreateObjectFromURL(  
    string pwszURL,  
    MFResolution dwFlags,  
    IPropertyStore pProps,  
    out MFObjectType pObjectType,  
    out object ppObject);
```

MF.Net C# Example:

```
int i;  
MFTopoStatus topoStatus = MFTopoStatus.Invalid; // this is an enum  
hr = pEvent.GetUINT32(MFAttributesClsid.MF_EVENT_TOPOLOGY_STATUS, out i);  
topoStatus = (MFTopoStatus)i;  
  
MFObjectType ObjectType = MFObjectType.Invalid; // enum  
hr = pSourceResolver.CreateObjectFromURL(  
    sURL,                // URL of the source.  
    MFResolution.MediaSource, // Create a source object.  
    null,                // Optional property store.  
    out ObjectType,      // Receives the created object type.  
    out pSource           // Receives a pointer to the media source.  
);
```

SETTING AN ENUM

C++ Prototype:

```
HRESULT MFCreateTopologyNode(  
    MF_TOPOLOGY_TYPE NodeType,  
    IMFTopologyNode **ppNode);
```

C++ Example:


```
IMFTopologyNode *pNode = NULL;
HRESULT hr = MFCreateTopologyNode(MF_TOPOLOGY_TRANSFORM_NODE, &pNode);
```

MF.Net Prototype:

```
public static HRESULT MFCreateTopologyNode(MFTopologyType NodeType,
    out IMFTopologyNode ppNode);
```

MF.Net C# Example:

```
IMFTopologyNode videoTransformNode = null;
HRESULT hr = MFExtern.MFCreateTopologyNode(MFTopologyType.TransformNode,
    out videoTransformNode);
```

GETTING A GUID

C++ Prototype:

```
HRESULT GetMajorType([out] GUID *pguidMajorType);
```

C++ Example:

```
GUID guidMajorType;
hr = pHandler->GetMajorType(&guidMajorType);
```

MF.Net Prototype:

```
HRESULT GetMajorType(out Guid pguidMajorType);
```

MF.Net C# Example:

```
Guid majorType = Guid.Empty;
hr = mediaTypeObj.GetMajorType(out majorType);
```

SETTING A GUID

C++ Prototype:

```
HRESULT MFGetService(
    IUnknown *punkObject,
    REFGUID guidService,
    REFIID riid,
    LPVOID *ppvObject
);
```

C++ Example:

```
MFGetService(session, MR_VIDEO_RENDER_SERVICE, IID_VideoDisplayControl,
    (void**)&service);
```

MF.Net Prototype:

```
HRESULT MFGetService(object punkObject, Guid guidService, Guid riid, out object
    ppvObject);
```

MF.Net C# Example:

```
hr = MFExtern.MFGetService(
    mediaSession,
    MFServices.MR_VIDEO_RENDER_SERVICE,
    typeof(IMFVideoDisplayControl).GUID,
    out evrVideoService
);
```

GETTING AN INT

C++ Prototype:

Converting Between C++ and C# Code Examples

```
HRESULT GetStreamDescriptorCount([out] DWORD *pdwDescriptorCount );
```

C++ Example:

```
DWORD cSourceStreams = 0;  
spPD->GetStreamDescriptorCount(&cSourceStreams);
```

MF.Net Prototype:

```
HResult GetStreamDescriptorCount(out int pdwDescriptorCount);
```

MF.Net C# Example:

```
int cSourceStreams = 0;  
hr = pSourcePD.GetStreamDescriptorCount(out cSourceStreams);
```

SETTING AN INT

C++ Prototypes:

```
HRESULT SetUINT32([in] REFGUID guidKey, [in] UINT32 unValue);
```

C++ Examples:

```
int bitrate;  
hr = pType2.SetUINT32(MFAttributesClsid.MF_MT_AVG_BITRATE, bitrate);
```

MF.Net Prototypes:

```
HResult SetUINT32(Guid guidKey, int unValue);
```

MF.Net C# Examples:

```
int targetBitRate;  
hr = encoderType.SetUINT32(MFAttributesClsid.MF_MT_AVG_BITRATE, targetBitRate);
```

GETTING AN INTPTR

C++ Prototype:

```
HRESULT GetCurrentImage(  
    [in, out] BITMAPINFOHEADER *pBih,  
    [out] BYTE **pDib,  
    [out] DWORD *pcbDib,  
    [in, out] LONGLONG *pTimeStamp  
);
```

C++ Example:

```
BITMAPINFOHEADER lpHeader = { 0 };  
BYTE* lpCurrImage = NULL;  
DWORD bitmapSize = 0;  
LONGLONG timestamp = 0;  
lpHeader.biSize = sizeof(BITMAPINFOHEADER);  
  
hr = pDisplay->GetCurrentImage(&lpHeader, &lpCurrImage, &bitmapSize, &timestamp));
```

MF.Net Prototype:

```
HResult GetCurrentImage(BitmapInfoHeader pBih,  
    out IntPtr pDib,  
    out int pcbDib,  
    out long pTimeStamp);
```

MF.Net C# Example:

```
BitmapInfoHeader workingBitmapInfoHeader = new BitmapInfoHeader();  
IntPtr bitmapData = IntPtr.Zero;  
int bitmapDataSize = 0;  
long bitmapTimestamp = 0;
```

```
workingBitmapInfoHeader.Size = Marshal.SizeOf(typeof(BitmapInfoHeader));

hr = evrVideoDisplay.GetCurrentImage(
    workingBitmapInfoHeader,
    out bitmapData,
    out bitmapDataSize,
    out bitmapTimestamp);

// bitmapData is an IntPtr. Use Marshal to copy the video data out
// into a byte array, bitmapDataSize is the length of bitmapData
byte[] managedArray = new byte[bitmapDataSize];
Marshal.Copy(bitmapData, managedArray, 0, bitmapDataSize);
```

GETTING A LONG:

C++ Prototype:

```
HRESULT GetCurrentImage(
    [in, out] BITMAPINFOHEADER *pBih,
    [out] BYTE **pDib,
    [out] DWORD *pcbDib,
    [in, out] LONGLONG *pTimeStamp
);
```

C++ Example:

```
BITMAPINFOHEADER lpHeader = { 0 };
BYTE* lpCurrImage = NULL;
DWORD bitmapSize = 0;
LONGLONG timestamp = 0;
lpHeader.biSize = sizeof(BITMAPINFOHEADER);

hr = pDisplay->GetCurrentImage(&lpHeader, &lpCurrImage, &bitmapSize, &timestamp));
```

MF.Net Prototype:

```
HResult GetCurrentImage(BitmapInfoHeader pBih,
    out IntPtr pDib,
    out int pcbDib,
    out long pTimeStamp);
```

MF.Net C# Example:

```
BitmapInfoHeader workingBitmapInfoHeader = new BitmapInfoHeader();
IntPtr bitmapData = IntPtr.Zero;
int bitmapDataSize = 0;
long bitmapTimestamp = 0;
workingBitmapInfoHeader.Size = Marshal.SizeOf(typeof(BitmapInfoHeader));

hr = evrVideoDisplay.GetCurrentImage(
    workingBitmapInfoHeader,
    out bitmapData,
    out bitmapDataSize,
    out bitmapTimestamp);

// bitmapData is an IntPtr. Use Marshal to copy the video data out
// into a byte array, bitmapDataSize is the length of bitmapData
byte[] managedArray = new byte[bitmapDataSize];
Marshal.Copy(bitmapData, managedArray, 0, bitmapDataSize);
```

SETTING AN MFINT (DWORD PTR)

C++ Prototypes:

```
HRESULT GetStreamCount([out] DWORD *pcInputStreams, [out] DWORD *pcOutputStreams);
```

C++ Examples:

<not available>

MF.Net Prototypes:

Converting Between C++ and C# Code Examples

```
public HRESULT GetStreamCount(MFInt pcInputStreams, MFInt pcOutputStreams);
```

MF.Net C# Examples:

```
public HRESULT GetStreamCount(MFInt pcInputStreams, MFInt pcOutputStreams)
{
    // This template requires a fixed number of input and output
    // streams (1 for each).
    if (pcInputStreams != null)
    {
        pcInputStreams.Assign(1);
    }
    ...
}
```

GETTING A STRING

C++ Prototype:

```
HRESULT GetAllocatedString([in] REFGUID guidKey,
    [out] LPWSTR *ppwszValue,
    [out] UINT32 *pcchLength );
```

C++ Example:

```
WCHAR *g_pwszSymbolicLink = NULL;
UINT32 g_cchSymbolicLink = 0;
pActivate->GetAllocatedString(
    MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_SYMBOLIC_LINK,
    &g_pwszSymbolicLink,
    &g_cchSymbolicLink );
```

MF.Net Prototype:

```
HResult GetAllocatedString(Guid guidKey, out string ppwszValue, out int pcchLength);
```

MF.Net C# Example:

```
int iSize;
string pwszSymbolicLink;
hr = pActivate.GetAllocatedString(
    MFAttributesClsid.MF_DEVSOURCE_ATTRIBUTE_SOURCE_TYPE_VIDCAP_SYMBOLIC_LINK,
    out pwszSymbolicLink,
    out iSize);
```

SETTING A STRING

C++ Prototypes:

```
HRESULT CreateObjectFromURL(
    [in] LPCWSTR pwszURL,
    [in] DWORD dwFlags,
    [in] IPropertyStore *pProps,
    [out] MF_OBJECT_TYPE *pObjectType,
    [out] IUnknown **ppObject
);
```

C++ Examples:

```
HRESULT hr = pSourceResolver->CreateObjectFromURL(
    sURL, // URL of the source.
    MF_RESOLUTION_MEDIASOURCE, // Create a source object.
    NULL, // Optional property store.
    &ObjectType, // Receives the created object type.
    &pSource // Receives a pointer to the media source.
);
```

MF.Net Prototypes:

```
HResult CreateObjectFromURL(
    string pwszURL,
```

```
MFResolution dwFlags,
IPropertyStore pProps,
out MFObjectType pObjectType,
out object ppObject);
```

MF.Net C# Examples:

```
string sUrl = @"C:\Dump\videoFile.mp4";
hr = pSourceResolver.CreateObjectFromURL(
    sURL, // URL of the source.
    MFResolution.MediaSource, // Create a source object.
    null, // Optional property store.
    out Object type, // Receives the created object type.
    out pSource // Receives a pointer to the media source.
);
```

GETTING A STRUCT

C++ Prototype:

```
HRESULT GetCurrentImage(
    [in, out] BITMAPINFOHEADER *pBih,
    [out] BYTE **pDib,
    [out] DWORD *pcbDib,
    [in, out] LONGLONG *pTimeStamp
);
```

C++ Example:

```
BITMAPINFOHEADER lpHeader = { 0 };
BYTE* lpCurrImage = NULL;
DWORD bitmapSize = 0;
LONGLONG timestamp = 0;
lpHeader.biSize = sizeof(BITMAPINFOHEADER);

hr = pDisplay->GetCurrentImage(&lpHeader, &lpCurrImage, &bitmapSize, &timestamp));
```

MF.Net Prototype:

```
HResult GetCurrentImage(BitmapInfoHeader pBih,
    out IntPtr pDib,
    out int pcbDib,
    out long pTimeStamp);
```

MF.Net C# Example:

```
BitmapInfoHeader workingBitmapInfoHeader = new BitmapInfoHeader();
IntPtr bitmapData = IntPtr.Zero;
int bitmapDataSize = 0;
long bitmapTimestamp = 0;
workingBitmapInfoHeader.Size = Marshal.SizeOf(typeof(BitmapInfoHeader));

hr = evrVideoDisplay.GetCurrentImage(
    workingBitmapInfoHeader,
    out bitmapData,
    out bitmapDataSize,
    out bitmapTimestamp);
```

SETTING A STRUCT

```
HRESULT GetCurrentImage(
    [in, out] BITMAPINFOHEADER *pBih,
    [out] BYTE **pDib,
    [out] DWORD *pcbDib,
    [in, out] LONGLONG *pTimeStamp
);
```

C++ Example:

```
BITMAPINFOHEADER lpHeader = { 0 };
BYTE* lpCurrImage = NULL;
DWORD bitmapSize = 0;
LONGLONG timestamp = 0;
```

Converting Between C++ and C# Code Examples

```
lpHeader.biSize = sizeof(BITMAPINFOHEADER);  
  
hr = pDisplay->GetCurrentImage(&lpHeader, &lpCurrImage, &bitmapSize, &timestamp));
```

MF.Net Prototype:

```
HResult GetCurrentImage(BitmapInfoHeader pBih,  
    out IntPtr pDib,  
    out int pcbDib,  
    out long pTimeStamp);
```

MF.Net C# Example:

```
BitmapInfoHeader workingBitmapInfoHeader = new BitmapInfoHeader();  
IntPtr bitmapData = IntPtr.Zero;  
int bitmapDataSize = 0;  
long bitmapTimestamp = 0;  
workingBitmapInfoHeader.Size = Marshal.SizeOf(typeof(BitmapInfoHeader));  
  
hr = evrVideoDisplay.GetCurrentImage(  
    workingBitmapInfoHeader,  
    out bitmapData,  
    out bitmapDataSize,  
    out bitmapTimestamp);
```

GETTING A TYPED OBJECT

C++ Prototype:

```
HRESULT MFCreateTopologyNode(  
    _In_ MF_TOPOLOGY_TYPE NodeType,  
    _Out_ IMFTopologyNode **ppNode);
```

C++ Example:

```
IMFTopologyNode *pNode = NULL;  
HRESULT hr = MFCreateTopologyNode(MF_TOPOLOGY_TRANSFORM_NODE, &pNode);
```

MF.Net Prototype:

```
public static HResult MFCreateTopologyNode(  
    MFTopologyType NodeType,  
    out IMFTopologyNode ppNode);
```

MF.Net C# Example:

```
IMFTopologyNode pNode = null;  
hr = MFExtern.MFCreateTopologyNode(MFTopologyType.SourcestreamNode, out pNode);
```

SETTING A TYPED OBJECT

C++ Prototype:

```
HRESULT AddNode([in] IMFTopologyNode *pNode);
```

C++ Example:

```
IMFTopologyNode *pNode = NULL;  
// ... the *pNode gets instantiated by other calls  
hr = pTopology->AddNode(pNode);
```

MF.Net Prototype:

```
HResult AddNode(IMFTopologyNode pNode);
```

MF.Net C# Example:

```
IMFTopologyNode pSourceNode = null;  
// ... the pSourceNode gets instantiated by other calls  
hr = pTopology.AddNode(pSourceNode);
```

MISC. CODE CONVERSIONS

CONVERTING A BYTE[] TO A STRUCT

This operation is surprisingly difficult in C#. There are a few ways this can be done (serialization of streams & etc.). Probably the simplest method is to use the static functions in the Marshal class. Below is a `ConvertStructureToByteArray()` function from the *TantaWMFUtils* library.

```
public static void ConvertByteArrayToStructure(byte[] bytearray, ref object
convertedStruct)
{
    int len = Marshal.SizeOf(convertedStruct);
    IntPtr i = Marshal.AllocHGlobal(len);
    Marshal.Copy(bytearray, 0, i, len);
    convertedStruct = Marshal.PtrToStructure(i, convertedStruct.GetType());
    Marshal.FreeHGlobal(i);
}
```

CONVERTING A STRUCT TO A BYTE[]

As with the reverse operation described above, this operation is also surprisingly difficult in C#. There are a few ways this can be done (serialization of streams & etc.). Probably the simplest method is to use the static functions in the Marshal class. Below is a `ConvertStructureToByteArray()` function from the *TantaWMFUtils* library.

```
public static byte[] ConvertStructureToByteArray(object structToConvert)
{
    int len = Marshal.SizeOf(structToConvert);
    byte[] arr = new byte[len];
    IntPtr ptr = Marshal.AllocHGlobal(len);
    Marshal.StructureToPtr(structToConvert, ptr, true);
    Marshal.Copy(ptr, arr, 0, len);
    Marshal.FreeHGlobal(ptr);
    return arr;
}
```

COPYING THE DATA FROM AN IntPtr

The data pointed to by an `IntPtr` will be in unmanaged memory. The `Marshal.Copy()` function can be used to copy the data into a datatype usable by C#. Note that when dealing with Windows Media Foundation, the memory pointed to by an `IntPtr` returned from a function call usually has to be released. This operation is not shown in the sample code below.

```
// bitmapData is an IntPtr. Use Marshal to copy the video data out
// into a byte array, bitmapDataSize is the length of bitmapData
byte[] managedArray = new byte[bitmapDataSize];
Marshal.Copy(bitmapData, managedArray, 0, bitmapDataSize);
```

GETTING THE SIZE OF A STRUCT

Unlike C or C++, C# cannot directly take the size of a struct. However the Marshal class provides a variety of static methods for this purpose and it is simple to use it for the task. The code below shows how to get the size of a Struct.

```
BitmapInfoHeader workingBitmapInfoHeader = new BitmapInfoHeader();
int bitMapHeaderSize = Marshal.SizeOf(typeof(BitmapInfoHeader));
```

CONVERTING AN ARRAY OF STRUCTS

Certain operations in WMF will return an array of structures in the form of a binary blob. This data can be converted for use using the technique below. In this particular example, an array of MFT_REGISTER_TYPE_INFO structs is returned when querying a Transform Activator for a list of its input media types.

```
MFTRegisterTypeInfo rtInfoList = new List<MFTRegisterTypeInfo>();

// get the data from the activator. This comes out in a blob which is actually
// an array of MFT_REGISTER_TYPE_INFO structs.
HRESULT hr = activatorObject.GetAllocatedBlob(
    MFAttributesClsid.MFT_INPUT_TYPES_Attributes,
    out outBlob,
    out outSize);
if (hr != HRESULT.S_OK) return HRESULT.E_FAIL;

// get the size of a MFTRegisterTypeInfo class. We have to
// use Marshal because GUIDs are supposedly not of fixed size.
// The only reason this works is because the MFTRegisterTypeInfo
// class has a "StructLayout(LayoutKind.Sequential)" decoration
int sizeOfMFTRegisterTypeInfo = Marshal.SizeOf(typeof(MFTRegisterTypeInfo));
if (sizeOfMFTRegisterTypeInfo <= 0) return HRESULT.E_FAIL;

// calculate the number of records in the blob
int numRecords = outSize / sizeOfMFTRegisterTypeInfo;

// to get at the information in the blob, we convert the start of each
// MFT_REGISTER_TYPE_INFO struct to an IntPtr then copy it into a
// MFTRegisterTypeInfo class
for (int i = 0; i < numRecords; i++)
{
    // get a pointer to the next MFT_REGISTER_TYPE_INFO struct
    IntPtr intPtrToStruct = new IntPtr(outBlob.ToInt64() + i *
        sizeOfMFTRegisterTypeInfo);

    // copy the contents at that pointer into a MFTRegisterTypeInfo class
    MFTRegisterTypeInfo tmpRTInfo =
        Marshal.PtrToStructure<MFTRegisterTypeInfo>(intPtrToStruct);
    if (tmpRTInfo == null) continue;

    // add it to our container
    rtInfoList.Add(tmpRTInfo);
}
```