# WDM USB Video Streaming Filter Driver

## A Technical Report

**Technical Expertise Level:** Intermediate  
**Requires knowledge of:** Windows driver development and Streaming Architecture

## INDEX

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>4</td>
</tr>
<tr>
<td>BASICS OF WDM AND STREAMING ARCHITECTURE</td>
<td>4</td>
</tr>
<tr>
<td>WHAT IS WDM ?</td>
<td>4</td>
</tr>
<tr>
<td>Architectural Overview of WDM Driver</td>
<td>5</td>
</tr>
<tr>
<td>Life Cycle of a WDM Driver</td>
<td>5</td>
</tr>
<tr>
<td>The DriverUnload Routine</td>
<td>8</td>
</tr>
<tr>
<td>AddDevice Routine</td>
<td>9</td>
</tr>
<tr>
<td>Handling Plug and Play &amp; Power IRPs</td>
<td>11</td>
</tr>
<tr>
<td>INSTALLATION OF A WDM DRIVER</td>
<td>11</td>
</tr>
<tr>
<td>WDM FILTER DRIVERS</td>
<td>13</td>
</tr>
<tr>
<td>Skeleton of a WDM Filter Driver</td>
<td>13</td>
</tr>
<tr>
<td>DriverEntry Routine</td>
<td>13</td>
</tr>
<tr>
<td>AddDevice Routine</td>
<td>15</td>
</tr>
<tr>
<td>Dispatch Routines</td>
<td>16</td>
</tr>
<tr>
<td>INSTALLATION OF A WDM FILTER DRIVER</td>
<td>19</td>
</tr>
<tr>
<td>WDM STREAMING ARCHITECTURE</td>
<td>22</td>
</tr>
<tr>
<td>Architecture Overview</td>
<td>24</td>
</tr>
<tr>
<td>Components of WDM Streaming</td>
<td>25</td>
</tr>
<tr>
<td>WDM Streaming Concepts</td>
<td>26</td>
</tr>
<tr>
<td>SYNCHRONIZATION</td>
<td>26</td>
</tr>
<tr>
<td>MEDIUMS AND INTERFACES</td>
<td>26</td>
</tr>
<tr>
<td>PROPERTY SETS</td>
<td>27</td>
</tr>
<tr>
<td>Property Set Description</td>
<td>27</td>
</tr>
<tr>
<td>CONNECTION PROCESS</td>
<td>28</td>
</tr>
<tr>
<td>I/O CONTROL CODES</td>
<td>29</td>
</tr>
<tr>
<td>PART II - USB CAMERA FILTER DRIVER IMPLEMENTATION</td>
<td>32</td>
</tr>
<tr>
<td>USB VIDEO STREAMING FILTER DRIVER-INTRODUCTION</td>
<td>32</td>
</tr>
</tbody>
</table>
DESCRIPTION OF USB VIDEO STREAMING FILTER DRIVER................................................................. 32
IMPLEMENTATION OF STREAMING FILTER DRIVER ........................................................................ 33
  DRIVER ENTRY ROUTINE ................................................................................................................. 34
  ADD DEVICE ROUTINE .................................................................................................................. 34
  SPECIFIC ROUTINES OF THE STREAMING FILTER DRIVER ......................................................... 34
DATAFLOW OF CAMERA FILTER DRIVER ...................................................................................... 40
INSTALLATION OF THE STREAMING FILTER DRIVER ................................................................. 41
IMPLEMENTATION OF NOISE FILTERING ALGORITHM ............................................................... 42
  PERFORMANCE SECTION ............................................................................................................... 44
CONCLUSION ..................................................................................................................................... 45
REFERENCES ...................................................................................................................................... 46
Introduction

Writing device drivers for certain specific applications is the latest trend now. And as a supportive act to this, the **WDM architecture** makes the task easier. Another addition to certain set of drivers is the **USB Camera Filter Driver**, which filters noise from a USB camera. The white paper leads through some of the basics of **WDM Drivers & Filter Drivers** to understand the underlying concepts. This section of the paper explains how to write a WDM Driver, build & install it. Later the concepts of Streaming are dealt in detail. Even the pros and cons of using Streaming and where it can be applied are dealt with. The second part of the paper deals with the **Implementation of the Camera Filter driver**, installation methods and the overall performance.

Basics of WDM and Streaming Architecture

In this section, we would be discussing the basics of WDM, its architecture, driver layering and the steps involved in implementing the WDM driver and filter driver. All the driver routines and the installation details are also covered. The concepts and components involved in Streaming Architecture are discussed here.

What is WDM?

**WDM is the new Windows Driver Model from Microsoft, which brings device driver binary compatibility between Windows NT and Windows 98. WDM incorporates plug and play, power management, and advanced bus management into a subset of the Windows NT device driver model.**

The new Plug and Play architecture, encompassing "**Ready to Run**" and "**Hot Swapping**", provides for the dynamic detection of devices being inserted or removed from the system while it is operational, and notifying the appropriate device drivers, and possibly applications, that a device has been added or removed.
from the system. Two new external bus standards Universal Serial Bus (USB) and IEEE 1394 support hot plugging (also called as Hot Swapping).

**Power Management** adds additional capabilities to the system and device drivers to allow the system to aggressively save electricity by selectively turning off the power to some or all hardware components in the system. Currently, most systems implement power management through a hybrid of hidden software (BIOS) and dedicated monitoring hardware, often causing problems in full-featured operating systems such as Windows NT. In the new model, the operating system is in control of all power management, making decisions based on policies set up by the user. Device drivers both generate and receive power management requests in the WDM environment.

**Windows Management and Instrumentation (WMI)** provides a standard interface between an instrumented driver and a browser (which may be running on the local system or on a remote node across the network). WMI provides not only counters and performance data, but also a means for controlling the settings and parameters used by a driver while it is operational. WMI is used by power management and is therefore required in WDM device drivers.

**Architectural Overview of WDM Driver**

**Life Cycle of a WDM Driver**

The life cycle of a typical WDM driver would proceed something like this:

- A bus driver detects a piece of hardware. (Bus drivers are required drivers which drives an I/O bus and provides per-slot functionality that is device-independent, there is one bus driver for each type of bus on a system)
- The PnP Manager locates a hardware instance key in the Enum branch of the registry. This registry key contains a pointer to another registry key that
designates the function driver (which drives an individual device and it is the main driver for the device) for the device. The PnP Manager dynamically loads the function driver. The first time this occurs for a particular driver, the I/O Manager creates a DRIVER_OBJECT data structure calls the main entry point of the newly loaded driver. Conventionally, this entry point is named DriverEntry, and its only purpose is to initialize the DRIVER_OBJECT by storing pointers to other functions contained within the driver.

- The PnP Manager calls the function driver's AddDevice function to create a DEVICE_OBJECT to represent the device. If a driver manages more than one actual device, the PnP Manager will call AddDevice once for each one.

From now on, all communication between the outside world and the driver uses an I/O Request Packet (IRP).

- The PnP Manager allocates the I/O resources (IRQ, port addresses, etc.) need and sends a request and the device gets initialized.
- Some devices can be removed from the system without shutting the computer down. When this occurs, the PnP Manager sends the driver a request (PnP IRP) and it is responded by deleting the device object, which was created in AddDevice.
- When the last device disappears, the I/O Manager calls the DriverUnload routine and then deletes the driver image from memory.

**Driver Layering**

In WDM, there can be any number of drivers for a single hardware device. The following figure illustrates the concept. The left-hand portion of the figure depicts a stack of DEVICE_OBJECT data structures, one for each driver that's involved in supporting one device. At the very bottom of the stack, there’s a Physical Device Object (PDO) that represents the connection between our device and a hardware
bus. Somewhere above the PDO, there’s a Function Device Object (FDO) that represents the logical functionality of the device. Elsewhere in the stack, either above or below the FDO, we may find various Filter Device Objects (FiDOs). Each device object in the stack belongs to a particular driver, as illustrated by the dotted lines in the figure. The PDO belongs to the bus driver. The FDO belongs to the function driver. The FiDO belongs to the filter driver.

The purpose of layering drivers like this is to divide up the work of handling IRPs according to expertise. IRPs flow in the first instance to the topmost FiDO. It can decide to handle the IRP all on its own, to pass the IRP down the stack, or to do a combination of both things. Each driver in turn can make similar choices. Some IRPs therefore percolate all the way to the bottom of the stack, while some IRPs stop (and get completed) partway down.

**Skeleton of a WDM Driver** In this section, the basics of the code fragments that are needed to write to transform our stodgy old KMD into a shiny new WDM driver are described.

**The Driver Entry Routine** A WDM DriverEntry routine need do nothing more than fill in function pointers within the driver object. If there's any other
global initialization that needs to be performed, DriverEntry would do that too.

extern "C" NTSTATUS
    DriverEntry
    (IN PDRIVER_OBJECT DriverObject,
     PUNICODE_STRING RegistryPath)
{

1) DriverObject->DriverUnload = DriverUnload;
    DriverObject->DriverExtension->AddDevice
        = AddDevice;
2) DriverObject->MajorFunction[IRP_MJ_PNP]
        = DispatchPnp;
    DriverObject->MajorFunction[IRP_MJ_POWER]
        = DispatchPower;
3). . .
    return STATUS_SUCCESS;
}

These three statements set the function pointers for entry points elsewhere in the driver.

Every WDM driver must handle PNP and POWER I/O requests; this is where we'd specify our dispatch functions for these requests.  In place of this ellipsis, we'd have code to set several additional MajorFunction pointers.

The DriverUnload Routine
The purpose of a WDM driver's DriverUnload function is to cleanup after any global initialization that DriverEntry might have done. There's almost nothing to do:

```c
VOID DriverUnload(PDRIVER_OBJECT DriverObject)
{
}
```

**AddDevice Routine**

The AddDevice function is brand new with WDM. That's the function the system will call to alert us when there's a hardware device to manage. In the WDM architecture, the AddDevice function that the PnP Manager calls once for each device that the driver manages. The function has the following prototype:

```c
NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject,
                   PDEVICE_OBJECT pdo)
{
}
```

The DriverObject argument points to the same driver object that we initialized in our DriverEntry routine. The pdo argument is the address of the physical device object at the bottom of the device stack, even if there are already filter drivers below. The basic responsibility of AddDevice in a function driver is to create a device object and link it into the stack rooted in this PDO. The steps involved here are as follows:

1. Call IoCreateDevice to create a device object and an instance of our own device extension object.
2. Register one or more device interfaces so that applications know about the existence of the device. Alternatively, give the device object a name and then create a symbolic link.

3. Initialize our device extension and the Flags member of the device object.

4. Call IoAttachDeviceToDeviceStack in order to put our new device object into the stack.

**Building the Device Stack**

Each filter and function driver has the responsibility of building up the stack of device objects, starting from the PDO and working upwards. This part of the work can be accomplished with a call to `IoAttachDeviceToDeviceStack`:

```c
NTSTATUS AddDevice( . . , PDEVICE_OBJECT pdo)
{
    PDEVICE_OBJECT fdo;
    IoCreateDevice( . . , &fdo);
    pdx->LowerDeviceObject =
        IoAttachDeviceToDeviceStack(fdo, pdo);
}
```

The first argument to `IoAttachDeviceToDeviceStack` (fdo) is the address of our own newly created device object.

The second argument is the address of the PDO, which we receive as an argument to AddDevice.

The return value is the address of whatever device object is immediately underneath ours, which may be the PDO or the address of some lower filter device object.
Handling Plug and Play & Power IRPs

The PnP Manager uses IRPs to direct drivers to start, stop, and remove devices and to query drivers about their devices. All PnP IRPs have the major function code IRP_MJ_PNP.

Drivers should handle PnP IRPs in a XxxDispatchPnp routine, where Xxx is a prefix identifying the driver. A driver sets the address of its DispatchPnp routine in DriverObject->MajorFunction[IRP_MJ_PNP] during driver initialization in its DriverEntry routine. The PnP manager, through the I/O Manager, calls a driver's DispatchPnp routine. To request a power operation on a device, the Power Manager or a device driver sends a power management IRP. All power management requests have the major IRP code IRP_MJ_POWER.

Installation of a WDM Driver

Installing a WDM driver involves creating a dual-personality INF file. One personality is for Windows 98; the other is for Windows 2000.

An INF File is made up of a set of named sections. A section contains one or more items. Each section begins with the section name enclosed in square brackets. There can be any number of sections in an INF file, but there are limited numbers of types of sections, and sections can be put in any order in an INF file.

The basic parts in a INF file includes

/Version/
Signature=$CHICAGO$
Class=UNKNOWN
Provider="CalSoft Software"

/Manufacturer/

CSWL Inc, Pleasanton, California
"CalSoft Software"=CSO

[DestinationDirs]
DefaultDestDir=10,System32\Drivers

[SourceDiskFiles]
sfdriver.sys=1

[SourceDiskNames]
1="Install disk",disk1

[CSO]
"WDM Streaming Filter Driver"=sfDriverInstall,*CSO1234

[sfDriverInstall]
AddReg=sfdriver.AddReg
CopyFiles=sfdriver.CopyFiles

[sfdriver.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,sfdriver.sys

[sfdriver.CopyFiles]
sfdriver.sys,,,2

[sfdriverInstall.NT]
CopyFiles=sfdriver.CopyFiles

[sfdriverInstall.NT.Services]
AddService=sfdriver,2,sfdriverService
[sfdriverService]
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%10\%\system32\drivers\sfdriver.sys

By launching the hardware wizard in either Windows 98 or Windows 2000 and use the "Have Disk" button to point to the directory containing the INF file, the driver will get installed.

WDM Filter Drivers

A Filter driver is a special type of driver that sits on the top of some other driver and intercept requests directed at the lower driver’s device objects. Users of the lower driver are completely unaware that this is going on. Filter Drivers work by attaching their device objects to a device object created by the lower level driver. A filter driver that’s above the function driver is called an Upper filter driver; a filter driver that’s below the function driver is called a Lower filter driver. The mechanics of building either type of filter are exactly the same. In fact, a filter driver can be built just like we build any other WDM driver (with a DriverEntry routine, an AddDevice routine, a bunch of dispatch functions, and so on). The intended purpose of an upper filter is to facilitate supporting a device that behaves in most respects like a generic device of its class but has some additional functionality. An upper filter driver can be used to intervene in the flow of I/O requests in order to deal with the extra functionality.

Skeleton of a WDM Filter Driver

DriverEntry Routine.
The DriverEntry routine for a filter driver is very similar to that for a function
driver. The major difference is that a filter driver must install dispatch routines for
every type of IRP, not just for the types of IRP it expects to handle.

```
extern "C" NTSTATUS DriverEntry (PDRIVER_OBJECT DriverObject,
    pUNICODE_STRING RegistryPath)
{
    DriverObject->DriverUnload = DriverUnload;
    DriverObject->DriverExtension->AddDevice = AddDevice;
    for (int i = 0; i MajorFunction) ++i)
        DriverObject->MajorFunction[i] = DispatchAny;
    DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
    DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;
    return STATUS_SUCCESS;
}
```

A filter driver has a DriverUnload and an AddDevice function like any other driver.
And the major function table should be filled with the address of a routine named
DispatchAny that would pass any random request down the stack. Specific dispatch
routines have to be specified for POWER and PNP.

The reason that a filter driver has to handle every conceivable type of IRP has to do
with the order in which driver AddDevice functions get called vis-à-vis
DriverEntry.

In general, a filter has to support all the same IRP types that the driver immediately
underneath it supports. If a filter were to leave a particular MajorFunction table
entry in its default state, IRPs of that type would simply get failed with
STATUS_INVALID_DEVICE_REQUEST. (There's a default dispatch function in
the I/O manager that simply completes a request with this status. The driver object
initially comes to you with all the MajorFunction table entries pointing to that
default routine.

\textit{AddDevice Routine}

Filter drivers have AddDevice functions that get called for each appropriate piece
of hardware. We'll be calling IoCreateDevice to create an unnamed device object
and IoAttachDeviceToDeviceStack to plug into the driver stack. In addition, we'll
need to copy a few settings from the device object underneath us.

\begin{verbatim}
NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject,
PDEVICE_OBJECT pdo)
{
PDEVICE_OBJECT fido;
NTSTATUS status = IoCreateDevice(DriverObject,
    sizeof(DEVICE_EXTENSION), NULL, FILE_DEVICE_UNKNOWN,
    0, FALSE, &fido);
if (!NT_SUCCESS(status))
    return status;
PDEVICE_EXTENSION pdx =
    (PDEVICE_EXTENSION) fido->DeviceExtension;
    
pdx->DeviceObject = fido;
pdx->Pdo = pdo;
IoInitializeRemoveLock(&pdx->RemoveLock, 0, 0, 255);
PDEVICE_OBJECT fdo = IoAttachDeviceToDeviceStack(fido,
pdo);
pdx->LowerDeviceObject = fdo;
fdo->Flags |= pdo->Flags &
    (DO_DIRECT_IO | DO_BUFFERED_IO | DO_POWER_PAGABLE
    | DO_POWER_INRUSH);
\end{verbatim}
The highlighted part that's different from a function driver is in bold face.
Basically, it can be seen that we're propagating a few flag bits, the DeviceType, the Characteristics value, and the buffer alignment requirements from the device object next beneath us.

Dispatch Routines

A filter driver in the first place is used to modify the behavior of a device in some way. Therefore, we have dispatch functions that do something with some of the IRPs that come our way. We'll merely be passing most IRPs down the stack using fairly generic code, though. The generic dispatch functions are shown here.

The DispatchAny Function

Here's the dispatch routine for all the types of IRP we don't specifically handle, other than IRP_MJ_PNP and IRP_MJ_POWER:

```c
NTSTATUS DispatchAny(PDEVICE_OBJECT fido, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
        return CompleteRequest(Irp, status, 0);
    IoSkipCurrentIrpStackLocation(Irp);
    status = IoCallDriver(pdx->LowerDeviceObject, Irp);
    return status;
}
```
IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
}

IoAcquireRemoveLock prevents the device object from being removed while we're accessing fields in it and in the device extension attached to it. CompleteRequest is a helper function that deals with the mechanics of completing an IRP.

Handling IRP_MJ_PNP

The dispatch function for Plug and Play requests is similar, but it includes code to delete the device object when an IRP_MN_REMOVE_DEVICE finally arrives:

```c
NTSTATUS DispatchPnp(PDEVICE_OBJECT fido, PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
        return CompleteRequest(Irp, status, 0);
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    ULONG fcn = stack->MinorFunction;
    IoSkipCurrentIrpStackLocation(Irp);
    status = IoCallDriver(pdx->LowerDeviceObject, Irp);
    if (fcn == IRP_MN_REMOVE_DEVICE)
    {
        IoReleaseRemoveLockAndWait(&pdx->RemoveLock, Irp);
        IoDetachDevice(pdx->LowerDeviceObject);
        IoDeleteDevice(fido);
    }
    else
        IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
```
These three statements are part of the standard way of handling a PNP request to remove a device object. IoReleaseRemoveLockAndWait waits until all pending IRPs complete and causes any newly arriving IRPs to complete immediately with STATUS_DELETE_PENDING. IoDetachDevice undoes the call to IoAttachDeviceToDeviceStack that appears in the AddDevice function. IoDeleteDevice deletes the device object, whereupon the filter's DriverUnload function will get called.

Handling IRP_MJ_POWERe="1">

The dispatch function for power management IRPs is also a bit different because of the special way you have to handle power IRPs:

```c
NTSTATUS DispatchPower(PDEVICE_OBJECT fido, PIRP Irp)
{
    PoStartNextPowerIrp(Irp);
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
        return CompleteRequest(Irp, status, 0);
    IoSkipCurrentIrpStackLocation(Irp);
    status = PoCallDriver(pdx->LowerDeviceObject, Irp);
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return status;
}
```
The two differences between this function and DispatchAny have been highlighted. We must call PoStartNextPowerIrp at some point when we own this IRP, even if we're going to fail it. In addition, we call PoCallDriver instead of IoCallDriver to actually forward the IRP to the next driver.

### Installation of a WDM Filter Driver

It's relatively easy to install a filter driver along with the function driver for a particular piece of hardware. Here's an INF file that accomplishes that feat:

```ini
[Version]
Signature=$Windows 98$
Class=UNKNOWN
Provider="CalSoft Software" e="1">

[DestinationDirs]
DefaultDestDir=10, System32\Drivers e="1">

[Manufacturer]
"CalSoft Software"=CSO e="1">

[WCO]
"Simple Device with Filter"=SimpleFilterDriverInstall, *WCO1234

[SourceDiskFiles]
simple.sys=1
filter.sys=1 e="1">

[SourceDiskNames]
1 = "Install disk", disk1 e="1">

; Win 98 install sections e="1">
```
[SimpleFilterDriverInstall]
CopyFiles=SimpleFilterDriverCopyFiles
AddReg=SimpleFilterDriverAddReg e="1">

[SimpleFilterDriverCopyFiles]
simple.sys
filter.sys e="1">

[SimpleFilterDriverAddReg]
HKR,,DevLoader,0,*ntkern
HKR,,NTMPDriver,0,simple.sys e="1">

[SimpleFilterDriverInstall.hw]
AddReg=FilterAddreg e="1">

[FilterAddreg]
HKR,,UpperFilters,0x00010000,filter.sys e="1">

; NT install sections e="1">

[SimpleFilterDriverInstall.nt]
CopyFiles=SimpleCopyFiles e="1">

[SimpleFilterDriverInstall.nt.Services]
AddService=Simple,2,SimpleService
AddService=Filter,,FilterService e="1">

[SimpleService]
ServiceType=1 ; SERVICE_KERNEL_DRIVER
StartType=3 ; SERVICE_DEMAND_START
ErrorControl=1 ; SERVICE_ERROR_NORMAL
ServiceBinary=%12%\simple.sys e="1">

[FilterService]
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%12%\filter.sys e="1">

[SimpleFilterDriverInstall.NT.hw]
AddReg=FilterAddreg.nt

[FilterAddreg.nt]
HKR,,UpperFilters,0x00010000,filter

The key, both for Windows 98 and Windows NT, is adding an UpperFilters value to the hardware registry key for the device. Appending ".HW" to the name of the install section is the trick that allows you to do that. To add a lower filter, you modify the LowerFilters value in the registry. In Windows 98, you can easily add a filter to an existing WDM driver by adding the UpperFilters or LowerFilters value to the hardware key for the device. This is the key below HKLM\Enum, and you can find it if you know who enumerates your device (e.g., USB, PCI, etc.) and what the hardware ID of the device is.
WDM Streaming Architecture

WDM Streaming is a kernel-mode solution to the general problem of synchronizing the rendering of large volumes of data in as close to real time as possible in Win98 and Win NT environments.

The goal of WDM Streaming is to enable the best possible real-time performance for media streaming, where low-latency and high throughput are the key factors. It also makes Windows an effective platform for time-sensitive data streaming.

As data stream transition from the non-kernel mode to the kernel-mode is time consuming, WDM Streaming is moving most data streaming tasks from the non-kernel mode code modules onto the kernel mode drivers or onto hardware, which Streaming also reduces latency.

Who should be interested in WDM Streaming?

Generally, developers of non-kernel software who were unable to meet their low latency and high throughput should consider using WDM Streaming. If the class driver supports hardware that requires low latency and high throughput, then it will support WDM Streaming Ioctls. Some of the potential WDM Streaming client technologies are Audio and Video rendering hardware, Video capture devices.

Why should WDM Streaming be used?

It is important to understand the limitations of the current Windows drivers when developing a streaming application with WDM Streaming.

Current drivers pass data either in or out of the kernel mode, resulting in many transitions between the user-mode and the kernel mode. This kind of transition involves the data to pass through several validation layers, making streaming data
inefficient. This also causes latency problems due to user mode scheduling and ring transitions, which make Windows unsuitable for writing programs that are dependent on Streaming and Synchronization.

The above diagram shows how kernel-mode streaming is done which results in low latency and efficient streaming.

WDM Streaming Vs DirectShow

The Microsoft DirectShow API is a multimedia technology designed to play audio, video and other time-stamped multimedia streams by using modular components called filters arranged in a configuration called filter graph. An object called filter graph manager oversees the connection of these filters and controls the data flow of the stream.

WDM Streaming adds a new streaming data transport, as compared to the DirectShow transports. DirectShow will continue to build filter graphs, negotiate media types, and provide control infrastructure. But, whenever a new WDM Streaming data transport is negotiated between two filters, the media streams will pass between the associated device drivers in the kernel, without any non-kernel
mode involvement. This feature of the WDM Streaming reduces the transfer latency and buffer sizes as well.

WDM Streaming also provides a non-kernel filter that acts as a generic proxy on behalf of the kernel mode drivers, to provide a transparent integration between DirectShow and WDM Streaming. The proxy marshals data between the COM-based interfaces of the DirectShow and the kernel’s use of IOCTLs and IRPs.

Architecture Overview

The WDM Connection and Streaming Architecture (WDM Streaming or WDM-CSA) built on top of the WDM, is designed to address certain limitations which make Windows an effective platform for time-sensitive data streaming.

Kernel-mode streaming can be achieved with the help of several components provided by WDM Streaming. This environment is made up of clients running in non-kernel mode and components running in kernel mode.

These components must work together to accomplish the following:

- Control (starting, stopping, pausing, speeding up, slowing down, amplifying, or attenuating components)
- Connection (dynamically connecting components that participate in the rendering of a particular data stream or streams)
- Streaming (passing the data to be rendered from component to component)
- Synchronization (making sure each component passes the right data at the right time)
- Quality Management (getting components to cooperatively share limited resources)
Components of WDM Streaming

WDM Streaming Library comprises of functions and structures used to connect kernel mode streaming components, stream data and synchronize data streams. Of the functions provided by the library, the non-kernel mode clients use filter connection functions and control functions, while the kernel mode drivers functions for connections, streaming and synchronization. The quality management functions are shared between the non-kernel mode clients and the kernel mode drivers.

KSProxy is a kernel streaming proxy module that exports COM interfaces to the standard property sets of the kernel mode components.

This proxy is another DirectShow filter, which allows video or audio filters to perform operations such as starting, stopping, pausing, speeding up, slowing down, amplifying, or attenuating a data stream. The proxy running in non-kernel mode moves data down on to the kernel mode drivers and hardware components. It also accepts existing control interfaces and translates them into Ioctl calls to the WDM Streaming library. The Streaming library in turn notifies the proxy of events.

Stream Class Driver is a generic class for streaming and time-stamped media. The class allows developers to create minidrivers that are hardware specific and lets the class driver handle the allocation of system resources and memory.

The Stream Class interface is a set of function calls between the class driver and the minidriver. The class driver controls the request flow, calling the adapter minidriver when access to the adapter hardware is necessary. The class driver is responsible for multiprocessor and interrupt synchronization. After both the class driver and the minidriver are initialized, the minidriver is passive and is called only by the class driver. Most of the function calls from the minidriver to the class driver are low-level service requests.
WDM Streaming Concepts

Let's look at some of the important concepts involved in Streaming apart from the components.

Synchronization

Synchronization is an important concept in WDM Streaming.

A master clock is an integral part of stream synchronization. It controls the rate of reference time within a group of filters, and is used by filters to synchronize presentation times. A master clock’s reference time is normally derived from the presentation time of a particular stream running through the group of filters. A master clock is represented by a file object just like any filter in WDM Streaming architecture, and filters can use that file object to perform operations on the master clock.

Presentation time can generally be thought of as file position in a file source, although the data stream source might not be from a file. If played forward in time, presentation time would proceed forward from that point. Presentation times are synchronized to a master clock reference time.

The presentation time is used to synchronize to the master clock's presentation time when a master clock is used. Altering the presentation time changes the synchronization between filter pins. This also enables the driver to easily relocate the synchronization points without modifying the data itself, and without maintaining an entire history of transforms to apply to an underlying physical clock in order to report that presentation time. The presentation time on a stream need not start at zero, but must always progress.

Mediums and Interfaces
A Medium defines a way of communication, such as the bus over which the communication occurs. The IRP-based medium is what would typically be used. Each filter and each pin on a filter must designate what mediums it supports. For example, a filter could specify that it communicates through IRPs.

An Interface defines what can actually be communicated. The interface is the group of methods and properties to use, implemented on the specified medium. Each interface has a medium-specific implementation, but the same interface can be implemented over several mediums.

**Property Sets**

In the WDM Streaming architecture, filters and pins can support property sets that represent elements such as the number of pins available on a filter and the types of connections a pin can make. Property sets are also used to configure filters and pins and to query information about a filter or pin. Property sets are a method for accessing a group of related data, and each property request can be fulfilled either synchronously or asynchronously.

Each property set has a unique identifier, which is used to access a requested property set. The method of using unique identifiers for property sets allows for custom driver extensions that will not conflict with any other custom extensions or future development. Filters and pins support property sets using the ioctl_ks_property device I/O control.

The following lists property sets on a KS filter that are accessed to implement methods in the proxy on behalf of a filter or a pin on a DirectShow filter.

**Property Set Description**
KSPROPSETID_Pin is used to present information on the filter as to its pins and instances that can be created, and information on each pin as to its data flow and compatible data types.

KSPROPSETID_Topology is used extensively to provide DirectShow internal connections information and to follow a filter's topology when attempting to forward state changes.

KSPROPSETID_MediaSeeking corresponds directly to IMediaSeeking, and is used to determine seeking capabilities and direct filter in media seeking.

KSPROPSETID_Stream is used to set or query the support of a clock on each KS pin, which is translated to such support on the proxy filter. It is also used to set or query the support of an allocator on each KS input pin, which translates directly to such support on the proxy-implemented pins.

KSPROPSETID_Connection reflects the state, the input allocator framing, the set dataformat, and the proposed dataformat.

Connection Process

The Connection Process in the Streaming Architecture takes places in a sequence of steps listed below:

The Client uses a KsCreatePin function to create a pin and establish a connection with the driver.

```c
UCHAR Connect[sizeof(KSPIN_CONNECT) + sizeof(KSDATAFORMAT)];

NTSTATUS KsCreatePin(
    FilterHandle,
    (PKSPIN_CONNECT) Connect
    AccessMask,
    PinHandle);
```
Where
FilterHandle is the handle of the driver object.
Data structure Connect contains both KSPin_Connect and the extended structure KSDataFormat passed to the KsCreatePin
PinHandle is the out parameter returned by the function once the pin is created.

- The Stream Class Driver does the connection between the source pin and the sink pin in the IRP_MJ_CREATE Irp dispatch routine. The connection structures passed to the Irp are validated using KsValidateConnectRequest function. This is the initial phase of Connection process and through which the driver would know the details about the pin and the dataformat supported by the client as well
- The source pin bases its stack size on the sink pin stack size plus one. The source pin maintains a pointer to the sink pin and can process incoming requests. It can also forward the requests to the lower device drivers using IoCallDriver function.
- Since the Client has a handle to the pin, it can query the pin by enumerating properties on each pin connection using the Win32 function DeviceIoControl. The filter has certain functions which allow the client to query pins regarding characteristics, status, connection, dataformats, time and timeformats as well as to set these parameters.

I/O Control Codes

WDM Streaming supports a set of predefined I/O control codes (IOCTLS). A device through a file object receives IRP_MJ_DEVICEIOCONTROL in order to support these IOCTLS. The parameters depend on the subfunction passed.

The IOCTLS covered here includes

- IOCTL_KS_READ_STREAM & IOCTL_KS_WRITE_STREAM These two IOCTLs are used to send and receive data in a buffered manner from the pin
connections. These requests contain a Stream Header list, each optionally pointing to the actual stream data.

- **IOCTL_KS_PROPERTY** Filters and pins support property sets using the IOCTL_KS_PROPERTY device I/O control. Although this IOCTL is defined with the METHOD_NEITHER flag, the property data is passed as the output buffer to the IOCTL and probed for read access. The KsPropertyHandler function handles properties requested through IOCTL_KS_PROPERTY. Property sets can represent items such as the number of pins available and the types of connections they can make. Property sets are also used to configure filters and pins. Property sets are a way to access a group of related data on an object, and each property request can be fulfilled either synchronously or asynchronously.

Each property set has a unique identifier that is used to access a requested property set. The method of using unique identifiers for property sets allows custom driver extensions that will not conflict with any other custom extension.

- **IOCTL_KS_METHOD** Filters and pins support method sets using the IOCTL_KS_METHOD device I/O control code. IOCTL_KS_METHOD is defined with the TransferType parameter set to METHOD_NEITHER. However, needed buffers are copied when processed using the KsMethodHandler service. The first buffer is copied and the second buffer is copied, if needed, according to the KSIDENTIFIER flags. Method sets are used to perform an action. In contrast, property sets are used to get and set attributes. If getting and setting it can manipulate a parameter associated with a device, then it is most likely a property. If a parameter is used to perform an action, such as allocating memory, then it is most likely a method.

Method sets provide a way to access related actions. Each method request can be fulfilled either synchronously or asynchronously. Method sets have unique identifiers that are used to retrieve the method set requested. The use of unique
identifiers for method sets allows custom driver extensions to be created that do not conflict with other custom extensions.

Although similar to a property set, the parameters for a method set differ in that a method can either pass no information, or pass information in only, out only, or pass information bidirectionally

- **IOCTL_KS_ENABLE_EVENT & IOCTL_KS_DISABLE_EVENT** Device objects support event sets using the IOCTL_KS_ENABLE_EVENT and IOCTL_KS_DISABLE_EVENT device I/O control codes. Event sets are groups of related events that can generate a notification of an event occurrence. For example, a change in a device state or the reaching of a specific sample position might cause a filter pin to generate an event notification to any clients that have registered to receive such a notification.

- **IOCTL_KS_RESET_EVENT** Resetting the state essentially returns the object (for example, a pin) to the state it was in when it was set to an Acquire state. Any reset must be propagated through a data stream from pin connection to pin connection. During a reset, any additional reset requests that a pin receives will be ignored. Consequently, a reset on a pin can affect the filter as a whole as it propagates the request through the effected pins in relation to the filter routing for that pin.

A reset is initiated on a pin by sending IOCTL_KS_RESET_STATE with a value of SRESET_BEGIN contained in the input buffer. This begins the Reset State and indicates to a pin that any pending or new data received while in a Reset state should be rejected with STATUS_DEVICE_NOT_READY. The reset state is ended by either a disconnection, or by receiving an IOCTL_KS_RESET_STATE with a value of KSRESET_END contained in the input buffer. It is not canceled by changing the Pin State. This indicates to a pin that new data can be accepted. It also indicates that the pin can once again pass data to its previous connection,
even if data had been rejected in a prior attempt. Typically, rejecting data indicates to a pin that it should halt sending more data.

Part II - USB Camera Filter Driver Implementation

USB Video Streaming Filter Driver-Introduction

This part of the paper deals with the procedure and methods of implementation of the filter driver. The USB Camera filter driver is basically designed to trap the incoming streaming video data. Operations on the data such as masking and filtering are done without buffering, which does not affect the system performance. As we have seen in Part I of the paper that streaming is carried on with the help of IOCTLs, the driver hooks the DeviceIoControl dispatch routine & sets a completion routine to trap the incoming data streams. Most of the work is done in the completion routines.

Description of USB Video Streaming Filter Driver

The USB Camera captures both the video and audio streams once it is powered. The Functional Device Object (FDO) is notified of the presence of the camera through a plug and play notification, which is a callback function. The FDO is responsible for initializing the stream header structures to store the incoming streams.

Separate streams are defined for both the Video and Audio data. Each stream is characterized by certain features which include bandwidth, size of the data packet, type of data transfer, data format and medium of data transfer such as through DMA channels or simple IO transfer etc…

The Client Application gets notified of the incoming streams with the help of Event notification routines defined by the application itself. The FDO signals the event
whenever a stream is read and this goes on asynchronously which internally increases the performance of the driver.

A Filter Device Object (FiDO) can be used to enhance the features of the FDO. This filter driver is implemented based on the WDM Architecture as an Upper filter driver, which would help us to modifying both the incoming and outgoing stream data. The I/O Manager generates the application Irps which are hooked by the Filter Device Object in the Filter Driver Module of the System.

![Fig 3-Context Diagram of USB Video Streaming Filter Driver](image)

**Implementation of Streaming Filter Driver**

Most of the routines in the Streaming Filter Driver are almost same as that of a Filter Driver as seen in previous sections. A Streaming Filter Driver should support certain predefined set of I/O Control Codes (IOCTLs) that are required to implement Kernel Streaming.

A brief description of the routines is given below.
Driver Entry Routine

The Streaming Filter Driver has to install Dispatch routines for all the Irps and not just for those it expects to handle. In simple terms, the filter driver should support all the Irps that the target device supports. If the filter driver leaves out any of the Irps, then that particular Irp gets failed with STATUS_INVALID_DEVICE_REQUEST.

Add Device Routine

Filter drivers have AddDevice functions that get called for each appropriate piece of hardware. We'll be calling IoCreateDevice to create an unnamed device object and IoAttachDeviceToDeviceStack to plug into the driver stack. In addition, we'll need to copy a few settings from the device object underneath us.

Specific Routines of the Streaming Filter Driver

Apart from these two routines most of the stream handling is done through control codes once the connection is established. In order to trap the stream data and modify the data, the driver has to handle all the control codes or IoctlS. Information about the pins supported can be obtained by trapping the IOCTL_KSPROPERTY control code and the stream data can be obtained by trapping both IOCTL_KS_READ_STREAM and IOCTL_KS_WRITE_STREAM control codes.

Let's look at how these control codes are handled and the routines that have to be implemented.

DeviceIoControl Dispatch Routine
All the Control Codes are passed as parameters to the IRP_MJ_DEVICE_CONTROL Irp, and the dispatch routine for this Irp is of great importance.

**Handling IRP_MJ_DEVICE_CONTROL**

```c
NTSTATUS
SFDDispatchDeviceIoControl ( 
    IN PDEVICE_OBJECT fido,
    IN PIRP Irp
)
{
    ULONG IoControlCode;

    1. IoSetCompletionRoutine registers the specified routine to be called when the next-lower-level driver has completed the requested operation in any or all of the following ways:
       o Successfully
       o With an error
       o Canceled the IRP

    In this case the function SFDDeviceIoCompletion is called.

    2. The next statement (IoCallDriver) illustrates the standard way of passing a request down to the next layer in a driver stack. The LowerDeviceObject pointer in the device extension gets initialized in AddDevice after calling IoAttachDeviceToDeviceStack

    Once the Completion routine is set for handling the Irp (IRP_MJ_DEVICE_CONTROL), most of the processing now focusses
around the Completion routine, where the data and the property sets are about to be handled.

**DeviceIoCompletion Routine for IRP_MJ_DEVICE_CONTROL**

**Handling IRP_MJ_DEVICE_CONTROL**

```c
NTSTATUS SFDDispatchDeviceIoControl(
    IN PDEVICE_OBJECT fido,
    IN PIRP Irp
)
{
    ULONG IoControlCode;

    PIO_STACK_LOCATION IrpStack = IoGetCurrentIrpStackLocation(Irp);
    PIO_STACK_LOCATION NextIrpStack = IoGetNextIrpStackLocation(Irp);
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
    NTSTATUS status;

    IoControlCode = IrpStack->Parameters.DeviceIoControl.IoControlCode;

    *NextIrpStack = *IrpStack;

    IoSetCompletionRoutine(
        Irp,
        SFDDeviceIoCompletion,
        NULL,
        TRUE,
        TRUE,
        TRUE
    );
```

CSWL Inc, Pleasanton, California - 36 -
2 status=IoCallDriver(pdx->LowerDeviceObject,Irp);
    return status;

}

1. IoSetCompletionRoutine registers the specified routine to be called when
the next-lower-level driver has completed the requested operation in any or
all of the following ways:
   o Successfully
   o With an error
   o Canceled the IRP

In this case the function SFDDeviceIoCompletion is called.

2. The next statement (IoCallDriver) illustrates the standard way of passing a
request down to the next layer in a driver stack. The LowerDeviceObject
pointer in the device extension gets initialized in AddDevice after calling
IoAttachDeviceToDeviceStack

Once the Completion routine is set for handling the Irp
(IRP_MJ_DEVICE_CONTROL), most of the processing now focusses
around the Completion routine, where the data and the property sets are
about to be handled.

DeviceIoCompletion Routine for IRP_MJ_DEVICE_CONTROL

Handling IOCTL_KS_READ_STREAM and IOCTL_KS_WRITE_STREAM
Two major control codes or Ioctls, where the driver can get access to the data stream are the IOCTL_KS_READ_STREAM and IOCTL_KS_WRITE_STREAM. These requests contain a Stream Header list, optionally pointing to the actual stream data. Whenever streaming is done under a standard interface, KSSTREAM_HEADER structure is used.

case IOCTL_KS_READ_STREAM:
{
  DbgPrint("SFDriver:IoCompletion
          IOCTL_KS_READ_STREAM \n");
  // The Stream data buffer is trapped,
  // and can be altered in
  // the SFDMODifyDataBuffer.
  if(sysBuffer!=NULL)
    SFDMODifyDataBuffer
      (sysBuffer,outBufferLength);
  break;
}

The KSSTREAM_HEADER structure describes a presentation time and options that refer to a data buffer and stream. This structure has a pointer to the virtual address of the actual data (KSSTREAM_HEADER.Data).

**Handling IOCTL_KS_PROPERTY**

Another important streaming control code is IOCTL_KS_PROPERTY, which helps the client to query information about the pins and the property sets. The Stream Class Driver uses a function KsPropertHandler, which handles all the property requests through IOCTL_KS_PROPERTY. However, the filter driver cannot call this function and handle the Ioctl. In order to hook the parameters
passed on to the Property Handler, the WDM Streaming library provides an Irp Management function KsDispatchSpecificProperty.

case IOCTL_KS_PROPERTY:
{
    DbgPrint("SFDriver:IoCompletion Of IOCTL_KS_PROPERTY \n");
    // Dispatch the property sets to ur Filter Handler
    2 KsDispatchSpecificProperty (Irps,PFNKSHANDLER) SFDPropKsHandler);
    break;
}

The KsDispatchSpecificProperty function dispatches the property to a specific handler. The function assumes that the caller has previously dispatched this IRP to a handler through the KsPropertyHandler function. The property set handler KsPropertyHandler (PFNKSHANDLER callback) takes three parameters, the current Irp, and a pointer to the KsIdentifier structure and a pointer to the data being passed. The data being passed to the callback depends on the type of request done.

SFDModifyDataBuffer is a helper function, which modifies the data buffer, and also implements noise filtering algorithm.
KsDispatchSpecificProperty is a Streaming library function, which helps the developer to enumerate the KsPropertyHandler callback.

PFNKSHANDLER CallBack

Most of the IOCTL_KS_PROPERTY requests are handled in the KsPropertyHandler callback function, which is of type PFNKSHANDLER. This function takes three parameters, an Irp, a KSIDENTIFIER structure and a data
buffer. Of these parameters, the second parameter contains information about the
type of request and purpose of the request.

The KsIdentifier structure is comprised of three variables. The first variable
KsIdentifier.Set is a GUID, which gives the type of property set supported. The
second variable KsIdentifier.Id is a ULONG, whose value varies depending on the
type of property set supported as provided by the first variable. The final variable
KsIdentifier.Flags is also a ULONG, which tells whether the property information
has to be set or has to be retrieved.

Pin properties and Connection states can be obtained by trapping
KSPROPSETID_Pin and KSPROPSETID_Connection property sets respectively.
The pin property set would give information such as the number of pins supported
and their instances, pin types, pin dataflow, the type of communication, data ranges
and interfaces. In the same way, the Connection property set is queried to get
information on the status of the streaming, the dataformat if changed, and also
about the pin priority settings if it exists. The type of query being done can be
obtained from KsIdentifier.Id value.
Installation of the Streaming Filter Driver

Before installing the driver, there are some pre-requisites that have to be met with.

- **System Requirements**
  - Pentium 200MHz or above
  - USB Port
  - Windows 98 or Windows 2000

- **Building Environment**
  - Microsoft Visual C++ 5.0 or above
  - Platform SDK
  - Win98 DDK or Win2000 DDK kit

- **Other Requirements For Debugging and Testing**
USB Camera Software
Numega SoftIce DriverBundle (Ver 3.24 or above)
Regedit

As this driver is an UpperFilter driver that is attached to an existing Function driver the traditional method of using INF files doesn’t work. Some modifications in the registry have to be done to load the driver.

The key, both for Windows 98 and Windows 2000, is adding an UpperFilters value to the hardware registry key for the device.

One can easily add a filter to an existing WDM driver by adding the UpperFilters to the hardware key for the device. This is the key below HKLM\Enum, and it depends on the enumeration of the device (e.g., USB, PCI, etc.) and what the hardware ID of the device is. The value of the UpperFilters is the name of the filter driver to be installed. Copy the driver file to the System directory (\Winnt\system32\drivers) and reboot the system.

In Win98, the UpperFilters value is of type REG_SZ whereas in Windows 2000 it’s of REG_MULTI_SZ.

For Uninstallation of the Filter driver, remove the entry from the registry and reboot the system.

Implementation of Noise Filtering Algorithm

Image Filtering is a process by which images can be enhanced. One of the techniques used for this purpose is the "Weighted average" method. In this method, the value of a pixel changes with respect to its surrounding pixel values. Depending on the impact of the surrounding pixel values on the center pixel or the hot pixel, the filtering effect also changes.
Our filter driver sits on the top of the class driver and it should trap all the incoming IOCTLs from the class driver. As far as our streaming architecture is concerned we have to trap necessarily the IOCTL_READ_STREAM & IOCTL_KS_WRITE_STREAM. These requests contain a stream header list(KSSTREAM_HEADER), each optionally pointing to the actual stream data. When submitting an IOCTL_KS_READ_STREAM, portions of the stream header are filled in by the call. The pIrp->IoStatus.Information element contains the total size of the header data to return, which is at least one sizeof KSSTREAM_HEADER. The headers could be longer if this is defined by the data format. And in our case we are getting the size of the stream header longer which specifies the inclusive size of the entire stream header and an extended portion.

The extended part is found to be KSDATAFORMAT(KSDATAFORMAT is a variable-length structure that describes a data format.). Moreover the KSSTREAM_HEADER contains the DATA field which points to the virtual address of the actual data. And we have to apply the Filtering algorithm to this actual data, which involves the following steps.

The process is fairly simple:

- Make a copy of the image (the actual data) in memory. We'll use this as the destination for our filter. (makes the filter work much better)
- Set aside a "final" pixel value variable. For height-based filters, this will probably be an integer and color-based will use a rgb triplet.
- For each pixel in the image,
  - For the pixel and those around it,
    - Get the pixel's value and multiply it by the corresponding value in the filter grid.
    - Add this result to the total.
  - Now divide this number by the "division factor". And put the "final" pixel value back in the image.
In C, the outermost layer would look something like this:

```c
for(y=top; y<=bottom; y++)
for(x=left; x<=right; x++)
{
...
}
```

This just sets up a two-dimensional loop to process each pixel in the image.

In order to apply the filter, and explain a "filter grid" is used which explains what the filter does. Here's a 3 * 3 example:

(An example grid)

```
1 1 1
1 2 1
1 1 1
```

For each pixel in the image, get the corresponding value in the filter grid and multiply it. Add the result to the total value. Divide this total by the "Division factor" and replace the pixel by this resultant value.

The "Division Factor" refers to the total of all the values in the filter grid which is 10 here.

**Performance Section**

Looking at the performance of the filter driver would clear doubts, pertaining to the system performance, speed and the CPU usage. Comparing the output of the USB
Camera before and after loading the filter shows that there is not much visible change. The incoming stream is not slowed down, by the filter driver and even the algorithm works fine. These results show that the Overall performance is appreciable.

Conclusion

Implementation of Streaming Filter Driver using the above specified method relieves burden faced by programmers in writing a WDM Filter Driver for an existing Function device. Most of the operations like filtering and Image processing are done on the incoming data on the fly, without buffering. This type of implementation improves the performance of the filter driver as a whole. A Setup Wizard can be written to make installation of the driver more easier. The filter driver can be enhanced to meet the requirements of complex filtering algorithms as well.
References

Website References

www.eu.microsoft.com\hwdev\siteinfo.htm
www.oneysoft.com
www.usb.org

Newsgroups

comp.os.ms-windows.programmer.nt.kernel-mode

Windows 98 DDK Documentation, Windows 2000 DDK Documentation

Kernel mode Streaming Reference
Stream Class Driver Reference

Copyright Notice:

2002 California Software Labs. All rights Reserved. The contents on the document are not to be reproduced or duplicated in any form or kind, either in part or full, without the written permission of California Software labs. Product and company names mentioned here in are the trademarks of their respective companies.