Analysis of Device Driver Design based on WDF

Fazhen Wang [a], Shaohui Cui [b], Cheng Wang [c] and Ran Zhao [d]

School of Shijiazhuang Campus, Army Engineering University, Shijiazhuang 050003, China.

[a]527316850@qq.com, [b]cuish@163.com, [c]32626364@qq.com, [d]newranzhao@163.com

Abstract. In order to realize the expected function of the hardware devices in a computer system, the device driver as an important part cannot be lost. The WDF is the modern standard for creating Windows drivers, and is the preferred way to implement most new drivers for Windows. WDF enables developers to write drivers that execute in either kernel-mode using the KMDF or user-mode using the UMDF V2. This paper firstly analyzes the importance of the driver from the Windows operating system architecture. Then, the paper discusses and analyzes the structure of WDF and its characteristics. Finally, the basic programming skills are given based on simple examples.

Keywords: WDF; KMDF; Driver structure.

1. Introduction

The device driver is a software interface that connects the hardware device to the computer operating system. Only the device has a corresponding driver to can it work normally on the computer system [1]. The driver is at the bottom of the operating system and is closely linked with the kernel of the operating system. So the hardware device can be directly operated by the driver, and the communication between the computer and the device can be realized. This reflects the extremely important position of the device driver in the system. The operating system must use the driver to control the hardware device. If the device driver fails to implement the correct installation, it will not work properly. Hardware devices such as hard disks, monitors, etc. have been installed during the system installation process. However, external hardware devices, such as printers and cameras, can only be used after the installation of corresponding drivers.

With the continuous updating of Windows systems, from the Windows 95/98 operating system to the Windows 10 operating system, Windows driver model development has also changed. The development of the driver model has evolved from VXD driver to WDM driver and then to WDF driver. This article mainly analyzes the WDF driver design.

2. Windows Operating System Structure

The Windows operating system is a layered operating system, and its operation depends on the call of the upper component to the lower component [2]. Each layer of the system contains several components and each of the components has its own fixed interface. The closer the component is to the bottom, the higher the operational privileges it has. And the components near the upper layer convert the task into a call to the underlying component.

The design philosophy of Windows operating system is to design the kernel as small as possible and the operating system adopts the "client-server" architecture. Each component or module of the operating system communicates with other component or module via messages.

Fig. 1 shows the Windows system architecture.

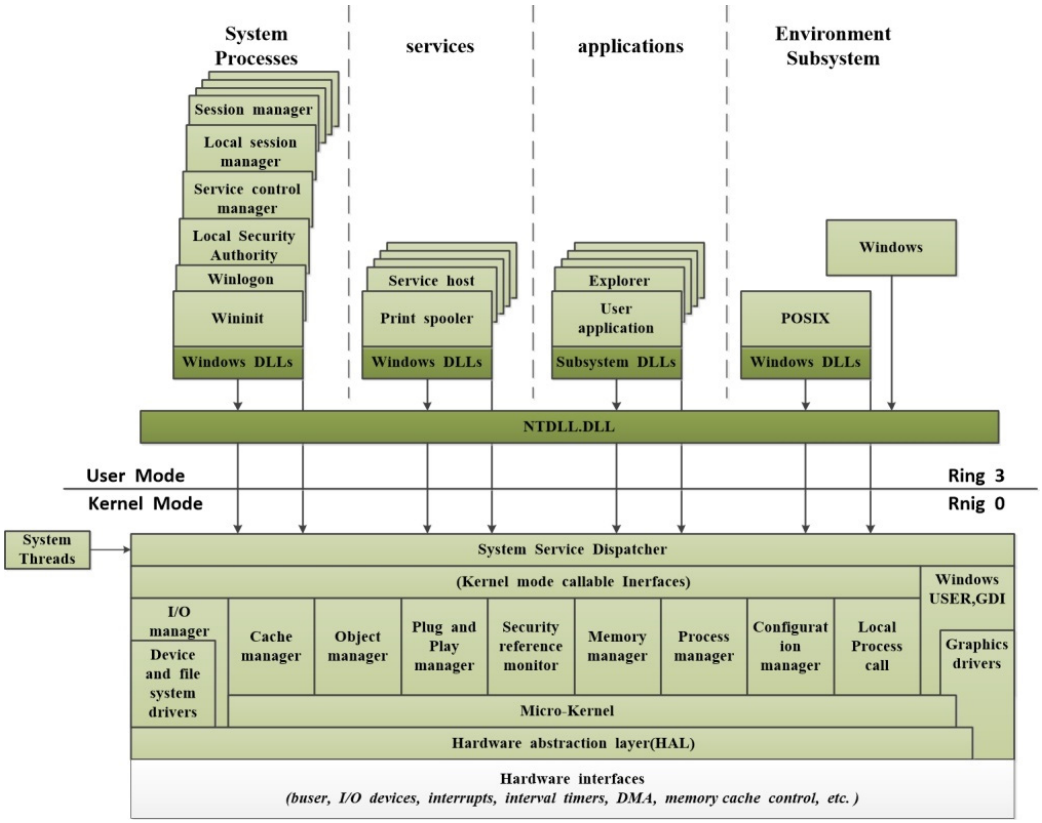

Fig. 1 Windows system architecture

It can be seen from the Window operating system architecture that it is generally divided into kernel mode and user mode. The kernel mode interfaces serve the user mode application.

The OS kernel mode runs in Ring 0 and the OS user mode runs in Ring 3. Because of the privileges restricted, the normal programs which run in Ring 3 have a number of restrictions. If an application wants to directly access sensitive operations, such as physical memory, physical ports, etc., it needs to make a request to a component in the kernel mode.

The Windows driver described in this article is a driver that executes in kernel mode.

3. WDF Driver

3.1 WDF Driver Introduction

Windows Driver Foundation (WDF) is the latest driver development framework. It develops based on the Windows Driver Model (WDM) and supports object-oriented, event-driven driver development [3].

The driver framework manages communication related to the operating system kernel. By isolating the driver from the system kernel, the driver's influence on the kernel is reduced and system stability is improved [4].

WDF defines a single driver model that is supported by two frameworks: Kernel-Mode Driver Framework (KMDF) and User-Mode Driver Framework (UMDF). While UMDF version 2 offers a significant subset of functionality that was previously available only to KMDF drivers [5].

If you need one of those features shown in Table 1, you still must write a KMDF driver.

Table 1. Features are available only to KMDF drivers

Feature	Related information
Direct memory access	Handling DMA Operations in KMDF Drivers
Bus enumeration	Enumerating the Devices on a Bus
Functional power states(limited support is available in UMDF)	Supporting Functional Power States
Access to WDM objects and IRPs	Obtaining WDM Information
Neither Buffered Nor Direct I/O	Accessing Data Buffers in WDF Drivers Intercepting an I/O Request before it is Queued
Internal device control requests	Sending I/O Requests Synchronously Sending I/O Requests Asynchronously
Remove lock opt-in for I/O requests	WdfDeviceInitSetRemoveLockOptions
Windows Management Instrumentation	provide data to WMI clients

3.2 WDF Driver Structure

The user program running in user mode implements interaction with the kernel driver through the API. After receiving the request from the user program, the kernel subsystem encapsulates the request into an IO Request Package (IRP) and sends it to the device for processing. The kernel subsystem is also responsible for processing the request after completion. The dispatch functions of various IRP commands are implemented inside the framework. These functions are responsible for receiving and sending IRP packets.

The WDF driver uses the device driver interface provided by the framework to accomplish the task. The WDF driver usually lets the WDF framework pass the IO requests to the bottom drivers and devices. But in some special cases, the WDF driver can also directly obtain the IRP and transfer it to the lower layer for processing [6]. The greatest value of the WDF framework is the realization of three important models, namely plug-and-play (PnP) / power model, I/O model and object model. Fig. 2 shows the architecture diagram of the WDF driver.

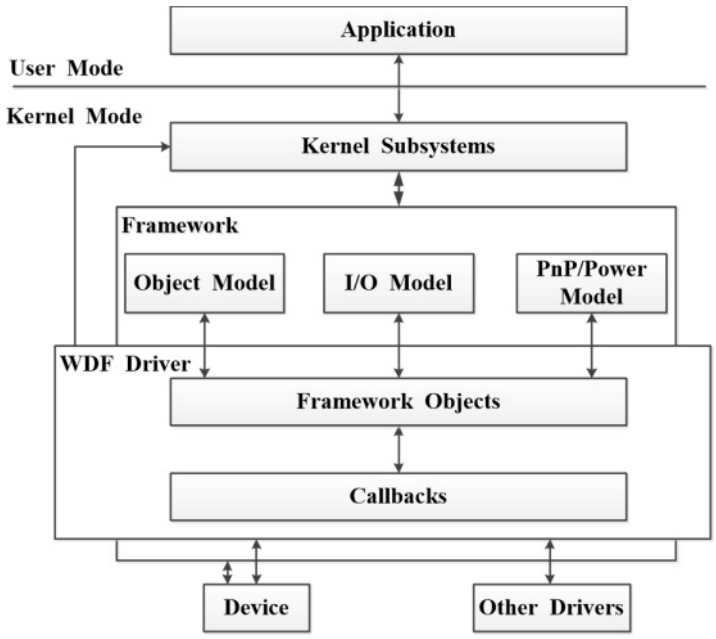

Fig. 2 WDF architecture

3.2.1 Object Model

The object model of WDF is a hierarchical model. Each driver has a unique drive object, and the WDFDRIVER object is the root object of all WDF objects. Each object has its own properties, methods, and events. Drivers can use these methods to set properties, create objects, and achieve inci–dent response [7]. The main objects defined in the WDF object model and their levels are shown in Fig. 3.

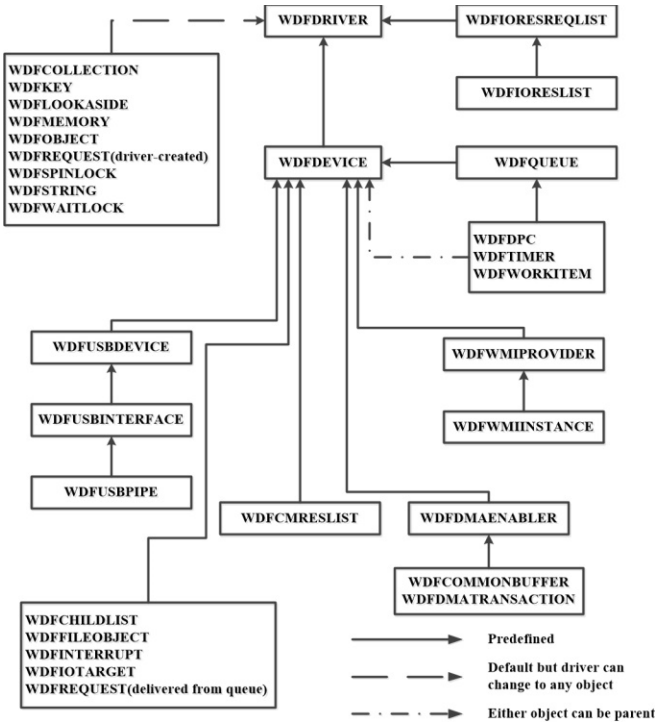

Fig. 3 The main objects defined in the WDF object model

3.2.2 I/O Model

Windows 2000 and later operating systems use IRP to communicate with kernel-mode drivers. When the upper application communicates with the bottom driver, the I/O manager converts the I/O request sent by the application into an IRP and sends it to the device driver. The MajorFunction is a basic attribute of the IRP. The IRP Dispatcher dispatches IRP according to its MajorFuntion [8].

Fig. 4 shows how the WDF driver processes the IRP.

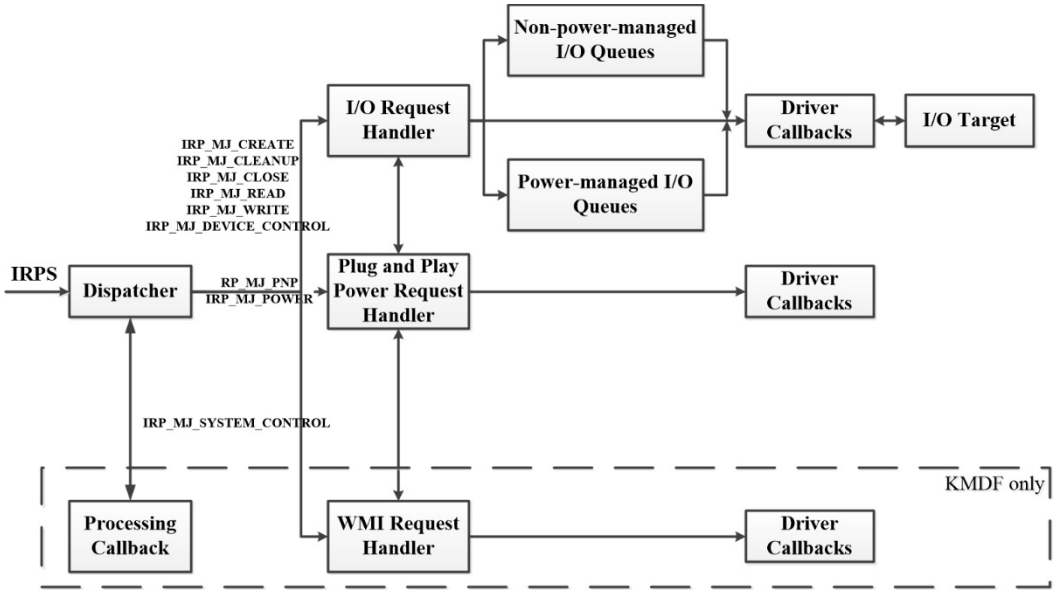

Fig. 4 Process for processing IRP

3.3 Driver Design based on WDF

A plug-and-play device driver includes:
(1) A DriverEntry routine;
(2) An EvtDriverDeviceAdd routine;
(3) One or more I/O queues;
(4) One or more I/O event callback routines;
(5) Plug-and-Play and power management callback routines;
(6) Interrupt processing routines;
(7) Other callback routines, such as DMA routines, cleanup routines, etc.

3.3.1 DriverEntry Routine

When the operating system loads the driver, it firstly completes the DriverEntry routine. This routine is the main entry address of the WDF driver, through which the driver creates the driver object and sets the EvtDriverDeviceAdd routine address [9].

```
NTSTATUS Driver Entry ( IN PDRIVER_OBJECT DriverObject,
                                IN PUNICODE_STRING RegistryPath )
{
WDF_DRIVER_CONFIG config;
NTSTATUS status=STATUS_SUCCESS;
WDF_DRIVER_CONFIG_INIT(&config, MyEvtDeviceAdd);
// The WDF_DRIVER_CONFIG_INIT function must be called to initialize the
                        WDF_DRIVER_CONFIG structure before calling WdfDriverCreate.
//    Create a device object
status=WdfDriverCreate( DriverObject, RegistryPath, WDF_NO_OBJECT_ATTRIBUTES,
                        &config, WDF_NO_HANDLE);
return status;
}
```

3.3.2 EvtDriverDeviceAdd Routine

After the driver is initialized, the PnP Manager calls the driver's EvtDriverDeviceAdd routine to initialize the device controlled by the driver [10]. Configuring object, creating an I/O queue, establishing a device GUID interface, seting various event callback routines and interrupting handling are completed by calling the EvtDriverDeviceAdd routine.

```
NTSTATUS EvtDriverDeviceAdd ( IN WDFDRIVER Driver,
                                    IN PWDFDEVICE_INIT DeviceInit)
{    // structure declaration
WDFSTATUS status;
WDF_PNPPOWER_EVENT_CALLBACKS pnpPowerCallbacks;
WDF_OBJECT_ATTRIBUTES objAttributes;
WDFDEVICE mydevice;
PMY_DEVICE_CONTEXT devContext;
WDF_IO_QUEUE_CONFIG ioqueueConfig;
WDF_INTERRUPT_CONFIG interruptConfig;
   // To set event callback functions related to PnP and power management
WDF_PNPPOWER_EVENT_CALLBACKS_INIT(&pnpPowerCallbacks);
pnpPowerCallbacks.EvtDevicePrepareHardware=MyEvtPrepareHareware;    // initialization
pnpPowerCallbacks.EvtDevicePrepareHardware=MyEvtReleaseHareware;    // stop
pnpPowerCallbacks.EvtDeviceSurpriseRemoval=MyEvtSurpriseRemoval; // abnormal removal
......
   // To create a WDF device.
status=WdfDeviceCreate(&DeviceInit, &attributes, &device);
```

```
    // To create a device GUID interface
status=WdfDeviceCreateDeviceInterface(device,(LPGUID)&My_DEVINTERFACE_GUID,
                                      NULL);
......
    // To create and initialize a request queue
WDF_IO_QUEUE_CONFIG_INIT (&ioqueueConfig, WdfIoQueueDispatchSequential);
ioqueueConfig.EvtIoDeviceControl=MyEvtDeviceControlIoctl;
status=WdfIoQueueCreate( myDevice, & ioqueueConfig, WDF_NO_OBJECT_ATTRIBUTES,
                         NULL );
.......
    // To create and initialize interrupt objects
WDF_INTERRUPT_CONFIG_INIT(&interruptConfig, MyEvtInterruptIsr,
                                      MyEvtInterruptDpc );
status = WdfInterruptCreate(mydevice, &interruptConfig, & objAttributes,
                         &devContext->WdfInterrupt);
......
    // Other initialization operations (omitted)
}
```

3.3.3 I/O Processing Routine

In the WDF driver, queues are used to process all I/O requests for the application. The WDF framework has a default queue or you can create your own queue. If these I/O requests are needed to queue separately, they can be managed separately by multiple queues.

3.3.4 Interrupt Service Routine

The hardware interrupt is handled by the WDFINTERRUPT object in the WDF driver, and the interrupt service routine (ISR) handles the hardware interrupt, which runs at the IRQL level. The deferred procedure call routine (DPC) handles the interrupt generated by the device itself, which runs at the DISPATCH_LEVEL level.

4. Summary

WDF, the latest generation driver development model launched by Microsoft, provides support for object-oriented, event-driven kernel-mode driver development frameworks. It greatly simplifies the development of device drivers and avoids the entire system crash due to driver errors. The object model and driver structure of WDF are analyzed in this paper, and some examples are given, which has certain significance for device driver development.

References

[1]. Yuhong Qian. Development of WDF Drivers for USB Data Transfer Card. Computer Applications and Software. Vol. 29 (2012) No. 6, p. 225-227.

[2]. Fan Zhang, Caicheng Shi, et al. Windows Driver Development Internals. Publishing House of Electronics Industry, 2008, p. 37.

[3]. Zhengping Li, Chao Xu, Junning Chen, et al. Design and Implementation of WDF Device Driver. COMPUTER TECHNOLOGY AND DEVELOPMENT. Vol. 17 (2007) No. 5, p. 228-230.

[4]. Jingxuan Zou, Wandong Cai. A USB Storage Device Monitor and Control System Based on the WDF Filter Driver. COMPUTER ENGINEERING & SCIENCE. Vol. 32 (2010) No. 3, p. 42-44.

[5]. Information on: https://docs.microsoft.com/zh-cn/windows-hardware/drivers/wdf/ comparing-umdf-2-0-functionality-to-kmdf.

[6]. Chengwei Yang: The Design and Implementation of PXIExpress Bus Interface to The DMA Controller and Driver (Master of Engineering, School of Aeronautics & Astronautics, China 2016). p.45.

[7]. Aimei Song, Jianjian Xu. A Method of USB Driver Program Design Based on WDF. Experiment Science and Technology. Vol. 10 (2012) No. 1, p. 58-63.

[8]. Afeng Yang: Development of WDF Driver Program for High-Speed Data Transmission Adapter on PCIe Interface (Master of Engineering, School of National University of Defense Technology, China 2008). p.45.

[9]. Wenjun Xiao, Wansong Liu, Wufeng Liu, et al. Design and Implementation of Driver Program for PCIe Based on WDF. MEASUREMENT & CONTROL TECHNOLOGY. Vol. 34 (2015) No. 7, p. 101-104.

[10]. Penny Orwick. Developing Drivers with the Microsoft Windows Driver Foundation. Microsoft Press, 2007, p. 30.