A&QT-R 2002 (THETA 13)
International Conference on Automation, Quality and Testing, Robotics
May 23-25, 2002, Cluj-Napoca, Romania

WINDOWS DRIVER MODEL AND IMPLEMENTATION FOR A DSP BOARD

Eng. Alin Inclezan, Technical University Cluj-Napoca
Eng. Anthony Hunt, Signatec U.S.A.
Prof.dr. eng. Gavril Toderean, Technical University Cluj-Napoca

Email: toderean@cluj.astral.ro
Web: http://pages.astral.ro/pro3soft

ABSTRACT

A Windows Driver Model (WDM) for a DSP board and its implementation of this model. The first five parts contains the description of the WDM from the point of view of synchronization, structure, routines, communication, plug and play. The last parts explain how this model was implemented for a PCI waveform capture board, and draw the conclusions.

KEYWORDS: Windows Driver Model

1. INTRODUCTION

A driver is a kernel-mode code used either to control or to emulate a hardware device. It provides a software interface to the hardware connected to a computer. A user application program can access the hardware in a well-defined manner, without having to worry about how the hardware must be controlled [3].

The driver becomes a part of the operating system kernel when it is loaded. It makes one or more devices available to the user-mode programmer.

In Windows, a driver always makes a device look like a file and an application program can open a handle to the device, issue read and write request and close the handle.

Within the Windows 2000 operating system, there are two basic kinds of drivers:

- User-mode drivers (VDD) are user-mode components that allow DOS-based applications to access hardware on x86 platforms
- Kernel-mode drivers for logical, virtual, or physical devices

Kernel-mode drivers share many of the design goals of Windows 2000. These design goals include the following [1]:

- Portability from one platform to another
- Configurability of hardware and software
- Always preemptible and always interruptible

- Multiprocessor-safe on multiprocessor platforms
- Object-based
- Packet-driven I/O with reusable *I/O request packets* (IRPs)
- Support for asynchronous I/O

2. SYNCHRONIZATION IN THE DRIVER

Microsoft Windows 2000 is a multitasking operating system that can run in a symmetric multiprocessor environment. In general, we need to assume two worst-case scenarios:

- The operating system can preempt any subroutine at any moment, for an arbitrarily long period of time, so that we cannot be sure of completing critical tasks without interference or delay.
- Even if we take steps to prevent preemption, simultaneous code executing on another CPU, in the same computer, can interfere with our code. It is even possible that the exact same set of instructions, belonging to one of our programs, could be executied in parallel in the context of two different threads.

These general synchronization problems may be solved by using the *interrupt request level* (IRQL) priority scheme and by claiming and releasing *spin locks* around critical code sections. IRQL avoids destructive preemption on a single CPU, while spin locks forestall interference among CPUs.

3. THE STRUCTURE OF A WDM (WINDOWS DRIVER MODEL)

In the WDM, each hardware device has at least two device drivers. One of these drivers is the *function driver*, i.e. what one always thought of as being "the" device driver. It understands all the details about how to make the hardware work. It is responsible for initiating I/O operations, for handling the interrupts that occur when those operations finish, and for providing a way for the end user to exercise whatever control over the device might be appropriate [1].

The other of the two drivers that every device has is the *bus driver*. It's responsible for managing the connection between the hardware and the computer. Above and below the FDO there might be a collection of *filter device objects.* Filter device objects from above the FDO are called *upper filters,* whereas filter device objects from below the FDO (but still above the PDO) are called *lower filters*.

For each of these drivers there is a device objects structure, that the system creates, to help the software to manage the hardware. The lowest-level device object in a stack is called the physical device object (PDO). Somewhere in the middle of a device object stack, there is the functional device object (FDO). Above and below the FDO there might be a collection of filter device objects (FiDO).

The I/O Manager uses a *driver object* data structure to represent each device driver and a *device object* structure to represent each of the devices managed by the driver.

4. THE MAIN ROUTINES OF THE DRIVER

4.1. The DriverEntry routine

Each driver must have a **DriverEntry** routine, which initializes global data structures and resources. The I/O Manager calls the **DriverEntry** routine when it loads the driver. The main job of this function is to set the driver's Dispatch, the AddDevice, the StartIo (if any), and the Unload (if any) entry points in the driver object. The purpose of the **Unload** function is to clean up after any global initialization, that DriverEntry might have done [2].

4.2. The AddDevice routine

The PnP Manager calls a driver's AddDevice routine for each device controlled by the driver. The basic responsibility of the AddDevice in a function driver is to create a device object and link it into the stack rooted in this PDO.

5. THE COMMUNICATION BETWEEN THE OS , THE APPLICATIONS AND THE WDM - THE I/O REQUEST PACKET

The I/O Request Packet (IRP) is a structure used by the operating system to communicate with a kernel-mode device driver.

When a driver receives an IRP, the Dispatch routine should get the current stack location and examine the type of IRP or determine the parameters. Then, this routine should decide how to handle the IRP. It can complete the request immediately, pass the request down to a lower-level driver, in the same driver stack, or queue the request for a later processing by other routines in this driver. Usually, when the IRP takes a long time to complete, it is queued for a later processing.

6. THE WDM AND PLUG AND PLAY

The Plug and Play Manager communicates with the device drivers using IRPs, with a major function code of IRP_MJ_PNP. This type of request is new in Windows 2000. In Windows NT, the driver should do most of the work of detecting and configuring their devices. The WDM drivers can let the PnP Manager do this work.

The PnP Manager sends IRP_MN_START_DEVICE to inform the function driver what I/O resources it has assigned to the hardware, and to instruct the function driver to do any necessary hardware and software setup so that the device can function.

The PnP Manager sends IRP_MN_STOP_DEVICE to stop a device in order to be able to reconfigure the device's hardware resources. The PnP Manager sends this IRP only if a prior IRP_MN_QUERY_STOP_DEVICE has been completed successfully. A driver should not complete the IRP_MN_QUERY_STOP_DEVICE successfully, if it can not stop the device. The device might be busy with a request that will take a long time and can't be stopped in the middle.

Another important PnP request is IRP_MN_REMOVE_ DEVICE. This request asks the driver to stop the device and to clean up any data structures associated with the device. The PnP Manager asks the permission to remove the device using an IRP_MN_QUERY_REMOVE_DEVICE request.

7. READING AND WRITING DATA BETWEEN THE DRIVER AND THE APPLICATIONS, AND BETWEEN THE DRIVER AND THE DEVICE

When an application initiates a read or write operation, it provides a data buffer by giving to the I/O Manager a user-mode virtual address and length. In general, a driver runs in an arbitrary thread context (not knowing if it is in the context of the thread which initiated the I/O Request), so it hardly ever accesses memory using a user-mode virtual address. There are three ways to access a user-mode data buffer :

- **the buffered method :** the I/O Manager creates a system buffer equal in size to the user-mode data buffer. You work with this system buffer. The I/O Manager takes care of copying data between the user-mode buffer and the system buffer
- **the direct method :** the I/O Manager locks the physical pages containing the user-mode buffer and creates an auxiliary data structure, called a memory descriptor list (MDL) to describe the locked pages. The driver works with the MDL.
- **the neither method :** the I/O Manager simply passes the user-mode virtual address to you.

A device receives its resources when the PnP Manager sends IRP_MN_STOP_DEVICE. These resources are ports and memory ranges, interrupts and DMA channels for devices that are not bus-master .

The Windows 2000 DDK provides functions for reading and writing ports and memory.

The driver should configure an interrupt resource in the StartDevice function, and it should provide an interrupt service routine (ISR) that will be executed when the device will generate an interrupt. The ISRs execute at an IRQL higher than DISPATCH_LEVEL. All code and data used in an ISR must therefore be in the nonpaged memory. Also, the set of kernel-mode functions that an ISR can call is very limited. Since an ISR executes at elevated IRQL, it freezes out other activities on its CPU that would require the same or a lower IRQL. Therefore, for best system performance, the ISR should execute as quickly as possible. Servicing a device interrupt often requires you to perform operations that are not legal inside an ISR, or that are too expensive to be carried out at the elevated IRQL of an ISR. To avoid these problems, Windows 2000 provides the deferred procedure call DPC mechanism. DPC represents a procedure that is to be called later. DPCs are executed in kernel mode at IRQL DISPATCH_LEVEL.

Performing DMA transfers depends on several factors:

- If the device has bus-mastering capability, it has the necessary electronics to access main memory.
- A device with scatter/gather capability can transfer large blocks of data to or from discontiguous areas of physical memory.

- If the device is not a bus master, one uses the system DMA controller on the motherboard of the computer.

8. IMPLEMENTATION

PDA500™ is a PCI waveform capture board, produced by Signatec Incorporated. The PDA500™ is a 32 Bit PCI board compatible with the revision 2.1 of the PCI local bus specification, and with "Plug and Play". The control and monitoring registers are accessed from the PCI bus with 32 bit I/O read/writes. The signal data can be read from, or written to the PDA500 with 32 bit memory transfers, or with 32 bit Bus Master Direct Memory Access (DMA) transfers. The board uses 1 MB memory, a 32 bytes I/O port for Control/Status Registers, and a 64 bytes I/O port PCI Controller [4].

In the driver's DriverEntry routine the driver initialises the dispatch functions for the IRP it receives: IRP_MJ_CREATE, IRP_MJ_CLOSE, IRP_MJ_DEVICE_CONTROL, IRP_MJ_READ, IRP_MJ_PNP, and the pointer to AddDevice.

When the PnP Manager detects a PDA500 board it calls the AddDevice routine. This routine creates and initializes the structures used by the device, puts the new device object into the stack, and creates a symbolic link that can be used by applications to access the driver's functions.

When the driver receives IRP_MN_START_DEVICE, it extracts the resources for the device. The PnP Manager sends these resources in an arbitrary order, so that the driver use the port range to tell what port it received. In the function that handles IRP_MN_START_DEVICE the driver configures the DMA adapter object and the interrupt resource.

The function that handles IRP_MN_QUERY_STOP_DEVICE does not permit the stopping of the device, if there are pending IRPs. Also, the function that handles IRP_MN_QUERY_REMOVE_DEVICE does not permit the removing of the device, if there are opened handles to the device.

An application does not use the driver's function directly. Between the driver and the application there is a Dynamic-Link Library (DLL). The application uses the high-level functions exported by this DLL, and the DLL uses the low-level function exported by the driver.

The main functions implemented by this driver are the acquisition functions. There are three acquisition types:

- the data is acquired and stored in the PDA500 signal RAM
- the data is transferred over the SAB (Signatec Auxiliary Bus)
- the data is transferred directly to the PCI bus.

9. CONCLUSIONS

The driver was implemented using the WDM model and it was proved that it is able to work with multiple PDA500 boards at a maximum digitization rate of 500 Megasamples per second, with 8 bits of resolution.

BIBLIOGRAPHY

1. Walter Oney, (1999), Programming the Microsoft Driver Model, Microsoft Press
2. Microsoft Windows 2000 Driver Development Kit
3. Chris Cant, (1999), Writing Windows WDM Device Drivers, R&D Books
4. PDA500 Operator's Manual