Print

ONE WAY FOR A VIRTUAL NUCLEAR INSTRUMENTATION.

Lic. Guillermo Mesa Pérez*, Ing. Raul Arteche**.
CEADEN P.O.B. 6122, Havana, Cuba. Tel: (537)221518. Fax: (537)221518/241188.
***guille@ceaden.edu.cu**
****arteche@ceaden.edu.cu**

ABSTRACT

This article discusses how to write a dynamical loadable VxDs for the Microsoft® Windows® 95 operating system using the VTOOLSD library, VxDs for Windows 95 may be written in C or C++, loading such VxDs using the Win32® application programming interface (API).

INTRODUCTION

Device drivers serve several different purposes. In their purest form, they are the link between software and hardware. Depending on your point of view, you can see them either as part of the software (because they are generally implemented in software) or as hardware (because they are closely coupled with the hardware device they support, and the rest of the software cannot easily figure out which task is done by the driver and which by the hardware device).

The concept of a virtual device driver arose in Windows mode as a way to "virtualize" hardware devices so that multiple DOS and Windows applications could share them. If I type on the keyboard, for example, my keystrokes might at one time belong to the active Windows application, and at another, to a character-mode program running in a DOS box. Microsoft's designers built a multitasking operating system--WIN386.EXE--around the idea of "virtual machines". Handling the virtual keyboard attached to a virtual machine calls for a virtual keyboard device (VKD) which can direct the actual keystrokes from physical hardware to the correct program in such a way that each consumer of keystrokes believes it's dealing directly with hardware. Handling some other hypothetical "x" device calls for a Virtual "x" Device--a VxD.

How to write a VXD?

VxDs are 32-bit, flat-model programs running in the same privileged *ring-0* (the operating system kernel in other words, the portion that allocates system resources; manages memory, files, and peripheral devices; maintains the time and date; and starts applications).

We have two type of VxD, *static* and *dynamic*.

The most frequently purpose of a static VxD is to virtualize a custom piece of hardware. Because the only way to install a piece of hardware into a computer is to shut down the machine, open up its cover, insert the hardware device, and then restart the computer, VxDs had to be loaded exclusively at system boot time and unloaded at shutdown time.

Dynamic VxDs are written for nonhardware virtualization purposes (for example, to support existing terminate-and-stay-resident programs (TSRs), to implement shared memory between virtual machines, to provide low-level services to Windows-based applications, and to provide 32-bit memory to virtual machines), it became apparent that it would make sense to provide a mechanism to dynamically load and unload VxDs. The dynamic VxD loader is also the heart of the layered device-driver architecture of Windows 95.

To explain how to make a VxD we can use the VTOOLSD libraries that generate automatically skeletons of VxDs. First we can explain how to initialize the virtual machine .

DriverVM::DriverVM(VMHANDLE hVM) : VVirtualMachine(hVM) {}

Define a user class *DriverVM* of the type *VVirtualMachine(hVM) {}* that provide a handler for a system message.

VVirtualMachine are responsible for handling these control messages. Each such message identifies a virtual machine by its handle. The internal control dispatcher automatically locates the class instance associated with the specified virtual machine handle, and invokes the appropriate member function.

When you allocate an instance of a class derived from VVirtualMachine, you invoke the constructor for VVirtualMachine. The constructor stores the pointer that is returned by the invocation of new inside the Virtual Machine Control Block (VMCB) associated with the virtual machine (VM). The VMM maintains a VMCB for each virtual machine, and allows VxDs to allocate small amounts of private storage in it at initialization time. The initialization code for the class library reserves space in the VMCB for a pointer to the VM class instance corresponding to the virtual machine. The virtual machine handle is a pointer to the VMCB. This relationship makes it possible to efficiently dispatch control messages to member functions of class VVirtualMachine.

To make your VxD dynamically loadable, the control procedure of your virtual device has to handle at least two system control messages: SYS_DYNAMIC_DEVICE_INIT and SYS_DYNAMIC_DEVICE_EXIT. If your VxD can be opened from inside a Win32 application, the control procedure must also handle the W32_DEVICEIOCONTROL. It is probably a good idea for your VxD to handle the WIN32_DEVICEIOCONTROL message in any case, just to be prepared to be called from a Win32 application.

This functions are define in VTOOLDs library.

```
BOOL DriverDevice::OnSysDynamicDeviceInit()
{
        return TRUE;
}
```

Process control message Sys_Dynamic_Device_Init.

The system invokes this member function to notify a dynamically loaded VxD that it should perform its initialization, returns TRUE if the VxD initialized successfully. Otherwise returns FALSE to prevent the device from loading.

```
BOOL DriverDevice::OnSysDynamicDeviceExit()
{
        return TRUE;
}
```

Process control message Sys_Dynamic_Device_Exit.

The system invokes this member function to notify a dynamically loaded VxD that it is about to be unloaded. The VxD should release any system resources that it holds at this time.

Another important control message is the W32_DEVICEIOCONTROL

The W32_DEVICEIOCONTROL message is sent to the VxD right after the SYS_DYNAMIC_DEVICE_INIT message, when the first Win32 application opens a handle to the VxD. Unlike SYS_DYNAMIC_DEVICE_INIT, the message is sent again for each application that opens the VxD and for each application that closes the VxD. To distinguish between control codes, you need to check the *dwIoControl* member of the **DIOCParams** structure that ESI points to when the VxD receives this message. The value of *dwIoControl* is set to DIOC_OPEN when the VxD is opened, and to DIOC_CLOSEHANDLE when it is closed. Application-specific initialization and cleanup can be done here.

```
DWORD DriverDevice::OnW32DeviceIoControl(PIOCTLPARAMS pDat)
{
        DWORD status;
switch(pDat->DIOC_IOCtlCode)
    {
    case DIOC_OPEN:

    case DIOC_CLOSEHANDLE:
         status=0;
         break;
    case COMMAND1:
         Dir_OF_APC=*(PVOID*)pDAT->DIOC_InBuf;
         HANDLER_OF_thread=Get_Cur_Thread_Handle();
         status=0;
         break;
    case COMMAND2:
         VWIN32_QueueUserApc(Dir_OF_APC,0,HANDLER_OF_thread);
         status=0;
         break;
  status=0xFFFFFFFF;
```

```
    }
    return status;
}
```

VWIN32_QueueUserApc permit to call the specified ring 3 code in the specified thread and returns a non-zero value if successful in queuing the asynchronous procedure call (APC) and if unable to allocate memory for the APC, returns 0.

In your software.

More than one application can open and close a dynamically loadable VxD. No matter how many applications open the VxD, it will be loaded into memory only once: The VxD will be loaded into memory when the first application opens it, and it will be unloaded after all applications that opened the VxD have closed it.

To open a dynamically loadable virtual device driver from within a Win32 application, you should use the Win32 **CreateFile** function. The *lpFileName* parameter to the **CreateFile** call should point to a string in the form "\\.*vxdname*," where *vxdname* is the actual filename of the VxD to be opened. The *dwDesiredAccess*, *dwShareMode*, *pSecurityAttributes*, and *hTemplateFile* parameters do not matter when opening a VxD and may, therefore, be set to 0. The *dwCreationDistribution* parameter should be set to OPEN_EXISTING, and *dwFlagsAndAttributes* should be set to FILE_FLAG_DELETE_ON_CLOSE.

For example:

```
HANDLE HANDLER_VxD = 0;

HANDLER_VxD = CreateFile("\\\\.\\MyDevice.VXD", 0, 0, NULL, OPEN_EXISTING,
FILE_FLAG_DELETE_ON_CLOSE, 0);
```

The **DeviceIoControl** function sends a control code directly to a specified device driver, causing the corresponding device to perform the specified operation. The *hDevice* parameter to this function is the handle to device of interest; the *dwIoControlCode* is the control code of operation to perform**.** The *lpInBuffer* is a pointer to buffer to supply input data **DWORD** *nInBufferSize***,** // size, in bytes, of input buffer **LPVOID** *lpOutBuffer***,** // pointer to buffer to receive output data **DWORD** *nOutBufferSize***,** // size, in bytes, of output buffer **LPDWORD** *lpBytesReturned***,** // pointer to variable to receive byte count **LPOVERLAPPED** *lpOverlapped* // pointer to structure for asynchronous operation**);**

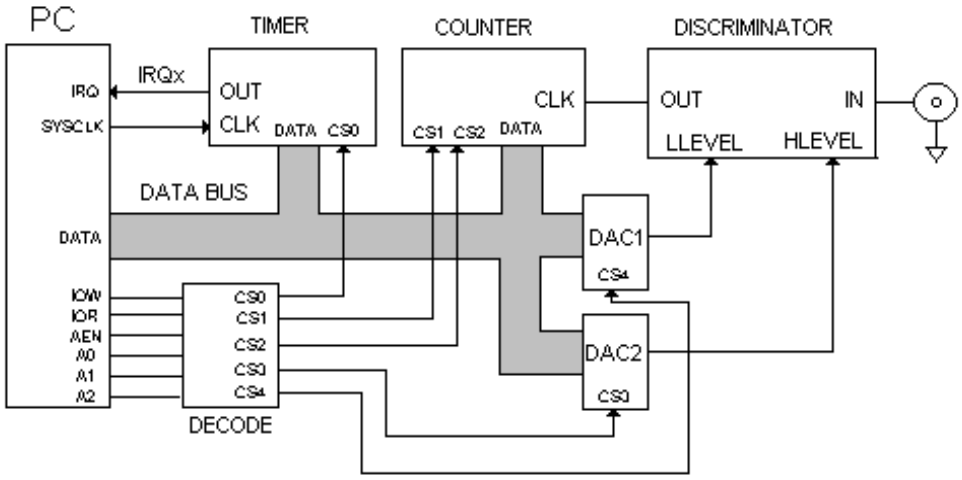
```
BOOL Success

Success = DeviceIoControl(HANDLE_VxD, COMMAND1, *pDAT ,sizeof(pDat), nil, 0,
OutBufferSize, nil);
```

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

To close the previously opened VxD, simply call the Win32 **CloseHandle** function with the handle returned by **CreateFile**.

CloseHandle(HANDLER_VxD);

Example VxD application.



In order to illustrate the above explanation we will take as example a PC card based Single Channel Analyzer. Basically a Single Channel Analyzer is composed by a discriminator, a counter and a timer(fig 1).

Fig.1

To take control of such a system we used 5 port addresses (3 undecoded address lines) and a Interrupt Request Line (IRQ). The first two addresses will control the two D/A converters used by the discriminator to fix the high and low level threshold. Other two addresses will reset and read the counter and the last one will be used to program the timer. The timer will handle the IRQ line to PC.(See fig.1).

The timer requests an interrupt once the programmed time has expired. The VxD driver “sees” the interrupt, immediately reads the counter and transfers the collected data to the software application. All these operations are going to be executed in ring 0 of the Operating System. The timer and converters will be programmed by an application software in ring 3.

To illustrate how can be processed interruption hardware (timer interruption) we include the virtual driver source.

Source code for include file, in this file we define a type of class to use.

// ChannelVxD.h - include file for VxD ChannelVxD

```
#include <vtoolscp.h>
#define DEVICE_CLASS            ChannelVxD_Device
#define ChannelVxD_DeviceID     UNDEFINED_DEVICE_ID
#define ChannelVxD_Init_Order   UNDEFINED_INIT_ORDER
#define ChannelVxD_Major        1
#define ChannelVxD_Minor        0
//----------------------------------------------------------------------
// Codigo de funciones para la comunicación con la aplicación.
#define COMMAND1   0x8100  // Command to register APC
#define COMMAND2   0x8101  // Command to define IRQ and data counter address
//----------------------------------------------------------------------
class ChannelVxD_Device : public VDevice
{
public:
        virtual BOOL OnSysDynamicDeviceInit();
        virtual BOOL OnSysDynamicDeviceExit();
        virtual DWORD OnW32DeviceIoControl(PIOCTLPARAMS pDIOCParams);
};

class ChannelVxD_VM : public VVirtualMachine
{
public:
        ChannelVxD_VM(VMHANDLE hVM);
};

class ChannelVxD_Thread : public VThread
{
public:
        ChannelVxD_Thread(THREADHANDLE hThread);
};

// Define the user's ChannelVxD class
class ChannelVxD_HwInt:public VHardwareInt
{
public:
        ChannelVxD_HwInt();         //Define constructor
        ~ChannelVxD_HwInt();        // Define Destructor
        // Define public events for hardware interruption
        virtual VOID OnHardwareInt(VMHANDLE);
};
```

Main source code of VxD.

```
// ChannelVxD.cpp - main module for VxD Channel
```

```
#define DEVICE_MAIN
#include "ChannelVxD.h"
Declare_Virtual_Device(ChannelVxD)
#undef DEVICE_MAIN

//----------------------Static Declaration .------------------------
ChannelVxD_HwInt   *HwInt=0;
int                NUM_IRQ=7;
int                DatCounterAdress=0x300;
PVOID              Dir_APC=0;
THREADHANDLE       Handler_Thread=0;
BOOL               FLAG=FALSE;



//Declare Virtual Machine class
ChannelVxD_VM::ChannelVxD_VM(VMHANDLE hVM) : VVirtualMachine(hVM) {}

//
ChannelVxD_Thread::ChannelVxD_Thread(THREADHANDLE  hThread)  :  VThread(hThread)
{}

// Initialize VxD
BOOL ChannelVxD_Device::OnSysDynamicDeviceInit()
{
        return TRUE;
}

// Destroy VxD and release resource
BOOL ChannelVxD_Device::OnSysDynamicDeviceExit()
{
        if (HwInt) delete HwInt;
        return TRUE;
}

//Read Port function
WORD ReadPort(int Adress)
{
        WORD Port;
        Port=(Adress);
        _asm
        {
         mov DX,Puerto
        in  AX,DX
        mov Puerto,AX
```

```
        }
        return Port;
}

//Driver functions
DWORD ChannelVxD_Device::OnW32DeviceIoControl(PIOCTLPARAMS pDat)
{
        DWORD status;
        switch(p->dioc_IOCtlCode)
        {
        case DIOC_OPEN:

        case DIOC_CLOSEHANDLE:
                status=0;
                break;

        // Command define by user. Register the Asynchronic Procedure Call(APC)
        case COMMAND1:
                Dir_APC=*(PVOID*)p->dioc_InBuf;
                Handler_Thread=Get_Cur_Thread_Handle();
                if (HwInt) delete HwInt;
                HwInt=new ChannelVxD_HwInt();

               //Hooks a virtualized IRQ to a set of handlers.
               //Returns 0 if the IRQ is successfully virtualized and unmask interrupt

                if ((HwInt==NULL)||!HwInt->hook()){status=0xf;}
                else {HwInt->physicalUnmask();status=0;}
                break;

        // User can redefine IRQ number and data counter address
        case COMMAND2:
                NUM_IRQ=*(int*)p->dioc_InBuf;
                DatCounterAdress=*(int*)p->dioc_OutBuf;
                break;
   status=0xffffffff;
        }
        return status;
}

//Constructor Hardware Interrupt class
ChannelVxD_HwInt::ChannelVxD_HwInt():VHardwareInt(NUM_IRQ,0,0,0)
{
}
```

```
//Destroy Hardware Interrupt class
ChannelVxD_HwInt::~ChannelVxD_HwInt()
{
        physicalMask;
}

//This function is called when an interrupt has been requested
VOID ChannelVxD_HwInt::OnHardwareInt(VMHANDLE)
{
        WORD DATA;
        DATA=ReadPort(DatCounterAdress);
        _VWIN32_QueueUserApc(Dir_APC,DATA,Handler_Thread);

/*Instructs VPICD to end interrupt processing. VPICD actually sends physical End Of Interrupt
at an earlier stage, and then masks the interrupt. This call causes VPICD to unmask the interrupt.
*/
        sendPhysicalEOI();
        ClearCarry();
}
```

SUMMARY

The information presented describes our experience to make and use of VxD which is the way to develop new PC hardware and software related.

The VTOOLDS library was selected because is easy to use and not require full knowledge in the VxD architecture.

REFERENCE

1. Microsoft Developer Network. 1998.

2. VTOOLSD Help. 1996.