PortTalk - A Windows NT/2000/XP I/O Port Device Driver

A problem that plagues Windows NT/2000 and Windows XP, is it's strict control over I/O ports. Unlike Windows 9x & ME, Windows NT/2000/XP will cause an exception (Privileged Instruction) if an attempt is made to access an IO port that a usermode program is not privileged to talk too. Actually it's not Windows NT that does this, but any 386 or higher processor running in protected mode.

Accessing I/O Ports in protected mode is governed by two events, The I/O privilege level (IOPL) in the EFLAGS register and the I/O permission bit map of a Task State Segment (TSS).

Under Windows NT, there are only two I/O privilege levels used, level 0 & level 3. Usermode programs will run in privilege level 3, while device drivers and the kernel will run in privilege level 0, commonly referred to as ring 0. This allows the trusted operating system and drivers running in kernel mode to access the ports, while preventing less trusted usermode processes from touching the I/O ports and causing conflicts. All usermode programs should talk to a device driver which arbitrates access.

The I/O permission bitmap can be used to allow programs not privileged enough (I.e. usermode programs) the ability to access certain I/O ports. When an I/O instruction is executed, the processor will first check if the task is privileged enough to access the ports. Should this be the case, the I/O instruction will be executed. However if the task is not allowed to do I/O, the processor will then check the I/O permission bitmap.

The I/O permission bitmap, as the name suggests uses a single bit to represent each I/O address. If the bit corresponding to a port is set, then the instruction will generate an exception however if the bit is clear then the I/O operation will proceed. This gives a means to allow certain processes to access certain ports. There is one I/O permission bitmap per task.

Accessing I/O Ports under NT/2000/XP

There are two solutions to solving the problem of I/O access under Windows NT/2000/XP. The first solution is to write a device driver which runs in ring 0 (I/O privilege level 0) to access your I/O ports on your behalf. Data can be passed to and from your usermode program to the device driver via IOCTL calls. The driver can then execute your I/O instructions. The problem with this, is that it assumes you have the source code to make such a change.

Another possible alternative is to modify the I/O permission bitmap to allow a particular task, access to certain I/O ports. This grants your usermode program running in ring 3 to do unrestricted I/O operations on selected ports, as per the I/O permission bitmap. This method is not really recommended, but provides a means of allowing existing applications to run under windows NT/2000/XP. Writing a device driver to support your hardware is the preferred method. The device driver should check for any contentions before accessing the port.

However, using a driver such as PortTalk can become quite inefficient. Each time an IOCTL call is made to read or write a byte or word to a port, the processor must switch from ring 3 to ring 0 perform the operation, then switch back. If your intentions were to write, for example a microcontroller programmer which is programmed serially using a parallel port pin, it would make better sense to send a pointer to a buffer of x many bytes. The device driver would then serialise the data and generate the handshake necessary in the programming of a PIC device.

Such an example is the USBLPTPD11 driver at *http://www.beyondlogic.org/usb/usblptpd11.htm*. This driver accepts a buffer of bytes via the IOCTL_WRITE_I2C IoDeviceCall and then big bangs this out in I2C format on a parallel port pin. The source code for this driver is available and is well worth a look.

The porttalk device driver comes complete with source code. It provides the facility to modify the IO permission bitmap and/or write and read to I/O ports via IOCTL calls.

Compatibility - Using existing applications under Windows NT/2000/XP

PortTalk can be used in conjunction with allowio to make existing programs that access the I/O ports work under Windows NT/2000/XP. As you already know, any 32bit program will cause a Privileged Instruction Exception. As most programming languages these days do not provide functions to read and write to ports, many hacks have been produced for Windows 9x and ME such as .DLL libraries. However should you need to run such a program under Windows NT, an exception will occur. This is where allowio and PortTalk will help.

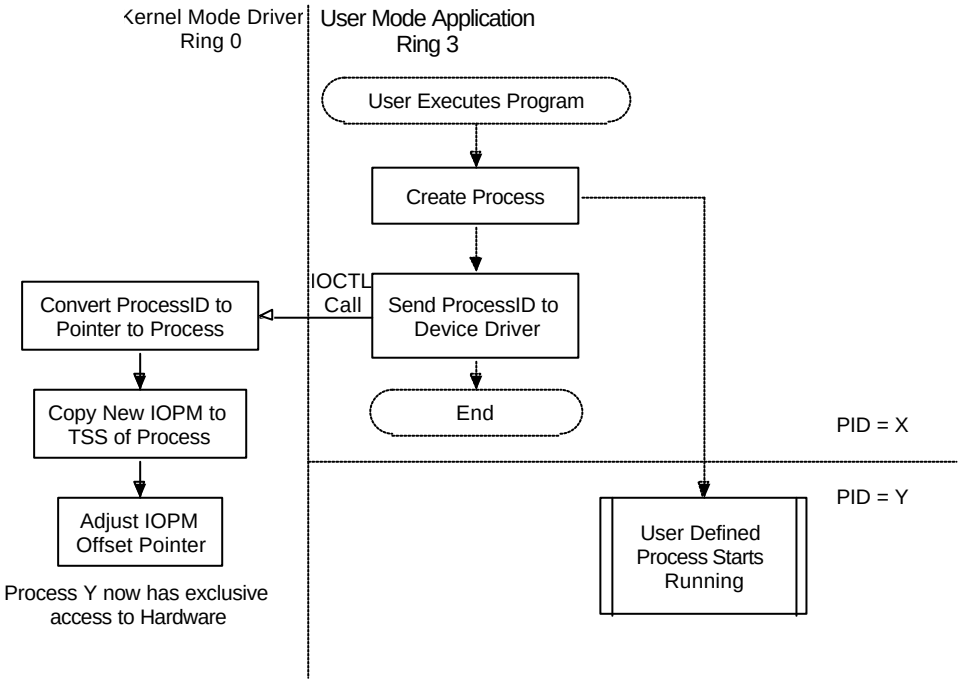
16 Bit Windows and DOS programs will run on virtual machines. In many cases existing applications should be transparent on Windows NT/2000/XP. However others just refuse to run. The virtual machines has support for communication ports, video, mouse, and keyboard. Therefore any program using these common I/O ports should run, however there is often a problem with timing. Other MS-DOS programs accessing specific hardware requires VDDs (Virtual Device Drivers) written to enable them to be used with Windows NT.

The Virtual Machine will intercept I/O operations and send them to an I/O handler for processing. The way the Virtual Machine does this, is by giving insufficient rights to I/O operations and creating an exception handler to dig back into the stack, find the last instruction and decode it. By giving the VDM full rights to I/O ports, it has no means of intercepting I/O operations, thus creating less problems with timing or the need to provide VDDs for obscurer hardware.

In order to change a processes IOPM, we must first have the process ID for the process we want to grant access too. This is accomplished by creating the process ourselves, so we can pass the ProcessID to our device driver. A small application is used which accepts the program name as an argument. This application then creates the process (i.e. executes the program) which starts and continues as another process in the system.

Note : We can also register a callback with the operating system which notifies our driver of any processes started and what their ID is. We can then keep a directory of processes that we want to have access to certain ports. When this process is executed, the callback informs the driver it has started and what it's process ID is. We could then automatically change the IOPM of this process. See the Process Monitor driver at Process.zip

When a Windows 32 bit program is started using CreateProcess(), it will return the ProcessID for the 32 Bit Program. This is passed to the Device Driver using an IOCTL Call.



DOS programs do not have their own ProcessID's. They run under the Windows NT Virtual DOS Machine (NTVDM.EXE) which is a protected environment subsystem that emulates MS-DOS. When a DOS program is called using this program, it will get the ProcessID for NTVDM.EXE and as a result changes NTVDM's IOPM.

However if NTVDM is already resident (if another DOS Program is running) it will return a process ID of zero. This doesn't cause a problem if the NT Virtual DOS Machine's IOPM is already set to allow any IO operation, however if the first DOS program was called from the command line without using "AllowIo", the NTVDM will not have the modified IOPM.

Windows 3.1 programs will run using WOW (Windows on Win32). This is another protected subsystem that runs within the NTVDM process. Running a Win3.1 program will return the ProcessID of NTVDM in accordance with the problem set out above.

When the device driver has the ProcessID, it finds the pointer to process for our newly created program and sets the IOPM to allow all I/O instructions. Once the ProcessID has been given to our PortTalk device driver, the allowio programs finishes.

Running a DOS/Win 3.1 program normally under NTVDM.EXE should not create any major problems. NTVDM will normally intercept most IO calls and check these resources against the registry to make sure they are not in use. Should they be in use, a message box will pop up giving the user the option to terminate the program or ignore the error. If the user chooses to ignore the error, access will NOT be granted to the offending I/O Port.

However using PortTalk to remove all I/O Protection will grant the application full rights to any port. As a result if it wants to talk to your mouse on COM1, it will. Result - Your mouse freezes. Using this program should be done at the discretion of the informed and educated user. If not, system instability will result. One solution to this problem is to be selective to what ports you allow full access too.

Calling your application using

C:\>allowio Test.exe /a

will grant test.exe exclusive access to all ports. However if you use,

C:\>allowio Test.exe 0x378

this will grant test.exe access only to 0x378 to 0x37F. As one byte represents 8 port addresses and that most devices will use a bank of 8 or 16 addresses, you need not specify every port address, only one port in the 8 byte boundary. Thus 0x378 will grant test.exe access to LPT1, including the data, status and control registers.

Manipulating the IOPM (I/O Permission Bitmap)

Changing the IOPM within your Kernel Mode Drivers requires the knowledge of a couple of undocumented calls. These are Ke386IoSetAccessProcess, Ke386SetIoAccessMap and PsLookupProcessByProcessId.

PsLookupProcessByProcessId(IN ULONG ulProcId,OUT struct _EPROCESS ** pEProcess);

The IOPM routines use a Pointer to Process and not the ProcessID. Therefore our first task is to convert the ProcessID to a Pointer to Process. There are documented calls such as PsGetCurrentProcess(), however we don't want the current process but rather the pointer to process of the process we wish to grant access to. This information is passed to the driver in the form of a processID. We must then use the undocumented call PsLookupProcessByProcessId to convert our ProcessID to a Pointer to Process.

Once we have a Pointer to Process we can start manipulating the I/O permission bitmap using the following undocumented calls

```
void Ke386SetIoAccessMap(int, IOPM *);
void Ke386QueryIoAccessMap(int, IOPM *);
void Ke386IoSetAccessProcess(PEPROCESS, int);
```

Ke386SetIoAccessMap will copy the IOPM specified to the TSS. Ke386QueryIoAccessMap will read it from the TSS. The IOPM is a 8192 byte array specifying which ports are allowed access and which ones aren't. Each address is represented by one bit, thus the 8192 bytes will specify access up to 64K. Any zero bit will allow access, while a one will deny access.

After the IOPM has been copied to the TSS, the IOPM offset pointer must be adjusted to point to our IOPM. This is done using the Ke386IoSetAccessProcess. The int parameter must be set to 1 to enable it to be set. Calling the function with zero will remove the pointer.

Talking to the Device Driver - User Mode APIs

PortTalk also has IOCTLs to allow reading and writing to I/O Ports. In this case, your usermode program would open the PortTalk device driver and pass data to the driver through IOCTL calls. The driver then talks to the I/O port(s) in ring 0.

Starting and installing the driver

In most cases, the PORTTALK.SYS driver isn't required to be explicitly installed. When running the usermode executable such as allowio.exe, it will check for the device driver and if it cannot be opened, it will install and start the driver for you. However for this to happen correctly, the PORTTALK.SYS driver must be in the same directory than the usermode executable ran and the user must have administrator privileges. Once the driver has been installed for the first time, any user with normal user privileges can access the device driver normally. This is ideal in classroom/corporate environments where security is paramount.

The driver can also be installed manually using the registry file included. Copy the PORTTALK.SYS to your /system32/drivers directory and click on the PORTTALK.REG file to load the required registry keys. Then reboot the computer and on boot-up the PORTTALK.SYS driver will load and start automatically. This is recommended for classroom/corporate use where the driver can be stored away securely in the system directory.

Check and Free Driver Versions

Two versions of the driver exist. The standard distribution is a free compiled version which has debugging statements removed and thus execute faster. However when writing your own code, or debugging problems such as buffer overuns, a checked version of the driver is provided which displays debugging. These debug messages can be read with any good debug viewer. One such recommended viewer is the System Internals DebugView which can be downloaded from their website (http://www.sysinternals.com) for free.

The checked build of the driver is provided in the checked folder of the distribution. Simply replace the PORTTALK.SYS with this driver and reload to display debug information.

Recompiling the code

The code for the device driver has been compiled using Microsoft Visual C and the Windows 2000 DDK. The source code is provided, but during the normal development cycle it is not required to be recompiled. It has also been built for testing purposes on the Windows XP DDK which includes build tools and is no longer dependent on Microsoft Visual C being installed. (Excellent for Borland Folks)

Revision History

12th January 2002 – Version 2.0, tested on Windows 2000 SP2 and Windows XP RTM.
Self installs driver for ease of use.
Improved type checking.
Distributed with IoExample code showing use of inportb/outportb() inp/outp() macros and IOCTL calls.

6th September 2001 – Version 1.02
Fixed .reg file after previous fix broke Windows 2000 Support. Now supports Windows NT/2000/XP.

26th June 2001 – Version 1.01
Fixed .reg file to support both Windows 2000 and Windows NT4.

13th March 1999 – Version 1.0 first public release for Windows NT4.

PortTalk IOCTL Call Summary

IOCTL_IOPM_RESTRICT_ALL_ACCESS

Operation

Fills the IO Permission Bitmap with 1's, denying I/O to all ports.

Input/Output

None

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS	Operation Succeeded

IOCTL_IOPM_ALLOW_EXCUSIVE_ACCESS

Operation

Fills the IO Permission Bitmap with 0's, allowing access to all ports.

Input/Output

None

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS	Operation Succeeded.

IOCTL_SET_IOPM

Operation

Sets a one byte chunk of the IOPM with a specified *offset* to *Value.*

The IOPM (I/O Permission Bitmap) contains one bit for every port value up to 64KB of I/O space. This equates to an array of 8192 bytes. For example, If it is desired to allow access to 0x378 then you will need to send a IOCTL_SET_IOPM DeviceIOControl call with an offset of 0x6F (0x378 / 8) and a value of 0xFE (bit zero cleared). The AllowIO program accompanying PortTalk simply allows access to blocks of 8 bytes sending a value of 0x00 for each call.

Input

The **AssociatedIrp.SystemBuffer** member points to three bytes. The first two bytes specify the offset within the IOPM. The third byte specifies the byte value to write to the IOPM.

The **Parameters.DeviceIoControl.InputBufferLength** member specifies the input parameter size. 3 Bytes.

Output

None.

Status I/O Block

The **Information** member is set to the number of bytes read.

The **Status** member is set to one of the following values:

STATUS_SUCCESS	The requested number of bytes were read from the I2C device.
STATUS_BUFFER_TOO_SMALL	The input or output buffers are too small.
STATUS_ARRAY_BOUNDS_EXCEEDED	The offset has exceeded the bounds of the IOPM (e.g >= 0x2000)

IOCTL_ENABLE_IOPM_ON_PROCESSID

Operation

Enables the current IOPM on the process indicated by ProcessID

Input

The **AssociatedIrp.SystemBuffer** member points to a ULONG holding the ProcessID.

The **Parameters.DeviceIoControl.InputBufferLength** member specifies the input parameter size. (4 bytes)

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS	The requested number of bytes were send to the I2C device.
STATUS_BUFFER_TOO_SMALL	The input buffer is too small.

IOCTL_READ_PORT_UCHAR

Operation

Reads a byte wide I/O Port indicated by *PortAddress.*

Input

The **AssociatedIrp.SystemBuffer** member points to a USHORT containing the PortAddress.

The **Parameters.DeviceIoControl.InputBufferLength** member specifies the size. 2 Bytes.

Output

The **AssociatedIrp.SystemBuffer** member points to the buffer of bytes read.

Status I/O Block

The **Information** member is set to 1. (1 byte returned)

The **Status** member is set to one of the following values:

STATUS_SUCCESS	Operation Succeeded.
STATUS_BUFFER_TOO_SMALL	Input or Output buffer length too small.

IOCTL_WRITE_PORT_UCHAR

Operation

Writes the byte, *value* to an I/O Port indicated by *PortAddress.*

Input

The **AssociatedIrp.SystemBuffer** member points to a USHORT containing the PortAddress.

The **Parameters.DeviceIoControl.InputBufferLength** member specifies the size. 2 Bytes.

Output

None.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS	Operation Succeeded.
STATUS_BUFFER_TOO_SMALL	Input buffer length too small.

Example I/O Program using PortTalk

The Porttalk driver contains two IOCTL calls to read from and write to I/O Ports. A c source file, pt_iotcl.c can be used to provide easy support based on the popular inportb/outportb and inp/outp calls supported in earlier programming environments. By simply including pt_ioctl.c and calling OpenPortTalk when you program starts and ClosePortTalk when your program finishes you can have the functionality of the inportb/outportb and inp/outp calls.

```
#include <stdio.h>
#include <windows.h>
#include <pt_ioctl.c>

void __cdecl main(void)
{
    unsigned char value;
    OpenPortTalk();
    outportb(0x378, 0xFF);
    value = inportb(0x378);
    printf("Value returned = 0x%02X \n",value);
    outp(0x378, 0xAA);
    value = inp(0x378);
    printf("Value returned = 0x%02X \n",value);
    ClosePortTalk();
}
```

The sample program above is included in the IoExample directory along with the pt_ioctl.c. The pt_ioctl can be used as an example of how to load and open the driver and then make IOCTL_WRITE_PORT_UCHAR and IOCTL_READ_PORT_UCHAR calls.