A Serial Port Spy for NT

Alex V. Bessonov

Under NT, serial ports are exclusive devices, and only one application can have a port open at a time. That's normally what you'd want, but there are many cases when it would be handy to be able to spy on the serial communications of another application. For example, you might be trying to debug your own high-level driver that uses the serial port to communicate with a particular device. Or, you might be trying to uncover the serial interface used by some commercial hardware (UPS, GPS, etc.) so that you can write your own custom software to manipulate the device instead.

Although NT normally prevents two applications from both accessing the same serial port, it's still possible to write a serial-port monitor that spies on the communications of another application. This article will provide the necessary device-driver support, as well as a simple application for displaying the resulting serial-port data.

Intercepting I/O Calls

The NT device driver serial.sys handles COM ports. During initialization, it enumerates all available COM ports and creates device objects for each one it can find. Device objects are created under the \Device subtree of the NT Object Manager's namespace. The driver gives them names like Serialx, where x is the number of the port (1, 2, 3, etc.). To let user-mode code access the ports, serial.sys also creates symbolic links under the \?? subtree called COM1, COM2, etc. The driver does not let user-mode code open these devices multiple times.

There are two basic approaches for spying on the serial-port data of a particular application. The first is to create a DLL with your own wrapper functions for CreateFile(), WriteFile(), SetCommState(), etc. and somehow force the application to call your wrapper functions instead. You could then inject the DLL into the application's address space (as described in Jeffrey Richter's *Advanced Windows*) and modify its import table to point to your wrapper functions. The wrapper functions would then call the original Win32 functions, but also communicate via some kind of interprocess communication with a monitor application, which would display the spied-on output. The second approach is to write a filter device driver that attaches itself to a serial device, processes all requests it deals with,

and transfers them to a user-mode application that provides the user interface.

I can't advise you which way is best because both have good and bad sides. The first method works under both Win95 and NT, runs in user-mode only, and doesn't require administrative privileges, but it is a little harder to implement. The second one is more "honest" because it uses only documented features, is easier to implement, and is much more interesting. This article uses the filter-driver approach.

Driver Source Code

The driver source code spans several files, and this month's code archive contains complete source code for both the device driver and the user-mode monitor application.

The driver's initialization and dispatch code resides in sermon.h (Listing 1) and sermon.cpp (Listing 2). drvclass.h and drvclass.cpp (available in the code archive) contain helper code to implement memory management and doubly linked lists. devext.h (Listing 3) and devext.cpp (Listing 4) contain the main classes that implement the device driver. sermon.rc (in this month's code archive)contains version information for the driver. sermonex.h (in the code archive) contains structures used by both the driver and application (using conditional compilation).

Initialization

Driver initialization starts in DriverEntry() in sermon.cpp (Listing 2). It first uses CreateDevices() to try to create its own device object named \ Device\ SerMon, as well as the symbolic link \ ??\ SerMon. The symbolic link is required to let user-mode code access the device object. If something fails, DriverEntry() returns a status other than STATUS_SUCCESS to the I/O manager. As soon as DriverEntry() creates the main device object, it fills the I/O manager's supplied array of major function handlers situated in the DRIVER_OBJECT structure. The driver has to handle all types of requests because it is going to receive all the requests that can be directed at a serial-port device driver.

The driver implements two types of devices (one filters requests to the serial driver and the other provides an interface to a user-mode monitoring application), so I used C++ inheritance to capture the commonality between the two implementations. devext.h (Listing 3) and devext.cpp (Listing 4) define a parent class (CDevice) and two derived classes (CAttachedDevice and CSERMONDevice). This gave me an opportunity to implement basic IRP processing functions as macros.

sermon.h (Listing 1) contains two macros used to declare and define IRP functions that DriverEntry() stores in the driver object's major function table:

```
#define DECLARE_FUNCTION(x) extern "C" NTSTATUS \
SERMON##x (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
#define IMPLEMENT_FUNCTION(x) NTSTATUS \
SERMON##x (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp) \
   { \
   return ((CDevice *)
(DeviceObject->DeviceExtension))->x(Irp); \
   }
```

The functions that these macros generate expect that the DeviceExtension field of the device object contains a pointer to a CDevice-derived object, and they use that pointer to invoke the appropriate virtual function. This is similar to techniques for mapping window messages onto C++ class virtual functions. For each request type, CDevice supplies a default implementation that simply completes the request and returns a non-error status code, STATUS_SUCCESS.

Finally, DriverEntry() constructs an empty linked list that will eventually contain pointers to created attached devices. DriverEntry() then returns STATUS_SUCCESS.

Communicating with the Driver

The SerMon device object exists to allow user-mode code to communicate with the filter driver. It implements four special I/O control requests, which user-mode code can access via Win32's DeviceIoControl(). An application opens the SerMon device by calling CreateFile():

```
hDevice=CreateFile(_T("\\\\.\\SerMon"),
   GENERIC_READ | GENERIC_WRITE | GENERIC_EXECUTE,
   FILE_SHARE_WRITE | FILE_SHARE_READ,
   NULL,
   OPEN_EXISTING,
   FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
   NULL);
```

CreateFile() returns a handle the application can then pass to DeviceIoControl(), along with a command code, pointers to input and output buffers, and so on:

```
BOOL res=DeviceIoControl(hDevice,
    IOCTL_SERMON_STARTMONITOR,
    (PVOID) (LPCTSTR) s,
    (s.GetLength()+1)*sizeof(WCHAR),
    &handle,sizeof(MHANDLE),
    &dw,
    NULL);
```

When the user-mode code calls DeviceIoControl(), control eventually transfers to CSERMONDevice::IoControl() in devext.cpp (Listing 3). This function examines the command code to decide how to handle the request. The four command codes the device handles are defined in sermonex.h.

IOCTL_SERMON_STARTMONITOR. To begin monitoring a device, pass this command code to DeviceIoControl(). The input parameter is a Unicode string identifying the device object the driver should attach to. It can be either in the form of \ Device\ Serialx or \ ??\ COMx, where x is the port number. The device must not be already opened before you execute this attach operation, since it will fail with a STATUS_ACCESS_VIOLATION status code.

The output from this command is an MHANDLE, which is internally a pointer to a newly created C++ object of type CAttachedDevice. The calling application must supply this handle in future calls to DeviceIoControl() to identify the monitoring session.

IOCTL_SERMON_STOPMONITOR. Pass this code to DeviceIoControl() to end a monitoring session. You must supply the MHANDLE for the session that was returned with the IOCTL_SERMON_STARTMONITOR command.

IOCTL_SERMON_GETINFOSIZE. SerMon produces a stream of variable-length event records that the calling application can interpret and display. Pass the IOCTL_SERMON_GETINFOSIZE command (along with the MHANDLE for the session) to obtain the size of the next event in the queue. It returns a DWORD that indicates the size in bytes of the next event.

IOCTL_SERMON_GETINFO. Once you know the size of the next event in the queue, you can use the IOCTL_SERMON_GETINFO command to copy the event data to a user-mode buffer. Pass the MHANDLE for the session along with a buffer that is big enough to handle the size of the next event from the queue.

Attaching to the Serial Device

When the application passes an IOCTL_SERMON_STARTMONITOR to DeviceIoControl(), it eventually results in a call to CSERMONDevice::IoControl() in devext.cpp (Listing 4). This function passes the name of the destination device to CSERMONDevice::TryConnectToSerialDevice(), which in turn calls the global function Attach().

Attach() first tries to find and open the destination device with a call to IoGetDeviceObjectPointer(), which returns a pointer to a device object and its associated file object. I don't need the file object, so I dereference it at the bottom of the function by calling ObDereferenceObject(). Then Attach() creates an unnamed device object and stores in its device extension field a pointer to a newly created CAttachedDevice object. Attach() then locks the CAttachedDevice object, since it is not yet ready to receive driver requests.

Next, Attach() calls IoAttachDeviceByPointer() to attach the created device to the destination device. The I/O manager fills all required fields in the device's DEVICE_OBJECT structure, making it capable of handling and forwarding other devices' requests. For simplicity, the device object is marked to do buffered I/O. Luckily, the serial device driver itself and most other similar drivers use this type of I/O.

If all the above operations were successful, Attach() unlocks the CAttachedDevice object making it ready to receive requests and returns a pointer to the CAttachedDevice object it created. This pointer will be the MHANDLE (typedef'ed as DWORD) that is returned to the calling application. To be able to distinguish between correct handles and junk, the driver does a little error checking by trying to read memory at a given address and comparing its contents with signature. This procedure doesn't guarantee that the given handle really is a pointer to a class, but at least it gives a good chance of it.

CAttachedDevice

The CAttachedDevice class is declared in devext.h (Listing 3). Each CAttachedDevice object contains two lists: an event list and a request list.

These lists are implemented as FIFO doubly linked lists, using the helper template class CDBLinkedList. Within CAttachedDevice, the Signature field is used to double-check that the calling application passed a valid MHANDLE (remember that the MHANDLE is just a pointer to a CAttachedDevice). The eres field is an ERESOURCE variable used to lock and unlock the entire CAttachedDevice object.

The class also contains the variable event of type KEVENT and initializes it in its constructor with a call to KeInitializeEvent(). This event is of great importance. While developing the driver, I found that if I detached the device from the serial device and unloaded the driver while some application had the port opened, then the next request sent to it caused NT to produce a BSOD (Blue Screen Of Death) in an error condition. That's why you can't detach the device from the destination device until all user-mode (and preferably kernel-mode, too) references to it are closed. That's the purpose of the Num and bFirstTime fields.

CAttachedDevice::Num contains the number of times the device was opened; serial devices can't be opened more than once, but you could use this code to spy on similar device drivers that do permit multiple opens. CAttachedDevice::bFirstTime is set to TRUE in the constructor preventing the close handler (CloseCompletion() in devext.cpp) from decrementing Num the first time, since the code will miss the initial open IRP. CloseCompletion() sets the event to signaled when Num is equal to zero and non-signaled when it's more than zero. This enables the call to KeWaitForSingleObject() in the destructor to wait for the port to be closed.

The CAttachedDevice constructor initializes all these fields and inserts itself into the internal driver's linked list of pointers to CAttachedDevice objects (created in DriverEntry()). This lets the driver unload cleanly.

Immediately after the attach operation, the driver is ready for monitoring. Each request forwarded by the I/O manager to the target serial driver goes instead to the attached driver, then is redirected to the corresponding virtual function in CAttachedDevice.

The driver monitors only open, close, write, read, and I/O control requests. Moreover, they ignore requests that the serial driver returns an unsuccessful status code for. In all cases, CAttachedDevice::Standard() is called to handle the incoming IRP, passing it the IRP being processed and a pointer to the completion routine for the current I/O stack location.

CAttached::Standard() copies the current stack location into the next and sets the completion routine for the IRP, using the macro

IoSetCompletionRoutine(). After bit-wise copying, the next stack location will contain the same value for its CompletionRoutine field as the current. If there is more than one driver layered above the destination driver, then it will cause multiple calls of the same function. The results are unpredictable. That's why it's much safer to always specify a completion routine. In cases where the driver doesn't need one (i.e., for flush and cleanup requests), it registers DefaultCompletion() as the completion routine; DefaultCompletion() does nothing besides returning STATUS_SUCCESS to the I/O manager.

The main processing can be found in ReadCompletion(), WriteCompletion(), OpenCompletion(), CloseCompletion(), and IOCompletion(). If the request was successful, they construct a new object of type IOReq, fill all necessary data within it, and then call CAttachedDevice::New(). CAttachedDevice::New() first locks the CAttachedDevice object (in order to eliminate internal lists inconsistencies), and then appends the last created IOReq object to the end of the events list.

The events list stores up information about serial port I/O requests so that the calling application can retrieve them later. However, requests from the calling application (IOCTL_SERMON_GETINFOSIZE and IOCTL_SERMON_GETINFO) may also be stored in a linked list — the request list. If the event list is empty when the user-mode application sends an IOCTL_SERMON_GETINFOSIZE or IOCTL_SERMON_GETINFO, then the request is marked pending and put at the end of the request list. When CAttachedDevice::New() appends a new IOReq structure to the event list, it also checks the request list. If there is a request already pending, then CAttachedDevice::New() goes ahead and processes it by calling either ProcessSize() or ProcessNext(). These functions retrieve the top IOReq from the event list, complete the request, and remove it from the request list. (ProcessNext() also removes the topmost event from the event list.)

Stopping Monitoring

When the application sends a IOCTL_SERMON_STOPMONITOR request, the driver simply deletes the CAttachedDevice object. All the work is done by its destructor.

CAttachedDevice::~CAttachedDevice() first changes the signature field so that if the application mistakenly uses the same MHANDLE again, the driver can detect it as an invalid pointer. It then waits for the previously explained event until the port is closed. Then it detaches the device and deletes it. After that, all pending IRPs are completed with

STATUS_CANCELLED. Finally, execution returns to the user-mode application.

Controlling Application

This month's code archive contains a simple controlling application (see Figure 1) that uses the SerMon driver to monitor ports. It uses MFC and implements an MDI window interface, allowing the user to monitor several ports concurrently.

When started, the program first tries to open the SerMon device driver by calling CreateFile() with a filename of \\.\SerMon. If the driver can not be opened (probably because it was not installed), the application tries to use the service control manager to install the driver. Installing device drivers using the SCM is a very effective and easy method.

When you select the start-monitor menu option, the application asks you to select a port. The application then creates a new document/view pair. The view contains two windows (containing bytes read and bytes written). Open, close, and I/O control requests are displayed in both windows.

The application then sends the IOCTL_SERMON_STARTMONITOR request to the driver and gets a handle to the monitor session. Then it creates a parallel thread, which enters the cycle of issuing two requests (IOCTL_SERMON_GETINFOSIZE and IOCTL_SERMON_GETINFO), using the OVERLAPPED structure to cause DeviceIoControl() to perform asynchronous I/O. When the thread receives data, it posts a message to the view, and it displays the data to the user.

When you select the stop-monitor menu option, or simply close the view, the application sends an IOCTL_SERMON_STOPMONITOR to the driver after displaying a warning message to the user. The warning tells the user to close the application that uses the port for reasons described earlier.

Conclusion

Creating a monitoring filter driver is an elegant solution to the problem of spying on serial ports under NT. Moreover, the technique can be used on almost any kernel-mode device driver. For example, you could also use this idea to monitor the IRP flow of a driver you are debugging. The serial port spy is just one of many applications made possible by NT's extensible device-driver framework.

Under NT, serial ports are exclusive devices, and only one application can have a port open at a time. That's normally what you'd want, but there are

many cases when it would be handy to be able to spy on the serial communications of another application. For example, you might be trying to debug your own high-level driver that uses the serial port to communicate with a particular device. Or, you might be trying to uncover the serial interface used by some commercial hardware (UPS, GPS, etc.) so that you can write your own custom software to manipulate the device instead.

Although NT normally prevents two applications from both accessing the same serial port, it's still possible to write a serial-port monitor that spies on the communications of another application. This article will provide the necessary device-driver support, as well as a simple application for displaying the resulting serial-port data.

Intercepting I/O Calls

The NT device driver serial.sys handles COM ports. During initialization, it enumerates all available COM ports and creates device objects for each one it can find. Device objects are created under the \Device subtree of the NT Object Manager's namespace. The driver gives them names like Serialx, where x is the number of the port (1, 2, 3, etc.). To let user-mode code access the ports, serial.sys also creates symbolic links under the \?? subtree called COM1, COM2, etc. The driver does not let user-mode code open these devices multiple times.

There are two basic approaches for spying on the serial-port data of a particular application. The first is to create a DLL with your own wrapper functions for CreateFile(), WriteFile(), SetCommState(), etc. and somehow force the application to call your wrapper functions instead. You could then inject the DLL into the application's address space (as described in Jeffrey Richter's *Advanced Windows*) and modify its import table to point to your wrapper functions. The wrapper functions would then call the original Win32 functions, but also communicate via some kind of interprocess communication with a monitor application, which would display the spied-on output. The second approach is to write a filter device driver that attaches itself to a serial device, processes all requests it deals with, and transfers them to a user-mode application that provides the user interface.

I can't advise you which way is best because both have good and bad sides. The first method works under both Win95 and NT, runs in user-mode only, and doesn't require administrative privileges, but it is a little harder to implement. The second one is more "honest" because it uses only documented features, is easier to implement, and is much more interesting. This article uses the filter-driver approach.

Driver Source Code

The driver source code spans several files, and this month's code archive contains complete source code for both the device driver and the user-mode monitor application.

The driver's initialization and dispatch code resides in sermon.h (Listing 1) and sermon.cpp (Listing 2). drvclass.h and drvclass.cpp (available in the code archive) contain helper code to implement memory management and doubly linked lists. devext.h (Listing 3) and devext.cpp (Listing 4) contain the main classes that implement the device driver. sermon.rc (in this month's code archive)contains version information for the driver. sermonex.h (in the code archive) contains structures used by both the driver and application (using conditional compilation).

Initialization

Driver initialization starts in DriverEntry() in sermon.cpp (Listing 2). It first uses CreateDevices() to try to create its own device object named \ Device\ SerMon, as well as the symbolic link \ ??\ SerMon. The symbolic link is required to let user-mode code access the device object. If something fails, DriverEntry() returns a status other than STATUS_SUCCESS to the I/O manager. As soon as DriverEntry() creates the main device object, it fills the I/O manager's supplied array of major function handlers situated in the DRIVER_OBJECT structure. The driver has to handle all types of requests because it is going to receive all the requests that can be directed at a serial-port device driver.

The driver implements two types of devices (one filters requests to the serial driver and the other provides an interface to a user-mode monitoring application), so I used C++ inheritance to capture the commonality between the two implementations. devext.h (Listing 3) and devext.cpp (Listing 4) define a parent class (CDevice) and two derived classes (CAttachedDevice and CSERMONDevice). This gave me an opportunity to implement basic IRP processing functions as macros.

sermon.h (Listing 1) contains two macros used to declare and define IRP functions that DriverEntry() stores in the driver object's major function table:

```
#define DECLARE_FUNCTION(x) extern "C" NTSTATUS \
SERMON##x (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
#define IMPLEMENT_FUNCTION(x) NTSTATUS \
SERMON##x (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp) \
   { \
```

```
   return ((CDevice *)
(DeviceObject->DeviceExtension))->x(Irp); \
   }
```

The functions that these macros generate expect that the DeviceExtension field of the device object contains a pointer to a CDevice-derived object, and they use that pointer to invoke the appropriate virtual function. This is similar to techniques for mapping window messages onto C++ class virtual functions. For each request type, CDevice supplies a default implementation that simply completes the request and returns a non-error status code, STATUS_SUCCESS.

Finally, DriverEntry() constructs an empty linked list that will eventually contain pointers to created attached devices. DriverEntry() then returns STATUS_SUCCESS.

Communicating with the Driver

The SerMon device object exists to allow user-mode code to communicate with the filter driver. It implements four special I/O control requests, which user-mode code can access via Win32's DeviceIoControl(). An application opens the SerMon device by calling CreateFile():

```
hDevice=CreateFile(_T("\\\\.\\SerMon"),
   GENERIC_READ | GENERIC_WRITE | GENERIC_EXECUTE,
   FILE_SHARE_WRITE | FILE_SHARE_READ,
   NULL,
   OPEN_EXISTING,
   FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
   NULL);
```

CreateFile() returns a handle the application can then pass to DeviceIoControl(), along with a command code, pointers to input and output buffers, and so on:

```
BOOL res=DeviceIoControl(hDevice,
   IOCTL_SERMON_STARTMONITOR,
   (PVOID) (LPCTSTR) s,
```

```
    (s.GetLength()+1)*sizeof(WCHAR),
    &handle,sizeof(MHANDLE),
    &dw,
    NULL);
```

When the user-mode code calls DeviceIoControl(), control eventually transfers to CSERMONDevice::IoControl() in devext.cpp (Listing 3). This function examines the command code to decide how to handle the request. The four command codes the device handles are defined in sermonex.h.

IOCTL_SERMON_STARTMONITOR. To begin monitoring a device, pass this command code to DeviceIoControl(). The input parameter is a Unicode string identifying the device object the driver should attach to. It can be either in the form of \ Device\ Serialx or \ ??\ COMx, where x is the port number. The device must not be already opened before you execute this attach operation, since it will fail with a STATUS_ACCESS_VIOLATION status code.

The output from this command is an MHANDLE, which is internally a pointer to a newly created C++ object of type CAttachedDevice. The calling application must supply this handle in future calls to DeviceIoControl() to identify the monitoring session.

IOCTL_SERMON_STOPMONITOR. Pass this code to DeviceIoControl() to end a monitoring session. You must supply the MHANDLE for the session that was returned with the IOCTL_SERMON_STARTMONITOR command.

IOCTL_SERMON_GETINFOSIZE. SerMon produces a stream of variable-length event records that the calling application can interpret and display. Pass the IOCTL_SERMON_GETINFOSIZE command (along with the MHANDLE for the session) to obtain the size of the next event in the queue. It returns a DWORD that indicates the size in bytes of the next event.

IOCTL_SERMON_GETINFO. Once you know the size of the next event in the queue, you can use the IOCTL_SERMON_GETINFO command to copy the event data to a user-mode buffer. Pass the MHANDLE for the session along with a buffer that is big enough to handle the size of the next event from the queue.

Attaching to the Serial Device

When the application passes an IOCTL_SERMON_STARTMONITOR to DeviceIoControl(), it eventually results in a call to CSERMONDevice::IoControl() in devext.cpp (Listing 4). This function passes the name of the destination device to CSERMONDevice::TryConnectToSerialDevice(), which in turn calls the global function Attach().

Attach() first tries to find and open the destination device with a call to IoGetDeviceObjectPointer(), which returns a pointer to a device object and its associated file object. I don't need the file object, so I dereference it at the bottom of the function by calling ObDereferenceObject(). Then Attach() creates an unnamed device object and stores in its device extension field a pointer to a newly created CAttachedDevice object. Attach() then locks the CAttachedDevice object, since it is not yet ready to receive driver requests.

Next, Attach() calls IoAttachDeviceByPointer() to attach the created device to the destination device. The I/O manager fills all required fields in the device's DEVICE_OBJECT structure, making it capable of handling and forwarding other devices' requests. For simplicity, the device object is marked to do buffered I/O. Luckily, the serial device driver itself and most other similar drivers use this type of I/O.

If all the above operations were successful, Attach() unlocks the CAttachedDevice object making it ready to receive requests and returns a pointer to the CAttachedDevice object it created. This pointer will be the MHANDLE (typedef'ed as DWORD) that is returned to the calling application. To be able to distinguish between correct handles and junk, the driver does a little error checking by trying to read memory at a given address and comparing its contents with signature. This procedure doesn't guarantee that the given handle really is a pointer to a class, but at least it gives a good chance of it.

CAttachedDevice

The CAttachedDevice class is declared in devext.h (Listing 3). Each CAttachedDevice object contains two lists: an event list and a request list. These lists are implemented as FIFO doubly linked lists, using the helper template class CDBLinkedList. Within CAttachedDevice, the Signature field is used to double-check that the calling application passed a valid MHANDLE (remember that the MHANDLE is just a pointer to a CAttachedDevice). The eres field is an ERESOURCE variable used to lock and unlock the entire CAttachedDevice object.

The class also contains the variable event of type KEVENT and initializes it in its constructor with a call to KeInitializeEvent(). This event is of great importance. While developing the driver, I found that if I detached the device from the serial device and unloaded the driver while some application had the port opened, then the next request sent to it caused NT to produce a BSOD (Blue Screen Of Death) in an error condition. That's why you can't detach the device from the destination device until all user-mode (and preferably kernel-mode, too) references to it are closed. That's the purpose of the Num and bFirstTime fields.

CAttachedDevice::Num contains the number of times the device was opened; serial devices can't be opened more than once, but you could use this code to spy on similar device drivers that do permit multiple opens. CAttachedDevice::bFirstTime is set to TRUE in the constructor preventing the close handler (CloseCompletion() in devext.cpp) from decrementing Num the first time, since the code will miss the initial open IRP. CloseCompletion() sets the event to signaled when Num is equal to zero and non-signaled when it's more than zero. This enables the call to KeWaitForSingleObject() in the destructor to wait for the port to be closed.

The CAttachedDevice constructor initializes all these fields and inserts itself into the internal driver's linked list of pointers to CAttachedDevice objects (created in DriverEntry()). This lets the driver unload cleanly.

Immediately after the attach operation, the driver is ready for monitoring. Each request forwarded by the I/O manager to the target serial driver goes instead to the attached driver, then is redirected to the corresponding virtual function in CAttachedDevice.

The driver monitors only open, close, write, read, and I/O control requests. Moreover, they ignore requests that the serial driver returns an unsuccessful status code for. In all cases, CAttachedDevice::Standard() is called to handle the incoming IRP, passing it the IRP being processed and a pointer to the completion routine for the current I/O stack location.

CAttached::Standard() copies the current stack location into the next and sets the completion routine for the IRP, using the macro IoSetCompletionRoutine(). After bit-wise copying, the next stack location will contain the same value for its CompletionRoutine field as the current. If there is more than one driver layered above the destination driver, then it will cause multiple calls of the same function. The results are unpredictable. That's why it's much safer to always specify a completion routine. In cases where the driver doesn't need one (i.e., for flush and cleanup requests), it registers DefaultCompletion() as the completion

routine; DefaultCompletion() does nothing besides returning STATUS_SUCCESS to the I/O manager.

The main processing can be found in ReadCompletion(), WriteCompletion(), OpenCompletion(), CloseCompletion(), and IOCompletion(). If the request was successful, they construct a new object of type IOReq, fill all necessary data within it, and then call CAttachedDevice::New(). CAttachedDevice::New() first locks the CAttachedDevice object (in order to eliminate internal lists inconsistencies), and then appends the last created IOReq object to the end of the events list.

The events list stores up information about serial port I/O requests so that the calling application can retrieve them later. However, requests from the calling application (IOCTL_SERMON_GETINFOSIZE and IOCTL_SERMON_GETINFO) may also be stored in a linked list — the request list. If the event list is empty when the user-mode application sends an IOCTL_SERMON_GETINFOSIZE or IOCTL_SERMON_GETINFO, then the request is marked pending and put at the end of the request list. When CAttachedDevice::New() appends a new IOReq structure to the event list, it also checks the request list. If there is a request already pending, then CAttachedDevice::New() goes ahead and processes it by calling either ProcessSize() or ProcessNext(). These functions retrieve the top IOReq from the event list, complete the request, and remove it from the request list. (ProcessNext() also removes the topmost event from the event list.)

Stopping Monitoring

When the application sends a IOCTL_SERMON_STOPMONITOR request, the driver simply deletes the CAttachedDevice object. All the work is done by its destructor.

CAttachedDevice::~CAttachedDevice() first changes the signature field so that if the application mistakenly uses the same MHANDLE again, the driver can detect it as an invalid pointer. It then waits for the previously explained event until the port is closed. Then it detaches the device and deletes it. After that, all pending IRPs are completed with STATUS_CANCELLED. Finally, execution returns to the user-mode application.

Controlling Application

This month's code archive contains a simple controlling application (see Figure 1) that uses the SerMon driver to monitor ports. It uses MFC and

implements an MDI window interface, allowing the user to monitor several ports concurrently.

When started, the program first tries to open the SerMon device driver by calling CreateFile() with a filename of \\.\SerMon. If the driver can not be opened (probably because it was not installed), the application tries to use the service control manager to install the driver. Installing device drivers using the SCM is a very effective and easy method.

When you select the start-monitor menu option, the application asks you to select a port. The application then creates a new document/view pair. The view contains two windows (containing bytes read and bytes written). Open, close, and I/O control requests are displayed in both windows.

The application then sends the IOCTL_SERMON_STARTMONITOR request to the driver and gets a handle to the monitor session. Then it creates a parallel thread, which enters the cycle of issuing two requests (IOCTL_SERMON_GETINFOSIZE and IOCTL_SERMON_GETINFO), using the OVERLAPPED structure to cause DeviceIoControl() to perform asynchronous I/O. When the thread receives data, it posts a message to the view, and it displays the data to the user.

When you select the stop-monitor menu option, or simply close the view, the application sends an IOCTL_SERMON_STOPMONITOR to the driver after displaying a warning message to the user. The warning tells the user to close the application that uses the port for reasons described earlier.

Conclusion

Creating a monitoring filter driver is an elegant solution to the problem of spying on serial ports under NT. Moreover, the technique can be used on almost any kernel-mode device driver. For example, you could also use this idea to monitor the IRP flow of a driver you are debugging. The serial port spy is just one of many applications made possible by NT's extensible device-driver framework.

Reference

Listing 1: sermon.h — Dispatch code declarations

```
#define DECLARE_FUNCTION(x) extern "C" NTSTATUS \
    SERMON##x(IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp);
#define IMPLEMENT_FUNCTION(x) NTSTATUS \
    SERMON##x(IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp) \
    { \
```

```
        return ((CDevice *) \
        (DeviceObject->DeviceExtension))->x(Irp); \
    }

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

NTSTATUS CreateDevices(IN PDRIVER_OBJECT DriverObject,
                       IN PUNICODE_STRING RegistryPath);
}     // extern "C"

DECLARE_FUNCTION(Open)
DECLARE_FUNCTION(Read)
DECLARE_FUNCTION(Write)
DECLARE_FUNCTION(Flush)
DECLARE_FUNCTION(Cleanup)
DECLARE_FUNCTION(Close)
DECLARE_FUNCTION(IoControl)

VOID SERMONUnload(IN PDRIVER_OBJECT DriverObject);
//End of File
```

Listing 2: sermon.cpp — Initialization and dispatch code

```
#include "stdafx.h"
#include "drvclass.h"
#include "sermon.h"
#include "devext.h"

extern "C"
{
#ifdef ALLOC_PRAGMA
// all auxiliary routines that are called during
// initialization should go in here.
#pragma alloc_text(INIT,DriverEntry)
#pragma alloc_text(INIT,CreateDevices)
#endif
}

PDEVICE_OBJECT deviceObject;
PDRIVER_OBJECT DriverObject;
CDBLinkedList<CAttachedDevice> *listAttached;
```

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
                     IN PUNICODE_STRING RegistryPath)
{
    NTSTATUS status = CreateDevices(DriverObject,RegistryPath);
    if (status==STATUS_SUCCESS)
    {
        ::DriverObject=DriverObject;

        DriverObject->MajorFunction[IRP_MJ_CREATE] = SERMONOpen;
        DriverObject->MajorFunction[IRP_MJ_CLOSE] = SERMONClose;
        DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
            SERMONIoControl;
        DriverObject->MajorFunction[IRP_MJ_READ] = SERMONRead;
        DriverObject->MajorFunction[IRP_MJ_WRITE] = SERMONWrite;
        DriverObject->MajorFunction[IRP_MJ_FLUSH_BUFFERS] =
            SERMONFlush;
        DriverObject->MajorFunction[IRP_MJ_CLEANUP] =
            SERMONCleanup;

        DriverObject->DriverUnload = SERMONUnload;

        status = STATUS_SUCCESS;
        listAttached=
            new (NonPagedPool) CDBLinkedList<CAttachedDevice>;
    }
    return(status);
}

NTSTATUS CreateDevices(IN PDRIVER_OBJECT DriverObject,
                       PUNICODE_STRING RegistryPath)
{
    NTSTATUS status;
    CUString *usName;
    CUString *usLinkName;

    usName = new(PagedPool) CUString(L"\\Device\\SerMon");
    if (!OK_ALLOCATED(usName))
        return STATUS_NO_MEMORY;
    status = IoCreateDevice(DriverObject,sizeof(CSERMONDevice *),
                &usName->m_String,FILE_DEVICE_UNKNOWN,0,
                FALSE,&deviceObject);
    if (!NT_SUCCESS(status))
    {
```

```
        delete usName;
        return status;
    }

    deviceObject->Flags |= DO_BUFFERED_IO;
    deviceObject->DeviceExtension=new (NonPagedPool)CSERMONDevice;

    usLinkName= new (PagedPool)CUString(L"\\??\\SerMon");
    if (!OK_ALLOCATED(usLinkName))
    {
        IoDeleteDevice(deviceObject);
        return STATUS_NO_MEMORY;
    }
    status = IoCreateSymbolicLink (&usLinkName->m_String,
        &usName->m_String);
    if (!NT_SUCCESS(status))
    {
        IoDeleteDevice(deviceObject);
        return status;
    }

    delete usLinkName;
    delete usName;

    return status;
}

VOID SERMONUnload(IN PDRIVER_OBJECT DriverObject)
{
    delete (deviceObject->DeviceExtension);
    IoDeleteDevice(deviceObject);
    CUString *usLinkName= new (PagedPool)CUString(L"\\??\\SerMon");
    IoDeleteSymbolicLink(&usLinkName->m_String);

    delete usLinkName;

    delete listAttached;     // this will kill all attached devices

    return;
}

IMPLEMENT_FUNCTION(Read)
IMPLEMENT_FUNCTION(Write)
IMPLEMENT_FUNCTION(Flush)
```

```
IMPLEMENT_FUNCTION(Cleanup)
IMPLEMENT_FUNCTION(Open)
IMPLEMENT_FUNCTION(Close)
IMPLEMENT_FUNCTION(IoControl)
//End of File
```

Listing 3: devext.h — Main driver class declarations

```
#include "sermonex.h"

struct ExIRP
{
    PIRP Irp;
    LIST_ENTRY entry;
};

class CDevice
{
protected:
    NTSTATUS DoDefault(PIRP Irp)
    {
        IoCompleteRequest(Irp,IO_NO_INCREMENT);
        Irp->IoStatus.Status = STATUS_SUCCESS;
        Irp->IoStatus.Information = 0;
        return STATUS_IO_DEVICE_ERROR;
    };

public:
    virtual NTSTATUS IoControl(PIRP Irp) { return DoDefault(Irp);};
    virtual NTSTATUS Read(PIRP Irp) { return DoDefault(Irp); };
    virtual NTSTATUS Write(PIRP Irp) { return DoDefault(Irp); };
    virtual NTSTATUS Open(PIRP Irp) { return DoDefault(Irp); };
    virtual NTSTATUS Close(PIRP Irp) { return DoDefault(Irp); };
    virtual NTSTATUS Cleanup(PIRP Irp) { return DoDefault(Irp); };
    virtual NTSTATUS Flush(PIRP Irp) { return DoDefault(Irp); };
};

class CSERMONDevice: public CDevice
{
public:
    CSERMONDevice();
    ~CSERMONDevice();

    virtual NTSTATUS IoControl(PIRP Irp);
```

```
    MHANDLE TryConnectToSerialDevice(LPCTSTR BufferName);
};

class CAttachedDevice;
extern CDBLinkedList<CAttachedDevice> *listAttached;

NTSTATUS ReadCompletion(IN PDEVICE_OBJECT DeviceObject,
                        IN PIRP Irp,IN PVOID Context);
NTSTATUS WriteCompletion(IN PDEVICE_OBJECT DeviceObject,
                         IN PIRP Irp,IN PVOID Context);
NTSTATUS CloseCompletion(IN PDEVICE_OBJECT DeviceObject,
                         IN PIRP Irp,IN PVOID Context);
NTSTATUS OpenCompletion(IN PDEVICE_OBJECT DeviceObject,
                        IN PIRP Irp,IN PVOID Context);
NTSTATUS IOCompletion(IN PDEVICE_OBJECT DeviceObject,
                      IN PIRP Irp,IN PVOID Context);

class CAttachedDevice : public CDevice
{
protected:
    WCHAR Signature[3];
    ERESOURCE eres;
    CDBLinkedList<IOReq> io;          // request queue
    CDBLinkedList<ExIRP> pending;     // event queue

public:
    KEVENT event;
    LONG Num;
    BOOLEAN bFirstTime;
    PDEVICE_OBJECT OriginalDevice;
    PDEVICE_OBJECT ThisDevice;          // device represented by
                                        // this object
    LIST_ENTRY entry;

public:
    BOOLEAN CheckValid(void)
    {
        return (MmIsAddressValid(this) &&
            Signature[0]==L'B' && Signature[1]==L'A'
            && Signature[2]==L'V');
    };

    CAttachedDevice()
    {
```

```
        Signature[0]=L'B';
        Signature[1]=L'A';
        Signature[2]=L'V';
        ExInitializeResourceLite(&eres);
        KeInitializeEvent(&event,NotificationEvent,TRUE);
        Num=0;
        bFirstTime=TRUE;

        listAttached->New(this);
    };

    ~CAttachedDevice()
    {
        Signature[0]++;
        Signature[1]++;
// The target device MUST be closed so much times it was opened
// so we don't return until this is met
        KeWaitForSingleObject(&event,Executive,KernelMode,
            FALSE,NULL);
        IoDetachDevice(OriginalDevice);
        IoDeleteDevice(ThisDevice);
        ExDeleteResourceLite(&eres);
        ExIRP *p;
        while (p=pending.RemoveHead())
        {
            p->Irp->IoStatus.Status=STATUS_CANCELLED;
            p->Irp->IoStatus.Information=0;
            IoCompleteRequest(p->Irp, IO_NO_INCREMENT);
            delete p;
        }

        listAttached->Remove(this);
    };

    void LockExclusive(void)
    {
        ExAcquireResourceExclusiveLite(&eres, TRUE);
    };

    void LockShared(void)
    {
        ExAcquireResourceSharedLite(&eres, TRUE);
    };
```

```
    void Unlock(void)
    {
        ExReleaseResourceForThreadLite(&eres,
            ExGetCurrentResourceThread());
    };

    NTSTATUS Standard(PIRP Irp,
        PIO_COMPLETION_ROUTINE Routine=NULL);

    void New(IOReq *req);
    NTSTATUS GetNext(PIRP Irp);
    NTSTATUS GetNextSize(PIRP Irp);
    NTSTATUS ProcessSize(PIRP Irp,IOReq *);
    NTSTATUS ProcessNext(PIRP Irp,IOReq *);

// Virtual functions from CDevice
    virtual NTSTATUS IoControl(PIRP Irp)
        { return Standard(Irp,IOCompletion); };
    virtual NTSTATUS Read(PIRP Irp)
        { return Standard(Irp,ReadCompletion); };
    virtual NTSTATUS Write(PIRP Irp)
        { return Standard(Irp,WriteCompletion); };
    virtual NTSTATUS Open(PIRP Irp)
        { return Standard(Irp,OpenCompletion); };
    virtual NTSTATUS Close(PIRP Irp)
        { return Standard(Irp,CloseCompletion); };
    virtual NTSTATUS Cleanup(PIRP Irp)
        { return Standard(Irp); };
    virtual NTSTATUS Flush(PIRP Irp)
        { return Standard(Irp); };
    virtual NTSTATUS Cancel(PIRP Irp)
        { return Standard(Irp); };
};

extern PDRIVER_OBJECT DriverObject;

NTSTATUS Attach(PUNICODE_STRING TargetDeviceName,DWORD DesiredAccess,
                PDEVICE_OBJECT     *PtrReturnedDeviceObject);
//End of File
```

Listing 4: devext.cpp — Main driver class code

```
#include "stdafx.h"
#include "sermon.h"
```

```
#include "drvclass.h"
#include "devext.h"
#include "serial.h"

CSERMONDevice::CSERMONDevice(){};

CSERMONDevice::~CSERMONDevice(){};

MHANDLE CSERMONDevice::TryConnectToSerialDevice(LPCTSTR Name)
{
    CUString str((PWCHAR) Name);
    PDEVICE_OBJECT pdo;
    NTSTATUS RC=Attach(&str.m_String,FILE_ALL_ACCESS,&pdo);
    if (RC==STATUS_SUCCESS)
        return (MHANDLE) pdo->DeviceExtension;
    else
        return NULL;
}

NTSTATUS CSERMONDevice::IoControl(PIRP Irp)
{
    PIO_STACK_LOCATION curIRPStack;

    curIRPStack = IoGetCurrentIrpStackLocation(Irp);
    switch (curIRPStack->Parameters.DeviceIoControl.IoControlCode)
    {
    case  IOCTL_SERMON_STARTMONITOR:
    {
        MHANDLE mh=TryConnectToSerialDevice(
            (LPCTSTR) Irp->AssociatedIrp.SystemBuffer);
        if (mh)
        {
            *((MHANDLE *) Irp->AssociatedIrp.SystemBuffer)=mh;
            Irp->IoStatus.Information = sizeof(MHANDLE);
            Irp->IoStatus.Status = STATUS_SUCCESS;
        } else
            Irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
        break;
    }
    case IOCTL_SERMON_STOPMONITOR:
    {
        if (curIRPStack->Parameters.DeviceIoControl.InputBufferLength
            ==sizeof(MHANDLE))
        {
```

```
            MHANDLE mh=*((MHANDLE *) Irp->AssociatedIrp.SystemBuffer);
            CAttachedDevice *ptr=(CAttachedDevice *) mh;
            if (ptr && ptr->CheckValid())
            {
                Irp->IoStatus.Status = STATUS_SUCCESS;
                delete ptr;
            } else
                Irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
        } else
            Irp->IoStatus.Status = STATUS_INVALID_HANDLE;

        Irp->IoStatus.Information = 0;
        break;
    }
    case IOCTL_SERMON_GETINFOSIZE:
    {
        if (curIRPStack->Parameters.DeviceIoControl.
            InputBufferLength==sizeof(MHANDLE))
        {
            MHANDLE mh=*((MHANDLE *)
                Irp->AssociatedIrp.SystemBuffer);
            CAttachedDevice *ptr=(CAttachedDevice *) mh;
            if (ptr && ptr->CheckValid() &&
                curIRPStack->Parameters.DeviceIoControl.
                OutputBufferLength==sizeof(ULONG))
            {
                return ptr->GetNextSize(Irp);
            } else
                Irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
        } else
            Irp->IoStatus.Status = STATUS_INVALID_HANDLE;

        Irp->IoStatus.Information = 0;
        break;
    }
    case IOCTL_SERMON_GETINFO:
    {
        if (curIRPStack->Parameters.DeviceIoControl.
            InputBufferLength==sizeof(MHANDLE))
        {
            MHANDLE mh=*((MHANDLE *) Irp->AssociatedIrp.
                SystemBuffer);
            CAttachedDevice *ptr=(CAttachedDevice *) mh;
            if (ptr && ptr->CheckValid())
```

```
            {
                return ptr->GetNext(Irp);
            } else
                Irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
        } else
            Irp->IoStatus.Status = STATUS_INVALID_HANDLE;

        Irp->IoStatus.Information = 0;
        break;
    }
    default:
        Irp->IoStatus.Status = STATUS_IO_DEVICE_ERROR;
        Irp->IoStatus.Information = 0;
        break;
    }

    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return(Irp->IoStatus.Status);
}

NTSTATUS Attach(PUNICODE_STRING TargetDeviceName,
                DWORD           DesiredAccess,
                PDEVICE_OBJECT* PtrReturnedDeviceObject)
{
    NTSTATUS        RC                      = STATUS_SUCCESS;
    PDEVICE_OBJECT  PtrNewDeviceObject      = NULL;
    CAttachedDevice* ptr                    = NULL;
    BOOLEAN         InitializedDeviceObject = FALSE;
    BOOLEAN         AcquiredDeviceObject    = FALSE;
    PDEVICE_OBJECT  PtrTargetDeviceObject   = NULL;
    PFILE_OBJECT    PtrTargetFileObject     = NULL;

    ASSERT(PtrReturnedDeviceObject);

    __try
    {
        if (!NT_SUCCESS(RC=IoGetDeviceObjectPointer(
            TargetDeviceName,
            DesiredAccess,&PtrTargetFileObject,
            &PtrTargetDeviceObject)))
            __leave;

        // Create a new device object.
        if (!NT_SUCCESS(RC = IoCreateDevice(
```

```
            DriverObject,
            sizeof(CAttachedDevice *),
            NULL,          // unnamed object
            PtrTargetDeviceObject->DeviceType,
            PtrTargetDeviceObject->Characteristics,
            FALSE,     // Not exclusive.
            &PtrNewDeviceObject)))
        {
            // failed to create a device object
            __leave;
        }

        PtrNewDeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;

        // Initialize the extension for the device object.
        (PtrNewDeviceObject->DeviceExtension)=ptr=
            new (NonPagedPool) CAttachedDevice;
        ptr->OriginalDevice=PtrTargetDeviceObject;
        ptr->ThisDevice=PtrNewDeviceObject;

        ptr->LockExclusive();
        AcquiredDeviceObject=TRUE;
        // attach to the target FSD.
        RC = IoAttachDeviceByPointer(PtrNewDeviceObject,
            PtrTargetDeviceObject);

        ASSERT(NT_SUCCESS(RC));

        PtrNewDeviceObject->Flags |= DO_BUFFERED_IO;
    } __finally {
        if (AcquiredDeviceObject)
        {
            ptr->Unlock();
            AcquiredDeviceObject = FALSE;
        }

        if (!NT_SUCCESS(RC) && PtrNewDeviceObject)
        {
            delete ptr;
        } else
        {
            *PtrReturnedDeviceObject = PtrNewDeviceObject;
        }
        if (PtrTargetFileObject)
```

```
        {
            ObDereferenceObject(PtrTargetFileObject);
            PtrTargetFileObject=NULL;
        }
    }

    return(RC);
}

NTSTATUS DefaultCompletion(IN PDEVICE_OBJECT DeviceObject,
                           IN PIRP Irp,IN PVOID Context)
{
    if (Irp->PendingReturned) {
        IoMarkIrpPending(Irp);
    }

// Nothing to do, simply return success
    return STATUS_SUCCESS;
}

NTSTATUS ReadCompletion(IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp,
                        IN PVOID Context)
{
    if (Irp->PendingReturned) {
        IoMarkIrpPending(Irp);
    }
    {
        if (Irp->IoStatus.Information)
        {
// There are bytes read, construct IOReq item and append it to the list
            PIO_STACK_LOCATION cur;
            cur = IoGetCurrentIrpStackLocation(Irp);
            IOReq *req=new (NonPagedPool) IOReq(REQ_READ,
                cur->Parameters.Read.Length,
                Irp->IoStatus.Information,
                Irp->AssociatedIrp.SystemBuffer);
            ((CAttachedDevice *) Context)->New(req);
        }
    }

    return STATUS_SUCCESS;
}

NTSTATUS WriteCompletion(IN PDEVICE_OBJECT DeviceObject,
```

```
                            IN PIRP Irp,IN PVOID Context)
{
    if (Irp->PendingReturned) {
        IoMarkIrpPending(Irp);
    }
    {
        if (Irp->IoStatus.Status==STATUS_SUCCESS &&
            Irp->IoStatus.Information)
        {
            PIO_STACK_LOCATION cur;
            cur = IoGetCurrentIrpStackLocation(Irp);
            IOReq *req=new (NonPagedPool) IOReq(REQ_WRITE,
                cur->Parameters.Read.Length,
                Irp->IoStatus.Information,
                Irp->AssociatedIrp.SystemBuffer);
            ((CAttachedDevice *) Context)->New(req);
        }
    }

    return STATUS_SUCCESS;
}

NTSTATUS OpenCompletion(IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp,
                        IN PVOID Context)
{
    if (Irp->PendingReturned) {
        IoMarkIrpPending(Irp);
    }
    {
        if (Irp->IoStatus.Status==STATUS_SUCCESS)
        {
            CAttachedDevice *p=(CAttachedDevice *) Context;
// Increase usage count
            if (InterlockedIncrement(&p->Num))
                KeResetEvent(&p->event);
            PIO_STACK_LOCATION cur;
            cur = IoGetCurrentIrpStackLocation(Irp);
            IOReq *req=new (NonPagedPool) IOReq(REQ_OPEN);
            ((CAttachedDevice *) Context)->New(req);
        }
    }

    return STATUS_SUCCESS;
}
```

```
NTSTATUS CloseCompletion(IN PDEVICE_OBJECT DeviceObject,
                         IN PIRP Irp,IN PVOID Context)
{
    if (Irp->PendingReturned) {
        IoMarkIrpPending(Irp);
    }
    {
        if (Irp->IoStatus.Status==STATUS_SUCCESS)
        {
            CAttachedDevice *p=(CAttachedDevice *) Context;
            if (!p->bFirstTime)
            {
// Decrease usage count and signal the event if it falls to zero
                if (!InterlockedDecrement(&p->Num))
                    KeSetEvent(&p->event,0,FALSE);
                PIO_STACK_LOCATION cur;
                cur = IoGetCurrentIrpStackLocation(Irp);
                IOReq *req=new (NonPagedPool) IOReq(REQ_CLOSE);
                ((CAttachedDevice *) Context)->New(req);
            } else p->bFirstTime=FALSE;

        }
    }

    return STATUS_SUCCESS;
}

NTSTATUS IOCompletion(IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp,
                      IN PVOID Context)
{
    if (Irp->PendingReturned) {
        IoMarkIrpPending(Irp);
    }
    {
        if (Irp->IoStatus.Status==STATUS_SUCCESS)
        {
// We process only IOCTL_SERIAL_SET_BAUD_RATE and
// IOCTL_SERIAL_SET_LINE_CONTROL requests
// (serial.h file from serial driver
// sample from DDK is used)
            IOReq *req;
            CAttachedDevice *p=(CAttachedDevice *) Context;
            PIO_STACK_LOCATION cur;
```

```
            cur = IoGetCurrentIrpStackLocation(Irp);
            switch(cur->Parameters.DeviceIoControl.IoControlCode)
            {
            case IOCTL_SERIAL_SET_BAUD_RATE:
                req=new (NonPagedPool) IOReq(REQ_SETBAUDRATE,
                    sizeof(ULONG),sizeof(ULONG),
                    Irp->AssociatedIrp.SystemBuffer);
                ((CAttachedDevice *) Context)->New(req);
                break;
            case IOCTL_SERIAL_SET_LINE_CONTROL:
                req=new (NonPagedPool) IOReq(REQ_SETLINECONTROL,
                    sizeof(SERIAL_LINE_CONTROL),
                    sizeof(SERIAL_LINE_CONTROL),
                    Irp->AssociatedIrp.SystemBuffer);
                ((CAttachedDevice *) Context)->New(req);
                break;
            }
        }
    }

    return STATUS_SUCCESS;
}

//////////////////////////////
// CAttachedDevice
NTSTATUS CAttachedDevice::Standard(PIRP Irp,
                                   PIO_COMPLETION_ROUTINE Routine)
{
// This function forwards the request
    PIO_STACK_LOCATION curIRPStack,nextIRPStack;
    curIRPStack = IoGetCurrentIrpStackLocation(Irp);
    nextIRPStack=IoGetNextIrpStackLocation(Irp);
    *nextIRPStack=*curIRPStack;
    IoSetCompletionRoutine(Irp,
        (Routine)?Routine:DefaultCompletion,
        this, TRUE, TRUE, TRUE);

    return IoCallDriver(OriginalDevice,Irp);
}

void CAttachedDevice::New(IOReq *req)
{
    LockExclusive();
    io.New(req);
```

```
// check for pending requests
    ExIRP *trp=pending.RemoveHead();
// If we have a pending request, waiting for smth to receive,
// so simply handle it
    if (trp)
    {
        IRP *Irp=trp->Irp;
        req=io.RemoveHead();
        Unlock();     // we don't need to lock anymore
        PIO_STACK_LOCATION curIRPStack;
        curIRPStack = IoGetCurrentIrpStackLocation(Irp);
// process the request
        switch(curIRPStack->Parameters.DeviceIoControl.
            IoControlCode)
        {
        case IOCTL_SERMON_GETINFOSIZE:
            ProcessSize(Irp,req);
            break;
        case IOCTL_SERMON_GETINFO:
            ProcessNext(Irp,req);
            break;
        }
        delete trp;
    } else
        Unlock();
}

NTSTATUS CAttachedDevice::ProcessSize(PIRP Irp,IOReq *q)
{
    Irp->IoStatus.Information=sizeof(ULONG);
    Irp->IoStatus.Status=STATUS_SUCCESS;
    LockExclusive();
    *((ULONG *) Irp->AssociatedIrp.SystemBuffer)=sizeof(IOReq)+
        q->SizeCopied;
    io.InsertHead(q);
    Unlock();
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

NTSTATUS CAttachedDevice::ProcessNext(PIRP Irp,IOReq *q)
{
    PIO_STACK_LOCATION curIRPStack;
    curIRPStack = IoGetCurrentIrpStackLocation(Irp);
```

```
    if (curIRPStack->Parameters.DeviceIoControl.OutputBufferLength
        < sizeof(IOReq)+q->SizeCopied)
    {
        delete q;
        Irp->IoStatus.Information=0;
        Irp->IoStatus.Status=STATUS_BUFFER_TOO_SMALL;
        IoCompleteRequest(Irp,IO_NO_INCREMENT);
        return STATUS_BUFFER_TOO_SMALL;
    }
    Irp->IoStatus.Information=sizeof(IOReq)+q->SizeCopied;
    Irp->IoStatus.Status=STATUS_SUCCESS;
    RtlCopyMemory(Irp->AssociatedIrp.SystemBuffer,q,sizeof(IOReq));
    if (q->pData)
        RtlCopyMemory((PCHAR) Irp->AssociatedIrp.SystemBuffer
        +sizeof(IOReq),q->pData,q->SizeCopied);
    delete q;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

NTSTATUS CAttachedDevice::GetNextSize(PIRP Irp)
{
    if (!io.IsEmpty())
    {
        LockExclusive();
        IOReq *q=io.RemoveHead();
        NTSTATUS ret=ProcessSize(Irp,q);
        Unlock();
        return ret;
    } else
    {
        ExIRP *irp=new (NonPagedPool) ExIRP;
        irp->Irp=Irp;
        pending.New(irp);

        Irp->IoStatus.Information=0;
        Irp->IoStatus.Status=STATUS_PENDING;
        IoMarkIrpPending(Irp);
        return STATUS_PENDING;
    }
}

NTSTATUS CAttachedDevice::GetNext(PIRP Irp)
{
```

```
    LockExclusive();
    if (!io.IsEmpty())
    {
        IOReq *q=io.RemoveHead();
        Unlock();
        return ProcessNext(Irp,q);
    } else
    {
        ExIRP *irp=new (NonPagedPool) ExIRP;
        irp->Irp=Irp;
        pending.New(irp);

        Irp->IoStatus.Information=0;
        Irp->IoStatus.Status=STATUS_PENDING;
        IoMarkIrpPending(Irp);
        Unlock();
        return STATUS_PENDING;
    }
}
//End of File
```