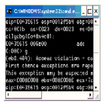
Driver Development Part 3: Introduction to driver contexts



Toby Opferman
20 Feb 2005

This article will go deeper into the basics of creating a simple driver.

Download source code - 21.8 Kb

Introduction

This is the third edition of the Writing Device Drivers articles. The first article helped to simply get you acquainted with device drivers and a simple framework for developing a device driver for NT. The second tutorial attempted to show how to use IOCTLs and display what the memory layout of Windows NT is. In this edition, we will go into the idea of contexts and pools. The driver we write today will also be a little more interesting as it will allow two user mode applications to communicate with each other in a simple manner. We will call this the "poor man's pipes" implementation.

What is a Context?

This is a generic question, and if you program in Windows, you should understand the concept. In any case, I will give a brief overview as a refresher. A context is a user-defined data structure (users are developers) which an underlying architecture has no knowledge of what it is. What the architecture does do is pass this context around for the user so in an event driven architecture, you do not need to implement global variables or attempt to determine what object, instance, data structure, etc. the request is being issued for.

In Windows, some examples of using contexts would be `SetWindowLong` with `GWL_USERDATA`, `EnumWindows`, `CreateThread`, etc. These all allow you to pass in contexts which your application can use to distinguish and implement multiple instances of functions using only one implementation of the function.

Device Context

If you recall, in the first article, we learned how to create a device object for our driver. The driver object contains information related to the physical instance of this driver in memory. There is obviously only one per driver binary and it contains things such as the function entry points for this binary. There can be multiple devices associated with the same binary as we know we can simply call "`IoCreateDevice`" to create any number of devices that are handled by a single driver object. This is the reason that all entry points send in a device object instead of a driver object, so you can determine which device the function is being invoked for. The device objects point back to the driver object so you can still relate back to it.

```
    NtStatus = IoCreateDevice(pDriverObject, sizeof(EXAMPLE_DEVICE_CONTEXT),
                      &usDriverName, FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN,
                      FALSE, &pDeviceObject);
```

```
...

    /*
    * Per-Device Context, User Defined
    */
    pExampleDeviceContext =
          (PEXAMPLE_DEVICE_CONTEXT)pDeviceObject->DeviceExtension;

    KeInitializeMutex(&pExampleDeviceContext->kListMutex, 0);
    pExampleDeviceContext->pExampleList  = NULL;
```

The "`IoCreateDevice`" function contains a parameter for the size of a "Device Extension". This can then be used to create the "deviceextension" member of the device object and this represents the user defined context. You can then create your own data structure to be passed around and used with each device object. If you define multiple devices for a single driver, you may want to have a single shared member among all your device contexts as the first member so you can quickly determine which device this function is being invoked for. The device represents the \Device\Name.

The context will generally contain any type of list which would need to be searched for this device, or attributes and locks for this device. An example of data which would be global per device would be free space on a disk drive. If you have three devices which each representing a particular disk drive image, the attributes which are particular for a certain device would be global for each instance of a device. As mentioned, the volume name, free space, used space, etc. would be per-device but global for all instances of the device.

Resource Context

This is something new but you can open a device with a longer string to specify a certain resource managed by the device itself. In the case of the file system, you would actually be specifying a file name and file path. As an example, the device can actually be opened by using *\Device\A\Program Files\myfile.txt*. Then the driver may want to allocate a context which is global for all processes who open this particular resource. In the example of the file system, items which may be global for an instance of a file could be certain cached items such as the file size, file attributes, etc. These would be unique per-file but shared among all instance handles to this file.

```
{   /*
     * We want to use the unicode string that was used to open the driver to
     * identify "pipe contexts" and match up applications that open the same name
     * we do this by keeping a global list of all open instances in the device
     * extension context.
     * We then use reference counting so we only remove an instance from the list
     * after all instances have been deleted.  We also put this in the FsContext
     * of the IRP so all IRP's will be returned with this so we can easily use
     * the context without searching for it.
     */
     if(RtlCompareUnicodeString(&pExampleList->usPipeName,
                                &pFileObject->FileName, TRUE) == 0)
     {
         bNeedsToCreate = FALSE;
         pExampleList->uiRefCount++;
         pFileObject->FsContext = (PVOID)pExampleList;
```

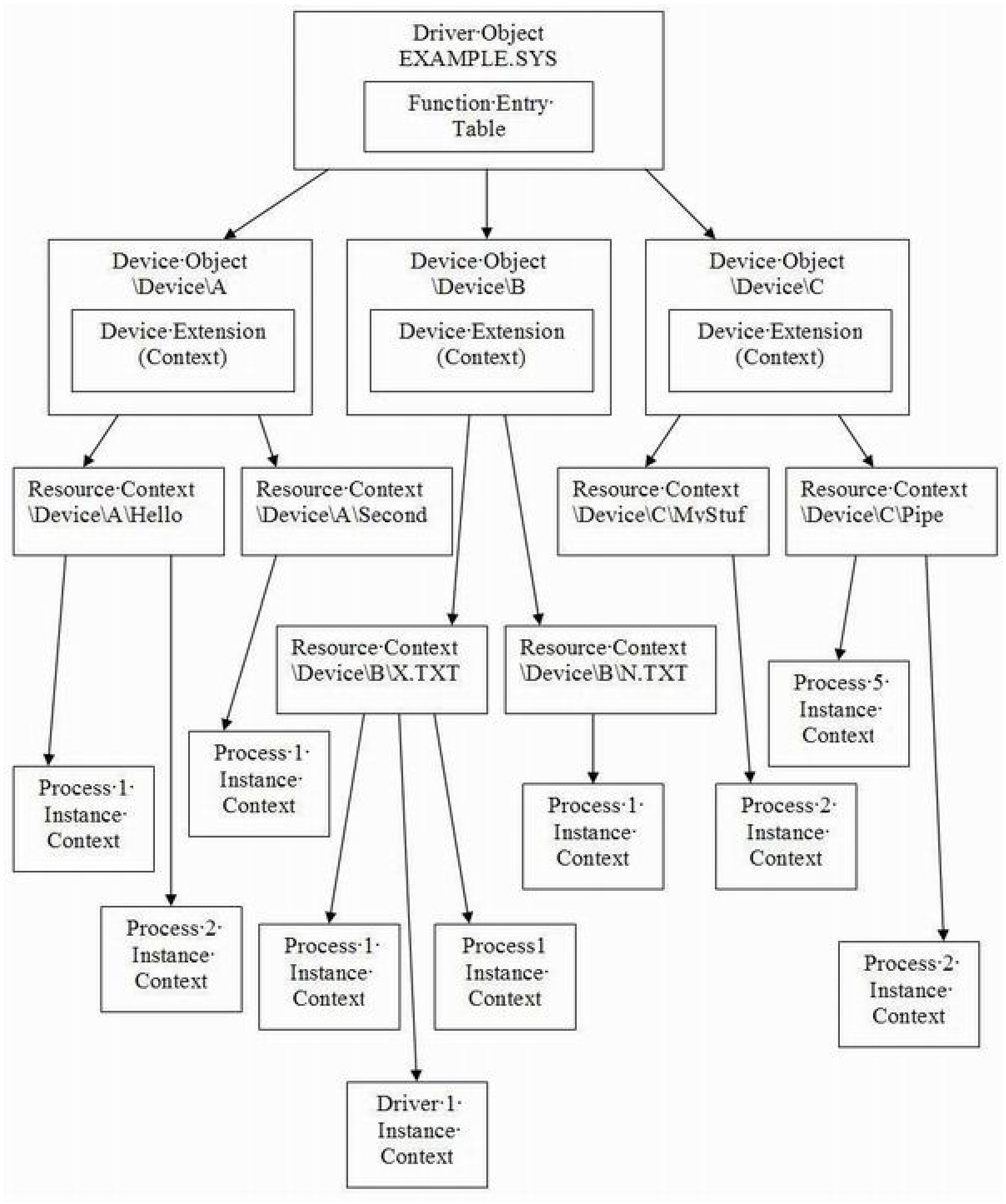
Instance Context

This is the most unique context that you may want to create. It is unique for every single handle created on the system. So if process 1 and process 2 both open a new handle to the same file, while their resource context may be the same their instance context will be unique. A simple example of items which may be unique for each instance could be the file pointer. While both processes have opened the same file, they may not be reading the file from the same location. That means that each open instance handle to the file must maintain its own context data that remembers the location of the file currently being read by each particular handle.

Instance contexts and any context can always have pointers back to resource contexts and device contexts just as the device object has a pointer back to the driver object. These can be used where necessary to avoid needing to use look up tables and search lists for the appropriate context.

The Big Picture

The following diagram outlines the big picture and relationships just described above. This should help you to visualize how you may want to structure relationships within your driver. The context relationship can be structured any way you want, this is just an example. You can even create contexts outside of the three mentioned here that have their own scopes you defined.


Driver Object EXAMPLE.SYS
Function Entry Table
Device Object \Device\A
Device Extension (Context)
Device Object \Device\B
Device Extension (Context)
Device Object \Device\C
Device Extension (Context)
Resource Context \Device\A\Hello
Resource Context \Device\A\Second
Resource Context \Device\C\MyStuf
Resource Context \Device\C\Pipe
Resource Context \Device\B\X.TXT
Resource Context \Device\B\N.TXT
Process 5 Instance Context
Process 1 Instance Context
Process 1 Instance Context
Process 1 Instance Context
Process 2 Instance Context
Process 2 Instance Context
Process 1 Instance Context
Process1 Instance Context
Process 2 Instance Context
Driver 1 Instance Context

Our Implementation

The implementation that will be used in our driver is to have a device context and a resource context. We do not need instance contexts for what we are doing.

We first create a device extension using `IoCreateDevice`. This data structure will be used to maintain the list of resource contexts so all calls to "`Create`" can then be associated with the proper resource context.

The second implementation we have is to simply create resource contexts. We first attempt to search the list on Create to determine if the resource already exists. If it does, we will simply increment the reference counter and associate it with that handle instance. If it does not, we simply create a new one and add it to the list.

The Close has the opposite operation. We will simply decrement the reference count and if it reaches 0, we then remove the resource context from the list and delete it.

The IRP's `IO_STACK_LOCATION` (if it) provides a pointer to a FILE_OBJECT which we can use as a handle instance. It contains two fields we can use to store contexts and we simply use one of them to store our resource context. We could also use these to store our instance contexts if we choose to. Certain drivers may have rules and be using this for different things, but we are developing this driver outside of any framework and there are no other drivers to communicate with. This means we are free to do whatever we want but if you choose to implement a driver of a particular class, you may want to make sure what is available to you.

To associate resources, we simply use the name of the device string being passed in. We now append a new string onto the end of our device name to create different resources. If two applications then open the same resource string, they will be associated and share the same resource context. This resource context we have created simply maintains its own locking and a circular buffer. This circular buffer, residing in kernel memory, is accessible from any process. Thus, we can copy memory from one process and give it to another.

Memory Pools

In this driver, we finally start to allocate memory. In the driver, allocations are called "pools" and you allocate memory from a particular pool. In user mode, you allocate memory from the heap. In this manner, they are essentially the same. There is a manager which keeps track of these allocations and provides you with the memory. In user mode, however, while there can be multiple heaps, they are essentially the same type of memory. Also, in user mode, each set of heaps used by a process is only accessible by that process. Two processes do not share the same heap.

```
pExampleList = (PEXAMPLE_LIST)ExAllocatePoolWithTag(NonPagedPool,
                          sizeof(EXAMPLE_LIST), EXAMPLE_POOL_TAG);

if(pExampleList)
{
```

In the kernel, things change a little bit. There are essentially two basic types of pools, paged and non-paged. The paged pool is essentially memory that can be paged out to disk and should only be used at IRQL < `DISPATCH_LEVEL` as explained in the first tutorial. Non-paged memory is different; you can access it anywhere at anytime because it's never paged out to disk. There are things to be aware of, though you don't want to consume too much non-paged pool for obvious reasons, you start to run out of physical memory.

The pools are also shared between all drivers. That being the case, there is something that you can do to help debug pool issues and that is specifying a "pool tag". This is a four byte identifier which is put in the pool header of your allocated memory. That way, if say, you overwrote your memory boundary, then all the sudden the file system driver crashes, you can look at the memory in the pool before the memory being accessed is invalid and notice that your driver possibly corrupted the next pool entry. This is the same concept as in user mode and you can even enable heap tagging there as well. You generally want to think of some unique name to identify your driver's memory. This string is also usually written backwards so it's displayed forwards when using the debugger. Since the debugger will dump the memory in `DWORD`s, the high memory will be displayed first.

In our driver, we allocate from the non-paged pool simply because we have a `KMUTEX` inside the data structure. We could have allocated this separately and maintained a pointer here, but for simplicity, we simply have one allocation. `KMUTEX` objects must be in non-paged memory.

Kernel Mutexes

In this article, we start to get into creating objects you may already be familiar with in user mode. The mutex is actually the same in the kernel as it was when you used it in user mode. In fact, each process actually has what is called a "handle table" which is simply a mapping between user mode handles and kernel objects. When you create a mutex in user mode, you actually get a mutex object created in the kernel and this is exactly what we are creating today.

The one difference we need to establish is that the mutex handle we create in the kernel is actually a data structure used by the kernel and it must be in non-paged memory. The parameters to wait on a mutex are a little more complicated than we are used to however.

The MSDN documentation for KeWaitForMutexObject can be found by clicking the link. The documentation does mention that this is simply a macro that is really `KeWaitForSingleObject`.

So, what do the parameters mean? These options are explained at MSDN, however here is essentially a summary.

The first one is obvious, it's the actual mutex object. The second parameter is a little stranger, it's either `UserRequest` or `Executive`. The `UserRequest` essentially means the wait is waiting for the user and `Executive` means the wait is waiting for the scheduler. This is simply an information field, and if a process queries for the reason why this thread is waiting, this is what is returned. It doesn't actually affect what the API does.

The next set of options specifies the wait mode. `KernelMode` or `UserMode` are your options. Drivers will essentially use "`KernelMode`" in this parameter. If you do your wait in `UserMode`, your stack could be paged out so you would be unable to pass parameters on the stack.

The third parameter is "`Alertable`" and this specifies if the thread is alertable while it waits. If this is true, then APCs can be delivered and the wait interrupted. The API will return with an APC status.

The last parameter is the timeout and it is a LARGE_INTEGER. If you wanted to set a wait, the code would be the following:

```
LARGE_INTEGER TimeOut;

TimeOut.QuadPart = 10000000L;
TimeOut.QuadPart *= NumberOfSeconds;
TimeOut.QuartPart = -(TimeOut.QuartPart);
```

The timeout value is relative time so it must be negative.

Our implementation attempts a simple approach and specifies `KernelMode`, non-alterable, and no timeout.

```
NtStatus = KeWaitForMutexObject(&pExampleDeviceContext->kListMutex,
                                Executive, KernelMode, FALSE, NULL);

if(NT_SUCCESS(NtStatus))
{
```

You can find detailed information about how mutexes work, at this location in MSDN.

Poor Man's Pipes Implementation

The project represents a very simple implementation. In this section, we will evaluate how the driver operates and some things to think about on how we could improve the implementation. We will also cover how to use the example driver.

Security

This is very simple, there is none! The driver itself sets absolutely no security so essentially we don't care who we allow to write or read from any buffer. Since we don't care, this IPC can be used by any processes to communicate regardless of the user or their privilege.

The question then becomes does this really matter? I am not a security expert but to me, it all really depends. If your intention is to allow this to be used by anyone, then you may not want to implement security. If you think the users want to enforce only SYSTEM processes or not allow cross user IPC, then this is something to consider. There are cases for both. The other could possibly be that you don't care about the user but rather you only wish that only two certain processes can communicate and no others. In this situation, perhaps you want to setup some type of registration or security so that you only allow the appropriate processes to open handles and the application then dictates what security it wants on its pipes. You could also have a model where you don't use names but rather do per-instance handling. In this case, they may be required to duplicate the handles into other processes, for example.

Circular Buffer

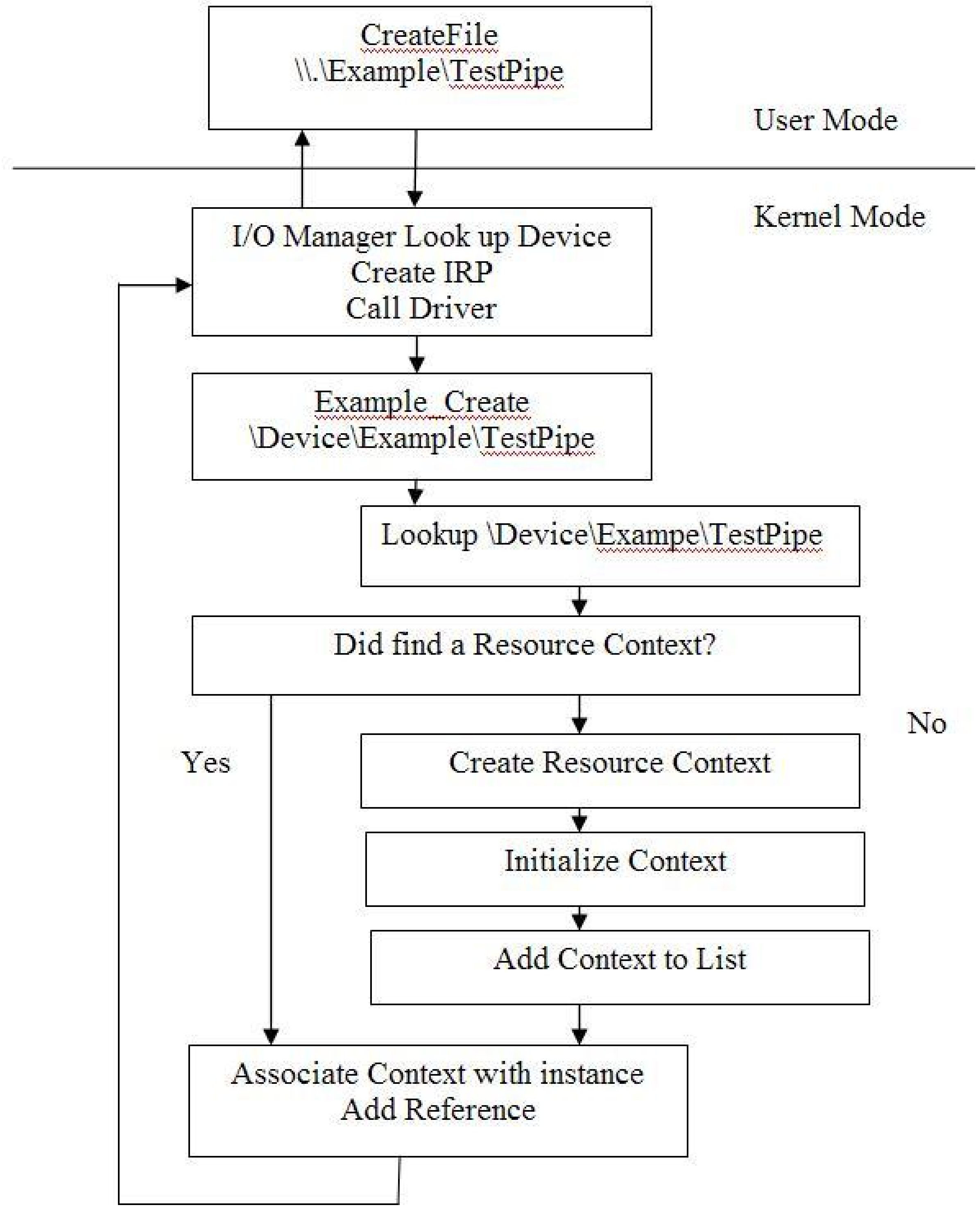
The circular buffer is a simple implementation, it never blocks a read or write and will simply ignore extra data. The buffer size is also not configurable so the application is stuck with the value we hard-coded.

Does this need to be the case? Definitely not, as we saw in Part 2, we can create our own IOCTLs to issue requests to the driver. An IOCTL could be implemented to do some configuration with the driver such as how big the buffer should be. The other part could be handling. Some circular buffers actually will start wrapping around and over writing old data with new data. This could be a flag on whether you want it to ignore new data or overwrite existing data with it.

The circular buffer implementation is not driver specific so I will not be going over its implementation in detail.

Graphical Flow of Example

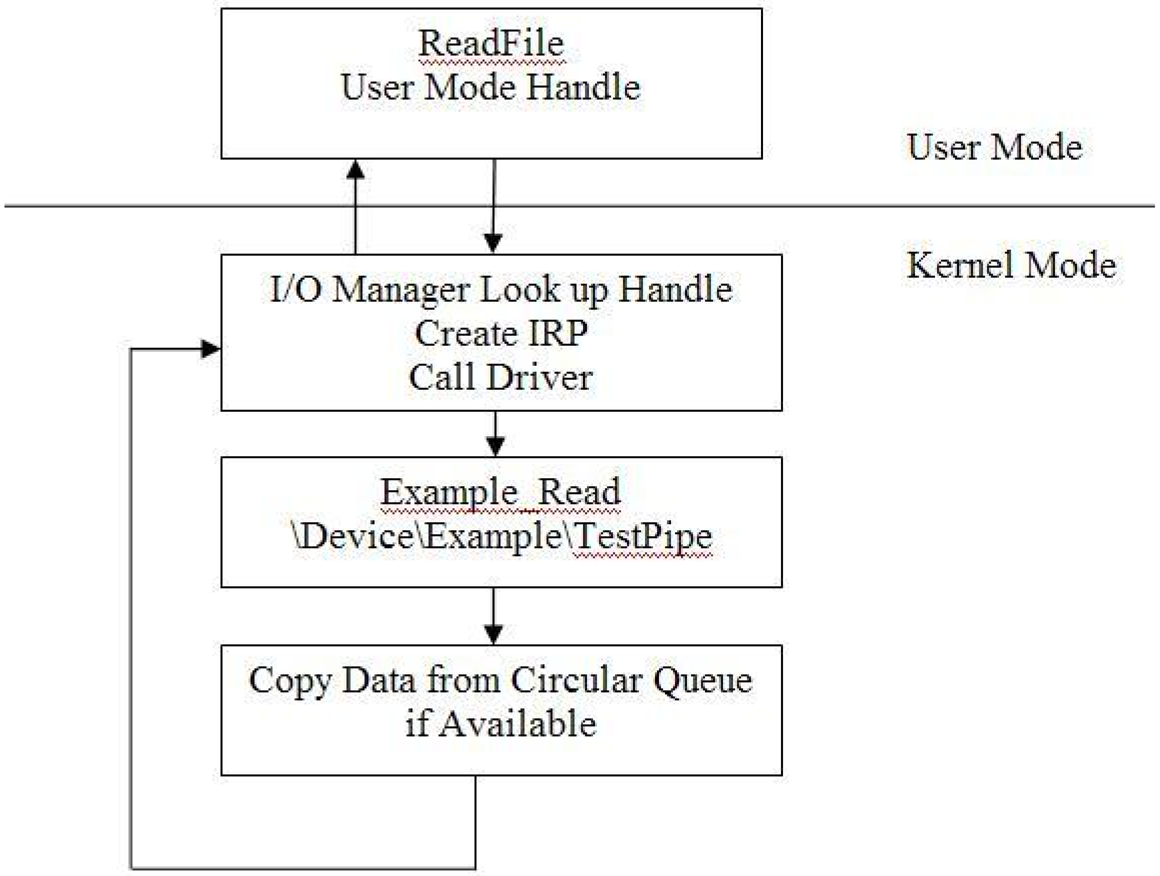
This is a simple illustration of the flow of this example. The `CreateFile()` API will reference this object using the symbolic linker "Example". The I/O manager will map the DOS device name to the NT Device "*\Device\Example*" and append any string we put beyond this name (like, "*\TestPipe*"). We get the IRP created by the device manager and we will first look up using the device string if we already have created a resource context. If yes, we simply use the `FileObject` of the I/O Stack Location to put our resource context after we add a reference. If not, then we need to create it first.



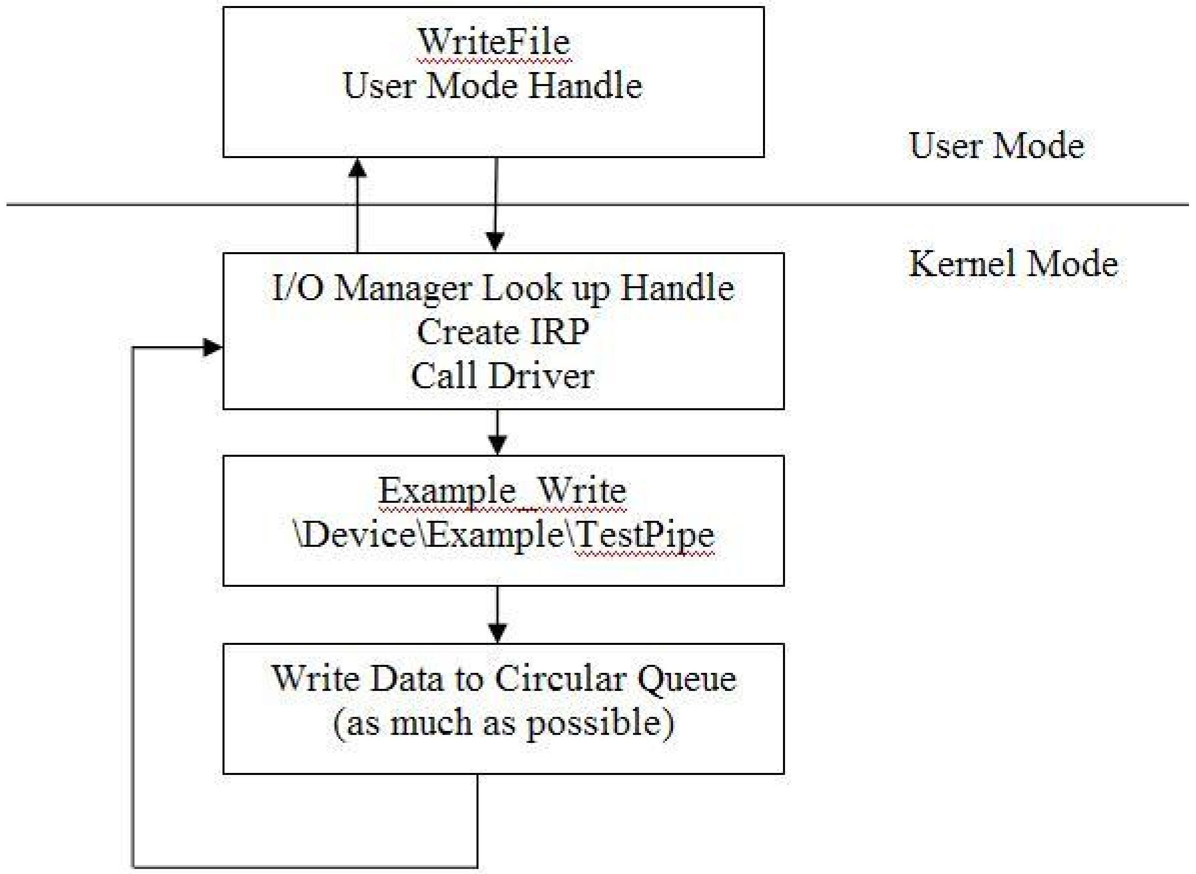
As a quick reference though, the `FILE_OBJECT` will actually only contain the extra "\TestPipe". Here is an example:

```
kd> dt _FILE_OBJECT ff6f3ac0
   +0x000 Type             : 5
   +0x002 Size             : 112
   +0x004 DeviceObject     : 0x80deea48
   +0x008 Vpb              : (null)
   +0x00c FsContext        : (null)
   +0x010 FsContext2       : (null)
   +0x014 SectionObjectPointer : (null)
   +0x018 PrivateCacheMap  : (null)
   +0x01c FinalStatus      : 0
   +0x020 RelatedFileObject : (null)
   +0x024 LockOperation    : 0 ''
   +0x025 DeletePending    : 0 ''
   +0x026 ReadAccess       : 0 ''
   +0x027 WriteAccess      : 0 ''
   +0x028 DeleteAccess     : 0 ''
   +0x029 SharedRead       : 0 ''
   +0x02a SharedWrite      : 0 ''
   +0x02b SharedDelete     : 0 ''
   +0x02c Flags            : 2
   +0x030 FileName         : _UNICODE_STRING "\HELLO"
   +0x038 CurrentByteOffset : _LARGE_INTEGER 0x0
   +0x040 Waiters          : 0
   +0x044 Busy             : 0
   +0x048 LastLock         : (null)
   +0x04c Lock             : _KEVENT
   +0x05c Event            : _KEVENT
   +0x06c CompletionContext : (null)
```

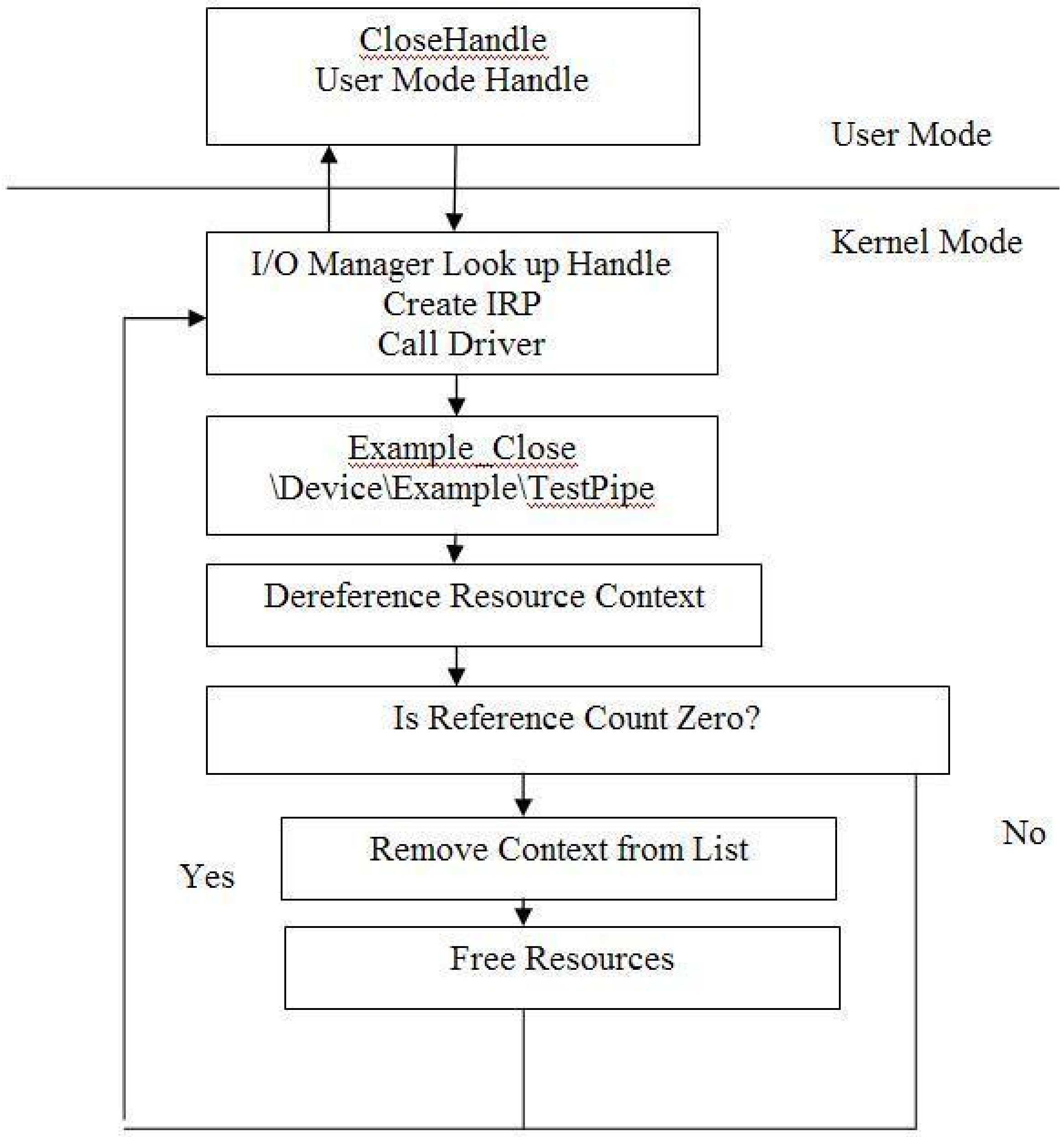
This is a simple illustration of how the `ReadFile` operation works. Since we associated our own context on the `FILE_OBJECT`, we do not need to perform look ups and we can simply access the appropriate circular buffer when we do the Read.



This is a simple illustration of how the `WrteFile` operation works. Since we associated our own context on the `FILE_OBJECT`, we do not need to perform look ups and we can simply access the appropriate circular buffer when we do the Write.



The close handle, we will simply dereference the resource context. If the context is now 0, we will delete it from the global list. If it is not, then we will simply do nothing more. One thing to remember is that this is a simple illustration and we are actually handling `IRP_MJ_CLOSE` and not `IRP_MJ_CLEANUP`. This code could has been put into either one since what we are doing does not interact with the user mode application. However, if we were freeing resources that should be done in the context of the application, we would need to move this to `IRP_MJ_CLEANUP` instead. Since IRP_MJ_CLOSE is not guaranteed to run in the context of the process, this illustration is more of how an `IRP_MJ_CLEANUP` could have occurred.



Although MSDN does state the `IRP_MJ_CLOSE` is not called in the context of the process, it doesn't mean that this is always true. The below stack trace shows it being called in the context of the application. If you debug and find this and think that you can simply ignore the warning on MSDN, I would think again. There is a reason that it is documented that way even if it does not always behave that way. There is another side of the coin which is, even if it doesn't behave that way, it doesn't mean things can't change in the future since they are documented that way. This is a general statement that you do not see something behave one way and expect it to always be the case. There is a document on Handling IRPs that describes the behavior of `IRP_MJ_CLOSE` and `IRP_MJ_CLEANUP`, at this location.

```
THREAD ff556020  Cid 0aa4.0b1c  Teb: 7ffde000
                 Win32Thread: 00000000 RUNNING on processor 0
        IRP List:
            ffa1b6b0: (0006,0094) Flags: 00000404  Mdl: 00000000
        Not impersonating
        DeviceMap                 e13b0d20
        Owning Process            ff57d5c8       Image:         usedriver3.exe
        Wait Start TickCount      26769661       Ticks: 0
        Context Switch Count      33
        UserTime                  00:00:00.0000
        KernelTime                00:00:00.0015
        Start Address kernel32!BaseProcessStartThunk (0x77e4f35f)
*** WARNING: Unable to verify checksum for usedriver3.exe
*** ERROR: Module load completed but symbols could not be loaded for usedriver3.exe
        Win32 Start Address usedriver3 (0x00401172)
        Stack Init faa12000 Current faa11c4c Base faa12000 Limit faa0f000 Call 0
        Priority 10 BasePriority 8 PriorityDecrement 2
        ChildEBP RetAddr
        faa11c70 804e0e0d example!Example_Close (FPO: [2,0,2])
                          (CONV: stdcall) [.\functions.c @ 275]
        faa11c80 80578ce9 nt!IofCallDriver+0x3f (FPO: [0,0,0])
        faa11cb8 8057337c nt!IopDeleteFile+0x138 (FPO: [Non-Fpo])
        faa11cd4 804e4499 nt!ObpRemoveObjectRoutine+0xde (FPO: [Non-Fpo])
        faa11cf0 8057681a nt!ObfDereferenceObject+0x4b (FPO: [EBP 0xfaa11d08] [0,0,0])
        faa11d08 8057687c nt!ObpCloseHandleTableEntry+0x137 (FPO: [Non-Fpo])
        faa11d4c 805768c3 nt!ObpCloseHandle+0x80 (FPO: [Non-Fpo])
        faa11d58 804e7a8c nt!NtClose+0x17 (FPO: [1,0,0])
        faa11d58 7ffe0304 nt!KiSystemService+0xcb (FPO: [0,0] TrapFrame @ faa11d64)
        0012fe24 77f42397 SharedUserData!SystemCallStub+0x4 (FPO: [0,0,0])
        0012fe28 77e41cb3 ntdll!ZwClose+0xc (FPO: [1,0,0])
        0012fe30 0040110d kernel32!CloseHandle+0x55 (FPO: [1,0,0])
WARNING: Stack unwind information not available. Following frames may be wrong.
        0012ff4c 00401255 usedriver3+0x110d
        0012ffc0 77e4f38c usedriver3+0x1255
        0012fff0 00000000 kernel32!BaseProcessStart+0x23 (FPO: [Non-Fpo])
```

Using the Example

The example is split into two new user mode processes, usedriver2 and usedriver3. The userdriver2 will allow you to type in data and it will send it to the driver. The userdriver3 source will allow you to press Enter and it will read data from the driver. Obviously, if it reads multiple strings the way it's currently implemented, you will only see the first string displayed.

There is one parameter that needs to be provided and this is the name of the resource to open. This is an arbitrary name that simply allows the driver to tie two handle instances together so multiple applications can share data at the same time! "usedriver 2 HELLO" "usedriver3 HELLO" "userdriver2 Temp" "usedriver3 Temp" will open \Device\Example\HELLO and "\Device\Example\Temp" and the appropriate versions will talk to the applications with the same handle. The current implementation creates resources **case insensitive**. It's very simple to change this, the `RtlCompareUnicodeString` function's last parameter specifies whether to compare strings case sensitive or case insensitive.

Building the Examples

This is something that I have not gone into in previous articles. The projects included with these articles can be unzipped using the directory structure in the ZIP itself. There are "makefiles" included in the project so you can simply do "nmake clean" then "nmake" to build these binaries.

The makefiles may need to be changed to point to the location of your DDK (which you can order from Microsoft for the cost of shipping and handling). These makefiles point to *C:\NTDDK\xxx*, you can then just change this to your location. If you do not have nmake in your path, you may want to make sure that the Visual Studio environment is setup in your command prompt. You go to the binaries directory of Visual Studio and just run "*VCVARS32.BAT*".

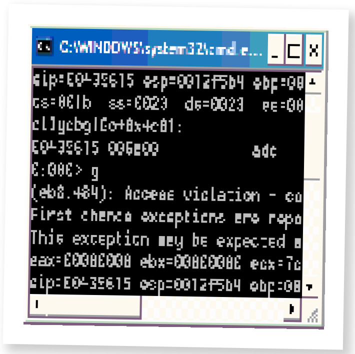
There may be an error when it attempts to use "rebase". These makefiles were simply copied from other projects so the rebase is actually not necessary. It was actually only being used before to strip out debug symbols. The error can be fixed by either removing the rebase sequence from the makefile or by creating the *SYMBOLS* directory under the *BIN* directory. The reason rebase is complaining is simply because the directory does not exist.

Conclusion

In this article, we learned a bit more about user-mode and kernel-mode interactions and how to create a very simple IPC. We learned about creating contexts in device drivers as well as how to allocate memory and use synchronization objects in the kernel.

License

About the Author



```
eip=E0435615 esp=0012f5b4 ebp=00
cs=001b  ss=0023  ds=0023  es=00
cl]ycbg!Go+0x4c01:
E0435615 0060009          adc
E:000> g
(eb8.484): Access violation - co
First chance exceptions are repo
This exception may be expected a
eax=E0000000 ebx=0000000E ecx=7c
eip=E0435615 esp=0012f5b4 ebp=00
```

Toby Opferman
Engineer Intel
United States

Toby Opferman has worked in just about all aspects of Windows development including applications, services and drivers.

He has also played a variety of roles professionally on a wide range of projects. This has included pure researching roles, architect roles and developer roles. He also was also solely responsible for debugging traps and blue screens for a number of years.

Previously of Citrix Systems he is very experienced in the area of Terminal Services. He currently works on Operating Systems and low level architecture at Intel.

He has started a youtube channel called "Checksum Error" that focuses on software.
https://www.youtube.com/channel/UCMN9q8DbU0dnllWpVRvn7Cw

Comments and Discussions

17 messages have been posted for this article Visit **https://www.codeproject.com/Articles/9636/Driver-Development-Part-3-Introduction-to-driver-c** to post and view comments on this article, or click **here** to get a print view with messages.