

# Driver Development Part 1: Introduction to Drivers



**Toby Opferman**

5 Feb 2005

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

- [Download source files - 10.4 Kb](#)

## Introduction

This tutorial will attempt to describe how to write a simple device driver for Windows NT. There are various resources and tutorials on the internet for writing device drivers, however, they are somewhat scarce as compared to writing a "hello world" GUI program for Windows. This makes the search for information on starting to write device drivers a bit harder. You may think that if there's already one tutorial, why do you need more? The answer is that more information is always better especially when you are first beginning to understand a concept. It is always good to see information from different perspectives. People write differently and describe certain pieces of information in a different light depending on how familiar they are with a certain aspect or how they think it should be explained. This being the case, I would recommend anyone who wants to write device drivers not to stop here or somewhere else. Always find a variety of samples and code snippets and research the differences. Sometimes there are bugs and things omitted. Sometimes there are things that are being done that aren't necessary, and sometimes there's information incorrect or just incomplete.

This tutorial will describe how to create a simple device driver, dynamically load and unload it, and finally talk to it from user mode.

## Creating a Simple Device Driver

### What is a subsystem?

I need to define a starting ground before we begin to explain how to write a device driver. The starting point for this article will be the compiler. The compiler and linker generate a binary in a format that the Operating System understands. In Windows, this format is "PE" for "Portable Executable" format. In this format, there is an idea called a **subsystem**. A subsystem, along with other options specified in the PE header information, describes how to load an executable which also includes the entry point into the binary.

Many people use the VC++ IDE to simply create a project with some default pre-set options for the compiler's (and linker) command line. This is why a lot of people may not be familiar with this concept even though they are most likely already using it if they have ever written Windows applications. Have you ever written a console application? Have you ever written a GUI application for Windows? These are different subsystems in Windows. Both of these will generate a PE binary with the appropriate subsystem information. This is also why a console application uses "main" where a WINDOWS application uses "WinMain". When you choose these projects, VC++ simply creates a project with /SUBSYSTEM:CONSOLE or /SUBSYSTEM:WINDOWS. If you accidentally choose the wrong project, you can simply change this in the linker options menu rather than needing to create a new project.

There's a point to all of this? A driver is simply linked using a different subsystem called "NATIVE". [MSDN Subsystem compiler options](#).

### The Driver's "main"

After the compiler is setup with the appropriate options, it's probably good to start thinking about the entry point to a driver. The first section lied a little bit about the subsystem. "NATIVE" can also be used to run user-mode applications which define an entry point called **"NtProcessStartup"**. This is the "default" type of executable that is made when specifying "NATIVE" in the same way **"WinMain"** and **"main"** are found when the linker is creating an application. You can override the default entry point with your own, simply by using the `"-entry:<functionname>"` linker option. If we know we want this to be a driver, we simply need to write an entry point whose parameter list and return type matches that of a driver. The system will then load the driver when we install it and tell the system that it is a driver.

The name we use can be anything. We can call it **BufferFly()** if we want. The most common practice used by driver developers and Microsoft is using the name **"DriverEntry"** as its initial entry point. This means we add `"-entry:DriverEntry"` to the linker's command line options. If you are using the DDK, this is done for you when you specify "DRIVER" as the type of executable to build. The DDK contains an environment that has pre-set options in the common *make* file directory which makes it simpler to create an application as it specifies the default options. The actual driver developer can then override these settings in the *make* file or simply use them as a convenience. This is essentially how **"DriverEntry"** became the somewhat "official" name for driver entry points.

Remember, DLLs actually are also compiled specifying "WINDOWS" as the subsystem, but they also have an additional switch called `/DLL`. There is a switch which can also be used for drivers: `/DRIVER:WDM` (which also sets NATIVE behind the scenes) as well as a `/DRIVER:UP` which means this driver cannot be loaded on a multi-processor system.

The linker builds the final binary, and based on what the options are in the PE header and how the binary is attempting to be loaded (run as an EXE through the loader, loaded by **LoadLibrary**, or attempting to be loaded as a driver) will define how the loading system behaves. The loading system attempts to perform some level of verification, that the image being loaded is indeed supposed to be loaded in this manner, for example. There is even, in some cases, startup code added to the binary that executes before your entry point is reached (**WinMainCRTStartup** calling **WinMain**, for example, to initialize the CRT). Your job is to simply write the application based on how you want it to be loaded and then set the correct options in the linker so it knows how to properly create the binary. There are various resources on the details of the PE format which you should be able to find if you are interested in further investigation into this area.

The options we will set for the linker will end up being the following:

```
/SUBSYSTEM:NATIVE /DRIVER:WDM -entry:DriverEntry
```

## Before creating the "DriverEntry"

There are some things we need to go over before we simply sit down and write the **"DriverEntry"**. I know that a lot of people simply want to jump right into writing the driver and seeing it work. This is generally the case in most programming scenarios as you usually just take the code, change it around, compile it, and test it out. If you remember back to when you were first learning Windows development, it was probably the same way. Your application probably didn't work right away, probably crashed, or just disappeared. This was a lot of fun and you probably learned a lot, but you know that with a driver, the adventure is a little different. Not knowing what to do can end up in blue screening the system, and if your driver is loaded on boot and executes that code, you now have a problem. Hopefully, you can boot in safe mode or restore to a previous hardware configuration. That being the case, we have a few things to go over before you write the driver in order to help educate you on what you are doing before you actually do it.

The first rule of thumb is do not just take a driver and compile it with some of your changes. If you do not understand how the driver is working or how to program correctly in the environment, you are likely to cause problems. Drivers can corrupt the integrity of the whole system, they can have bugs that don't always occur but in some rare circumstances. Application programs can have the same type of bugs in behavior but not in root cause. As an example, there are times when you cannot access memory that is pageable. If you know how Virtual Memory works, you know that the Operating System will remove pages from memory to pull in pages that are needed, and this is how more applications can run than would have been physically possible given the memory limitations of the machine. There are places, however, when pages cannot be read into memory from disk. At these times, those "drivers" who work with memory can only access memory that cannot be paged out.

Where am I going with this? Well, if you allow a driver which runs under these constraints to access memory that is "pageable", it may not crash as the Operating System usually tries to keep all pages in memory as long as possible. If you close an application that was running, it may still be in memory, for example! This is why a bug like this may go undetected (unless you try doing things like driver verifier) and eventually may trap. When it does, if you do not understand the basic concepts like this, you would be lost as to what the problem is and how to fix it.

There are a lot of concepts behind everything that will be described in this document. On IRQL alone, there is a twenty page document you can find on MSDN. There's an equally large document on IRP. I will not attempt to duplicate this information nor point out every single little detail. What I will attempt to do is give a basic summary and point you in the direction of where to find more information. It's important to at least know that these concepts exist and understand some basic idea behind them, before writing the driver.

## What is IRQL?

The IRQL is known as the "Interrupt **Re**Quest **L**evel". The processor will be executing code in a thread at a particular IRQL. The IRQL of the processor essentially helps determine how that thread is allowed to be interrupted. The thread can only be interrupted by code which needs to run at a higher IRQL on the same processor. Interrupts requiring the same IRQL or lower are masked off so only interrupts requiring a higher IRQL are available for processing. In a multi-processor system, each processor operates independently at its own IRQL.

There are four IRQL levels which you generally will be dealing with, which are "Passive", "APC", "Dispatch" and "DIRQL". Kernel APIs documented in MSDN generally have a note which specifies the IRQL level at which you need to be running in order to use the API. The higher the IRQL you go, the less APIs that are available for use. The documentation on MSDN defines what IRQL the processor will be running at when the particular entry point of the driver is called. "**DriverEntry**", for example, will be called at **PASSIVE\_LEVEL**.

### PASSIVE\_LEVEL

This is the lowest IRQL. No interrupts are masked off and this is the level in which a thread executing in user mode is running. Pagable memory is accessible.

### APC\_LEVEL

In a processor running at this level, only APC level interrupts are masked. This is the level in which [Asynchronous Procedure Calls](#) occur. Pagable memory is still accessible. When an APC occurs, the processor is raised to APC level. This, in turn, also disables other APCs from occurring. A driver can manually raise its IRQL to APC (or any other level) in order to perform some synchronization with APCs, for example, since APCs can't be invoked if you are already at APC level. There are some APIs which can't be called at APC level due to the fact that APCs are disabled, which, in turn, may disable some I/O Completion APCs.

### DISPATCH\_LEVEL

The processor running at this level has DPC level interrupts and lower masked off. Pagable memory **cannot** be accessed, so all memory being accessed must be non-paged. If you are running at Dispatch Level, the APIs that you can use greatly decrease since you can only deal with non-paged memory.

### <A name=OLE\_LINK4>DIRQL </A>(Device IRQL)

Generally, higher level drivers do not deal with IRQLs at this level, but all interrupts at this level or less are masked off and do not occur. This is actually a range of IRQLs, and this is a method to determine which devices have priority over other devices.

In this driver, we will basically only be working at **PASSIVE\_LEVEL**, so we won't have to worry about the gotchas. However, it is necessary for you to be aware of what IRQL is, if you intend to continue writing device drivers.

For more information on IRQLs and thread scheduling, refer to the following [documentation](#), and another good source of information is [here](#).

## What is an IRP?

The "IRP" is called the "I/O Request Packet", and it is passed down from driver to driver in the driver stack. This is a data structure that allows drivers to communicate with each other and to request work to be done by the driver. The I/O manager or another driver may create an IRP and pass it down to your driver. The IRP includes information about the operation that is being requested.

A description of the IRP data structure can be found [here](#).

The description and usage of an IRP can go from simple to complex very easily, so we will only be describing, in general, what an IRP will mean to you. There is an article on MSDN which describes in a lot more detail (about twenty pages) of what exactly an IRP is and how to handle them. That article can be found [here](#).

The IRP will also contain a list of “sub-requests” also known as the “IRP Stack Location”. Each driver in the device stack will generally have its own “sub request” of how to interpret the IRP. This data structure is the “[IO\\_STACK\\_LOCATION](#)” and is described on MSDN.

To create an analogy of the IRP and [IO\\_STACK\\_LOCATION](#), perhaps you have three people who do different jobs such as carpentry, plumbing and welding. If they were going to build a house, they could have a common overall design and perhaps a common set of tools like their tool box. This includes things like power drills, etc. All of these common tools and overall design of building a house would be the IRP. Each of them has an individual piece they need to work on to make this happen, for example, the plumber needs the plans on where to put the pipe, how much pipe he has, etc. These could be interpreted as the [IO\\_STACK\\_LOCATION](#) as his specific job is to do the piping. The carpenter could be building the framework for the house and the details of that would be in his [IO\\_STACK\\_LOCATION](#). So, while the entire IRP is a request to build a house, each person in the stack of people has their own job as defined by the [IO\\_STACK\\_LOCATION](#) to make this happen. Once everyone has completed their job, they then complete the IRP.

The device driver we will be building will not be that complex and will basically be the only driver in the stack.

## Things to Avoid

There are a lot of pitfalls that you will need to avoid but they are mostly unrelated to our simple driver. To be more informed, however, here is a list of items called “[things to avoid](#)” when it comes to driver development.

## Create the DriverEntry routine

There is so much to explain, however, I think it’s time we simply started to develop the driver and explain as we go. It is hard to digest theory or even how code is supposed to work, without actually doing anything. You need some hands on experience so you can bring these ideas out of space and into reality.

The prototype for the [DriverEntry](#) is the following.

```
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegistryPath);
```

The [DRIVER\\_OBJECT](#) is a data structure used to represent this driver. The [DriverEntry](#) routine will use it to populate it with other entry points to the driver for handling specific I/O requests. This object also has a pointer to a [DEVICE\\_OBJECT](#) which is a data structure which represents a particular device. A single driver may actually advertise itself as handling multiple devices, and as such, the [DRIVER\\_OBJECT](#) maintains a linked list pointer to all the devices this particular driver services request for. We will simply be creating one device.

The “Registry Path” is a string which points to the location in the registry where the information for the driver was stored. The driver can use this location to store driver specific information.

The next part is to actually put things in the [DriverEntry](#) routine. The first thing we will do is create the device. You may be wondering how we are going to create a device and what type of device we should create. This is generally because a driver is usually associated with hardware but this is not the case. There are a variety of different types of drivers which operate at different levels, not all drivers work or interface directly with hardware. Generally, you maintain a stack of drivers each with a specific job to do. The highest level driver is the one that communicates with user mode, and the lowest level drivers generally just talk to other drivers and hardware. There are network drivers, display drivers, file system drivers, etc., and each has their own stack of drivers. Each place in the stack breaks up a request into a more generic or simpler request for the lower level driver to service. The highest level drivers are the ones which communicate themselves to user mode, and unless they are a special device with a particular framework (like display drivers), they can behave generally the same as other drivers just as they implement different types of operations.

As an example, take the hard disk drive. The driver which communicates to user mode does not talk directly to hardware. The high level driver simply manages the file system itself and where to put things. It then communicates where it wants to read or write from the disk to the lower level driver which may or may not talk directly to hardware. There may be another layer which then communicates that request to the actual hardware driver which then physically reads or writes a particular sector off a disk and then returns it to the higher level. The highest level may interpret them as file data, but the lowest level driver may simply be stupid and only manage requests as far as when to read a sector based off where the read/write head is located on the disk. It could then determine what sector read requests to service, however, it has no idea what the data is and does not interpret it.

Let’s take a look at the first part of our “[DriverEntry](#)”.

```

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegistryPath)
{
    NTSTATUS NtStatus = STATUS_SUCCESS;
    UINT uiIndex = 0;
    PDEVICE_OBJECT pDeviceObject = NULL;
    UNICODE_STRING usDriverName, usDosDeviceName;

    DbgPrint("DriverEntry Called \r\n");

    RtlInitUnicodeString(&usDriverName, L"\\Device\\Example");
    RtlInitUnicodeString(&usDosDeviceName, L"\\DosDevices\\Example");

    NtStatus = IoCreateDevice(pDriverObject, 0,
                            &usDriverName,
                            FILE_DEVICE_UNKNOWN,
                            FILE_DEVICE_SECURE_OPEN,
                            FALSE, &pDeviceObject);
}

```

The first thing you will notice is the **DbgPrint** function. This works just like "**printf**" and it prints messages out to the debugger or debug output window. You can get a tool called "DBGVIEW" from [www.sysinternals.com](http://www.sysinternals.com) and all of the information in those messages will be displayed.

You will then notice that we use a function called "**RtlInitUnicodeString**" which basically initializes a **UNICODE\_STRING** data structure. This data structure contains basically three entries. The first is the size of the current Unicode string, the second is the maximum size that the Unicode string can be, and the third is a pointer to the Unicode string. This is used to describe a Unicode string and used commonly in drivers. The one thing to remember with **UNICODE\_STRING** is that they are **not** required to be **NULL** terminated since there is a size parameter in the structure! This causes problems for people new to driver development as they assume a **UNICODE\_STRING** is **NULL** terminated, and they blue-screen the driver. Most Unicode strings passing into your driver will not be **NULL** terminated, so this is something you need to be aware of.

Devices have names just like anything else. They are generally named `\\Device\\<somename>` and this is the string we were creating to pass into **IoCreateDevice**. The second string, "`\\DosDevices\\Example`", we will get into later as it's not used in the driver yet. To the **IoCreateDevice**, we pass in the driver object, a pointer to the Unicode string we want to call the driver, and we pass in a type of driver "**UNKNOWN**" as it's not associated with any particular type of device, and we also pass in a pointer to receive the newly created device object. The parameters are explained in more detail at "[IoCreateDevice](#)".

The second parameter we passed 0, and it says to specify the number of bytes to create for the device extension. This is basically a data structure that the driver writer can define which is unique to that device. This is how you can extend the information being passed into a device and create device contexts, etc. in which to store instance data. We will not be using this for this example.

Now that we have successfully created our `\\Device\\Example` device driver, we need to setup the Driver Object to call into our driver when certain requests are made. These requests are called IRP Major requests. There are also Minor requests which are sub-requests of these and can be found in the stack location of the IRP.

The following code populates certain requests:

```

for(uiIndex = 0; uiIndex < IRP_MJ_MAXIMUM_FUNCTION; uiIndex++)
    pDriverObject->MajorFunction[uiIndex] = Example_UnsupportedFunction;

pDriverObject->MajorFunction[IRP_MJ_CLOSE] = Example_Close;
pDriverObject->MajorFunction[IRP_MJ_CREATE] = Example_Create;
pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Example_IoControl;
pDriverObject->MajorFunction[IRP_MJ_READ] = Example_Read;
pDriverObject->MajorFunction[IRP_MJ_WRITE] = USE_WRITE_FUNCTION;

```

We populate the **Create**, **Close**, **IoControl**, **Read** and **Write**. What do these refer to? When communicating with the user-mode application, certain APIs call directly to the driver and pass in parameters!

- `CreateFile` -> `IRP_MJ_CREATE`
- `CloseHandle` -> `IRP_MJ_CLEANUP` & `IRP_MJ_CLOSE`
- `WriteFile` -> `IRP_MJ_WRITE`
- `ReadFile` -> `IRP_MJ_READ`
- `DeviceIoControl` -> `IRP_MJ_DEVICE_CONTROL`

To explain, one difference is `IRP_MJ_CLOSE` is not called in the context of the process which created the handle. If you need to perform process related clean up, then you need to handle `IRP_MJ_CLEANUP` as well.

So as you can see, when a user mode application uses these functions, it calls into your driver. You may be wondering why the user mode API says "file" when it doesn't really mean "file". That is true, these APIs can talk to any device which exposes itself to user mode, they are not only for accessing files. In the last piece of this article, we will be writing a user mode application to talk to our driver and it will simply do `CreateFile`, `WriteFile`, `CloseHandle`. That's how simple it is. `USE_WRITE_FUNCTION` is a constant I will explain later.

The next piece of code is pretty simple, it's the driver unload function.

```
pDriverObject->DriverUnload = Example_Unload;
```

You can technically omit this function but if you want to unload your driver dynamically, then it must be specified. If you do not specify this function once your driver is loaded, the system will not allow it to be unloaded.

The code after this is actually using the `DEVICE_OBJECT`, not the `DRIVER_OBJECT`. These two data structures may get a little confusing since they both start with "D" and end with "\_OBJECT", so it's easy to confuse which one we're using.

```
pDeviceObject->Flags |= IO_TYPE;
pDeviceObject->Flags &= (~DO_DEVICE_INITIALIZING);
```

We are simply setting the flags. `"IO_TYPE"` is actually a constant which defines the type of I/O we want to do (I defined it in *example.h*). I will explain this in the section on handling user-mode write requests.

The `"DO_DEVICE_INITIALIZING"` tells the I/O Manager that the device is being initialized and not to send any I/O requests to the driver. For devices created in the context of the `"DriverEntry"`, this is not needed since the I/O Manager will clear this flag once the `"DriverEntry"` is done. However, if you create a device in any function outside of the `DriverEntry`, you need to manually clear this flag for any device you create with `IoCreateDevice`. This flag is actually set by the `IoCreateDevice` function. We cleared it here just for fun even though we weren't required to.

The last piece of our driver is using both of the Unicode strings we defined above. `"\Device\Example"` and `"\DosDevices\Example"`.

```
IoCreateSymbolicLink(&usDosDeviceName, &usDriverName);
```

`"IoCreateSymbolicLink"` does just that, it creates a "Symbolic Link" in the object manager. To view the object manager, you may download my tool `"QuickView"`, or go to [www.sysinternals.com](http://www.sysinternals.com) and download `"WINOBJ"`. A Symbolic Link simply maps a "DOS Device Name" to an "NT Device Name". In this example, `"Example"` is our DOS Device Name and `"\Device\Example"` is our NT Device Name.

To put this into perspective, different vendors have different drivers and each driver is required to have its own name. You cannot have two drivers with the same NT Device name. Say, you have a memory stick which can display itself to the system as a new drive letter which is any available drive letter such as E:. If you remove this memory stick and say you map a network drive to E:. Application can talk to E: the same way, they do not care if E: is a CD ROM, Floppy Disk, memory stick or network drive. How is this possible? Well, the driver needs to be able to interpret the requests and either handle them within themselves such as the case of a network redirector or pass them down to the appropriate hardware driver. This is done through symbolic links. E: is a symbolic link. The network mapped drive may map E: to `\Device\NetworkRedirector` and the memory stick may map E: to `\Device\FujiMemoryStick`, for example.

This is how applications can be written using a commonly defined name which can be abstracted to point to any device driver which would be able to handle requests. There are no rules here, we could actually map `\Device\Example` to E:. We can do whatever we wish to



do, but in the end, however, the application attempts to use the device as how the device driver needs to respond and act. This means supporting IOCTLs commonly used by those devices as applications will try to use them. COM1, COM2, etc. are all examples of this. COM1 is a DOS name which is mapped to an NT Device name of a driver which handles serial requests. This doesn't even need to be a real physical serial port!

So we have defined "Example" as a DOS Device which points to "\Device\Example". In the "communicating with usermode" portion, we will learn more about how to use this mapping.

## Create the Unload Routine

The next piece of code we will look at is the unload routine. This is required in order to be able to unload the device driver dynamically. This section will be a bit smaller as there is not much to explain.

```
VOID Example_Unload(PDRIVER_OBJECT DriverObject)
{
    UNICODE_STRING usDosDeviceName;

    DbgPrint("Example_Unload Called \r\n");

    RtlInitUnicodeString(&usDosDeviceName, L"\\DosDevices\\Example");
    IoDeleteSymbolicLink(&usDosDeviceName);

    IoDeleteDevice(DriverObject->DeviceObject);
}
```

You can do whatever you wish in your unload routine. This unload routine is very simple, it just deletes the symbolic link we created and then deletes the only device that we created which was \Device\Example.

## Creating the IRP\_MJ\_WRITE

The rest of the functions should be self explanatory as they don't do anything. This is why I am only choosing to explain the "Write" routine. If this article is liked, I may write a second tutorial on implementing the IO Control function.

If you have used **WriteFile** and **ReadFile**, you know that you simply pass a buffer of data to write data to a device or read data from a device. These parameters are sent to the device in the IRP as we explained previously. There is more to the story though as there are actually three different methods that the I/O Manager will use to marshal this data before giving the IRP to the driver. That also means that how the data is marshaled is how the driver's Read and Write functions need to interpret the data.

The three methods are "Direct I/O", "Buffered I/O" and "Neither".

```
#ifdef __USE_DIRECT__
#define IO_TYPE DO_DIRECT_IO
#define USE_WRITE_FUNCTION Example_WriteDirectIO
#endif

#ifdef __USE_BUFFERED__
#define IO_TYPE DO_BUFFERED_IO
#define USE_WRITE_FUNCTION Example_WriteBufferedIO
#endif

#ifndef IO_TYPE
#define IO_TYPE 0
#define USE_WRITE_FUNCTION Example_WriteNeither
#endif
```

The code was written so if you define "**\_\_USE\_DIRECT\_\_**" in the header, then **IO\_TYPE** is now **DO\_DIRECT\_IO** and **USE\_WRITE\_FUNCTION** is now **Example\_WriteDirectIO**. If you define "**\_\_USE\_BUFFERED\_\_**" in the header, then **IO\_TYPE** is now **DO\_BUFFERED\_IO** and **USE\_WRITE\_FUNCTION** is now **Example\_WriteBufferedIO**. If you don't define

`__USE_DIRECT__` or `__USE_BUFFERED__`, then `IO_TYPE` is defined as 0 (neither) and the write function is `Example_WriteNeither`.

We will now go over each type of I/O.

## Direct I/O

The first thing I will do is simply show you the code for handling direct I/O.

```
NTSTATUS Example_WriteDirectIO(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
    NTSTATUS NtStatus = STATUS_SUCCESS;
    PIO_STACK_LOCATION pIoStackIrp = NULL;
    PCHAR pWriteDataBuffer;

    DbgPrint("Example_WriteDirectIO Called \r\n");

    /*
     * Each time the IRP is passed down
     * the driver stack a new stack location is added
     * specifying certain parameters for the IRP to the driver.
     */
    pIoStackIrp = IoGetCurrentIrpStackLocation(Irp);

    if(pIoStackIrp)
    {
        pWriteDataBuffer =
            MmGetSystemAddressForMdlSafe(Irp->MdlAddress, NormalPagePriority);

        if(pWriteDataBuffer)
        {
            /*
             * We need to verify that the string
             * is NULL terminated. Bad things can happen
             * if we access memory not valid while in the Kernel.
             */
            if(Example_IsStringTerminated(pWriteDataBuffer,
                pIoStackIrp->Parameters.Write.Length))
            {
                DbgPrint(pWriteDataBuffer);
            }
        }
    }

    return NtStatus;
}
```

The entry point simply provides the device object for the device for which this request is being sent for. If you recall, a single driver can create multiple devices even though we have only created one. The other parameter is as was mentioned before which is an IRP!

The first thing we do is call `"IoGetCurrentIrpStackLocation"`, and this simply provides us with our `IO_STACK_LOCATION`. In our example, the only parameter we need from this is the length of the buffer provided to the driver, which is at `Parameters.Write.Length`.

The way buffered I/O works is that it provides you with a `"MdlAddress"` which is a "Memory Descriptor List". This is a description of the user mode addresses and how they map to physical addresses. The function we call then is `"MmGetSystemAddressForMdlSafe"` and we use the `Irp->MdlAddress` to do this. This operation will then give us a system virtual address which we can then use to read the memory.

The reasoning behind this is that some drivers do not always process a user mode request in the context of the thread or even the process in which it was issued. If you process a request in a different thread which is running in another process context, you would not



be able to read user mode memory across process boundaries. You should know this already, as you run two applications they can't just read/write to each other without Operating System support.

So, this simply maps the physical pages used by the user mode process into system memory. We can then use the returned address to access the buffer passed down from user mode.

This method is generally used for larger buffers since it does not require memory to be copied. The user mode buffers are locked in memory until the IRP is completed which is the downside of using direct I/O. This is the only downfall and is why it's generally more useful for larger buffers.

## Buffered I/O

The first thing I will do is simply show you the code for handling buffered I/O.

```
NTSTATUS Example_WriteBufferedIO(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
    NTSTATUS NtStatus = STATUS_SUCCESS;
    PIO_STACK_LOCATION pIoStackIrp = NULL;
    PCHAR pWriteDataBuffer;

    DbgPrint("Example_WriteBufferedIO Called \r\n");

    /*
     * Each time the IRP is passed down
     * the driver stack a new stack location is added
     * specifying certain parameters for the IRP to the driver.
     */
    pIoStackIrp = IoGetCurrentIrpStackLocation(Irp);

    if(pIoStackIrp)
    {
        pWriteDataBuffer = (PCHAR)Irp->AssociatedIrp.SystemBuffer;

        if(pWriteDataBuffer)
        {
            /*
             * We need to verify that the string
             * is NULL terminated. Bad things can happen
             * if we access memory not valid while in the Kernel.
             */
            if(Example_IsStringTerminated(pWriteDataBuffer,
                pIoStackIrp->Parameters.Write.Length))
            {
                DbgPrint(pWriteDataBuffer);
            }
        }
    }

    return NtStatus;
}
```

As mentioned above, the idea is to pass data down to the driver that can be accessed from any context such as another thread in another process. The other reason would be to map the memory to be non-paged so the driver can also read it at raised IRQL levels.

The reason you may need to access memory outside the current process context is that some drivers create threads in the SYSTEM process. They then defer work to this process either asynchronously or synchronously. A driver at a higher level than your driver may do this or your driver itself may do it.

The downfall of using "Buffered I/O" is that it allocates non-paged memory and performs a copy. This is now overhead in processing every read and write into the driver. This is one of the reasons this is best used on smaller buffers. The whole user mode page doesn't need to be locked in memory as with Direct I/O, which is the plus side of this. The other problem with using this for larger buffers is that since it allocates non-paged memory, it would need to allocate a large block of sequential non-paged memory.

## Neither Buffered nor Direct

The first thing I will do is show you the code for handling neither Buffered nor Direct I/O.

```
NTSTATUS Example_WriteNeither(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
    NTSTATUS NtStatus = STATUS_SUCCESS;
    PIO_STACK_LOCATION pIoStackIrp = NULL;
    PCHAR pWriteDataBuffer;

    DbgPrint("Example_WriteNeither Called \r\n");

    /*
     * Each time the IRP is passed down
     * the driver stack a new stack location is added
     * specifying certain parameters for the IRP to the driver.
     */
    pIoStackIrp = IoGetCurrentIrpStackLocation(Irp);

    if(pIoStackIrp)
    {
        /*
         * We need this in an exception handler or else we could trap.
         */
        __try {

            ProbeForRead(Irp->UserBuffer,
                pIoStackIrp->Parameters.Write.Length,
                TYPE_ALIGNMENT(char));
            pWriteDataBuffer = Irp->UserBuffer;

            if(pWriteDataBuffer)
            {
                /*
                 * We need to verify that the string
                 * is NULL terminated. Bad things can happen
                 * if we access memory not valid while in the Kernel.
                 */
                if(Example_IsStringTerminated(pWriteDataBuffer,
                    pIoStackIrp->Parameters.Write.Length))
                {
                    DbgPrint(pWriteDataBuffer);
                }
            }
        } __except( EXCEPTION_EXECUTE_HANDLER ) {

            NtStatus = GetExceptionCode();
        }
    }

    return NtStatus;
}
```

In this method, the driver accesses the user mode address directly. The I/O manager does not copy the data, it does not lock the user mode pages in memory, it simply gives the driver the user mode address buffer.

The upside of this is that no data is copied, no memory is allocated, and no pages are locked into memory. The downside of this is that you must process this request in the context of the calling thread so you will be able to access the user mode address space of the correct process. The other downside of this is that the process itself can attempt to change access to the pages, free the memory, etc., on another thread. This is why you generally want to use "**ProbeForRead**" and "**ProbeForWrite**" functions and surround all the

code in an exception handler. There's no guarantee that at any time the pages could be invalid, you can simply attempt to make sure they are, before you attempt to read or write. This buffer is stored at **Irp->UserBuffer**.

### What's this #pragma stuff?

These directives you see simply let the linker know what segment to put the code and what options to set on the pages. The **"DriverEntry"**, for example, is set as **"INIT"** which is a discardable page. This is because you only need that function during initialization.

### Homework!

Your homework is to create the Read routines for each type of I/O processing. You can use the Write routines as reference to figure out what you need to do.

## Dynamically Loading and Unloading the Driver

A lot of tutorials will go and explain the registry, however, I have chosen not to at this time. There is a simple user mode API that you can use to load and unload the driver without having to do anything else. This is what we will use for now.

```
int _cdecl main(void)
{
    HANDLE hSCManager;
    HANDLE hService;
    SERVICE_STATUS ss;

    hSCManager = OpenSCManager(NULL, NULL, SC_MANAGER_CREATE_SERVICE);

    printf("Load Driver\n");

    if(hSCManager)
    {
        printf("Create Service\n");

        hService = CreateService(hSCManager, "Example",
                                "Example Driver",
                                SERVICE_START | DELETE | SERVICE_STOP,
                                SERVICE_KERNEL_DRIVER,
                                SERVICE_DEMAND_START,
                                SERVICE_ERROR_IGNORE,
                                "C:\\example.sys",
                                NULL, NULL, NULL, NULL, NULL);

        if(!hService)
        {
            hService = OpenService(hSCManager, "Example",
                                   SERVICE_START | DELETE | SERVICE_STOP);
        }

        if(hService)
        {
            printf("Start Service\n");

            StartService(hService, 0, NULL);
            printf("Press Enter to close service\n");
            getchar();
            ControlService(hService, SERVICE_CONTROL_STOP, &ss);

            DeleteService(hService);

            CloseServiceHandle(hService);
        }
    }
}
```

```

    }

    CloseServiceHandle(hSCManager);
}

return 0;
}

```

This code will load the driver and start it. We load the driver with "**SERVICE\_DEMAND\_START**" which means this driver must be physically started. It will not start automatically on boot, that way we can test it, and if we blue-screen, we can fix the issue without having to boot to safe mode.

This program will simply pause. You can then run the application that talks to the service, in another window. The code above should be pretty easy to understand that you need to copy the driver to *C:\example.sys* in order to use it. If the service fails to create, it knows it has already been created and opens it. We then start the service and pause. Once you press Enter, we stop the service, delete it from the list of services, and exit. This is very simple code and you can modify it to serve your purposes.

## Communicating to the Device Driver

The following is the code that communicates to the driver.

```

int _cdecl main(void)
{
    HANDLE hFile;
    DWORD dwReturn;

    hFile = CreateFile("\\\\.\\Example",
        GENERIC_READ | GENERIC_WRITE, 0, NULL,
        OPEN_EXISTING, 0, NULL);

    if(hFile)
    {
        WriteFile(hFile, "Hello from user mode!",
            sizeof("Hello from user mode!"), &dwReturn, NULL);
        CloseHandle(hFile);
    }

    return 0;
}

```

This is probably simpler than you thought. If you compile the driver three times using the three different methods of I/O, the message sent down from user mode should be printed in DBGVIEW. As you notice, you simply need to open the DOS Device Name using `\\. \DosName`. You could even open `\\Device\Nt Device Name` using the same method. You will then create a handle to the device and you can call **WriteFile**, **ReadFile**, **CloseHandle**, **DeviceIoControl**! If you want to experiment, simply perform actions and use `DbgPrint` to show what code is being executed in your driver.

## Conclusion

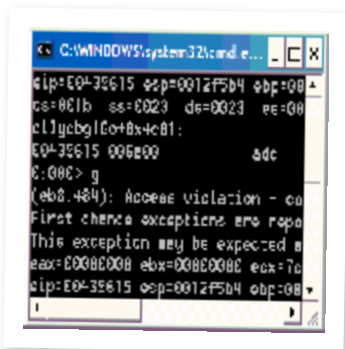
This article showed a simple example of how to create a driver, install it, and access it via a simple user mode application. You may use the associated source files to change and experiment. If you wish to write drivers, it's best to read up on many of the basic concepts of drivers, especially, some of the ones linked to in this tutorial.

## License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found [here](#)

## About the Author



### 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/UCMN9q8DbU0dnIIWpVRvn7CW>

## Comments and Discussions

 **440 messages** have been posted for this article Visit <https://www.codeproject.com/Articles/9504/Driver-Development-Part-1-Introduction-to-Drivers> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#)

[Advertise](#)

[Privacy](#)

[Cookies](#)

[Terms of Use](#)

Article Copyright 2005 by Toby Opferman  
Everything else Copyright © [CodeProject](#), 1999-2021

Web05 2.8.20210412.1