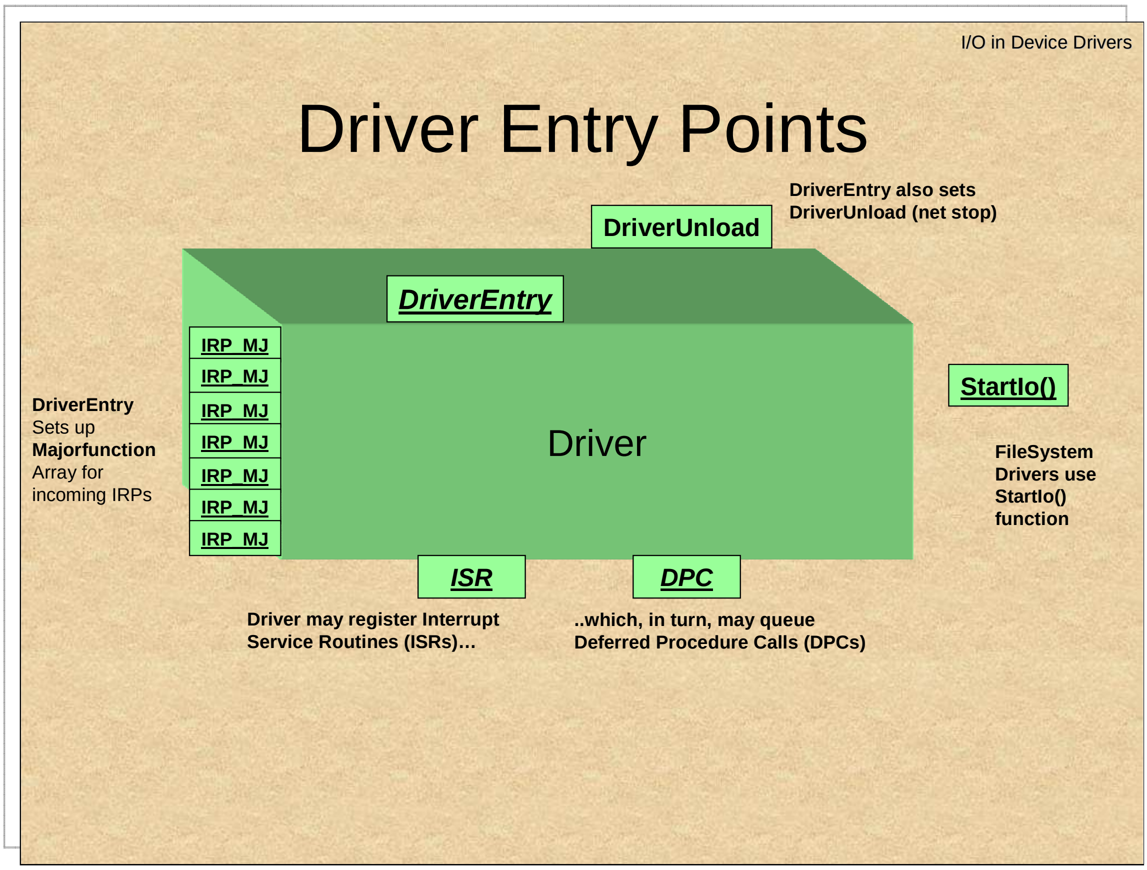
# I/O in Device Drivers

Creating a Driver that actually DOES something

This section extends the sample driver by introducing I/O processing via IRPs, and interacting with the driver from user mode and from other drivers.

**Key Concepts**:IRP, IRP Dispatching, Buffered I/O, Direct I/O, IoControlCodes (IOCTLs)




The Kernel defines two callback interfaces for drivers:

- Fast I/O
  - Rapid synchronous I/O only, mostly for File System Drivers
  - Direct from user buffers to system cache (less copying)
- I/O Request Packets
  - Default I/O for most operations:
    - Both synchronous and asynchronous I/O
    - Page faults implemented by IRPs to file system
    - Networking – send/recv implemented as IRPs

Driver may define additional entry points/callbacks. Fast I/O is used primarily for File System Drivers (FSDs), and is left out of the scope of this course.

I/O in Device Drivers

# IRPs

- I/O operations are put into “I/O Request Packets”
- IRPs pass up and down the driver stack
- Every driver owns an “IO_STACK_LOCATION” in IRP
- Top level (creator of IRP) must set up IRP “stack size”
- Structure documented, but remains semi-opaque
  - Structs of Unions of Structs – ***very*** volatile

A fundamental concept in the Windows I/O architecture is that of an **I/O Request Packet**, or **IRP**.

# IRPs - I/O Request Packets

- IRP_MJ: "Major" Requests
- IRP_MN: "Minor" (sub) Requests (e.g. for IRP_MJ_PNP)

- Common Major request types:

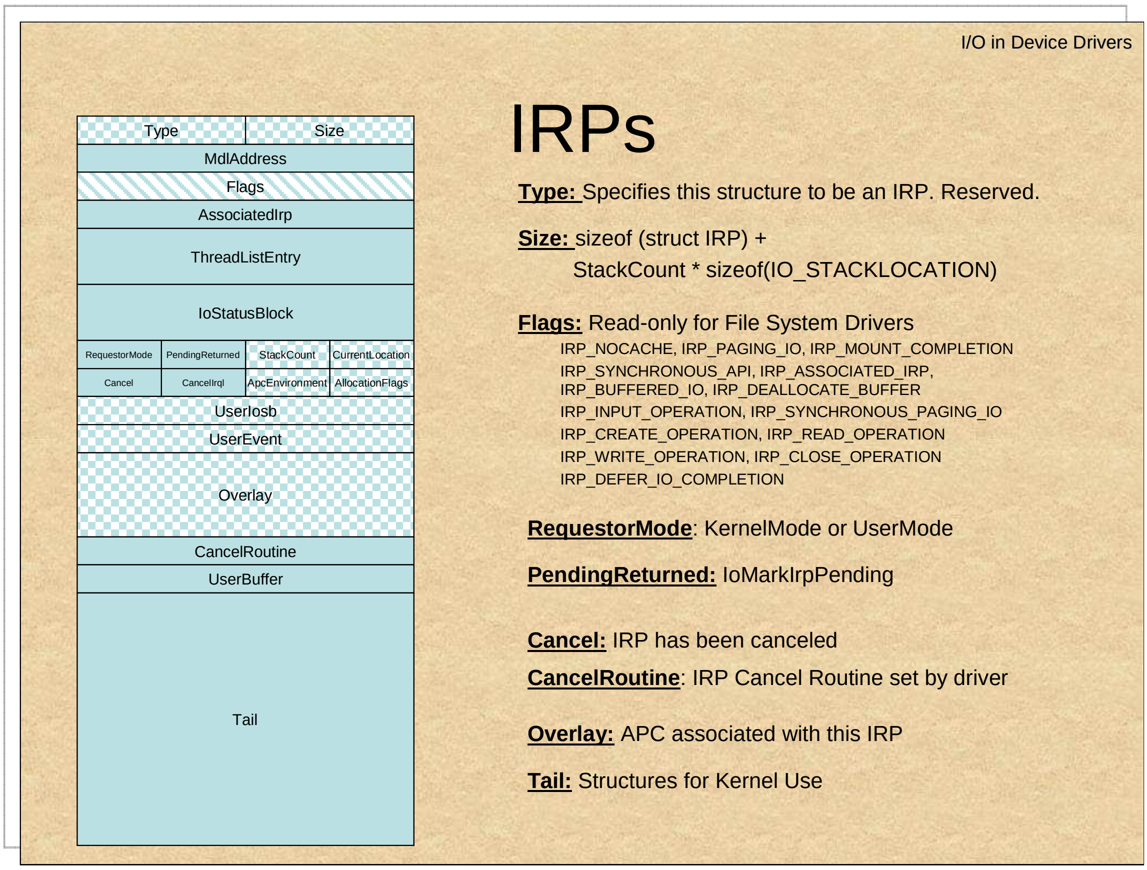
| IRP_MJ_ | Use |
|---|---|
| CREATE | File/Socket/Dir creation open |
| CLOSE | File/Socket/Dir close |
| DEVICE_CONTROL | Ioctl/DeviceIoControl |
| FILESYSTEM_CONTROL | Various FSD operations |
| READ | Read operation |
| QUERY_INFORMATION | Get information on descriptor |
| SET_INFORMATION | Set information of descriptor |
| WRITE | Write operation |

Kernel drivers (with the exception of NDIS and FSD) generally communicate through I/O Request Packets. These "packets" are semi opaque objects.

The Kernel defines IRP_MJ_ types, corresponding to "Major" codes, and IRP_MN_ types, corresponding to "Minor" codes.

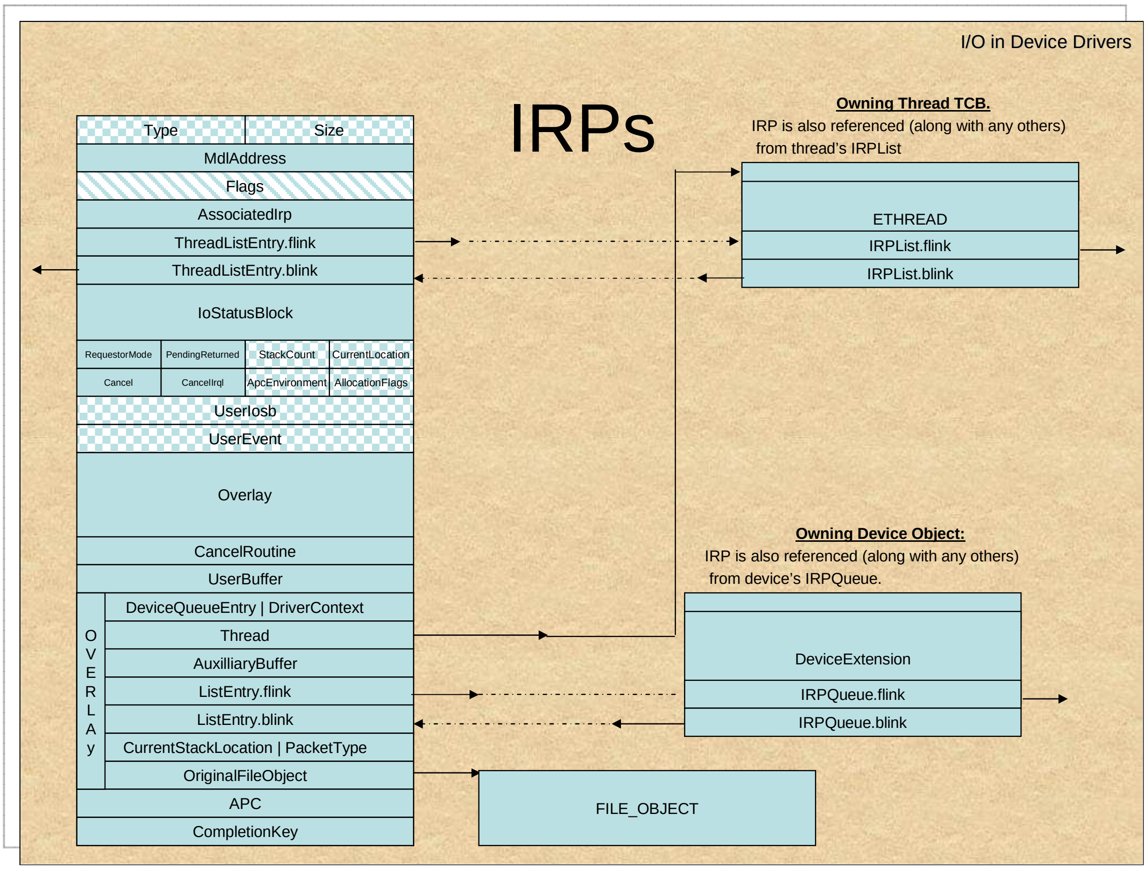
The Major codes are for the various request operations, the important ones of which are shown above. The Minor codes are sub codes for a particular Major – for example, Plug and Play operations all have the same Major code, IRP_MJ_PNP, but specific minor codes for starting/stopping devices, etc.

```
typedef struct _IRP {
  …
  PMDL  MdlAddress;
  ULONG  Flags;
  union {
    …
    PVOID  SystemBuffer;
  } AssociatedIrp;
  …
  IO_STATUS_BLOCK  IoStatus;
  KPROCESSOR_MODE  RequestorMode;
   …
  BOOLEAN  Cancel;   // The cancel bit
  …
  PDRIVER_CANCEL  CancelRoutine;
  PVOID UserBuffer;
  union {
    struct { ..
     union {
      KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
      struct {
        PVOID  DriverContext[4];
      };
    };
    …
    PETHREAD  Thread;
    LIST_ENTRY  ListEntry;
    ..    } Overlay;
    } Tail;
} IRP, *PIRP;
```




The illustrations on the next three pages show the IRP structure. Notice how it is aligned in memory for fast access.

It should be noted that most of these fields are NOT to be handled directly, and there exist functions and macros for that.




Lastly, the IRP "Tail" is a union containing all the fields that the IRP cannot hope to align. Most of these fields are in an "overlay" struct, and they link the IRP to its corresponding device and thread.

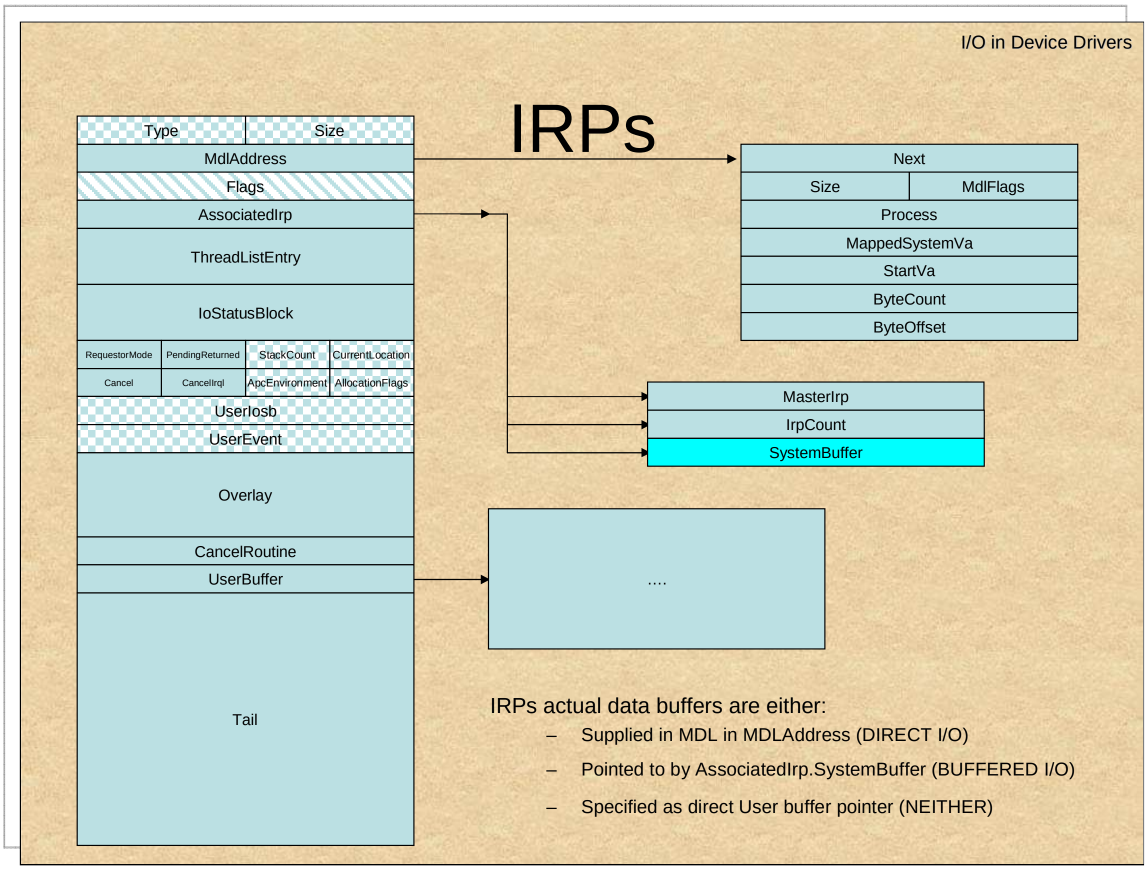
```
union {
    struct {
        union {
            KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
            struct {            PVOID DriverContext[4]; } ;
        } ;

        PETHREAD Thread;
        PCHAR AuxiliaryBuffer;

        struct {
            LIST_ENTRY ListEntry;
            union {
                struct _IO_STACK_LOCATION *CurrentStackLocation;
                ULONG PacketType;
            };
        };
        PFILE_OBJECT OriginalFileObject;
    } Overlay;

    KAPC Apc;
    PVOID CompletionKey;

} Tail;
```



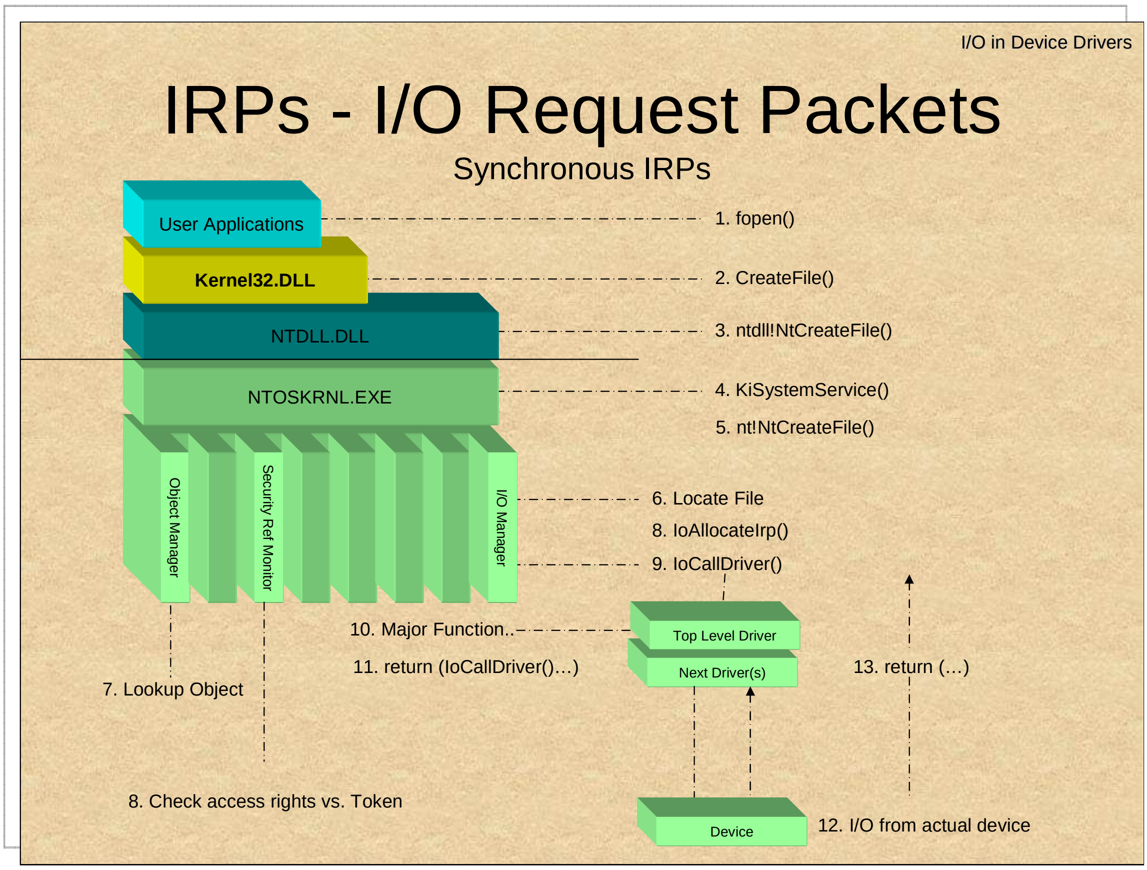

IRPs relate to I/O requests, and therefore point to data buffers. It's not that simple, however, as there are three modes of access to data buffers:

DIRECT I/O: In which the IRP contains a pointer to an MDL (in the MdlAddress field, as shown above). This MDL contains the virtual pages associated with the IRP, and it is the device driver's responsibility to lock these pages in memory.

BUFFERED I/O: In which the IRP contains a pointer to locked in memory pages – the I/O manager takes care of all the lock operations, etc. However, this involves buffering and therefore an extra copy operation. So, while it is easier to handle, it is also more expensive performance-wise.

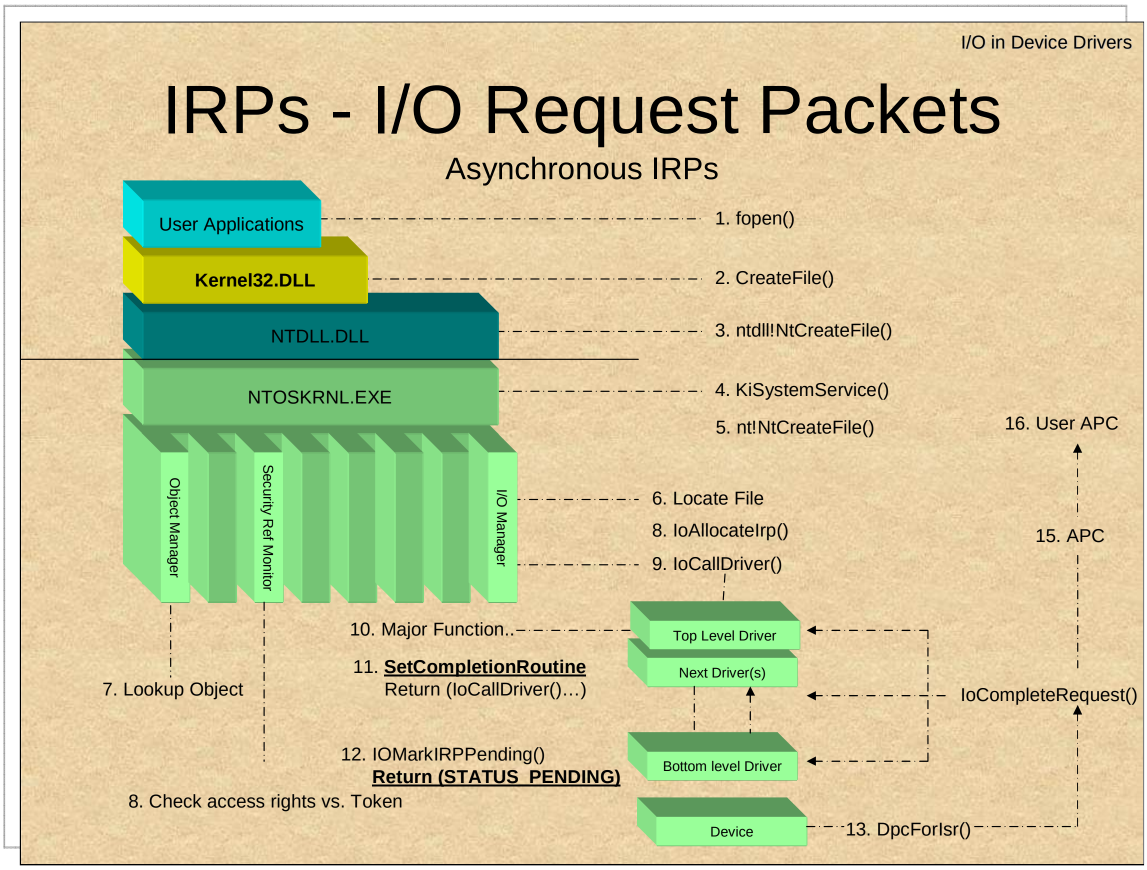
NEITHER: Used in I/O Control codes (IOCTLs, described later), this mode simply passes the user buffer address to the driver. The driver needs to prepare and handle the MDL.

The Device object specifies the preferred mode of operation in its Flags (see page 80). IOCTLs, however, may use any of the modes.




After getting familiar with the IRP structure, we can next look at the typical flow of an I/O request – from inception (usually, in user mode) down to the device.

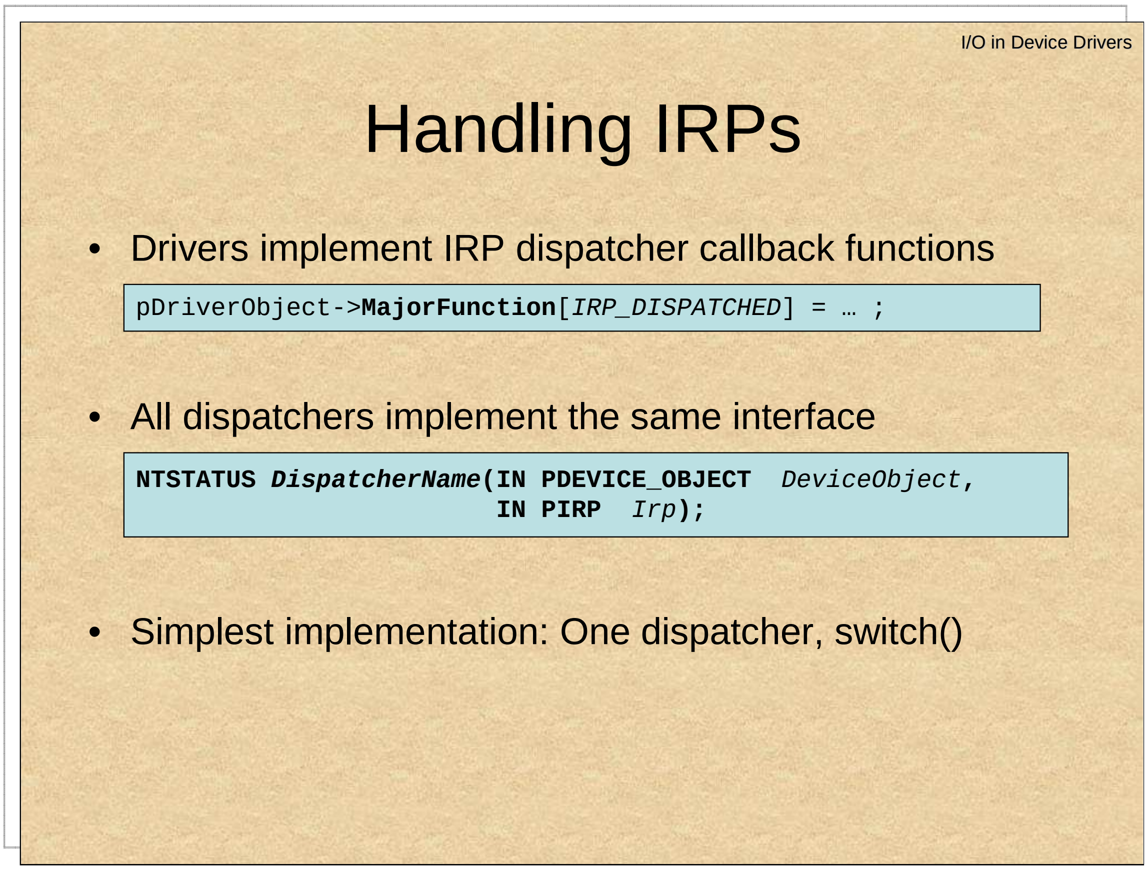
This slide depicts the process of such a typical I/O request – in this case a file “fopen()” from the C standard library. This is a synchronous request, meaning the process blocks until the I/O returns.




Processing of Asynchronous IRPs is similar all the way up to the first driver called by the I/O manager. Drivers determining a request to be potentially asynchronous still push the IRP down the stack, but may each opt to set an IRP Completion Routine of their own.

When the bottom driver processes the request, it usually submits a request to the hardware device, IoMarkIrpPending() and returns a STATUS_PENDING. This bubbles up the driver chain back to the I/O manager.

The IRP is completed in a truly asynchronous manner when some other entity chooses to call **IoCompleteRequest()** on it. This function is usually called from the device driver who set the interrupt handler (ISR) on the device. **IoCompleteRequest**() then calls the completion routines of the IRP, in reverse order. Finally, it signals the I/O manager that the IRP is indeed complete, which in turn schedules an Asynchronous Procedure Call (APC), and completes any user APCs that may have been scheduled, as well.




The DRIVER_OBJECT struct has an array of **MajorFunction**s, with indices corresponding to all the major IRP types shown so far – that is, the IRP_MJ constants, which are actually implemented as an enum.

To handle IRPs, we would have our driver main function look something like this:

```
// Prototype – We'll implement this later..
NTSTATUS driverIRPDispatcher(PDEVICE_OBJECT DeviceObject, PIRP Irp);

// Our DriverEntry function:
NTSTATUS DriverEntry (IN PDRIVER_OBJECT pDriverObject,
                      IN PUNICODE_STRING strRegistryPath )
{
    pDriverObject->DriverUnload = driverCleanupFunction;

   // Register Callback some IRP, say, IRP_MJ_READ
   pDriverObject->MajorFunction[IRP_MJ_READ] = driverIRPDispatcher;

    DbgPrint("Driver:: Hello, Kernel!\n");
    return STATUS_SUCCESS;
}
```

**Listing 1**: Stub Driver, Entry Function, IRP Dispatcher registration

Since all dispatchers implement the same prototype, it's often simpler to just implement one, and register it to multiple IRPs, perhaps even all of them.To register more than one IRP to the same IRP Dispatcher you could use something like:

```
pDriverObject->MajorFunction[IRP_TO_PROCESS] =
                 ..
pDriverObject->MajorFunction[IRP_TO_ALSO_PROCESS] = driverIRPDispatcher;
```

Not all IRPs must be handled. As the following example shows, IRPs that do not have installed handlers are handled by IopInvalidDeviceRequest, which returns an error code for the IRP.

```
kd> !drvobj kmixer 7
Driver object (8697a988) is for:
 \Driver\kmixer
Driver Extension List: (id , addr)

Device Object list:
Device object (8653f528) is for:
  \Driver\kmixer DriverObject 8697a988
Current Irp 00000000 RefCount 0 Type 0000002f Flags 00002010
DevExt 8653f5e0 DevObjExt 8653f5e8
ExtensionFlags (0000000000)
AttachedTo (Lower) 866d9570 \Driver\swenum
Device queue is not busy.

DriverEntry:   b7203105   kmixer!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  b71ea610   kmixer!DriverUnload

Dispatch routines:
[00] IRP_MJ_CREATE                      f7039fe2    ks!DispatchCreate
[01] IRP_MJ_CREATE_NAMED_PIPE           804fb8de    nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       f7039711    ks!DispatchClose
[03] IRP_MJ_READ                        804fb8de    nt!IopInvalidDeviceRequest
[04] IRP_MJ_WRITE                       f70391cc    ks!DispatchWrite
[05] IRP_MJ_QUERY_INFORMATION           804fb8de    nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             804fb8de    nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    804fb8de    nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      804fb8de    nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               804fb8de    nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    804fb8de    nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      804fb8de    nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           804fb8de    nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         804fb8de    nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              f7038f60    ks!DispatchDeviceIoControl
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     804fb8de    nt!IopInvalidDeviceRequest
[10] IRP_MJ_SHUTDOWN                    804fb8de    nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                804fb8de    nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     804fb8de    nt!IopInvalidDeviceRequest
[13] IRP_MJ_CREATE_MAILSLOT             804fb8de    nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              804fb8de    nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                804fb8de    nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER                       f70327cf    ks!KsDefaultDispatchPower
[17] IRP_MJ_SYSTEM_CONTROL              b72014d0    kmixer!PerfWmiDispatch
[18] IRP_MJ_DEVICE_CHANGE               804fb8de    nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 804fb8de    nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   804fb8de    nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         b71ea570    kmixer!DispatchPnp
```

# IRP Debugging

Useful Tools:

- KD:
  - !irpfind: Find all active IRPs in the system
  - !irp: Display IRP specific data
- OSR: IRPTracker

The Kernel Debugger offers powerful extensions for diagnosing and debugging IRPs. The first is "!irpfind", that searches the non-paged pool for memory allocations with a tag of "Irp", and then walks them, and provides summary data:

```
kd> !irpfind
Searching NonPaged pool (81337000 : 82400000) for Tag: Irp?

  Irp    [ Thread ] irpStack: (Mj,Mn)   DevObj  [Driver]         MDL Process
81d674e8 [81f24558] irpStack: ( e,2d)  821612a0 [ \Driver\AFD]
81e2eb28 [81f23368] irpStack: ( e,2d)  821612a0 [ \Driver\AFD]
81e332b0 [8226dda8] irpStack: ( c, 2)  8232e020 [ \FileSystem\Ntfs]
81e3c008 [00000000] Irp is complete (CurrentLocation 3 > StackCount 2) 0x823b4788
81e3c528 [00000000] Irp is complete (CurrentLocation 3 > StackCount 2) 0x823b4788
81e3cd78 [00000000] Irp is complete (CurrentLocation 3 > StackCount 2) 0x823b4788
81e63570 [81e3d560] irpStack: ( c, 2)  8232e020 [ \FileSystem\Ntfs]
81e6a238 [82336a08] irpStack: ( e,43)  821612a0 [ \Driver\AFD]
81e70528 [00000000] Irp is complete (CurrentLocation 3 > StackCount 2) 0x823b4788
81e70950 [00000000] Irp is complete (CurrentLocation 3 > StackCount 2) 0x823b4788
81e70be0 [00000000] Irp is complete (CurrentLocation 3 > StackCount 2) 0x823b4788

81e8c3a0 [00000000] irpStack: ( 0, 0)  823078a8 [ \Driver\Cdrom]
```

The table lists the IRPs found, their owning threads (a PETHREAD), the owning device object, Device Driver, IRP Major and Minor code, and MDL, if any.

Specific detail for a particular IRP can then be displayed using "!irp" on the IRP address:

```
kd> !irp 81e332b0
Irp is active with 8 stacks 8 is current (= 0x81e3341c)
 No Mdl: No System Buffer: Thread 8226dda8:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                                       Args: 00000000 00000000 00000000 00000000
    ……
>[  c, 2]   1  1 8232e020 822f3400 00000000-00000000    pending
                    \FileSystem\Ntfs
                                       Args: 00000020 00000017 00000000 00000000
```

As well as, of course, other commands, like !thread and !devobj. !thread is especially useful, as it shows the thread's entire IRPList, as well as the process name.

```
kd> !thread 8226dda8
THREAD 8226dda8  Cid 0670.06ac  Teb: 7ffd6000 Win32Thread: e195aad0 WAIT:
……
IRP List:
    82297248: (0006,0190) Flags: 00000000  Mdl: 00000000
    82261b68: (0006,0190) Flags: 00000000  Mdl: 00000000
    81ea9dd8: (0006,0190) Flags: 00000000  Mdl: 00000000
    82035e70: (0006,0190) Flags: 00000000  Mdl: 00000000
    81e41008: (0006,0190) Flags: 00000000  Mdl: 00000000
    822db3e0: (0006,0190) Flags: 00000000  Mdl: 00000000
    81e332b0: (0006,0190) Flags: 00000000  Mdl: 00000000
Not impersonating
DeviceMap                 e1cc4470
Owning Process            0       Image:         <Unknown>
Attached Process          820cbda0       Image:         explorer.exe
……
```

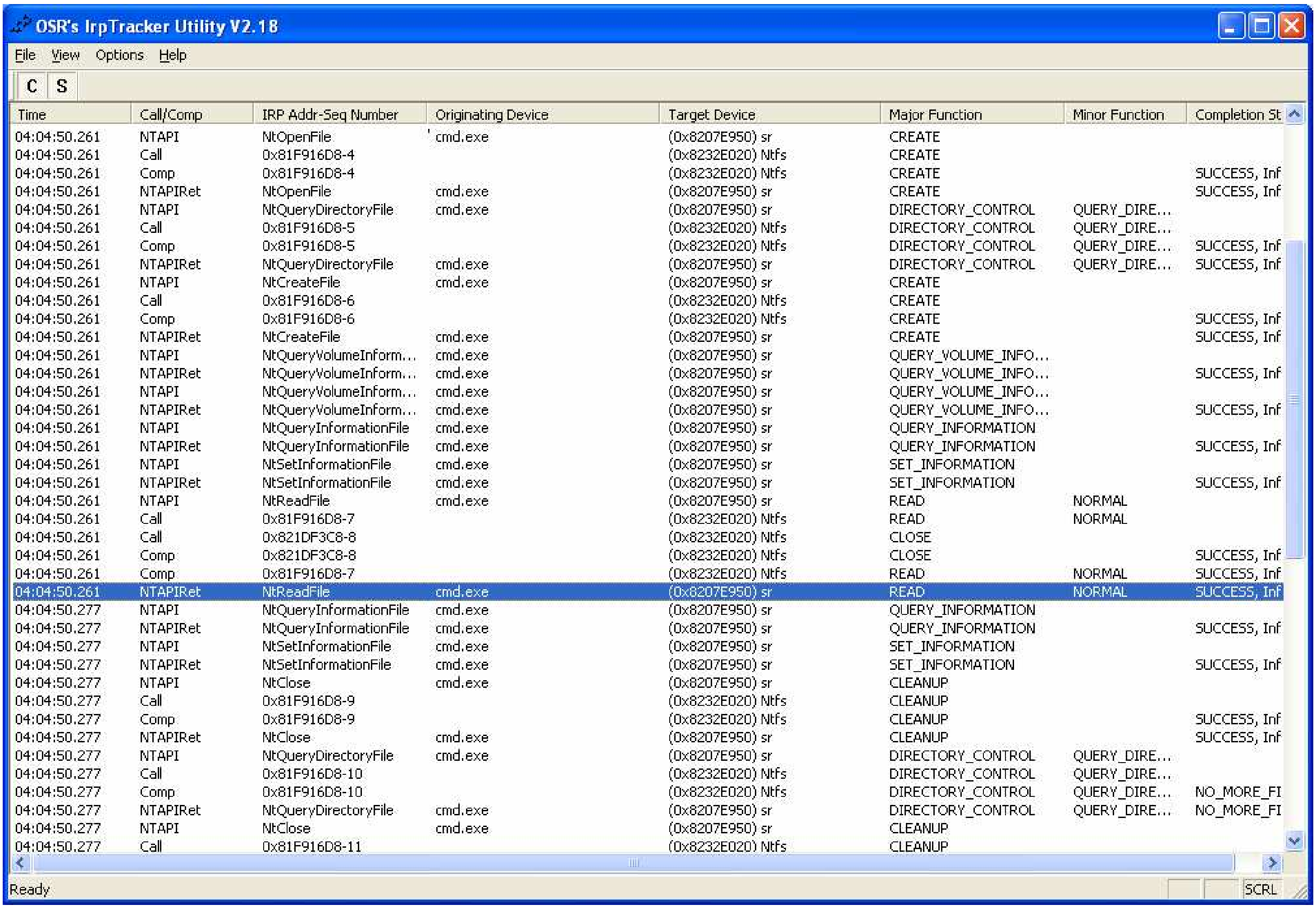
DT'ing also helps:

```
kd> dt !_IRP 81e332b0
ntdll!_IRP
   +0x000 Type             : 6
   +0x002 Size             : 0x190
   +0x004 MdlAddress       : (null)
   +0x008 Flags            : 0
   +0x00c AssociatedIrp    : __unnamed
   +0x010 ThreadListEntry  : _LIST_ENTRY [ 0x8226dfb8 - 0x822db3f0 ]
   +0x018 IoStatus         : _IO_STATUS_BLOCK
   +0x020 RequestorMode    : 1 ''
   +0x021 PendingReturned  : 0 ''
   +0x022 StackCount       : 8 ''
   +0x023 CurrentLocation  : 8 ''
   +0x024 Cancel           : 0 ''
   +0x025 CancelIrql       : 0 ''
   +0x026 ApcEnvironment   : 0 ''
   +0x027 AllocationFlags  : 0xc ''
   +0x028 UserIosb         : 0x7c8837e0 _IO_STATUS_BLOCK
   +0x02c UserEvent        : (null)
   +0x030 Overlay          : __unnamed
   +0x038 CancelRoutine    : 0x80512601     void  nt!FsRtlCancelNotify+0
   +0x03c UserBuffer       : 0x7c883800
   +0x040 Tail             : __unnamed
```

Because, remembering that “Tail” contains many useful parameters the Kernel associates this IRP with, one can quickly deduce:

```
kd> dd 81e332b0 + 0x40
81e332f0  00000000 00000000 00000000 00000000
81e33300  8226dda8 00000000 e187ab68 e187ab68
81e33310  81e3341c 822f3400 00000000 00000000
81e33320  00000000 00000000 00000000 00000000
81e33330  00000000 00000000 00000000 00000000
81e33340  00000000 00000000 00000000 00000000
81e33350  00000000 00000000 00000000 00000000
81e33360  00000000 00000000 00000000 00000000
lkd> dt _FILE_OBJECT 822f3400
ntdll!_FILE_OBJECT
   +0x000 Type             : 5
   +0x002 Size             : 112
    ……
   +0x02c Flags            : 0x40000
   +0x030 FileName         : _UNICODE_STRING "\Docume~1\All Users\Desktop"
   ……
```

For real time statistics, either attach a Kernel Debugger, or use OSR’s “IRPTracker” Utility. The figure below shows a capture of a “type C:\temp.txt” command from cmd.exe.


OSR's IrpTracker Utility V2.18

File View Options Help

C S

| Time | Call/Comp | IRP Addr-Seq Number | Originating Device | Target Device | Major Function | Minor Function | Completion St |
|---|---|---|---|---|---|---|---|
| 04:04:50.261 | NTAPI | NtOpenFile | cmd.exe | (0x8207E950) sr | CREATE | | |
| 04:04:50.261 | Call | 0x81F916D8-4 | | (0x8232E020) Ntfs | CREATE | | |
| 04:04:50.261 | Comp | 0x81F916D8-4 | | (0x8232E020) Ntfs | CREATE | | SUCCESS, Inf |
| 04:04:50.261 | NTAPIRet | NtOpenFile | cmd.exe | (0x8207E950) sr | CREATE | | SUCCESS, Inf |
| 04:04:50.261 | NTAPI | NtQueryDirectoryFile | cmd.exe | (0x8207E950) sr | DIRECTORY_CONTROL | QUERY_DIRE... | |
| 04:04:50.261 | Call | 0x81F916D8-5 | | (0x8232E020) Ntfs | DIRECTORY_CONTROL | QUERY_DIRE... | |
| 04:04:50.261 | Comp | 0x81F916D8-5 | | (0x8232E020) Ntfs | DIRECTORY_CONTROL | QUERY_DIRE... | SUCCESS, Inf |
| 04:04:50.261 | NTAPIRet | NtQueryDirectoryFile | cmd.exe | (0x8207E950) sr | DIRECTORY_CONTROL | QUERY_DIRE... | SUCCESS, Inf |
| 04:04:50.261 | NTAPI | NtCreateFile | cmd.exe | (0x8207E950) sr | CREATE | | |
| 04:04:50.261 | Call | 0x81F916D8-6 | | (0x8232E020) Ntfs | CREATE | | |
| 04:04:50.261 | Comp | 0x81F916D8-6 | | (0x8232E020) Ntfs | CREATE | | SUCCESS, Inf |
| 04:04:50.261 | NTAPIRet | NtCreateFile | cmd.exe | (0x8207E950) sr | CREATE | | SUCCESS, Inf |
| 04:04:50.261 | NTAPI | NtQueryVolumeInform... | cmd.exe | (0x8207E950) sr | QUERY_VOLUME_INFO... | | |
| 04:04:50.261 | NTAPIRet | NtQueryVolumeInform... | cmd.exe | (0x8207E950) sr | QUERY_VOLUME_INFO... | | SUCCESS, Inf |
| 04:04:50.261 | NTAPI | NtQueryVolumeInform... | cmd.exe | (0x8207E950) sr | QUERY_VOLUME_INFO... | | |
| 04:04:50.261 | NTAPIRet | NtQueryVolumeInform... | cmd.exe | (0x8207E950) sr | QUERY_VOLUME_INFO... | | SUCCESS, Inf |
| 04:04:50.261 | NTAPI | NtQueryInformationFile | cmd.exe | (0x8207E950) sr | QUERY_INFORMATION | | |
| 04:04:50.261 | NTAPIRet | NtQueryInformationFile | cmd.exe | (0x8207E950) sr | QUERY_INFORMATION | | SUCCESS, Inf |
| 04:04:50.261 | NTAPI | NtSetInformationFile | cmd.exe | (0x8207E950) sr | SET_INFORMATION | | |
| 04:04:50.261 | NTAPIRet | NtSetInformationFile | cmd.exe | (0x8207E950) sr | SET_INFORMATION | | SUCCESS, Inf |
| 04:04:50.261 | NTAPI | NtReadFile | cmd.exe | (0x8207E950) sr | READ | NORMAL | |
| 04:04:50.261 | Call | 0x81F916D8-7 | | (0x8232E020) Ntfs | READ | NORMAL | |
| 04:04:50.261 | Call | 0x821DF3C8-8 | | (0x8232E020) Ntfs | CLOSE | | |
| 04:04:50.261 | Comp | 0x821DF3C8-8 | | (0x8232E020) Ntfs | CLOSE | | SUCCESS, Inf |
| 04:04:50.261 | Comp | 0x81F916D8-7 | | (0x8232E020) Ntfs | READ | NORMAL | SUCCESS, Inf |
| 04:04:50.261 | NTAPIRet | NtReadFile | cmd.exe | (0x8207E950) sr | READ | NORMAL | SUCCESS, Inf |
| 04:04:50.277 | NTAPI | NtQueryInformationFile | cmd.exe | (0x8207E950) sr | QUERY_INFORMATION | | |
| 04:04:50.277 | NTAPIRet | NtQueryInformationFile | cmd.exe | (0x8207E950) sr | QUERY_INFORMATION | | SUCCESS, Inf |
| 04:04:50.277 | NTAPI | NtSetInformationFile | cmd.exe | (0x8207E950) sr | SET_INFORMATION | | |
| 04:04:50.277 | NTAPIRet | NtSetInformationFile | cmd.exe | (0x8207E950) sr | SET_INFORMATION | | SUCCESS, Inf |
| 04:04:50.277 | NTAPI | NtClose | cmd.exe | (0x8207E950) sr | CLEANUP | | |
| 04:04:50.277 | Call | 0x81F916D8-9 | | (0x8232E020) Ntfs | CLEANUP | | |
| 04:04:50.277 | Comp | 0x81F916D8-9 | | (0x8232E020) Ntfs | CLEANUP | | SUCCESS, Inf |
| 04:04:50.277 | NTAPIRet | NtClose | cmd.exe | (0x8207E950) sr | CLEANUP | | SUCCESS, Inf |
| 04:04:50.277 | NTAPI | NtQueryDirectoryFile | cmd.exe | (0x8207E950) sr | DIRECTORY_CONTROL | QUERY_DIRE... | |
| 04:04:50.277 | Call | 0x81F916D8-10 | | (0x8232E020) Ntfs | DIRECTORY_CONTROL | QUERY_DIRE... | |
| 04:04:50.277 | Comp | 0x81F916D8-10 | | (0x8232E020) Ntfs | DIRECTORY_CONTROL | QUERY_DIRE... | NO_MORE_FI |
| 04:04:50.277 | NTAPIRet | NtQueryDirectoryFile | cmd.exe | (0x8207E950) sr | DIRECTORY_CONTROL | QUERY_DIRE... | NO_MORE_FI |
| 04:04:50.277 | NTAPI | NtClose | cmd.exe | (0x8207E950) sr | CLEANUP | | |
| 04:04:50.277 | Call | 0x81F916D8-11 | | (0x8232E020) Ntfs | CLEANUP | | |

Ready SCRL

# Windows Debugger Cheat Sheet

| Command | Use |
|---|---|
| d | Dump memory address. Can further specify:<br>a – ASCII<br>d – Dword<br>t - as type – provide a structure name to overlay (needs symbols)<br>v - Variables local to scope (processes only)<br>ps – Pointers and Symbols<br>u – Unicode |
| k | Dump current thread stack |
| lm | List loaded and unloaded modules (useful to find drivers) |
| ln | List Nearest Symbols to address or symbol |
| r | Show/set registers |
| s | Search memory |
| u | Unassemble memory address or symbol.<br>Also: ub (unassemble backwards from address/symbol) |
| | |
| | |
| .sympath+ | Fix symbol path and append MS Symbol Server |
| .reload | Force reloading of Kernel Symbols |

## Useful Debugger Extensions:

| Command | Use |
|---|---|
| !analyze | Crash dump analysis. The author's favorite ☺ |
| !drvobj | Show Drive object of name (from lm) |
| !devobj | Show Device object at address |
| !devstack | Show Device Driver Stack for a given device |
| !idt | Show Kernel's Interrupt descriptor table (e.g. INT 2e) |
| !process | Show PEB at address (try "0") |
| !thread | Show TEB at address (try "0") |
| !pool, !pooltag, !poolfind | Pool debugging |
| !irpfind, !irp | Find IRPs in NonPagedPool, Display IRP contents |
| | |
| | |

*…If you liked this course, consider…*

## Networking:

Networking Protocols – OSI Layers 2-4:
Focusing on - Ethernet, Wi-Fi, IPv4, IPv6, TCP, UDP and SCTP

Application Protocols – OSI Layers 5-7:
Including - DNS, FTP, SMTP, IMAP/POP3, HTTP and SSL

VoIP:
In depth discussion of H.323, SCCP, SIP and RTP/RTCP, down to the packet level.

Windows Networking Internals:
NetBIOS/SMB, CIFS, DCE/RPC, Kerberos, NTLM, and networking architecture

---

## Linux:

Linux Survival and Basic Skills:
Graceful introduction into the wonderful world of Linux for the non-command line oriented user. Basic skills and commands, work in shells, redirection, pipes, filters and scripting

Linux Administration:
Follow up to the Basic course, focusing on advanced subjects such as user administration, software management, network service control, performance monitoring and tuning.

Linux User Mode Programming:
Programming POSIX and UNIX APIs in Linux, including processes, threads, IPC mechanisms and networking. Linux User experience required.

Linux Kernel Programming:
Guided tour of the Linux Kernel, 2.4 and 2.6, focusing on design, architecture, writing device drivers (character, block), performance and network devices

Embedded Linux Kernel Programming:
Similar to the Linux Kernel programming course, but with a strong emphasis on development on non-intel and/or tightly constrained embedded platforms

---

## Windows:

Windows Programming:
Windows Application Development, focusing on Processes, Threads, DLLs, Memory Management, and Winsock

Windows Kernel Programming (this course):
Windows Kernel Architecture and Device Driver development – focusing on Network Device Drivers (in particular, NDIS) and the Windows Driver Model. Updated to include NDIS 6 and Winsock Kernel

---

## Security:

Cryptography:
From Basics to implementations in 5 days: foundations, Symmetric Algorithms, Asymmetric Algorithms, Hashes, and protocols. Design, Logic and implementation

Application Security
Writing secure code – Dealing with Buffer Overflows, Code, SQL and command Injection, and other bugs… before they become vulnerabilities that hackers can exploit.