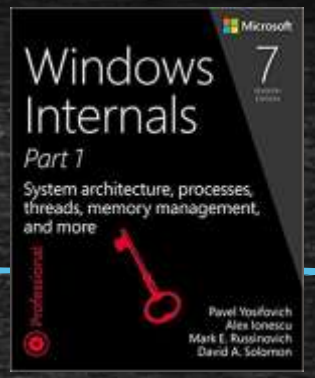
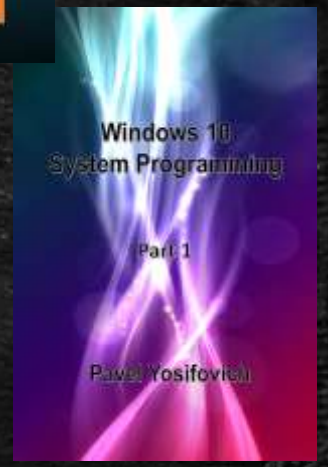
RECON 2023

Writing a Windows Kernel Driver in an Hour (or so)

Pavel Yosifovich
@zodiacon
zodiacon@live.com

About Me





- Developer, Trainer, Author, Speaker
- Book author
  - "Windows Internals 7th edition, Part 1" (co-author, 2017)
  - "Windows 10 System Programming, Part 1" (2020)
  - "Windows 10 System Programming, Part 2" (2021)
  - "Windows Kernel Programming, 2nd ed." (2023)
- *Pluralsight* and *Pentester Academy* course author
- Author of several open-source tools (http://github.com/zodiacon)
- Website: http://scorpiosoftware.net

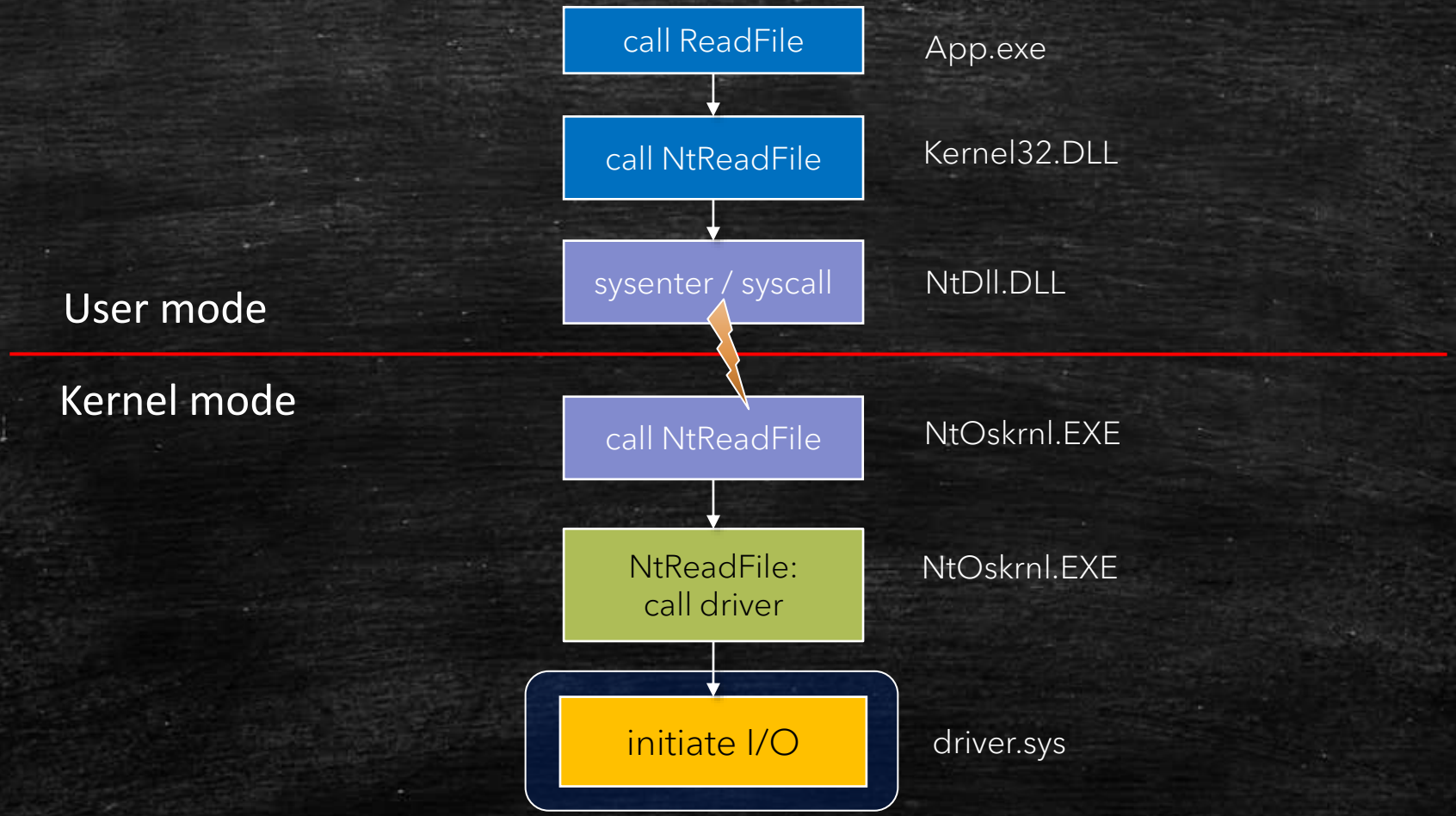
Overview

- Kernel Driver Basics
- Invoking Drivers
- **`DriverEntry`**
- Drivers, Devices, and Symbolic Links
- Handling Requests
- Testing and Debugging

Kernel Device Drivers

- Always execute in kernel mode
  - Use the kernel mode stack of a thread
  - Image part of system space
  - Unhandled exceptions will crash the system
- Typically has a SYS file extension
- Usually invoked by client code
  - e.g. `ReadFile`, `WriteFile`, `DeviceIoControl`
- Exports entry points for various functions
  - Called by system code when appropriate
- System handles all device independent aspects of I/O
  - No need for hardware specific code or assembly

Invoking a Driver



Accessing Devices

- A client that wants to communicate with a device, must open a handle to the device
  - `CreateFile` or `CreateFile2` from user mode
  - `ZwCreateFile` or `ZwOpenFile` from kernel mode
- `CreateFile` accepts a "filename" which is actually a device symbolic link
  - "file" being just one specific case
  - For devices, the name should have the format `\\.\name`
    - Cannot access non-local device

The `CreateFile` API

- Creates a file object
  - Creation flags must specify `OPEN_EXISTING` for devices
  - Driver sees **`CreateFile`** as an IRP with major code **`IRP_MJ_CREATE`**
  - Returns a handle to the file object
    - Returns `INVALID_HANDLE_VALUE` (-1) if fails
    - Call `GetLastError` to get reason for failure

```
HANDLE CreateFile(
    _In_          LPCTSTR lpFileName,             // name of file or device
    _In_          DWORD dwDesiredAccess,          // access mode (read, write, etc.)
    _In_          DWORD dwShareMode,              // share mode (for files)
    _In_opt_ LPSECURITY_ATTRIBUTES sa,            // security descriptor and handle inheritance
    _In_          DWORD dwCreationDisposition,    // creation flags (OPEN_EXISTING for devices)
    _In_          DWORD dwFlagsAndAttributes,     // more flags
    _In_opt_      HANDLE hTemplateFile);          // template file to copy attributes from
```

Setting Up For Kernel Development

- Install Visual Studio 2022 (any edition)
- Install the latest Windows 11 Software Development kit
- Install the latest Windows 11 Driver Kit (WDK)
  - The latest WDK allows building drivers for Windows 10 and later
  - The WDK installs a bunch of project templates for driver development
- Optionally, install the new WinDbg

The Kernel API

- Most of the kernel code is in *NtOsKrnl.exe*
- Some implementation is in *Hal.dll*
- The WDK documents about third of exported functions
- Most functions have a prefix indicating the component they are implemented in
- Main header file: <*ntddk.h*>

Some Kernel API Prefixes

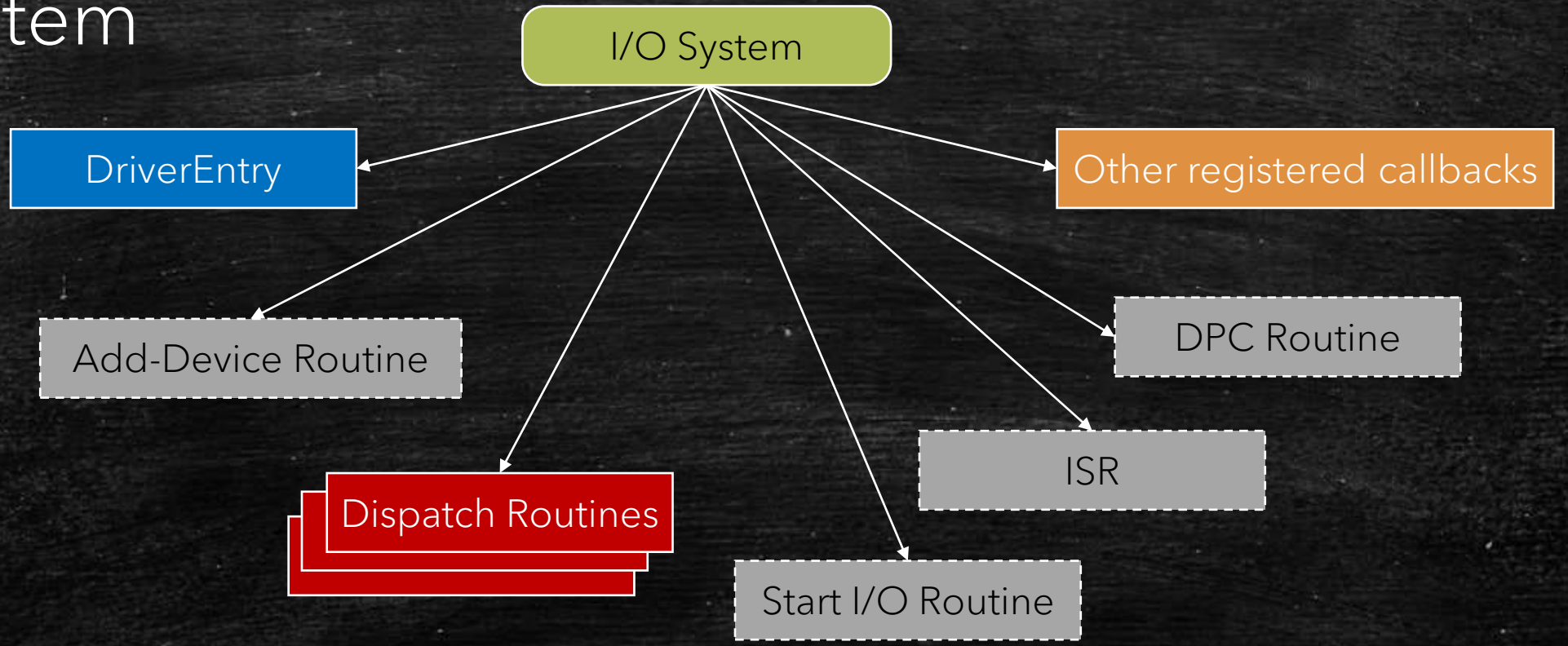
```
Ex              - General executive functions
Ke              - General kernel functions
Cc              - Cache Controller (Manager)
Mm              - Memory Manager
Rtl             - General runtime library
FsRtl           - File System Runtime Library
Flt             - File system mini-filters
Ob              - Object Manager
Io              - I/O Manager
Se              - Security
Ps              - Process Structure
Po              - Power Manager
Wmi             - Windows Management Instrumentation
Zw              - Native API wrappers
Hal             - Hardware Abstraction Layer
```

Functions and Error Codes

- Most kernel functions return **NTSTATUS**
  - 32-bit integer
  - MSB=1 indicates error
- Many status values
  - **STATUS_SUCCESS** = 0 (standard success)
  - Many failure status values
    - **STATUS_UNSUCCESSFUL** being the most generic
- If returned to user mode, turned into **ERROR_xxx**
- Check success with **NT_SUCCESS**(status) macro

Anatomy of a Driver

- A driver exports functionality, callable by the I/O system



Strings

- Most kernel APIs work with **UNICODE_STRING** structures
  - Length and MaximumLength are in bytes
  - Need ***not*** be NULL terminated
- Useful functions
  - RtlInitUnicodeString
    - Initializes a pre-allocated string (e.g. constant string)
  - RtlCopyUnicodeString
    - Copy from an existing **UNICODE_STRING**
    - Destination string must be initialized
  - RtlCompareUnicodeString
    - Case sensitive or insensitive comparison
  - RTL_CONSTANT_STRING macro to initialize with a literal string

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWCH   Buffer;
} UNICODE_STRING;
typedef UNICODE_STRING *PUNICODE_STRING;
```

The `DriverEntry` Function

- Must have for any kernel driver

```
NTSTATUS DriverEntry(
    _Inout_     PDRIVER_OBJECT  DriverObject,
    _In_        PUNICODE_STRING RegistryPath);
```

- Called with the driver object partially initialized using a system worker thread at IRQL 0
- Registry key points to the "Software" key
  - HKLM\System\CurrentControlSet\Services*ServiceName*
  - String is valid in **`DriverEntry`** only (deep copy if needed in other functions)
- Returning anything other than `STATUS_SUCCESS` causes driver to unload
  - Unload routine ***not*** called

Initializing the Driver Object

- Set the Unload routine
  - `DriverUnload` member of `DRIVER_OBJECT`
- Set supported major functions
  - `MajorFunction` array of function pointers
  - Unset major functions are initialized to "unsupported function"

Driver and Device Objects

- Drivers are represented in memory using a `DRIVER_OBJECT` structure
  - Created by the I/O system
  - Provided to the driver in the `DriverEntry` function
  - Holds all exported functions
- Device objects are created by the driver on a per-device basis
  - Within the **`AddDevice`** driver callback (for P&P drivers) or in **`DriverEntry`** (for software drivers)
  - Represented by the `DEVICE_OBJECT` structure
- I/O system is device-centric, not driver-centric

Exporting Entry Points

- `DriverEntry`'s main job is to export entry points for its main functions
  - `DriverUnload`
    - The unload routine, to be called when driver is about to be unloaded from memory
  - `MajorFunction`
    - An array of function pointers for the various types of IRPs the driver is going to handle

Creating a Device Object

- Define a device extension structure to hold all device-specific data
- Call `IoCreateDevice`

```
NTSTATUS IoCreateDevice(
    _In_        PDRIVER_OBJECT DriverObject,
    _In_        ULONG DeviceExtensionSize,
    _In_opt_    PUNICODE_STRING DeviceName,
    _In_        DEVICE_TYPE DeviceType,
    _In_        ULONG DeviceCharacteristics,
    _In_        BOOLEAN Exclusive,
    _Outptr_    PDEVICE_OBJECT *DeviceObject);
```

Device Symbolic Link

- Drivers should define a second name, if the device object is created with a name
- The kernel device name is not visible from user mode (`CreateFile`)
  - However, it is possible to access with the native `NtOpenFile`
  - Must create a symbolic link to the kernel device object
  - The name should be under the "\??\" directory in the object manager's namespace
  - The suffix of both names need not be the same, but usually is

Creating a Symbolic Link

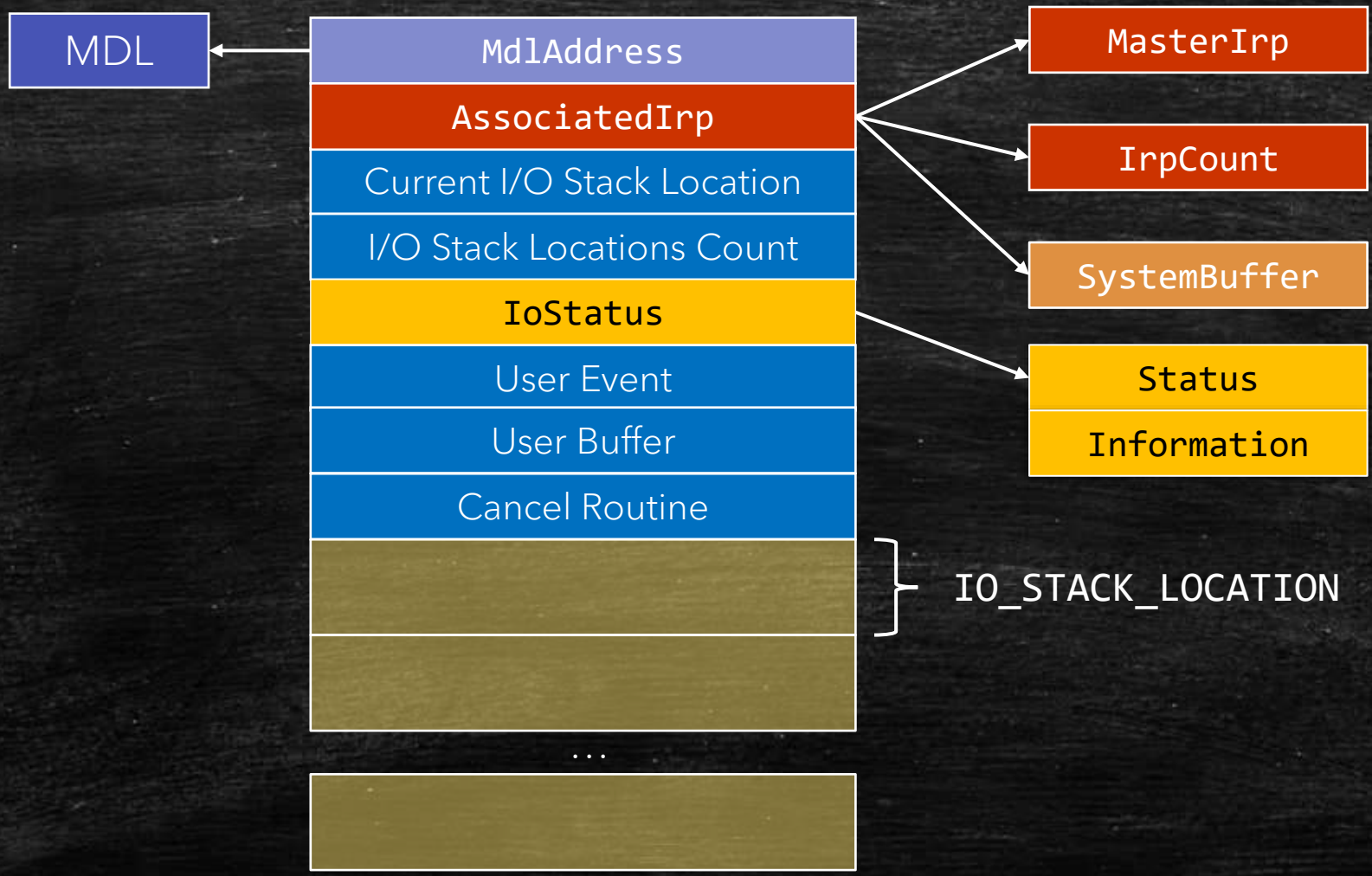
- Call `IoCreateSymbolicLink`
- "\??\" is a subdirectory in the object manager's namespace
  - Run *WinObj* tool from SysInternals to view
  - User mode apps can enumerate with **`QueryDosDevice`**

```
UNICODE_STRING Win32DevName;
RtlInitUnicodeString(&Win32DevName, L"\\??\\MyDev1");
IoCreateSymbolicLink(&Win32DevName, &DeviceName);
```

I/O Request Packet (IRP)

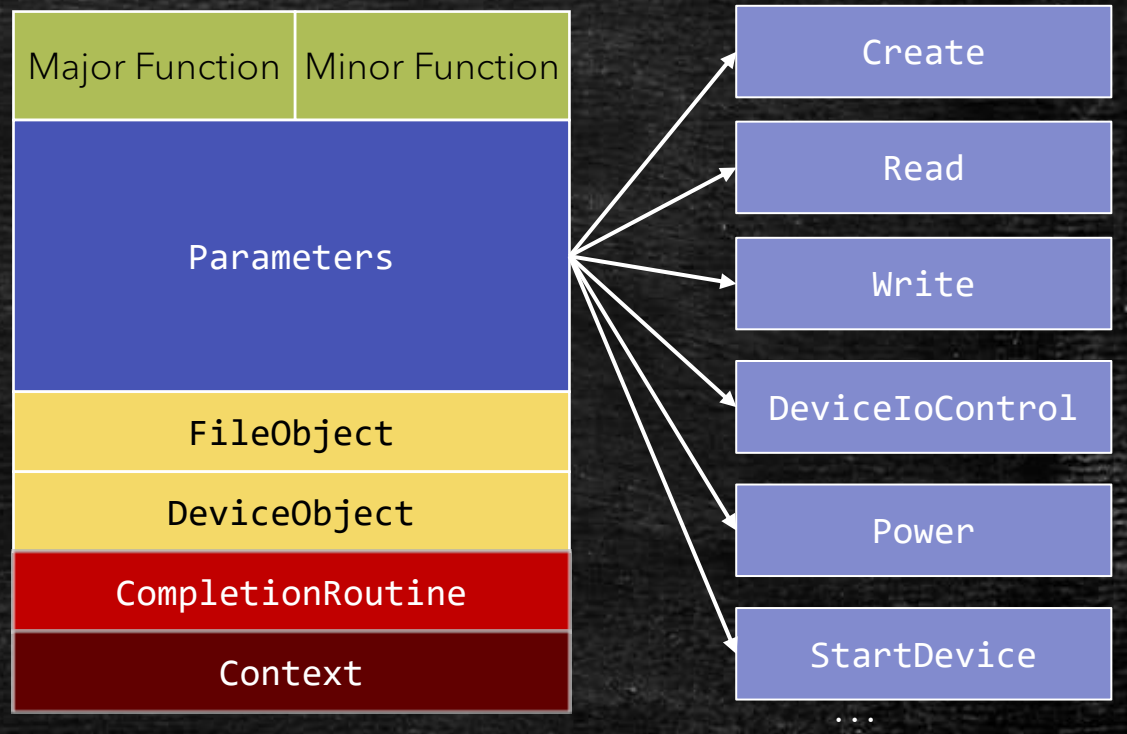
- A structure representing some request
  - Represented by the IRP structure
  - Contains all details needed to handle the request (codes, buffers, sizes, etc.)
- Always allocated from non-paged pool
- Accompanied by a set of structures of type IO_STACK_LOCATION
  - Complements the data in the IRP
- IRPs are typically created by the I/O Manager, P&P Manager or the Power Manager

The IRP Structure



The I/O Stack Location

- Each created IRP comes with an array of **`IO_STACK_LOCATION`** structures
- Basically, contain more info about the request



Dispatch Routines

- Located through the Driver object's `MajorFunction` array
  - Selected by the IRP major function code (located in the current I/O stack location)
- An unassigned entry means "unimplemented function"
  - I/O Manager returns error automatically

```
NTSTATUS SomeDispatchRotine(
    _In_    PDEVICE_OBJECT DeviceObject,
    _In_    PIRP           Irp);
```

Common IRP Types

- `IRP_MJ_CREATE`
  - Invoked for `CreateFile` calls
- `IRP_MJ_CLOSE`
  - Invoked when last handle closed to the file object
- `IRP_MJ_READ`, `IRP_MJ_WRITE`
  - Invoked for `ReadFile`, `WriteFile`
- `IRP_MJ_DEVICE_CONTROL`
  - Invoked for `DeviceIoControl`

Completing the Request

```
Irp->IoStatus.Status = STATUS_XXX;
Irp->IoStatus.Information = NumberOfBytesTransfered;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
return STATUS_XXX;
```

- Returned status is mapped to a Win32 error code
  - There is no one to one mapping relationship
  - See *NTSTATUS.H* for possible error codes
- `NumberOfBytesTransfered` should be zero if error occurred
  - Might mean something else depending on the type of IRP
- Do not boost thread priority on immediate completion

Installing a Software Driver

- Normally, an INF file is not needed
- Use the `CreateService` API to install
  - Or a comparable tool, such as *Sc.exe*
- Example
  - Use elevated command window
  - Mind the spaces (or lack thereof)

```
sc.exe create MyDriver type= kernel binPath= c:\Source\MyDriver.sys
```

Starting and Stopping the Driver

- By default, driver is installed with start type "on demand"
- To load the driver, use the `StartService` API
  - Or "`sc start mydriver`"
- Driver binary is loaded into kernel space
- `DriverEntry` is now called
- To stop the driver
  - Call the `ControlService` API with `SERVICE_CONTROL_STOP`
  - Or use "`sc stop mydriver`"
  - Unload routine is called before the driver binary is unloaded

Testing the Driver

- Copy the binary (SYS) file to the target system
  - Preferably a virtual machine
- Enable test signing mode so that unsigned drivers can be loaded
  - `bcdedit /set testsigning on` (and restart)
- Install the driver on the target system
- Copy a test application to the target system
- Use the test application to send commands to the driver

Debugging the Driver

- Install the driver on the target system
- Create a debugger connection to the target system
- Make sure the debugger symbol path points to where the PDB file is
  - Or copy the PDB file to the same location as the driver
- Start remote kernel debugging
- Set breakpoints as appropriate
  - Use the **bu** command to set a breakpoint at **`DriverEntry`**
- Debug!

`KdPrint` / `DbgPrint` Output

- Using the `KdPrint` macro or `DbgPrint` function sends output to a debugger, if connected
- Requires creating a registry key *HKLM\System\CurrentControlSet\Control\Session Manager\Debug Print Filter*
  - Add a DWORD value named *DEFAULT* and set its value to 8
  - Restart the system
- Can capture debug output with the *DbgView* Sysinternals tool
  - Use the option "Enable Verbose Kernel Output" to capture calls regardless of the above Registry key/value settings

Remote Kernel Debugging

- Target machine
  - Configure for debugging as before
  - Select a communication medium
    - *MsConfig.exe* or *bcdedit.exe /dbgsettings*
    - Serial, USB, Network (Windows 8+)
- Host machine
  - File | Kernel Debug…
  - Select configured communication medium
- If target is a virtual machine
  - Can expose a VM COM port as a host named pipe
  - Or use network if host and target are Windows 8+

Remote Kernel Debugging (Network)

- Target
  - Elevated command window
  - `bcdedit /dbgsettings net hostip:<ip> port:<port> [key:<key>]`
  - If key is not provided, a random key is generated
  - Restart the system
- Host (debugger)
  - Select network debugging
  - Set same port and key
- The *KdNet.exe* tool simplifies the above procedure somewhat