

Creating Device Drivers

Creating a sample “Hello, Kernel” driver

This section demonstrates how to create a simple device driver, and one method of installing it.

Key Concepts: DriverEntry, DRIVER_OBJECT, SCM

Writing a basic Device Driver

- Device Drivers all implement a standard interface

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

- Driver is required to:
 - populate pDriverObject
 - Set callback routines
 - Perform any driver specific (NOT device specific) initializations
 - Return an NTSTATUS response code (e.g. STATUS_SUCCESS)
- To link to an entry point with a different name: /entry

Just like any user mode application has an entry point, usually “int main(int argc, char **argv, char **envp)”, a driver is expected to likewise implement a standard interface – called “DriverEntry”. The name itself may be changed, but if so, the linker has to be told about it with a “/entry” switch defining the new entry point.

The DriverEntry function will be passed two arguments from the Kernel:

PDRIVER_OBJECT: A pointer to a DRIVER_OBJECT structure. This is discussed shortly.

PUNICODE_STRING: A pointer to a UNICODE_STRING representing the Driver's Registry Entry. This is a path name in the system's registry, under the key:

\Registry\Machine\System\CurrentControlSet\Services\DriverName,

in the SYSTEM hive. The path name is where the Driver's configuration entries are saved, and may be tweaked by the System Administrator, or the Driver's installation function. It's important to save this Unicode String (i.e. wstrcpy() it to some Driver global buffer) since the I/O Manager will free this string upon the DriverEntry function's return.

Device specific initializations are handled by an addDevice routine, and not by the driver entry.

The Driver Object

- A semi-opaque object used by the I/O manager

<u>Field</u>	<u>Use</u>
PDEVICE_OBJECT DeviceObject	Linked list of Driver's devices
PDRIVER_EXTENSION DriverExtension	Used for AddDevice
PUNICODE_STRING HardwareDatabase	\Registry\Machine\Hardware path
PFAST_IO_DISPATCH FastIoDispatch	Fast I/O (File Systems/Network drivers)
PDRIVER_INITIALIZE DriverInit	Pointer to DriverEntry
PDRIVER_STARTIO DriverStartIo	Pointer to Driver StartIO function, or NULL.
PDRIVER_UNLOAD DriverUnload	Pointer to DriverUnload function, or NULL.
PDRIVER_DISPATCH MajorFunction	Array of Major function codes corresponding to IRPs handled by dispatcher function(s)

- Populated by *DriverEntry*

The **DRIVER_OBJECT** is a semi-opaque struct that the I/O manager passes to the device driver. Upon first invocation of the driver – in the *DriverEntry* – the driver is expected to populate it with whatever data it requires for further callbacks. From that point on, the same struct will be passed on to the respective callbacks.

The structure is semi-opaque on purpose – Microsoft keeps many details and fields for its own internal use. We will demonstrate one of them later on, when we talk about drivers operating in “stealth” mode – hiding their presence from others, including the Kernel itself.

A sample driver, then, that does nothing but initialize, and clean up would look like this:

```
NTSTATUS DriverEntry (IN PDRIVER_OBJECT pDriverObject,
                    IN PUNICODE_STRING strRegistryPath )
{
    pDriverObject->DriverUnload = driverCleanupFunction;

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

Listing 1: Stub Driver, demonstrating a DriverEntry

And for the cleanup:

```
NTSTATUS DriverCleanupFunction (IN PDRIVER_OBJECT pDriverObject)
{
    DbgPrint("Driver:: Exit, Stage Left..\n");
    return STATUS_SUCCESS;
}
```

Listing 2: Stub Driver, demonstrating a Driver Cleanup function

Controlling Driver Paging

- Use `#pragma alloc_text` where available:
 - `alloc_text(init, function)` on discardable initialization functions
 - `alloc_text(page, function)` on pageable functions
 - Note: Pageable functions MUST run at `IRQL == PASSIVE_LEVEL`
- Selectively lock/unlock pages (`#pragma data_seg()` or `code_seg()`)

```
PVOID MmLockPagableCodeSection (IN PVOID AddressInSection);
PVOID MmLockPagableDataSection (IN PVOID AddressInSection);
VOID MmUnlockPagableImageSection(IN PVOID ImageHandle);
```

- Entire Driver can be paged or locked

```
PVOID MmPageEntireDriver(IN PVOID AddressofDriverEntry);
PVOID MmResetDriverPaging(IN PVOID AddressWithinSection);
```

- Note: Driver cannot be paged if it installed Interrupt Handlers!

Even though most systems today sport Physical memory in the GB range, it's a recommended practice to be very conservative with memory usage at the driver level. Visual Studio supports a `#pragma` called **`alloc_text`**, that defines functions as discardable or pageable.

Functions that are used only during the driver initialization phase (i.e. `DriverEntry` and whatever functions it calls) can be defined as init functions. Other functions, used at `IRQL == PASSIVE_LEVEL`, can be pageable. The `IRQL` requirement is, to remind you, because the system page swapper runs at `IRQL == APC_LEVEL`.

This `pragma` only applies to C-linkage functions. To use it, you must define the function prototype, and place the `#pragma` setting in between the function prototype and definition. During runtime, you can also override any `pragma` settings and force paging using **`MmPageEntireDriver()`**, by supplying it with the address of your *DriverEntry* or any other function in the section. This technique must NOT be used if you have registered any Interrupt Handlers (ISRs), as it will crash the system.

Conversely, you can lock your sections in memory by calling **`MMResetDriverPaging()`**.

Caution: Incorrectly marking sections of your driver as pageable will quickly lead to Bug Check 0xD3: **`DRIVER_PORTION_MUST_BE_NONPAGED`**

Installing the Device Driver

Device drivers may be installed:

- By using an INF file:
 - The preferred, professional method
 - Allows for automatic installation by Windows setup
 - Device Manager can add/remove/update driver
- Programmatically:
 - The deprecated method, that should be avoided
 - Installation must be performed manually
 - (extremely) Useful if you're a trojan installing a rootkit..

Creating a Device Driver

Installing the Device Driver

- Start by obtaining a handle to the Service Control Manager:

```
SC_HANDLE WINAPI OpenSCManager(OPTIONAL IN LPCTSTR lpMachineName,
                                OPTIONAL IN LPCTSTR lpDatabaseName,
                                IN DWORD dwDesiredAccess );
```

- Then:

```
SC_HANDLE WINAPI CreateService
( IN SC_HANDLE hSCManager,
  IN LPCTSTR lpServiceName,
  OPTIONAL IN LPCTSTR lpDisplayName,
  IN DWORD dwDesiredAccess,
  IN DWORD dwServiceType,
  IN DWORD dwStartType,
  IN DWORD dwErrorControl,
  OPTIONAL IN LPCTSTR lpBinaryPathName,
  OPTIONAL IN LPCTSTR lpLoadOrderGroup,
  _out OPTIONAL LPDWORD lpdwTagId,
  OPTIONAL IN LPCTSTR lpDependencies,
  OPTIONAL IN LPCTSTR lpServiceStartName,
  OPTIONAL IN LPCTSTR lpPassword );
```

A Windows Kernel Device Driver is considered a Windows “Service”, dating back to the old days of Windows NT, where Drivers were viewable in a similar manner to services, via the Control Panel.

The simplest way to install a Driver, albeit deprecated, is by using the Service Control Manager. Much like any user mode service, this requires two calls. The first is a call to OpenSCManager:

```
hSCM = OpenSCManager(NULL, /* Local Machine */
                     NULL, /* Local Machine */
                     SC_MANAGER_ALL_ACCESS); /* or READ | WRITE */
```

Assuming this call succeeds (it would, of course, require Administrator privileges), the returned handle can be used to install the driver:

```
SC_HANDLE hDriver = CreateService(hSCM,
                                   L"My Kernel Driver",
                                   L"Driver Display Name",
                                   SERVICE_ALL_ACCESS,
                                   /* This makes the difference: */ SERVICE_KERNEL_DRIVER,
                                   SERVICE_DEMAND_START,
                                   SERVICE_ERROR_NORMAL,
                                   "C:\\driver.sys",
                                   NULL,
                                   NULL,
                                   NULL,
                                   NULL,
                                   NULL);
```

Installing the Device Driver

“Stealth” Mode:

- Undocumented call: ZwSetSystemInformation
- Code 38: Loads AND calls an image
- Leaves no registry trace, no SCM entry
- Caveats:
 - Driver is *pageable*.
 - Unreliable

A well known method of installing a driver without any Registry or Service Control Manager interface involves using an undocumented function, ZwSetSystemInformation.

<http://archives.neohapsis.com/archives/ntbugtraq/2000-q3/0114.html>

```
typedef struct _SYSTEM_LOAD_AND_CALL_IMAGE
{ UNICODE_STRING ModuleName; } SYSTEM_LOAD_AND_CALL_IMAGE;
SYSTEM_LOAD_AND_CALL_IMAGE MyDeviceDriver;
WCHAR imagepath[] = L"\\??\\C:\\driver.sys"; /* Path to driver */
RtlInitUnicodeString = (void*)GetProcAddress(GetModuleHandle("ntdll.dll"),
                                              "RtlInitUnicodeString");
ZwSetSystemInformation=(void*)GetProcAddress(GetModuleHandle("ntdll.dll"),
                                              "ZwSetSystemInformation");

if( RtlInitUnicodeString && ZwSetSystemInformation )
{
    RtlInitUnicodeString( &(amp; MyDeviceDriver.ModuleName), imagepath );
    status = ZwSetSystemInformation(38,
                                    &MyDeviceDriver,
                                    sizeof(SYSTEM_LOAD_AND_CALL_IMAGE));
}
```


Starting/Stopping the Driver

- Drivers may be controlled by:
 - “**Net start/stop**” – from any console (command line) window
 - Programmatically accessing the SCM:

```
BOOL WINAPI ControlService(IN SC_HANDLE hService,  
                           IN DWORD     dwControl,  
                           OUT LPSERVICE_STATUS lpServiceStatus );
```

After a driver is installed with the SCM, it still needs to be installed. This can be done, like any Windows Service, with a “net start” command:

```
E:\WINDOWS\system32> net start "My kernel Driver"  
The My kernel Driver service is starting.  
The My kernel Driver service was started  
successfully.  
  
E:\WINDOWS\system32> net stop "My kernel Driver"  
The My kernel Driver service is stopping.  
The My kernel Driver service was stopped  
successfully.
```

or programmatically:

```
if(0 == StartService(hService, 0,  
NULL))  
{  
    /* Great! */  
}  
else {  
    // Call GetLastError()..  
}
```

Creating System Threads

- Creating system threads is straightforward:

```
NTSTATUS PsCreateSystemThread(  
    OUT PHANDLE ThreadHandle, // close with ZwClose()  
    IN ULONG DesiredAccess,  
    IN POBJECT_ATTRIBUTES ObjectAttributes, // OBJ_KERNEL_HANDLE  
    IN HANDLE ProcessHandle OPTIONAL,  
    OUT PCLIENT_ID ClientId OPTIONAL,  
    IN PKSTART_ROUTINE StartRoutine,  
    IN PVOID StartContext);
```

- Thread priority may be further controlled:

```
KPRIORITY KeSetPriorityThread(IN PKTHREAD Thread,  
                             IN KPRIORITY Priority);
```

- No terminate API - Thread must terminate itself

The Device Driver will generally act as a service – meaning it will respond to requests coming from user mode (via System calls and I/O Request Packets, or IRPs), or interrupts coming from a device. Sometimes, however, a device driver needs to create its own independent thread for whatever purpose. For this, the Windows Kernel Process Manager (the Ps subsystem) offers a full thread API, chief amongst which is the **PsCreateSystemThread** call.

The call is very similar to Win32's `CreateThread()`, with the exception that it allows for a process handle, as well. If the Process Handle is set to `NULL`, the thread is created under the System (`Id=4`) process. It's possible, however, to create threads in any process, if a handle to that process can be obtained. If creating threads in other processes aside from the System one, the "Object Attributes" must be set to `OBJ_KERNEL_HANDLE` – or else the thread will be accessible to the process in which it is running.

There is no known API to terminate a Kernel thread – the thread must terminate itself, by calling **PsTerminateThread()**.

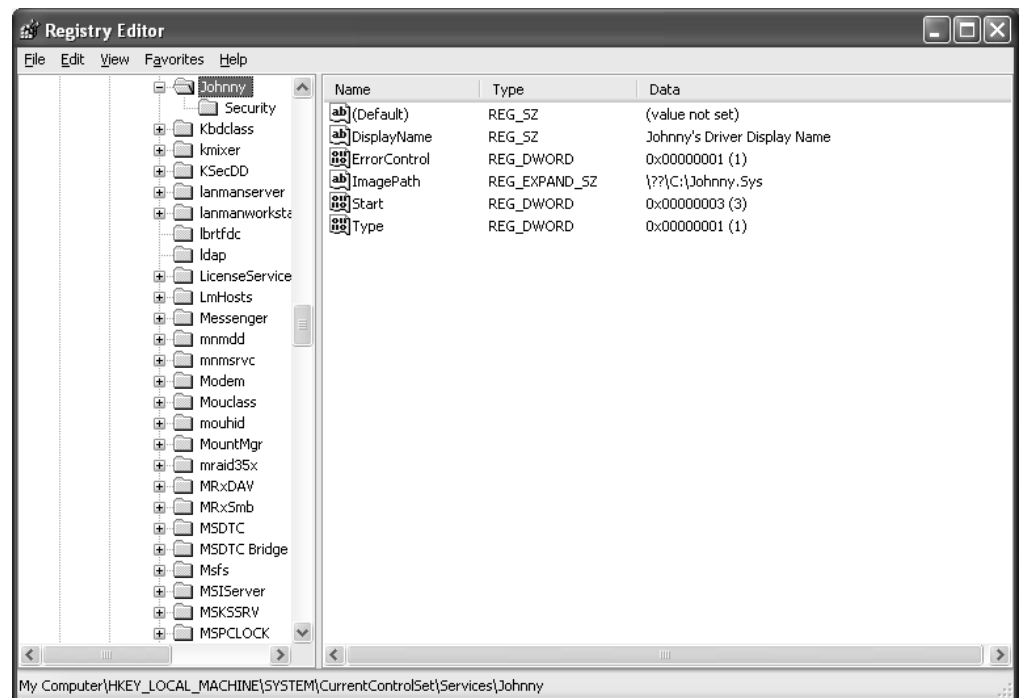
Exercises

1. In this exercise we will create a very basic driver (that does absolutely nothing), compile and build it.
 - i. Open a Windows XP "Checked Build" command prompt. Make sure your PATH settings allow you to invoke the "build" script.
 - ii. Create the basic driver shown in Listing I.
 - iii. Create a SOURCES file to build your driver.
 - iv. Run "build" and examine the resulting SYS file. Use DumpBin to verify its imports and exports. What are its dependencies?

2. We will now take the simple driver and install it, using the Service Control Manager.
 - i. Create a main program to invoke the Service Control Manager and install your driver.

 - ii. Run your program to install the Driver. Now run "Services.msc" and/or "net start" and look for your driver. What do you see?

- iii. Using RegEdit, search for your driver in the registry, in HKLM\System\CCS\Services. Explain the parameters you see:



- iv. What other tool could you use to see if your driver has been loaded successfully?

Exercises

2. In this exercise we will trace the linked list of Driver objects manually, and see their interrelations with their devices, and each other.
 - i. Start LiveKD, as in the previous exercises
 - ii. Use !drvobj tcpip, and record the address of the DRIVER_OBJECT. You should see something like

```
kd> !drvobj tcpip
Driver object (86be3ca8) is for:
\Driver\Tcpip
Driver Extension List: (id , addr)

Device Object list:
86c3fd80 86cbfd80 86c31af0 86c2ef18
86be2bc0
```
 - iii. Try the "!devobj" command on the entries in the "Device Object List". What do you see?

 - iv. Do a "dd" on the driver object + 20 bytes. Record this address. This is the address of the linked list of drivers.
 - v. Next, do a dd or two on the address you just figured out. Somewhere around there lies a Unicode string which tells you what the pathname of this driver is. What offset is it? Try the "du" command to see what the pathname is.
 - vi. Claim: That address indeed holds a linked list, you should be able to see two entries – the PREV and the next. Follow the linked list by applying (iii-iv) iteratively and figuring out the names of the next drivers in sequence.

Kernel Survival Guide

This section discusses the constraints of Kernel Mode programming: The APIs exposed by the Kernel executive, memory allocation, IPC & synchronization objects, outputting messages to user space, and crashing.

Key Concepts: Rtl functions, Ex Functions, Mutexes, Events, Timers, Spinlocks, DbgPrint, Event Logging, Bug Check

Kernel APIs

- Kernel APIs prefixed by “subsystem” identifier
 - See below table
- Most are directly exported from NTOSKRNL.exe
 - Few from HAL.DLL
- Fairly well documented in the MSDN

This table lists some of the Kernel API identifiers in Windows XP. These are very closely tied to the various Kernel “Subsystems” that we discussed in the architectural overview. The exception to this are the Rtl functions, which serve as basic “Run Time Library” support the Kernel offers in the absence of a full fledged C-level API.

Aux	Auxiliary Library
Clfs	Common Log File System
Cc	Cache Manager
Cm	Configuration Manager
Ex	Executive (Memory Allocation wrappers, etc)
Flt	Filter Manager
Hal	Hardware Abstraction Layer
Io	I/O Manager
Ke	Kernel Core
Mm	Memory Manager
Nt	Native Services (User Mode)
Ob	Object Manager
Po	PnP/Power Manager
Ps	Processes and Threads
Rtl	Run Time Library
Se	Security Reference Monitor
Wmi	Windows Management Instrumentation
Zw	Kernel Mode Wrappers for Nt*

Windows Version

- Always useful to know what you're running in:

```
NTSTATUS RtlGetVersion(IN OUT PRTL_OSVERSIONINFOW lpVersionInfo);
```

- Optionally, use RtlVerifyVersionInfo()

```
NTSTATUS RtlVerifyVersionInfo(IN PRTL_OSVERSIONINFOEXW lpVerInfo,
                           IN ULONG                    TypeMask,
                           IN ULONGLONG                CondMask);
```

Most drivers need to tweak their functionality to the exact version of Kernel they are running in. Microsoft Operating Systems do differ in their Kernel implementations in between Windows Versions and even Service Packs. Therefore it's standard practice to call **RTLGetVersion()**, or the now deprecated **PsGetVersion()** (used prior to Windows XP). This function is essentially the Kernel equivalent of **GetWindowsVersion()**, but operates slightly differently: It accepts a pointer to a struct of RTL_OSVERSIONINFO, or a slightly more advanced RTL_OSVERSIONINFOEX. Which looks as follows:

```
typedef struct _OSVERSIONINFOEXW {
    ULONG    dwOSVersionInfoSize;
    ULONG    dwMajorVersion;
    ULONG    dwMinorVersion;
    ULONG    dwBuildNumber;
    ULONG    dwPlatformId;
    WCHAR    szCSDVersion[ 128 ];
    USHORT   wServicePackMajor;
    USHORT   wServicePackMinor;
    USHORT   wSuiteMask;
    UCHAR    wProductType;
    UCHAR    wReserved;
} RTL_OSVERSIONINFOEXW;
```

The function relies on the first field, dwOSVersionInfoSize, to be set to the sizeof() the structure before the call. It can use the size to tell which of the two structs was passed to it.

It is then possible to verify what version of Windows you are in with something like the following code:

```
RTL_OSVERSIONINFOWEX osv;

osv.dwOSVersionInfoSize = sizeof(RTL_OSVERSIONINFOWEX);

NTSTATUS status;
status = RtlGetVersion((RTL_OSVERSIONINFOW *)&osv);

switch (osv.dwMajorVersion)
{
    case 5:
        if (osv.dwMinorVersion == 1) { /* XP */ }
        if (osv.dwMinorVersion == 2) { /* 2003 */ }
        break;

    case 6:
        if (osv.dwMinorVersion == 0) { /* 2008 */ }
        break;

    default:
        ..
} /* end Switch */
```

Listing 1: Verifying Windows Kernel Version

Kernel File and Dir Access

- Win32 CreateFile and friends are still available – As Zw*
- These are Kernel mode wrappers to the Nt* versions
 - Nt* cannot be called directly (return to User mode)
- Zw* functions allow for most operations, including:
 - File and directory access
 - Registry access

While the Kernel does not allow calling system calls from within Kernel space, sometimes there has to be a method to access user space objects, most notably files and registry keys. For this, the Kernel offers the Zw* API, which is a set of wrappers over the NT apis. These calls are actually faster than their Nt* counterparts, as they bypass parameter validation and access right checks.

```
NTSTATUS ZwCreateFile(OUT PHANDLE FileHandle,
                    IN ACCESS_MASK DesiredAccess,
                    IN POBJECT_ATTRIBUTES Attributes,
                    OUT PIO_STATUS_BLOCK IoStatusBlock
                    ,
                    OPTIONAL IN PLARGE_INTEGER AllocSize ,
                    IN ULONG FileAttributes,
                    IN ULONG ShareAccess,
                    IN ULONG CreateDisposition,
                    IN ULONG CreateOptions,
                    IN PVOID EaBuffer OPTIONAL,
                    IN ULONG EaLength);
```

Kernel Registry Access

- ADVAPI's Registry interface is implemented:

```
NTSTATUS RtlCheckRegistryKey(IN ULONG   RelativeTo,
                          IN PWSTR   Path);

NTSTATUS RtlCreateRegistryKey(IN ULONG   RelativeTo,
                           IN PWSTR   Path);

NTSTATUS RtlQueryRegistryValues(IN ULONG   RelativeTo,
                              IN PCWSTR  Path,
                              IN PRTL_QUERY_REGISTRY_TABLE QueryTable,
                              IN PVOID   Context,
                              IN PVOID   Environment OPTIONAL);

NTSTATUS RtlWriteRegistryValue(IN ULONG   RelativeTo,
                              IN PCWSTR  Path,
                              IN PCWSTR  ValueName,
                              IN ULONG   ValueType,
                              IN PVOID   ValueData,
                              IN ULONG   ValueLength);
```

The Windows Registry.. Can't live with it, can't live without it. The Kernel's RunTime Library exports an API that is nearly 1:1 that of ADVAPI32's RegXXX functions. The Kernel also offers an executive interface, via Zw functions, but since these work with keys as objects, the approach requires interaction with the Object Manager by creating and initializing an OBJECT_ATTRIBUTES structure.

Object Access

- User mode HANDLES can be used in the Kernel

```
NTSTATUS ObReferenceObjectByHandle(
    IN HANDLE                Handle,
    IN ACCESS_MASK           DesiredAccess,
    IN POBJECT_TYPE          ObjectType OPTIONAL,
    IN KPROCESSOR_MODE       AccessMode,
    OUT PVOID                *Object,
    OUT POBJECT_HANDLE_INFORMATION HandleInfo OPTIONAL);
```

Win32 “Handles” are actually implemented as void pointers, that are opaque and manipulated by the Kernel. Objects from user mode can thus be accessed in Kernel mode, by using the Object Manager’s **ObReferenceObjectByHandle** function. The “Handle” is the user mode handle.

The AccessMode enum, **KPROCESSOR_MODE**, may be **UserMode** or **KernelMode**

ObjectType may be one of the following:

***IoFileObjectType** - PFILE_OBJECT (File Handle)

***ExEventObjectType** - PKEVENT (Event Handle)

***ExSemaphoreObjectType** – PKSEMAPHORE (Sempahore)

***PsProcessType** PEPROCESS or PKPROCESS (Handle from OpenProcess)

***PsThreadType** PETHREAD or PKTHREAD (Handle from OpenThread)

And AccessMode should be KernelMode. HandleInfo is left NULL.

Memory Copy/Move

- Memcpy(), memset() are implemented as:

```
VOID RtlCopyMemory(IN VOID UNALIGNED *Destination,
                  IN CONST VOID UNALIGNED *Source,
                  IN SIZE_T Length);

VOID RtlFillMemory(IN VOID UNALIGNED *Destination,
                  IN SIZE_T Length,
                  IN UCHAR Fill);
```

Pool Type	Purpose
NonPagedPool	Memory that is always resident and never paged out. Always accessible – but considered scarce. Call may fail.
NonPagedPoolMustSucceed	As NonPaged, but if call fails system blue screens with code 0x41.
PagedPool	Normal system memory – not guaranteed to be accessible. May trigger a pagefault. Must be running at a lower priority than dispatcher to access this memory.

Kernel String Manipulation

- The Kernel offers full ANSI and Unicode String support

```
VOID RtlInitString(IN OUT PSTRING DestinationString,  
                  IN PCSZ   SourceString);
```

- Strcpy(), strcmp() are supported as:

```
VOID RtlCopyString(IN OUT PSTRING DestinationString,  
                  IN const STRING * SourceString);  
  
LONG RtlCompareString(IN PSTRING String1,  
                     IN PSTRING String2,  
                     BOOLEAN CaseInsensitive);
```

- Unicode can only be manipulated at passive IRQLs

The Windows Kernel, unlike Linux, has surprisingly advanced string manipulation functions. The Kernel RunTime is not only string-capable, but can also handle Unicode, as well.

To initialize a String, you'd use RtlInitString(), which automatically resolves to the ANSI or UNICODE variant, depending on the string initializer value (Unicode strings have an uppercase "L" right before them).

Strcpy and Strcmp are also available, although under different names.

The only caveat to string manipulation in the Kernel is, that Unicode operations cannot take place when running at a higher IRQL level. IRQL is discussed later, but for now we can "get away" with saying that the Unicode functions can only be used in "standard" Kernel scenarios, i.e. when running as a normal priority thread under the dispatcher, and not in any elevated context such as that of an Interrupt Handler.

Interfacing with HAL

- Hardware access is performed via the HAL:
 - READ_REGISTER and WRITE_REGISTER
 - READ_PORT and WRITE_PORT
- Most other functionality obsoleted by I/O Manager

The Hardware Abstraction Layer provides the only interface to the physical, or hardware layer. Kernel components may still use direct calls to I/O and hardware ports, but not using inline assembly sequences.

Most of the HAL exported functionality has been rendered obsolete as the I/O manager has picked up more and more responsibilities, but the HAL still exports macros to read and write register values and/or port values. Specific macros exist for the specific datatypes. For example:

```
USHORT READ_REGISTER_USHORT(IN PUSHORT  Register);  
UCHAR  READ_REGISTER_UCHAR(IN PCHAR    Register );
```

And, for string:

```
VOID READ_REGISTER_BUFFER_UCHAR(IN PCHAR  Register,  
                                IN PCHAR  Buffer,  
                                IN ULONG  Count);
```

And similarly for ports:

```
UCHAR READ_PORT_UCHAR (IN PCHAR  Port);  
VOID WRITE_PORT_ULONG(IN PULONG  Port, IN ULONG  Value);
```

Synchronization - Mutexes

- The Kernel exports three types of Mutexes
 - “classic” mutexes: non performant, effectively deprecated

```
VOID KeInitializeMutex(IN PRKMUTEX mMutex, IN ULONG Reserved);
...
```

- Fast Mutexes

```
VOID ExInitializeFastMutex(IN PFAST_MUTEX fmMutex);
VOID ExAcquireFastMutex(IN PFAST_MUTEX fmMutex);
BOOLEAN ExTryToAcquireFastMutex(IN PFAST_MUTEX fmMutex);
VOID ExReleaseFastMutex(IN PFAST_MUTEX fmMutex);
```

- Guarded Mutexes (Windows 2003 and later)

```
VOID KeInitializeGuardedMutex(IN PKGUARDED_MUTEX gmMutex);
VOID KeAcquireGuardedMutex(IN PKGUARDED_MUTEX gmMutex);
BOOLEAN KeTryToAcquireGuardedMutex(IN PKGUARDED_MUTEX gmMutex);
VOID KeReleaseGuardedMutex(IN PKGUARDED_MUTEX gmMutex);
```

The Kernel supports several synchronization mechanisms for drivers and modules to use. The first is the classic Mutex object, which is available in three varieties:

- **Mutexes:** Using `KeInitializeMutex`, and `KeWaitForMutexObject()`.
- **Fast Mutexes:** Which are implemented by “bumping up” a special thread priority value known as the *IRQL* to a higher level (`APC_LEVEL`) rather than usual (`PASSIVE_LEVEL`). The exact meaning of this is discussed shortly (as *IRQLs* deserve their own notes) – but suffice it to say a Mutex holder will run at a higher priority so long as the mutex remains in its possession. If the mutex is unavailable (i.e. owned by another thread) the requesting thread is suspended until the mutex is released.
- **Guarded Mutexes:** Which have exactly the same interface, but are implemented with “Guarded Sections” which are quicker to enter and leave than the *IRQL* level raising.

The Mutex objects are defined globally in non-paged memory, as `FAST_MUTEX` or `KGUARDED_MUTEX`, respectively. A driver or Kernel component usually follows the Initialize→Acquire→Release pattern, but may also opt to use the `TryToAcquire` functions, that return immediately, if it cannot or will not block execution.

Caution: Attempting to acquire a Mutex object you already own will result in a Bug Check (= Blue Screen of Death) `0xBF: MUTEX_ALREADY_OWNED`.

Synchronization - Events

- Like the Win32 API, the Kernel also supports Events

```
VOID KeInitializeEvent(IN PRKEVENT Event,
                     IN EVENT_TYPE Type,
                     IN BOOLEAN State);

/* To signal an event: */
LONG KeSetEvent(IN PRKEVENT Event,
               IN KPRIORITY Increment,
               IN BOOLEAN Wait);
```

- Synchronization events auto-reset. Notification needs:

```
VOID KeClearEvent(IN PRKEVENT Event);
LONG KeResetEvent(IN PRKEVENT Event);
```

- Wait for one or more events with KeWaitForXXX

The Kernel enables Drivers and components to use Event based synchronization, in an API that is virtually identical to the Win32 API – with good reason – The User mode calls are simply pass through calls to their Kernel implementations.

Two types of events are defined, and the Type parameter of KeInitializeEvent can be either:

- SynchronizationEvent**: for events that are auto resetting “flags”, that may be signalled once (by **KeSetEvent()**) before being reset. These allow a single consumer to awaken, and service consumers one at a time.
- NotificationEvent** : for events/flags which do not reset. As soon as they are signaled, all waiting consumers awaken, and the flag remains until explicitly cleared by **KeClearEvent()** or **KeResetEvent()** (The latter being a slower function, that also recovers the value prior to reset).

Consumers wait for one or more events simultaneously by calling **KeWaitForSingleObject()** or **KeWaitForMultipleObjects()**, respectively.

When signaling an event, it is possible to specify a two parameters: A *priority* increment for threads that have been waiting on it, and a Boolean *wait* Value if the signaling thread immediately wishes to enter a wait state (i.e. call **KeWaitForXXX()**).

Synchronization - Timers

- For delayed execution, or watchdogs, use timers:

```
VOID KeInitializeTimerEx(IN PKTIMER    Timer,
                       IN TIMER_TYPE Type); /* Synch/Notif. */

BOOLEAN KeSetTimer(IN PKTIMER    Timer,
                  IN LARGE_INTEGER DueTime, /* (x 100ns) */
                  IN PKDPC       Dpc OPTIONAL);
```

- I/O Manager offers an automatic, 1HZ timer*:

```
NTSTATUS IoInitializeTimer(IN PDEVICE_OBJECT DeviceObject,
                        IN PIO_TIMER_ROUTINE TimerRoutine,
                        IN PVOID Context);
```

- May be stopped/resumed (IoStopTimer/IoStartTimer)

* - Limit one timer per device, please.

Timers are another useful mechanism the Kernel offers. A driver may set a timer by defining a (global) KTIMER object. This object is opaque, and can be manipulated by calls to the Kernel timer functions. The first, **KeInitializeTimer**, does just that. The Ex variant (shown above) allows to select one of two timer types: **NotificationTimer** or **SynchronizationTimer**, which follow the same principle as Notification and Synchronization Events, discussed previously.

The timer may be set by calling **KeSetTimer** and providing a DueTime argument. The argument may be positive (in which case it is interpreted as an absolute timestamp), or negative (in which case it is considered an offset from the current time when KeSetTimer was called). The Deferred Procedure Call (DPC) supplied as the third argument will be called upon expiry. **KeSetTimerEx** inserts another argument in the third position, *Period*, which is a value in milliseconds the timer will fire at, periodically.

The I/O Manager offers a simple, watchdog oriented timer. Each device object may register a single timer function. The timer function will be called by the I/O manager once every second. This is useful for making sure the driver is still functional, and threads in it have not deadlocked.

The PIO_TIMER_ROUTINE is a pointer to a function implementing the following interface:

```
VOID IoTimer(IN struct DEVICE_OBJECT *DeviceObject,
             IN PVOID Context);
```

With Context being the argument set in the 3rd parameter to **IoInitializeTimer**.

Synchronization - Spinlocks

- Effective Synchronization objects for SMP

```
VOID KeInitializeSpinLock(IN PKSPIN_LOCK SpinLock);  
  
VOID KeAcquireSpinlock();  
VOID KeReleaseSpinLock();  
  
/* XP and later - There are preferred */  
VOID KeAcquireInStackQueuedSpinLock (IN PKSPIN_LOCK SpinLock,  
                                      IN PKLOCK_QUEUE_HANDLE qslHandle);  
VOID KeReleaseInStackQueuedSpinLock (IN PKLOCK_QUEUE_HANDLE qslHandle);
```

- Initialize Spinlocks from non-paged areas only
- Consider Try functions, whenever possible

Spinlocks are thus called because threads “spin” while trying to acquire them – that is, run in a tight loop. In an SMP environment, this makes sense, as the spinlocks are generally held for very short time periods, which do not merit having the thread lose execution rights.

In Windows XP and later, Queued Spin Locks were introduced. These, allegedly, provide for better performance, and deprecate the “classic” SpinLocks. Queued Spin Locks work are also fairer than their predecessors – as they are implemented in a FIFO, guaranteeing acquisition in the order of calls to Acquire..().

Acquiring a queued SpinLock is only slightly more troublesome than a normal one: The driver needs to additionally allocate and pass a KLOCK_QUEUE_HANDLE structure.

As with all synchronization objects, all Spinlock data must be allocated on non paged data, as a Spinlock absolutely cannot trigger a page fault This means the memory should be allocated from the NonPagedPool using ExAllocatePoolWithTag..

IRQLs

- The Kernel maintains Interrupt Request Levels for threads
- Threads with low IRQL may be preempted for higher ones
- Each processor maintains its own IRQL

IRQL	x86	x64	Use
PASSIVE	0	0	User threads & default Kernel mode
APC	1	1	APC, Page faults
DISPATCH	2	2	Thread scheduler & DPC
DIRQL	3-26	3-11	Devices (= Interrupt handlers)
CLOCK2	28	--	Clock timer
SYNCH	28	13	SMP – Instruction Stream Sync
IPI	29	14	SMP – Interprocessor (Cache)
POWER	30	15	UPS Power Failure notification
HIGH	31	15	XP Profiling timer; System failure

A key concept in Kernel mode programming is that of **Interrupt Request Levels**, or **IRQLs**: This is a range of values each processor uses when running threads, in either Kernel or User mode, to enable or disable preemption as necessary. The values start at the basic PASSIVE level (0) and go all the way up to the HIGH level (31), with a simple but important policy: A thread running at a given IRQL, call it n , will be preempted for any thread that becomes runnable with an IRQL of $n+1$.

Most threads run in the PASSIVE level. Being at level 0, this means they can be preempted for pretty much any other thread on the system that is non-Passive. However, since most Kernel mode threads also run at PASSIVE, this doesn't happen all too much.

The levels above PASSIVE are reserved for very specific use cases:

APC: is reserved for Asynchronous Procedure Calls (callbacks) and page faults. The former must be handled as soon as possible, and will temporarily preempt other threads. The latter also need "immediate gratification", as the appropriate page must be fetched for the thread to continue its proper execution. Fast Mutexes are also implemented by an IRQL change to this level.

DISPATCH: is the level in which the Thread Scheduler itself executes. Deferred procedure calls (DPCs) also execute at this level, since they are handled by the scheduler. Code here CANNOT wait for objects since the code will not be preempted by the dispatcher if it blocks.

Caution: Code running at IRQL_DISPATCH or above **CANNOT:**

- Block
- Wait for any non zero amount of time
- Trigger a page fault (because the Page swap occurs at IRQL_APC)
- Release a spinlock (**KeReleaseSpinLock**) not acquired (i.e. called **KeAcquireSpinLock**)
- Acquire a spinlock if already running at this level – use **KeAcquireSpinLockAtDpcLevel()** or **KeAcquireInStackQueuedSpinLockAtDpcLevel()** instead.
- Format Unicode (this includes calling DbgPrint/DbgPrintEx with Unicode % specifiers)

DIRQL: is reserved for Interrupt Handlers (also called Interrupt Service Routines, or ISRs). These are architecture dependent, and for the x86 architecture are reserved at 3-26 (mapping to IRQs 0-15 and then some), or when the driver calls **KeSynchronizeExecution** (which, in turn, calls SynchCritSection).

Higher IRQLs are usually dangerous territory you do *not* want to find yourself in. 28+ interferes with the system timer itself, SMP and power management. Most Kernel code runs at IRQL_PASSIVE, and that's the recommended way of going about things.

A Good reference on IRQLs can be found in the Microsoft White Paper “**Scheduling, Thread Context, and IRQL**” (downloadable from Microsoft.com).

Changing your IRQL

- You really shouldn't need this.. But..

```
KIRQL KeGetCurrentIrql(VOID);  
VOID KeRaiseIrql(IN KIRQL NewIrql,  
                OUT PKIRQL OldIrql);  
  
VOID KeLowerIrql(IN KIRQL NewIrql);
```

- Play at your own risk, but remember you **CANNOT**:
 - RaiseIRQL to a lower IRQL than current
 - Call LowerIRQL on an IRQL that was not previously raised
- **IRQL_NOT_LESS_OR_EQUAL** is a common BSOD
 - Caused by memory faults (paged/non paged)
 - Buggy Drivers messing with their IRQs.

Normally, you should be happy at your own IRQL, and would not need to change it in any way. That said, the Kernel does expose interfaces to get and set the IRQL if required.

Caution: Raising your IRQL can have severe impact on system performance and stability, ***especially*** when raised above IRQL_DISPATCH – since this, effectively disables any scheduling by the Thread Scheduler – which will not get to execute, as it would be of lesser IRQL and priority!

The IRQL_NOT_LESS_OR_EQUAL Blue Screen of Death is commonly the result of executing in the wrong IRQL – greater than APC_LEVEL (i.e. DISPATCH_LEVEL or above) and accessing paged (or invalid) memory. The Page fault that occurs cannot be serviced by the system pager, that is designed to run as the lower APC_LEVEL.

SpinLocks ↔ DeadLocks

- Incorrect usage of Spinlocks leads to deadlocks, or worse
 - Initialize Spinlocks ONLY on non-paged data
 - Don't trigger page faults, hardware or software exceptions
 - You cannot release a SpinLock you have not acquired
 - IRQL = DISPATCH_LEVEL requires AcquireAtDPC() calls
 - Queued/Classic calls cannot be combined
 - Spinlocks are NOT recursive
 - Multiple Spin Locks, if needed, should be called in same order
- Holding a Spinlock will mutually exclude:
 - Other code waiting for same spinlock – on all CPUs
 - Code at a lower IRQL than that of spinlock holder on same CPU
- Hold spinlocks for as little as required (< 25mS)

Naturally, all these rules also apply to calling external functions. A common mistake made by Kernel coders is adhering to these rules, but calling some external function that does not.

Good references on using Spinlocks properly: <http://go.microsoft.com/fwlink/?LinkId=57456> and <http://msdn.microsoft.com/en-us/library/aa490225.aspx>.

DbgPrint

- Kernel drivers can print debug output, when required

```
ULONG DbgPrint(IN PCHAR Format,
               ... [arguments]);

ULONG DbgPrintEx(IN ULONG ComponentId,
                 IN ULONG Level,
                 IN PCHAR Format,
                 ... [arguments]);
```

- Usable at IRQL <= DIRQL – unless formatting Unicode
- Use KdPrint, KdPrintEx for both checked/free builds
- View output with Kernel Debugger, or DbgView
- XP/Vista need registry enablement for each component

Much like Linux has its “printk” for printf() like output in the Kernel, so does Windows with DbgPrint()/DbgPrintEx(). DbgPrint is used to print out messages that are normally ignored, unless a Kernel Debugger is attached – in which case the messages can be read. The usage is straightforward – use it exactly as you would printf(). DbgPrintEx() adds two arguments – ComponentId and Level. DbgPrint(*Format, arguments*) is exactly equivalent to **DbgPrintEx (DPFLTR_DEFAULT_ID, DPFLTR_INFO_LEVEL, *Format, arguments*);**

Component IDs are defined as follows:

Constant	Purpose
IHVVIDEO	Video driver
IHVAUDIO	Audio driver
IHVNETWORK	Network driver
IHVSTREAMING	Kernel streaming driver
IHVBUS	Bus driver
IHVDriver	Any other type of driver

Level is anywhere between 0-31 (which is actually bit-shifted by the OS), or 32-0xFFFFFFFF.

Setting the Component and the Level is useful for Kernel Debuggers with filtering capabilities.

DbgPrint()s are available in most Kernel code – for IRQLs less than or equal to than DIRQL. Looking back at the IRQL notes, you can see that this would cover almost all Kernel code – including Interrupt Handlers – but not SYNCH, CLOCK2, POWER, IPI or HIGH. Chances are, however, your Kernel code won't go anywhere near these IRQLs anyway, so you should be fine. Calling at an IRQL greater than DIRQ risks causing a Kernel deadlock – so be warned.

Caution: DbgPrint() is that it is so like printf() you could find yourself printing out debug messages that contain Unicode strings (%S, %ls, %C, %lc, %ws, %wc and %wZ) – and that's something you can do only if the IRQL is IRQL_PASSIVE.

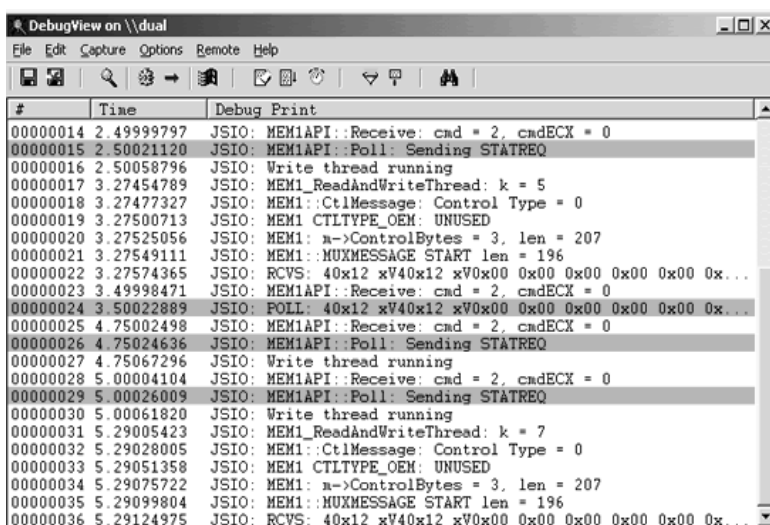
If you compile the same code for a Checked and a Free build, use KdPrint and KdPrintEx, respectively. These are macros that expand normally in a checked build, but compile out in a free build.

In XP, Vista and later, a specific registry key needs to be created:

HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter.

In which a DWORD value needs to be further be defined for each component ID (usually DEFAULT suffices) and mask (usually 0xFFFFFFFF) to enable DbgPrint messages to be sent.

To view Debug messages, either attach a Kernel Debugger, or – better yet – use DebugView from the former SysInternals (<http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx>).



The screenshot shows the DebugView application window titled "DebugView on \\dual". The window has a menu bar (File, Edit, Capture, Options, Remote, Help) and a toolbar with icons for file operations, search, and capture. The main pane displays a list of debug print messages with columns for #, Time, and Debug Print. The messages are from the JSIO driver and include various operations like Receive, Poll, Send, and Write thread running.

#	Time	Debug Print
00000014	2.49999797	JSIO: MEM1API::Receive: cmd = 2, cmdECC = 0
00000015	2.50021120	JSIO: MEM1API::Poll: Sending STATREQ
00000016	2.50058796	JSIO: Write thread running
00000017	3.27454789	JSIO: MEM1_ReadAndWriteThread: k = 5
00000018	3.27477327	JSIO: MEM1::CtlMessage: Control Type = 0
00000019	3.27500713	JSIO: MEM1_CTLTYPE_OEM: UNUSED
00000020	3.27525056	JSIO: MEM1: n->ControlBytes = 3, len = 207
00000021	3.27549111	JSIO: MEM1::MUXMESSAGE START len = 196
00000022	3.27574365	JSIO: RCVS: 40x12 xV40x12 xV0x00 0x00 0x00 0x00 0x00 0x...
00000023	3.49998471	JSIO: MEM1API::Receive: cmd = 2, cmdECC = 0
00000024	3.50022889	JSIO: POLL: 40x12 xV40x12 xV0x00 0x00 0x00 0x00 0x00 0x...
00000025	4.75002498	JSIO: MEM1API::Receive: cmd = 2, cmdECC = 0
00000026	4.75024636	JSIO: MEM1API::Poll: Sending STATREQ
00000027	4.75067296	JSIO: Write thread running
00000028	5.00004104	JSIO: MEM1API::Receive: cmd = 2, cmdECC = 0
00000029	5.00026009	JSIO: MEM1API::Poll: Sending STATREQ
00000030	5.00061820	JSIO: Write thread running
00000031	5.29005423	JSIO: MEM1_ReadAndWriteThread: k = 7
00000032	5.29028005	JSIO: MEM1::CtlMessage: Control Type = 0
00000033	5.29051358	JSIO: MEM1_CTLTYPE_OEM: UNUSED
00000034	5.29075722	JSIO: MEM1: n->ControlBytes = 3, len = 207
00000035	5.29099804	JSIO: MEM1::MUXMESSAGE START len = 196
00000036	5.29124975	JSIO: RCVS: 40x12 xV40x12 xV0x00 0x00 0x00 0x00 0x00 0x...

Writing to the Event Logs

- Drivers may also write to the system log:

```
PVOID IoAllocateErrorLogEntry(IN PVOID IoObject,
                             IN UCHAR EntrySize);

VOID IoWriteErrorLogEntry(IN PVOID ElEntry);
```

- PVOID is actually a PIO_ERROR_LOG_PACKET
- IoAllocate(..), populate fields, then IoWrite(..)
- If not writing, must free by calling IoFreeErrorLogEntry

```
VOID IoFreeErrorLogEntry(IN PVOID ElEntry);
```

Another option for communicating with the outside world is by writing to the System's Event Log. This is almost as simple as user-mode's LogEvent API:

Begin by allocating an Error Log Entry. This is done by calling IoAllocateErrorLogEntry. The arguments here are the PDEVICE_OBJECT or PDRIVER_OBJECT reporting the error, and a size for the entry (as a UCHAR – up to 255 bytes and under ERROR_LOG_MAXIMUM_SIZE).

Despite the API definition, the pointer returned is actually a PIO_ERROR_LOG_PACKET:

```
typedef struct _IO_ERROR_LOG_PACKET {
    UCHAR MajorFunctionCode;
    UCHAR RetryCount;
    USHORT DumpDataSize;
    USHORT NumberOfStrings;
    USHORT StringOffset;
    USHORT EventCategory;
    NTSTATUS ErrorCode;
    ULONG UniqueErrorValue;
    NTSTATUS FinalStatus;
    ULONG SequenceNumber;
    ULONG IoControlCode;
    LARGE_INTEGER DeviceOffset;
    ULONG DumpData[1];
} IO_ERROR_LOG_PACKET, *PIO_ERROR_LOG_PACKET;
```

The `IO_ERROR_LOG_PACKET` fields are initialized by the driver, as shown in the following example:

```
VOID LogEvent(NTSTATUS code, PWSTR userString, PDEVICE_OBJECT fdo)
{
    ULONG packetlen = (wcslen(userString) + 1) * sizeof(WCHAR)
        + sizeof(IO_ERROR_LOG_PACKET);

    PIO_ERROR_LOG_PACKET p = (PIO_ERROR_LOG_PACKET)
        IoAllocateErrorLogEntry(fdo, (UCHAR) ERROR_LOG_MAXIMUM_SIZE);

    if (!p) {
        DEBUGP (MP_ERROR, ( "Can't write to Event Log\n"));
        return; }

    memset(p, 0, sizeof(IO_ERROR_LOG_PACKET));
    p->ErrorCode = code;

    /* Optional "Dump Data may be set */
    p->DumpDataSize = 1;
    p->DumpData[0] = '\0';

    /* Strings may be set – these correlate to Message file "%1" entries.
    p->StringOffset = sizeof(IO_ERROR_LOG_PACKET) + p->DumpDataSize;
    p->NumberOfStrings = 1;

    /* Copy strings – This example demonstrates only one user supplied String */
    wcscpy((PWSTR) ((PUCHAR) p + p->StringOffset), userString);

    IoWriteErrorLogEntry(p); /* No need to free */

} /* end LogEvent */
```

The real trick, however, is to prepare a “message file” for the driver. This is a separate file, with a “.mc” extension, that looks something like the example on the next page. This file is compiled into a resource script using the “mc” tool, which in turn creates an .rc file, and a corresponding .h file. The .rc is then added to the driver SOURCES.

The MC file format is described in <http://msdn.microsoft.com/en-us/library/aa489593.aspx>. Here is a sample file:

MessageIdTypedef = NTSTATUS

SeverityNames = (

Success = 0x0:STATUS_SEVERITY_SUCCESS
Informational = 0x1:STATUS_SEVERITY_INFORMATIONAL
Warning = 0x2:STATUS_SEVERITY_WARNING
Error = 0x3:STATUS_SEVERITY_ERROR
)

FacilityNames = (

System = 0x0
Eventlog = 0x2A:FACILITY_EVENTLOG_ERROR_CODE
)

LanguageNames = (

English = 0x0409:msg00001
French = 0x040C:msg00003
)

MessageId = 0x0001

Facility = Eventlog
Severity = Informational
SymbolicName = DRIVER_MSG_INIT
Language = English
NdisCom Driver is loaded. Embed strings with %1, %2, %3. End with a "." on a line by itself

.

MessageId = 0x0002

Facility = Eventlog
Severity = Error
SymbolicName = DRIVER_MSG_SOME_OTHER
Language = English
Example of another message

.

Finally, to enable the Windows NT Event Log Viewer to display the messages, add a registry value for your driver, under:

HKLM\SYSTEM\CurrentControlSet\Services\Eventlog\System\<driverName>\EventMessageFile
 pointing to your .sys file. Otherwise, The Event Log viewer will display messages like:

"The description for Event ID (10) in Source (driverName) cannot be found. The local computer may not have the necessary registry information or message DLL files to display messages from a remote computer. You may be able to use the /AUXSOURCE= flag to retrieve this description; see Help and Support for details."

Crash and Burn

- To crash the system with a BSOD:

```
VOID KeBugCheckEx (IN ULONG   BugCheckCode,  
                  IN ULONG_PTR BugCheckParameter1,  
                  IN ULONG_PTR BugCheckParameter2,  
                  IN ULONG_PTR BugCheckParameter3,  
                  IN ULONG_PTR BugCheckParameter4) ;
```

- Codes 0x01-0x12C (as well as 0xDEADDEAD ☺)
 - Manual crash: 0xE2 – Ctrl-ScrlLk/ScrlLk
- List of Bug Check Codes:
 - <http://msdn.microsoft.com/en-us/library/ms789516.aspx>
- May also register a bug check Call Back

You are encouraged to handle exceptions in your driver by using structured exception handling (i.e. `__try/__except/__finally` blocks) whenever possible. But when a driver detects some horrendous, catastrophic, uncorrectable error that compromises system integrity, sometimes the only way to go is down – by crashing the system. In the UNIX world this is a panic situation – and Windows calls this a BugCheck.

BugChecks are more commonly known as Windows “Blue Screens of Death”, and probably need no introduction (you’re truly exceptional if you’ve never seen one ☺). These screens are the last thing Windows displays before the system is halted, and usually rebooted.

The only required argument for a BugCheck is the BugCheckCode, which is usually one of the documented MSDN codes (at <http://msdn.microsoft.com/en-us/library/ms789516.aspx>). The code will be translated to its #define name and displayed in the Blue Screen, with an additional line for the four parameters. There are over 250 codes, and more are added with every release of Windows, so any attempt to explain them all would almost immediately be outdated. Still, the following table lists some of the common ones you’re likely to encounter:

Code	#define	Meaning
0x0A 0xD1	IRQL_NOT_LESS_OR_EQUAL DRIVER_IRQL_NOT_LESS_OR_EQUAL	Attempt by Kernel (or by device driver) to access paged/invalid memory at an IRQL >= DISPATCH_LEVEL. Parameters are: (Addr, IRQL, 0=read/1=write, EIP at fault)
0x1E	KMODE_EXCEPTION_NOT_HANDLED	Exception that wasn't caught in a __try/__catch. E.g. from ProbeForRead()
0x24	NTFS_FILE_SYSTEM	Error in NTFS.sys. Usually due to bad sectors
0x41	MUST_SUCCEED_POOL_EMPTY	A Must Succeed Allocation didn't..Parameters: (Request Size, # Pages, .., # pages avail)
0x50	PAGE_FAULT_IN_NON_PAGED_AREA	Attempt to access invalid system memory.Parameters: (Addr, 0=read/1=write, EIP at fault, Reserved)
0x7E	..THREAD_EXCEPTION_NOT_HANDLED	Usually, your driver's fault: Arguments: (Exc Code, Address of Exc, Exc Rec, Context Rec) (use .exr on arg3, .cxr on arg4)
0xC8	IRQL_UNEXPECTED_VALUE	IRQL changed by some driver, but not restored.
0xE2	USER_GENERATED	User pressed Ctrl-Scroll (twice) and registry is configured for dumps (HKLM\System\CCS\i8042prt\Parameters] "CrashOnCtrlScroll"=dword:00000001
0x109	CRITICAL_STRUCTURE_CORRUPTION	PatchGuard (Vista) reporting suspected patching of Kernel. Parameters are: (0,0,0,corruption) where: 0 = Generic Data 2=IDT 3=GDT 4,5=Process List 6=Debug Routine 7=MSR

It's also possible for a driver to register a Bug Check callback function, for post-dump processing. This is done by the following steps:

1. Initialize a Callback Record

```
VOID KeInitializeCallbackRecord(IN PKBUGCHECK_CALLBACK_RECORD CallbackRecord);
```

2. Register the call back:

```
BOOLEAN KeRegisterBugCheckCallback
```

```
(IN PKBUGCHECK_CALLBACK_RECORD CallbackRecord,  
IN PKBUGCHECK_CALLBACK_ROUTINE CallbackRoutine,  
IN PVOID Buffer,  
IN ULONG Length,  
IN PCHAR Component);
```

3. Implement the call back:

```
VOID BugCheckCallback(IN PVOID Buffer,  
IN ULONG Length);
```

Exercises

1. In this exercise we will utilize SysInternals' "LiveKD" extension to The Windows Debugger to view the behind-the-scenes implementation of Kernel IRQLs and Spinlocks, thereby learning one or two important things.. Follow these steps:

- i. Start LiveKD:

- a) Make sure you are running as an Administrator. If not, use the "runas" command to start a command prompt (cmd.exe)
 - b) Make sure to set your Symbol path correctly, using the environment variable `_NT_SYMBOL_PATH`. The easiest way to do that is to use the DOS "subst" command to assign a logical drive, say K:, to the LiveKD directory, and set `_NT_SYMBOL_PATH` to K:\Symbols.

- ii. Unassemble HAL's KeRaiseIrql and KeLowerIrql. How are they implemented? Specifically, Where is the IRQL value stored in memory? Make note of this address. Verify this by unassembling KeGetCurrentIrql.

- iii. Unassemble hal!KeAcquireSpinLock, and follow the trace. How is the spin lock acquired? How does that affect the IRQL?

- iv. Next, Unassemble NT's function for SpinLocks at the IRQL of Dispatch - nt!KeAcquireSpinLockAtDpcLevel and nt!KeReleaseSpinLockFromDpcLevel. How are they implemented? Can you explain why?

- v. Why are the IRQL function implemented inside HAL, with the exception of the DPCLevel ones? How would a different HAL, e.g. SMP vs UP, be different?

Exercises (cont)

2. In this exercise we will examine the difference between the Zw* functions and their Rtl counterparts. Again, using LiveKD, Unassemble RtlDeleteRegistryValue and nt!ZwDeleteValueKey:

```
kd> u nt!ZwDeleteValueKey
nt!ZwDeleteValueKey:
804dcbd0 b841000000      mov     eax,41h
804dcbd5 8d542404      lea     edx,[esp+4]
804dcbd9 9c           pushfd
804dcbda 6a08         push    8
804dcbdc e8501a0000   call    nt!KiSystemService (804de631)
804dcbe1 c20800      ret     8
```

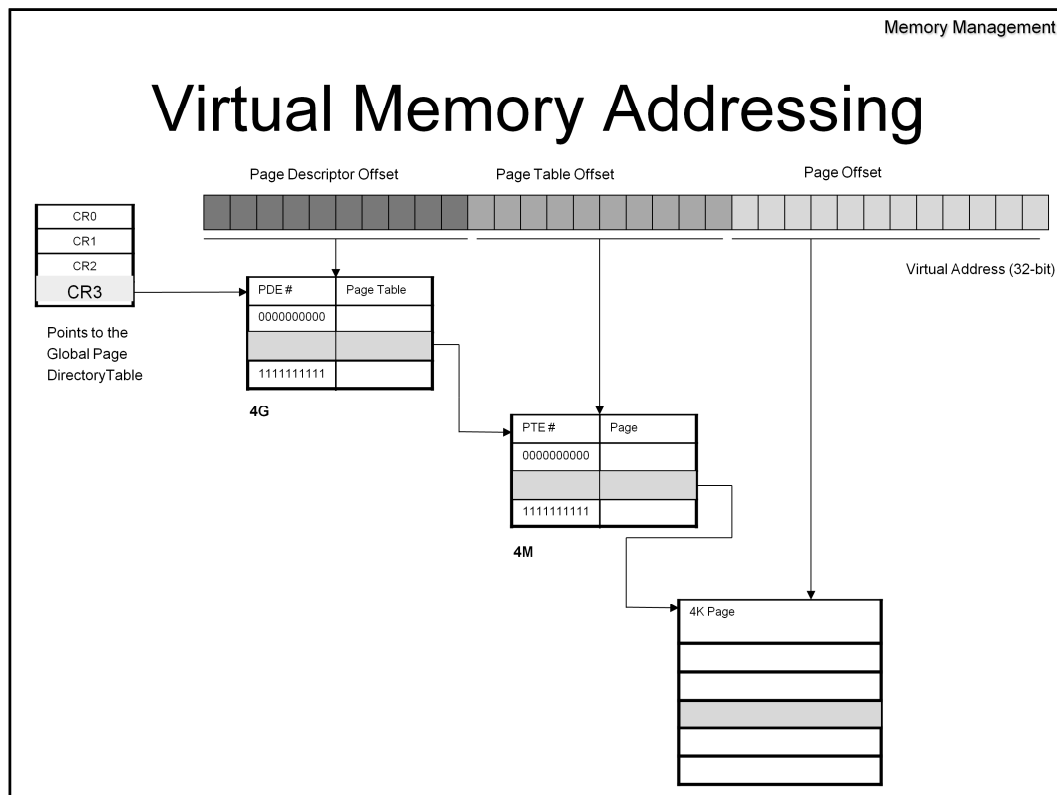
```
kd> u ntdll!RtlDeleteRegistryValue
ntdll!RtlDeleteRegistryValue:
7c933da0 8bff         mov     edi,edi
7c933da2 55           push    ebp
7c933da3 8bec         mov     ebp,esp
7c933da5 51           push    ecx
7c933da6 51           push    ecx
7c933da7 8d450c       lea     eax,[ebp+0Ch]
7c933daa 50           push    eax
7c933dab 6a01         push    1
kd> u
( .. More )
```

What's the difference between the two functions? Unassemble several lines to figure this out.

Memory Management

This section describes the Windows Memory Management mechanism, and explains how low level operations using the Mm* API work

Key Concepts: Virtual Memory, MDL



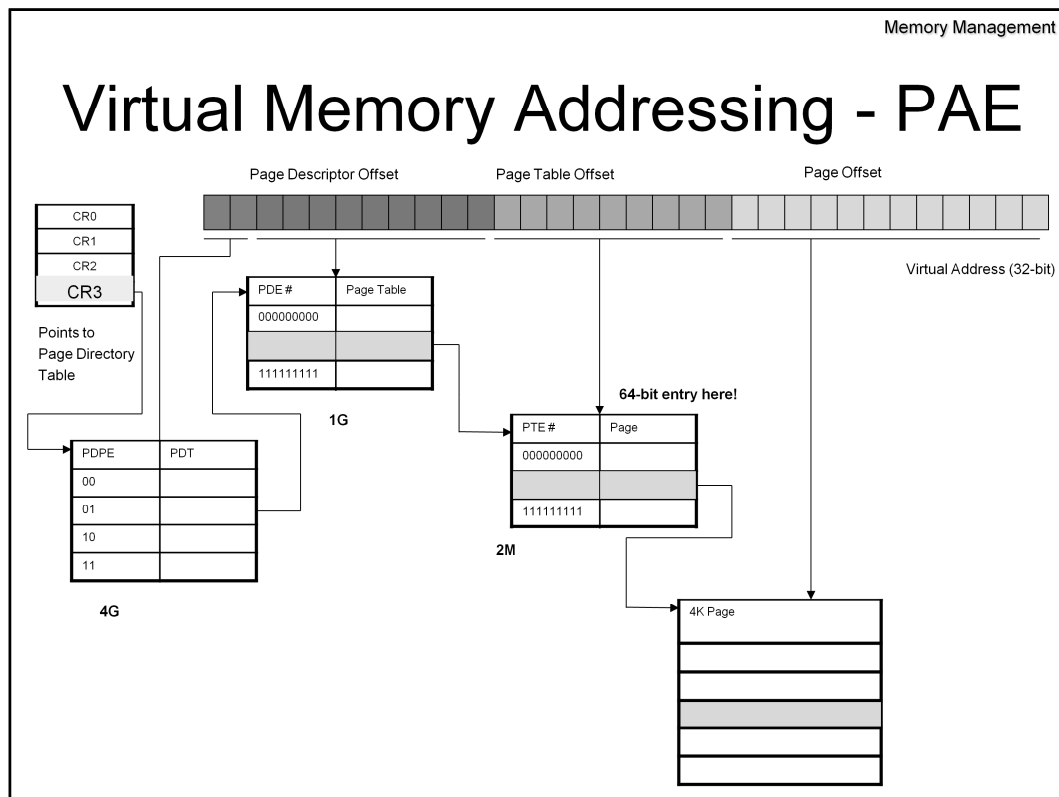
The translation of Virtual Addresses into physical ones is a three staged process. Given a 32-bit address, The CPU segments the address into three separate parts:

The first 10 bits – point to one of 2^{10} entries in a global **Page Directory Table**. This table is, in effect, a table of page tables, and the 10 bits select a specific page table index by a **Page Directory Entry** or **PDE**. This table is defined per process, and maintained in a Page Descriptor Base Register, which on the Intel architectures is Control Register #3 (CR3). This register is reloaded on each process context switch from the KPROCESS object, since each process has a different virtual memory image.

The next 10 bits – point to a specific page (a.k.a **Page Table Entry** - PTE) in the **Page Table** that was selected by the previous 10 bits. 10 bits again mean 2^{10} – so each page table maintains the addresses of 4 MB ($=2^{10} * 4KB$) of memory.

The last 12 bits – are the specific offset in the page itself. Since the page itself is 4KB ($=4096$ bytes) this works out perfectly with 4096 being 2^{12} . However, most addresses are aligned on a DWORD boundary, which allows the system to reserve the last two bits for its own internal use.

Each page table maintains 4MB, and there are 2^{10} tables in the Page Descriptor Table – so $2^{10} * 4MB = 4GB$, which is the size of the virtual address space of the process. Things look somewhat different when Physical Address Extensions* (PAE) are employed, but are sufficiently similar – as is shown next.



Intel's Physical Address Extensions (PAE) extend Virtual Memory addressing to systems with more than 4GB of physical memory. Because of the limitations of 32-bits, this isn't as simple as it seems. Given a 32-bit address, The CPU now segments the address into four, not three separate parts:

CR3 – Now points to a table of 4 ($=2^2$) entries, called the **Page Directory Pointer Table**.

The first 2 bits – point to one of the four entries in the Page Directory Pointer Table – which will serve as the usual **Page Descriptor Table**.

The next 9 bits – point to one of 2^9 entries in the **Page Descriptor Table**. Remember that this is one of four tables. However, each PDE is now 64-bits. Note the size of the table is the same, because $2^9 * 2^6 = 2^{10} * 2^5$.

The next 9 bits – point to a specific page (this is the **Page Table Entry - PTE**) in the **Page Table** that was selected by the previous 9 bits. This page is, again, 64-bits – and 9 bits again mean 2^9 – so each page table maintains the addresses of 2 MB ($=2^9 * 4\text{KB}$) of memory. However, the address here **can be up to 64-bits – allowing for physical addresses over 4GB**.

The last 12 bits – are the specific offset in the page itself. Since the page itself is 4KB ($=4096$ bytes) this works out perfectly with 4096 being 2^{12} . However, most addresses are aligned on a DWORD boundary, which allows the system to reserve the last two bits for its own internal use.

Each page table maintains 2MB, and there are 2^9 tables in the Page Descriptor Table – so $2^9 * 2\text{MB} = 1\text{GB}$ - But there are 4 PDE tables – so we're back to the 4GB of memory.

Virtual Memory Addresses

- Revisit the last page, and you'll see a waste of 12 bits
- Those 12-bits are redefined by the OS for valid pages:



Flag	Meaning
Global	Page belongs to Kernel, and is thus global across all processes
Dirty	Page has been modified and cannot be reused until committed
Accessed	Page has been recently accessed (for LRU "clock" algorithm)
Cache Disable	Page may not be cached
Write Through	Write this page to disk (disables write caching)
Owner	User-mode (Ring 3) page or Kernel-Mode (Ring 0) page
Writable	Is page writable or read only
Valid	Page is a valid page, mapping to a physical. Always set to "1"

If you did the math on the last pages, you might have noticed something a little bit troubling:

- Page addresses must start on a page boundary.
- Pages are 4KB in size
- $4KB = 4,096 = 2^{12}$.
 - ➔ Page addresses have their 12 lowermost bits always set to 0
 - ➔ Page Table Entries are effectively only 20 bits out of the 32-bits.

This means that, indeed, using 32-bits for the Page Table Entries would be wasteful – after all, the last 12 bits would be unused! The system therefore redefines the Page Table Entry to be two parts: The first 20-bits, which are the actual physical address of the page (called "**Page Frame Number**" or **PFN**), and the last 12 bits, that are used as flags, as shown above.

When using PAE, page directory entries have two more bits: 63 – NX (No Execute) – to defeat buffer overflow attacks, and bit 7- PS – to allow for 2MB pages rather than Page table entries.

64-bit Addressing

- 16TB/16EB Address space, 128GB System PTEs
- 64-bit addressing extends PAE:
 - Page sizes are 4K, 2M, or 1G(!)
 - 4 Levels:
 - PDPE – from 2 bits to 9
 - New Level 4 table – also 9 bits
- Actual addresses are currently 48 bits:
 - Addresses: User mode: 0-‘7FFFFFFFFFFFFF
Kernel mode: FFFF80000000-FFFFFFFFFFFFFFFFFFFF

(leaving a “hole” in the middle due to sign-extension)

From <http://support.microsoft.com/kb/294418>:

Limitation	On 32-bits	On 64-bits
VM	4GB	16TB
PTEs	660MB	128GB
Cache	1GB	1TB
Paged Pool	470MB	128GB
Non Paged Pool	256MB	128GB

Memory Management APIs

- The Kernel offers two memory management APIs:
 - High level allocation, using “Pools”
 - No Physical/Virtual mess.
 - Limited types of memory, pre-allocated and managed by system
 - Direct allocation – using “Memory Descriptor Lists”
 - Finer, low-level control of pages
 - More complicated

Memory Operations

- Windows Defines several “Pools” of memory for allocations:
 - **Non Paged Pools**: Memory pages are always resident. Small.
 - **Paged Pools**: Larger pool, but pages may be swapped out.
- Memory allocated from pools, and may be “tagged”

```
PVOID ExAllocatePoolWithTag(IN POOL_TYPE PoolType,
                           IN SIZE_T NumberOfBytes,
                           IN ULONG AscII7BitTag);

VOID ExFreePool(IN PVOID pPool);
VOID ExFreePoolWithTag(IN PVOID pPool, IN ULONG AscII7BitTag);
```

The Kernel stack is extremely limited: 12K in size. Therefore, most allocation of memory is done explicitly, from one of two “Pools” of memory.

ExAllocatePoolWithTag() is the Windows Kernel version of Linux’s `kmalloc()`. It is very much like any other `malloc()` – in that the Number of Bytes is specified (second parameter), and a void pointer is returned. There are a couple of subtle differences, however:

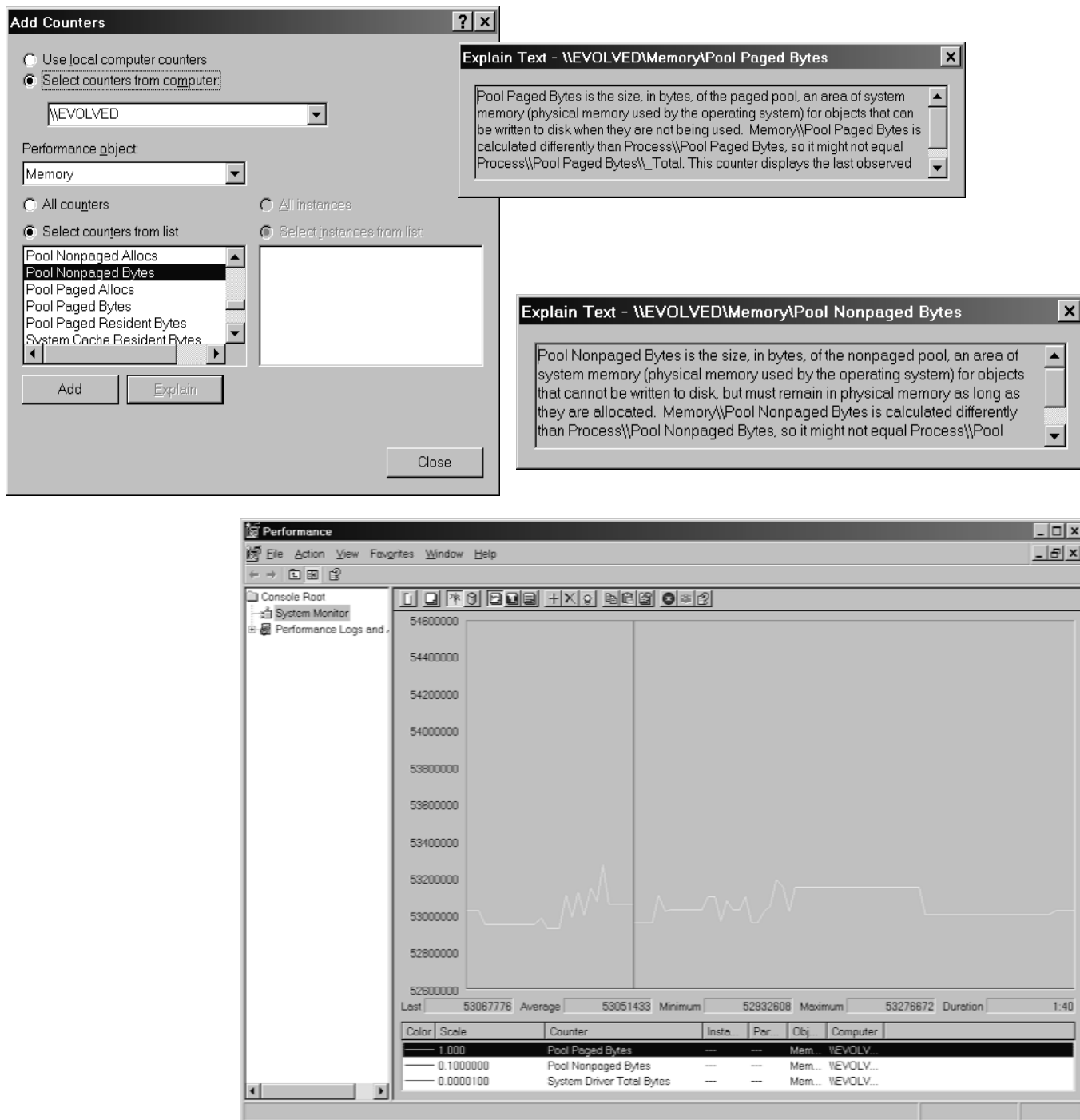
- The Number Of Bytes should be very close to a multiple of the page size. Otherwise, the request is rounded up to the nearest multiple. Windows does not have a slab allocator like Linux for allocations of under a single page.
- Memory may be “tagged” by up to four 7-bit ASCII characters (hence the parameter is defined as a ULONG).. This is useful for debugging only, and has no effect on the memory. In fact, you can just call **ExAllocatePool()**, which tags the last argument as “656E6F4E” (None). Microsoft keeps track of all its drivers’ pools in a file called “pooltag.txt”. WinDBG can use this file when analyzing Kernel dumps.
- Last, but most important, the `POOL_TYPE` parameter is an enum, containing several values – of which the following are usable by drivers:

Pool Type	Purpose
NonPagedPool	Memory that is always resident and never paged out. Always accessible – but considered scarce. Call may fail.
NonPagedPoolMustSucceed	As NonPaged, but if call fails system blue screens with code 0x41.
PagedPool	Normal system memory – not guaranteed to be accessible. May trigger a pagefault. Must be running at a lower priority than dispatcher to access this memory.

Values in HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management - "NonPagedPoolSize" and "PagedPoolSize" control the size of the pools. At any rate, the NonPaged may not exceed 256MB, and the PagedPool - ~492MB (2000/XP) or 650MB (2003). Windows Vista and beyond have dynamic pool sizes.

Experiment:

You can see the two pools by opening up Performance Monitor, and selecting the counters under "Memory". You'll have to play with the scale and graph minimum/maximum settings for best visibility. Then, press ALT-TAB every once in a while to switch between applications, noting what happens.



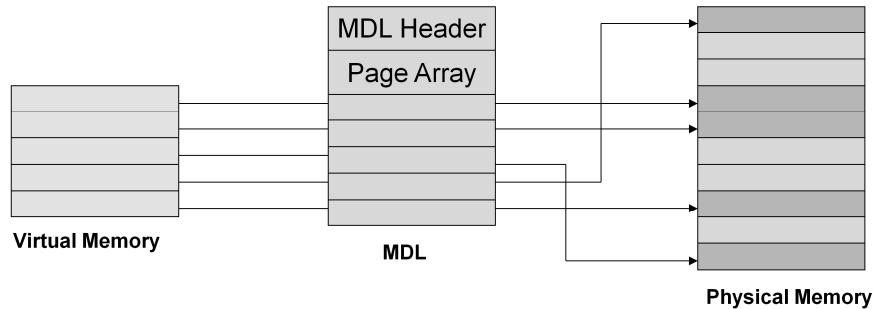
Debugging Pool Allocations

- Debugger Commands:
 - **!pool [Address [flags]]** – Display pool headers for Address
Flags: 0x1 – Contents 0x2 – This pool only
 - **!poolused* [flags [Tag]]** – Display pool allocation by Pool Tag
Flags: 0x1 – verbose 0x2 – Sort NonPaged 0x4 Sort Paged 0x8: Session
 - **!poolfind Tag [PoolType]** – Find pages matching Pool Tag
Pooltype: 0: NonPaged 1: PagedPool 2: Special 4: Session
- OSR's PoolTag: www.osronline.com/article.cfm?article=98

* Requires GFlags Pool Tagging on XP and earlier

Memory Descriptor Lists

- MDLs show the physical representation of virtual buffers
 - Contiguous in Virtual Address, may be fragmented physically



- Structure is documented, but should be treated as opaque
 - Structure defines the header only. Pages are in adjacent array
 - All manipulation of MDL should be through Mm functions/macros

The Windows Kernel maintains and manages its virtual memory by using “**Memory Descriptor Lists**” or **MDLs**. An MDL is a descriptor of a single, virtually contiguous buffer of virtual memory, and its mapping to physical memory pages.

By “virtually contiguous” we mean that, even though the buffer can be treated as a single contiguous range of addresses, this range may be spread over non-contiguous pages in physical memory. Conceptually, this means an MDL might look something like the illustration above. The MDL contains the mapping from the virtual pages to the locked-in-memory physical pages.

The pages are in an array that immediately follows the MDL in memory. I.e. to access them, one can simply increment the MDL header pointer, and cast to a `PPFN_NUMBER`. This can be done “quick and dirty” in code, but the recommended way is to call **MmGetMdlPfnArray()**.

```
/* Quick and dirty, as per DDK header file */
PPFN_NUMBER Pages = (PPFN_NUMBER) (Mdl + 1);
/* Recommended way, preserving opacity */
PPFN_NUMBER pPageDesc = MmGetMdlPfnArray(pMdl);
```

The structure is listed in the WinDDK header files. But here, too, are macros used in the interest of opacity. The definition below, annotated, shows the fields and their macros:

```

typedef struct _MDL {
    struct _MDL *Next;
    CSHORT Size;
    CSHORT MdlFlags;
    struct _EPROCESS *Process; /* Owning process of this MDL */
    PVOID MappedSystemVa;
    PVOID StartVa; /* Beginning of Buffer - use MmGetMdlVirtualAddress */
    ULONG ByteCount; /* sizeof buffer - use MmGetMdlByteCount */
    ULONG ByteOffset; /* First address in buffer - use MmGetMdlByteOffset */
} MDL, *PMDL;
/* Pages immediately follow this header - use MmGetMdlPfnArray here.. */

```

The virtual memory buffer may or may not be aligned on a page boundary. Further, it may span any number of pages, and not necessarily fill its last page. The MDL thus contains two properties, shown in the structure above: ByteCount (accessible by **MmGetMdlByteCount()**) – which is the size of the buffer, and ByteOffset (accessible by **MmGetMdlByteOffset()**), which is the offset of the buffer start from the first page boundary. In a sense, the virtual address this MDL describes can be thought of as $\text{StartVa} \mid \text{ByteOffset}$, since StartVa is guaranteed to be a 20-bit address – as it is the address of a page, and ByteOffset is necessarily inside a page, thus under the size of one, which – if you recall – is 4KB on intel architectures, and thus in the least significant 20 bits.

Next	
Size	MdlFlags
Process	
MappedSystemVa	
StartVa	
ByteCount	
ByteOffset	

A ByteOffset of 0 means the address is aligned on a page boundary. Similarly, a ByteCount divisible by 4KB means that the buffer spans an integer multiple of whole pages. Since more often than not, however, that is not the case, the ADDRESS_AND_SIZE_TO_SPAN_PAGES macro can be used to calculate the number of the entries in the array. This macro takes two arguments (surprisingly enough, the address and size):

```

ULONG ADDRESS_AND_SIZE_TO_SPAN_PAGES (IN PVOID Va,
                                       IN ULONG Size);

```

And returns a ULONG which is the size of the array. So using it on a particular MDL would look like so:

```

arraySize = ADDRESS_AND_SIZE_TO_SPAN_PAGES (MmGetMdlVirtualAddress(pMdl),
                                             MmGetMdlByteCount(pMdl));

```

Working with MDLs

I. Creating MDLs

- Create an MDL with:

```
PMDL IoAllocateMdl(IN PVOID    VirtualAddress,
                  IN ULONG     Length,
                  IN BOOLEAN    SecondaryBuffer,
                  IN BOOLEAN    ChargeQuota,
                  IN OUT PIRP   Irp OPTIONAL);
```

- An MDL may be reused:

```
VOID MmInitializeMdl(IN PMDL MemoryDescriptorList,
                   IN PVOID BaseVa,
                   IN SIZE_T Length);
```

- And, eventually, freed:

```
VOID IoFreeMdl(IN PMDL Mdl);
```

MDLs may be allocated by calling **IoAllocateMdl()**. This function (exported by the I/O Manager) is the preferred way of creating a new MDL (the other, deprecated way being **MmCreateMdl()**). The MDL is allocated from non-paged memory (since it, itself, describes paged memory and therefore cannot be paged). The function takes the following arguments:

```
PMDL IoAllocateMdl(IN PVOID    VirtualAddress, /* start addr of buffer */
                  IN ULONG     Length,        /* Length of buffer */
                  IN BOOLEAN    SecondaryBuffer, /* for IRPs, else FALSE */
                  IN BOOLEAN    ChargeQuota,    /* charge user memory quota? */
                  IN OUT PIRP   Irp OPTIONAL); /* IRP to assoc. MDL with */
```

The first two parameters are straightforward: *VirtualAddress* and *Length* initialize the MDL's *StartVa* and *ByteCount* fields, respectively. The fourth parameter, *ChargeQuota*, is used to charge the MDL virtual memory to the owning process/thread's quota allowance.

The third parameter, *SecondaryBuffer*, only has meaning if the fifth parameter (*Irp*) is not null. If this MDL is associated with an I/O Request Packet (IRP), it may be a primary buffer, or (one of potentially several) secondary buffers. Every IRP has a list of MDLs, and the I/O manager adds the MDL to the IRP's list – at the head of the list (for a primary buffer) or at its tail (for a secondary).

Even though the MDL typedef only accounts for the header, recall that the actual structure allocated in memory is comprised of the header and the list of physical pages, that follows it. This means that the actual memory allocation by the I/O manager accounts for that, satisfying the following formula:

$$(\text{sizeof}(\text{MDL}) + \text{sizeof}(\text{PFN_NUMBER}) * \text{ADDRESS_AND_SIZE_TO_SPAN_PAGES}(\text{BaseVa}, \text{Length}))$$

That is, the size of the MDL header, plus the size of the page lists that follow. In the interest of opacity, the function **MmSizeOfMdl()** can be used to perform this calculation, and will return the size of the MDL required to hold the address.

```
ULONG MmSizeOfMdl(IN PVOID Base,
                  IN SIZE_T Length);
```

We can now attempt to construct the pseudo code for **IoAllocateMdl()**:

```
PMDL IoAllocateMdl(IN PVOID VirtualAddress, /* start addr of buffer */
                  IN ULONG Length, /* Length of buffer */
                  IN BOOLEAN SecondaryBuffer, /* for IRPs, else FALSE */
                  IN BOOLEAN ChargeQuota, /* charge user memory quota? */
                  IN OUT PIRP Irp OPTIONAL); /* IRP to assoc. MDL with */
{
    ULONG sizeAllocated = MmSizeOfMdl(VirtualAddress, Length);
    PMDL returned = (PMDL) ExAllocatePoolWithTag(NonPagedPool,
                                                sizeAllocated,
                                                "Tag ");

    /* Initialize fields */
    returned->Size = sizeAllocated;
    returned->StartVa = VirtualAddress & 0xFFFFF000;
    returned->ByteOffset = VirtualAddress & 0x00000FFF;
    returned->ByteCount = Length;
    returned->Process = PsGetCurrentProcess();
    if (ChargeQuota)
    {
        /* Charge Length bytes to process quota */
    }
    if (Irp)
    {
        if (SecondaryBuffer)
        {
            /* Add to end of MDL list */
            PMDL listMDL = Irp->MdlAddress;
            while (listMDL->Next) { listMDL = listMDL->Next; }
            listMDL->Next = returned;
        }
        else
        {
            /* Add at head */
            Irp->MdlAddress = returned;
        }
    }
    return (returned);
}
```

Of course, MDLs must be freed using the inverse function, **IoFreeMdl()**. Instead of freeing MDLs and allocating new ones, however, it often makes sense to reuse the existing MDLs and just reinitialize their page lists. This can be done by calling **MmInitializeMdl()** with new values for **VirtualAddress** and **Length**.

Note: If an MDL is reused, by calling **MmInitializeMdl**, special care must be taken to ensure that the size of the buffer pointed to also accounts for the physical page table! Remember to verify with **MmSizeOfMdl()**

(C) 2009 JL@HisOwn.com - Feel free to use, replicate, but please don't modify. Questions/Comments welcome!

Working with MDLs

II. Allocating Pages

For non paged memory:

```
VOID MmBuildMdlForNonPagedPool (IN OUT PMDL pMdl);
```

For pageable memory:

```
VOID MmProbeAndLockPages (IN OUT PMDL pMdl,
                          IN KPROCESSOR_MODE AccessMode,
                          IN LOCK_OPERATION Operation);
```

- Function may throw exception
- Caller must remember to MmUnlockPages()

Map into Kernel Space:

```
PVOID MmGetSystemAddressForMdlSafe (IN PMDL pMdl,
                                     IN MM_PAGE_PRIORITY Priority);
```

MDLs may describe memory originally allocated from either pool: Paged or Non-Paged. To work with the MDLs, they must be initialized by one of two functions:

- **MmBuildMdlForNonPagedPool()**: which takes the MDL and initializes it with the appropriate flags corresponding to Non Paged Pool values.
- **MmProbeAndLockPages()**: which attempts to lock the pages described by the MDL, so they may be safely used, if they are from the Paged Pool.

Special care must be taken when locking pages, as a lock is an inherently risky operation – when a driver locks a given MDL's pages, with **MmProbeAndLockPages()**, it must be aware of two major caveats:

- A page fault may be triggered (since **MmProbeAndLock()** calls ProbeFor..) which, in turn, may throw the exception. As such, calls to this function must be made within a `__try/__catch` block
- The caller must remember to also unlock the pages, i.e. call **MmUnlockPages()** when done. The calls must match exactly 1:1, however: Forgetting to call **MmUnlockPages()** will result in a `DRIVER_LEFT_PAGES_IN_MEMORY` bugcheck, whereas calling it one time too many will corrupt the system Page Frame Number Database (PFN Database), resulting in a `PFN_LIST_CORRUPT` bugcheck.

...If you liked this course, consider...

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

Networking:

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 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:

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 Programming:

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

Windows:

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

Cryptography:

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

Security:

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.