

Basic Knowledge for Writing Driver Programs

主講人：虞台文

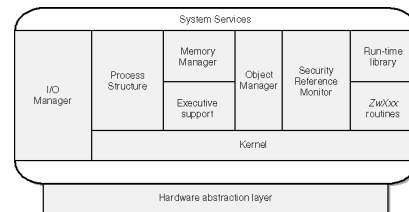
Content

- ≠ The Kernel-Mode Programming Environment
- ≠ Memory/Buffer Management
- ≠ Linked Lists
- ≠ String Handling
- ≠ Registry Access
- ≠ File Access

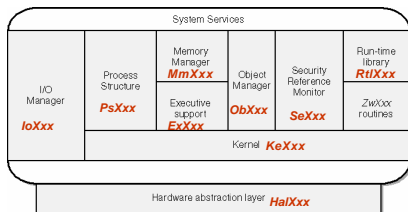
Basic Knowledge for Writing Driver Programs

The Kernel-Mode Programming Environment

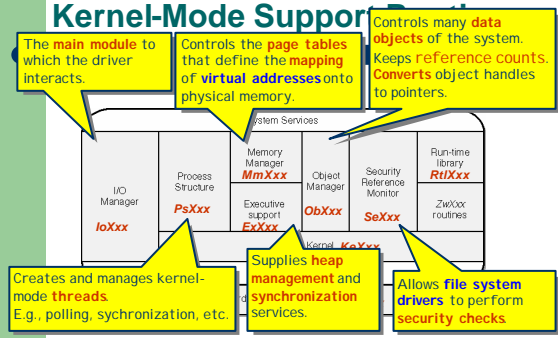
Kernel-Mode Support Routines



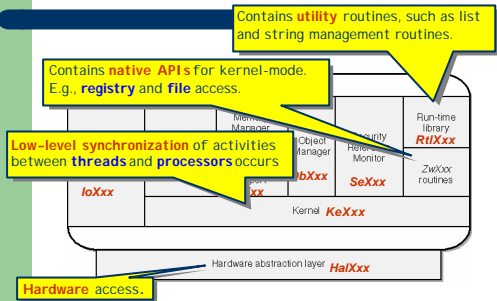
Kernel-Mode Support Routines



Kernel-Mode Support Routines



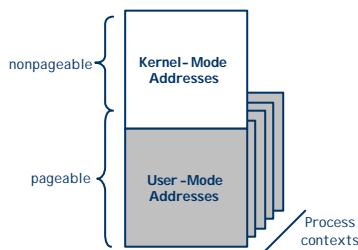
Kernel-Mode Support Routines



Basic Knowledge for Writing Driver Programs

Memory/Buffer Management

User-Mode and Kernel-Mode Address Spaces



Physical Memory vs. Virtual Memory

- ✦ In a **virtual memory system**, the operating system organizes **physical memory** and the swap file into like-size **page frames**.
- ✦ The **continuous addresses** in the virtual memory space are, hence, **not necessary continuous** in the physical memory space.

Page Size for Intel Platform

```
// wdm.h
// Define the page size for the Intel 386 as 4096 (0x1000).
//
#define PAGE_SIZE 0x1000

//
// Define the number of trailing zeroes in a page aligned virtual address.
// This is used as the shift count when shifting virtual addresses to
// virtual page numbers.
//
#define PAGE_SHIFT 12L
```

Marcos for Buffer and MDL Management

- ✦ **ADDRESS_AND_SIZE_TO_SPAN_PAGES**
 - Returns the **number of pages** required to contain a given virtual address and size in bytes.
- ✦ **BYTE_OFFSET**
 - Returns the **byte offset** of a given virtual address within the page.
- ✦ **BYTES_TO_PAGES**
 - Returns the **number of pages** necessary to contain a given number of bytes.
- ✦ **PAGE_ALIGN**
 - Returns the **page-aligned virtual address** for the page that contains a given virtual address.
- ✦ **ROUND_TO_PAGES**
 - Rounds a given size in bytes up to a **page-size multiple**.

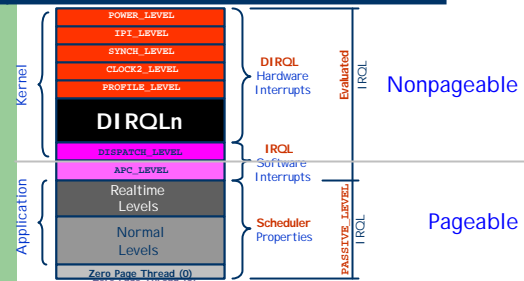
Paged and Nonpaged Memory

Where will your driver code be located?

General Rule

Code executing at or above interrupt request level (IRQL) `DISPATCH_LEVEL` cannot cause page faults.

Paged and Nonpaged Memory



PAGE_CODE() Macro

Conditional Compilation

```
#if DBG
#define PAGE_CODE() \
    if (KeGetCurrentIrql() > APC_LEVEL) { \
        KdPrint(("EX: Pageable code called at IRQL %d\n", KeGetCurrentIrql())); \
        ASSERT(FALSE); \
    }
#else
#define PAGE_CODE()
#endif
```

In retail build, this macro do nothing.

Debugging Paged Code Violation

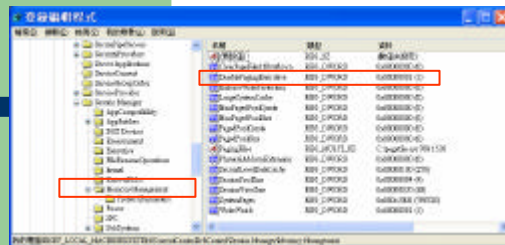
```
NTSTATUS DispatchPower(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PAGE_CODE()
    . . . . .
}
```

If page fault occurs at IRQL? `IRQL_DISPATCH`, bug check will be resulted.

`IRQL_NOT_LESS_OR_EQUAL` or `DRIVER_IRQL_NOT_LESS_OR_EQUAL`

Compile-Time Control of Pageability

- Win32 executable files, including kernel-mode drivers, are internally composed of one or more sections.
- A section is also the smallest unit that you can designate when you're specifying page-ability
- When loading a driver image, the system puts sections beginning with `PAGE` or `.EDA` (the start of `.EDATA`) into the paged pool
 - unless the `DisablePagingExecutive` value in the `HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management` key happens to be set (in which case no driver paging occurs).
- Note that these names are case sensitive!



- unless the `DisablePagingExecutive` value in the `HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management` key happens to be set (in which case no driver paging occurs).
- Note that these names are case sensitive!

The Control of Pageability

- ≪ Compiler time
 - Assign your code and data to the paged and nonpaged pools.
 - By instructing the compiler how to apportion your code and data among various sections.
 - The run-time loader uses the names of the sections to put parts of your driver in the places you intend.
- ≪ Run time
 - Calling various Memory Manager routines.
 - E.g., to allocate memory in the designate memory pool, and to lock memory.

```
#pragma alloc_text(PAGExxxx, RoutineName)
```

Compile-Time Control of Pageability

To isolate the pageable code into a named section, mark it as above.

```
// Note: function declaration appears before the following
#ifdef ALLOC_PRAGMA
#pragma alloc_text(PAGE, AddDevice)
#pragma alloc_text(PAGE, DispatchPnp)
. . . . .
#endif
// append the driver code from here
```

Compile-Time Control of Pageability

Create a pageable data section as follows:

At the beginning of the data module:

```
#pragma data_seg("PAGE DATA")
```

At the end of the module:

```
#pragma data_seg()
```

Compile-Time Control of Pageability

You can also create a pageable code section as follows:

```
#pragma code_seg("PAGE CODE")
NTSTATUS AddDevice(...){...}
NTSTATUS DispatchPnp(...){...}
. . . . .
#pragma code_seg()
```

Locking Pageable Code or Data

- ≪ Certain drivers, such as the serial and parallel drivers, need not be resident unless the devices are open.
- ≪ On the other hand, if the port or connection is not in use, the driver code is not required.
- ≪ It is better to mark such pieces of code pageable, but to lock them whenever needed.

Locking Pageable Code or Data

- ≪ Group correlated codes that must be resident (nonpageable) when they are activated in the same section with name PageXxxx.
- ≪ These codes are, hence, pageable.
- ≪ However, the section have to be locked into memory before invoking any code in the section.

Run-Time Control of Pageability

- Group correlated codes that **must** be resident (nonpageable) when they are activated in the **same section** with name **PageXxxx**.
- These codes are, hence, **pageable**.
- However, the **section** have to be **locked** into memory **before invoking** any code in the section.

Isolating Pageable Code

- Sometimes, the driver's code **has to be modified** so as to apply the aforementioned strategy.



Dynamically Locking and Unlocking Driver Pages

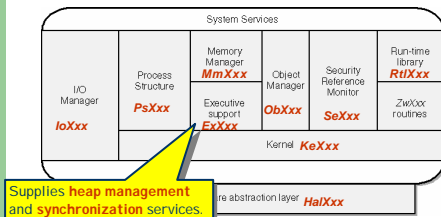
Service Function	Description
MmLockPagableCodeSection	Locks a code section given an address inside it
MmLockPagableDataSection	Locks a data section given an address inside it
MmLockPagableSectionByHandle	Locks a code section by using a handle from a previous MmLockPagableCodeSection call (Windows 2000 and Windows XP only)
MmPageEntireDriver	Unlocks all pages belonging to driver
MmResetDriverPaging	Restores compile-time pageability attributes for entire driver
MmUnlockPagableImageSection	Unlocks a locked code or data section

Dynamically Locking and Unlocking Driver Pages

```

VOID MmLockPagableCodeSection(IN PVOID AddressWithinSection);
VOID MmLockPagableDataSection(IN PVOID AddressWithinSection);
VOID MmLockPagableSectionByHandle(IN PVOID ImageSectionHandle);
VOID MmUnlockPagableImageSection(IN PVOID ImageSectionHandle);
VOID MmPageEntireDriver(IN PVOID AddressWithinSection);
VOID MmResetDriverPaging(IN PVOID AddressWithinSection);
    
```

Basic Buffer Allocation Routines



Basic Buffer Allocation Routines

Service Function	Description
ExAllocatePoolWithTag	Allocates (optionally cache-aligned) pool memory from paged or nonpaged system space. The caller-supplied tag is put into any crash dump of memory that occurs.
ExFreePoolWithTag	Releases memory with the specified pool tag .
ExFreePool	Releases memory to paged or nonpaged system space.



Callers' **IRQL ? DISPATCH_LEVEL**

ExAllocatePoolWithTag Routine

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

Allocates pool memory of the **specified type**, and returns a pointer to the allocated block.

Callers' **IRQL ? DISPATCH_LEVEL**

ExAllocatePoolWithTag Routine

The type of pool memory to allocate.

```
PVOID ExAllocatePoolWithTag(
    IN POOL_TYPE PoolType,
```

Pool Type	Description
NonPagedPool	from the nonpaged pool
PagedPool	from the paged pool
NonPagedPoolCacheAligned	nonpaged pool and aligned with the CPU cache
PagedPoolCacheAligned	from the paged pool of memory and aligned with the CPU cache

and returns a pointer to the allocated block.

Callers' **IRQL ? DISPATCH_LEVEL**

ExAllocatePoolWithTag Routine

The number of bytes to allocate.

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

The pool tag for the allocated memory. Normally specify as a string, in reversed order, of up to four characters, delimited by single quotation marks. **WinDbg** will use.

Callers' **IRQL ? DISPATCH_LEVEL**

ExAllocatePoolWithTag Routine

```
PVOID ExAllocatePoolWithTag(
    IN POOL_TYPE PoolType,
    IN SIZE_T NumberOfBytes,
```

- NumberofBytes > PAGE_SIZE
 - Page aligned
 - May cause **waste** in the last page
- NumberofBytes = PAGE_SIZE or less (a few)
 - Page aligned
- NumberofBytes << PAGE_SIZE
 - Aligned on a **8-byte** boundary
 - Allocate **exactly** the number of requested bytes of memory

Callers' **IRQL ? DISPATCH_LEVEL**

ExFreePool Routine

```
VOID ExFreePool(
    IN PVOID P
);
```

Releases memory allocated by **ExAllocatePool**, **ExAllocatePoolWithTag**, **ExAllocatePoolWithQuota**, or **ExAllocatePoolWithQuotaTag**

Callers' **IRQL ? DISPATCH_LEVEL**

ExFreePoolWithTag Routine

```
VOID ExFreePoolWithTag(
    IN PVOID P,
    IN ULONG Tag
);
```

Bug checks if the specified value for Tag does **not match** the tag value passed to the routine that originally allocated the memory block.

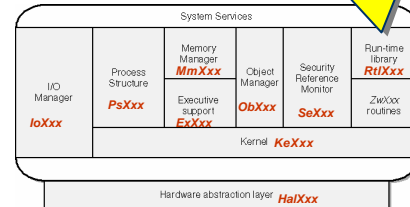
Otherwise, the behavior of this routine is **identical** to **ExFreePool**.

Basic Knowledge for Writing Driver Programs

Linked List

Run-Time Library Routines

Contains **utility** routines, such as **list** and **string** management routines.



Linked Lists

- ⌚ A way to **organize collections of similar data structures**.
- ⌚ Extensively used in Windows XP, such as **shared** IRP queues.
- ⌚ **Synchronization** is an important issue.
- ⌚ **Two Types** of Linked lists
 - doubly-linked
 - singly-linked

Data Structures

Doubly-Linked List

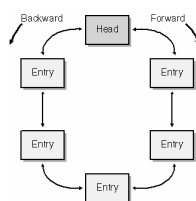
```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

Singly-Linked List

```
typedef struct _SINGLE_LIST_ENTRY {
    struct _SINGLE_LIST_ENTRY *Next;
} SINGLE_LIST_ENTRY, *PSINGLE_LIST_ENTRY;
```

Topologies

Doubly-Linked List



Singly-Linked List



Using the Link Lists

```
typedef struct _TWOWAY
{
    . . . . .
    LIST_ENTRY linkfield;
    . . . . .
} TWOWAY, *PTWOWAY;
```

Embed the list in another structure

```
LIST_ENTRY DoubleHead;
```

Declare a link list head

```
typedef struct _ONEWAY
{
    . . . . .
    SINGLE_LIST_ENTRY linkfield;
    . . . . .
} ONEWAY, *PONEWAY;
```

Embed the list in another structure

```
SINGLE_LIST_ENTRY SingleHead;
```

Declare a link list head

Example

```
typedef struct _IRP {
    .
    .
    LIST_ENTRY ListEntry;
    .
    .
} IRP, *PIRP;
```

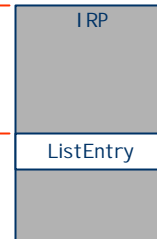


Example

Don't use this

Use this

When calling **list-management service functions** (to be discussed), always work with the **linking field** or the **list head** – *never* directly with the containing **structures themselves**.



CONTAINING_RECORD Macro

```
#define CONTAINING_RECORD(address, type, field) \
((type *) ( \
(PCHAR)(address) - \
(ULONG_PTR)((type *)0->field)))
```

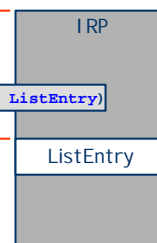


CONTAINING_RECORD Macro

```
#define CONTAINING_RECORD(address, type, field) \
((type *) ( \
(PCHAR)(address) - \
(ULONG_PTR)((type *)0->field)))
```

```
q = CONTAINING_RECORD(p, IRP, ListEntry)
```

p



Service Functions of Doubly-Linked List

Service Function or Macro	Description
InitializeListHead	Initializes the LIST_ENTRY at the head of the list
InsertHeadList	Inserts element at the beginning
InsertTailList	Inserts element at the end
IsListEmpty	Determines whether list is empty
RemoveEntryList	Removes element
RemoveHeadList	Removes first element
RemoveTailList	Removes last element

Example

```
typedef struct _TWOWAY {
    .
    .
    LIST_ENTRY linkfield;
} TWOWAY, *PTWOWAY;

LIST_ENTRY DoubleHead;

InitializeListHead(&DoubleHead);
ASSERT(IsListEmpty(&DoubleHead));
.
.
.
PTWOWAY pdElement = (PTWOWAY) ExAllocatePool(PagedPool,
sizeof(TWOWAY));
InsertTailList(&DoubleHead, &pdElement->linkfield);
.
.
.
if (!IsListEmpty(&DoubleHead)){
    PLIST_ENTRY pdLink = RemoveHeadList(&DoubleHead);
    pdElement = CONTAINING_RECORD(pdLink, TWOWAY, linkfield);
    .
    .
    ExFreePool(pdElement);
}
```

Service Functions of Singly-Linked List

Service Function or Macro	Description
<code>PushEntryList</code>	Adds element to top of list
<code>PopEntryList</code>	Removes topmost element



Example

```

typedef struct _ONEWAY {
    . . . . .
    SINGLE_LIST_ENTRY linkfield;
} ONEWAY, *PONEWAY;

SINGLE_LIST_ENTRY SingleHead;
SingleHead.Next = NULL;

PONEWAY psElement = (PONEWAY) ExAllocatePool(PagedPool,
    sizeof(ONEWAY));
PushEntryList(&SingleHead, &psElement->linkfield);
. . . . .
SINGLE_LIST_ENTRY psLink = PopEntryList(&SingleHead);
if (psLink){
    psElement = CONTAINING_RECORD(psLink, ONEWAY, linkfield);
    ExFreePool(psElement);
}
  
```

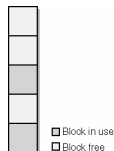
The Issue

⚡ What problem will we encounter if using the above approach to **allocate memory**?

Fragmentation

⚡ What will you do?

- Allocate a large block.
- Is that good?



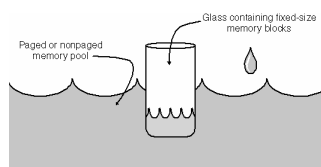
Lookaside Lists

⚡ To deal with the allocation/deallocation of **fixed-size** memory blocks **effectively** and **dynamically**.

⚡ **Executive** supports lookaside list routines

- to manage any **dynamically sized set of fixed-size buffers** or structures with caller-determined contents.

Basic Concept



Service Functions for Lookaside List

Service Function	Description
<code>ExInitializeNPagedLookasideList</code>	Initialize a lookaside list
<code>ExInitializePagedLookasideList</code>	Initialize a lookaside list
<code>ExDeleteNPagedLookasideList</code>	Destroy a lookaside list
<code>ExDeletePagedLookasideList</code>	Destroy a lookaside list
<code>ExAllocateFromNPagedLookasideList</code>	Allocate a fixed-size block
<code>ExAllocateFromPagedLookasideList</code>	Allocate a fixed-size block
<code>ExFreeToNPagedLookasideList</code>	Release a block back to a lookaside list
<code>ExFreeToPagedLookasideList</code>	Release a block back to a lookaside list

Callers' **IRQL ? DISPATCH_LEVEL**

Initialize a Lookaside List

```

VOID ExInitializeNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside,
    IN PALLOCATE_FUNCTION Allocate OPTIONAL,
    IN PFREE_FUNCTION Free OPTIONAL,
    IN ULONG Flags,
    IN SIZE_T Size,
    IN ULONG Tag,
    IN USHORT Depth
);

```

Callers' **IRQL ? DISPATCH_LEVEL**

Initialize a L

Pointer to the caller-supplied memory for the lookaside list head to be initialized.

```

Reserved. Must be zero.
VOID ExInitializeNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside,
    IN PALLOCATE_FUNCTION Allocate OPTIONAL,
    IN PFREE_FUNCTION Free OPTIONAL,
    IN ULONG Flags,
    IN SIZE_T Size,
    IN ULONG Tag,
    IN USHORT Depth
);

```

Pool tag.

The size in bytes for each nonpaged entry to be allocated subsequently.

Reserved. Must be zero.

Callers' **IRQL ? DISPATCH_LEVEL**

Pointer to either a caller-supplied function for allocating an entry when the lookaside list is empty, or to NULL.

If non-NULL, the pointer is to a function with the prototype as follows:

```

VOID ExInitializeNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside,
    IN PALLOCATE_FUNCTION Allocate OPTIONAL,
    IN PFREE_FUNCTION Free OPTIONAL,
    PVOID XxxAllocate (
        IN POOL_TYPE PoolType, // NonPagedPool
        IN SIZE_T NumberOfBytes, // value of Size
        IN ULONG Tag // value of Tag
    );

```

Callers' **IRQL ? DISPATCH_LEVEL**

Pointer to either a caller-supplied function for freeing an entry whenever the lookaside list is full, or to NULL.

If non-NULL, the pointer is to a function with the prototype as follows:

```

VOID ExInitializeNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside,
    IN PALLOCATE_FUNCTION Allocate OPTIONAL,
    IN PFREE_FUNCTION Free OPTIONAL,
    IN ULONG Flags,
    IN SIZE_T Size,
    IN ULONG Tag,
    IN USHORT Depth,
    VOID XxxFree (
        PVOID Buffer
    );

```

Callers' **IRQL ? DISPATCH_LEVEL**

Free a Lookaside List

```

VOID ExInitializeNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside,
    IN PALLOCATE_FUNCTION Allocate OPTIONAL,
    IN PFREE_FUNCTION Free OPTIONAL,
    IN ULONG Flags,
    IN SIZE_T Size,
    IN ULONG Tag,
    IN USHORT Depth
);

```

```

VOID ExDeleteNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside
);

```

Allocation/Free an Entry

```

PVOID ExAllocateFromNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside
);

```

```

VOID ExFreeToNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside,
    IN PVOID Entry
);

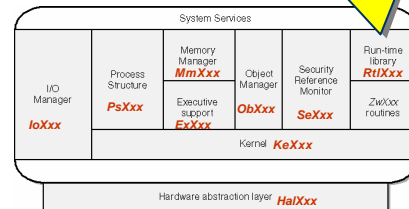
```

Basic Knowledge for Writing Driver Programs

String Handling

String Utilities

Contains **utility** routines, such as **list** and **string** management routines.



String Formats

WDM drivers work with **4 formats** of string:

- ⚡ **Unicode** strings
 - UNICODE_STRING
- ⚡ **ANSI** strings
 - ANSI_STRING
- ⚡ **Null-terminated** string of **characters**
 - PCHAR
 - "Hello WDM"
- ⚡ **Null-terminated** string of **wide characters**
 - PWCHAR
 - L"Hello WDM"

Unicode

- ⚡ A standard, fixed-width, **16-bit** character encoding scheme that the system uses to provide **NLS** support for **locale-specific** natural languages.
- ⚡ **NLS**
 - National Language Support.
 - A **set of routines** that give applications access to locale-specific information.

Null-Terminated String Manipulation

ntstrsafe.h

Standard function (deprecated)	Safe UNICODE Alternative	Safe ANSI Alternative
strcpy, wcsncpy, strncpy, wcsncpy	RtlStringCbCopyW, RtlStringCchCopyW	RtlStringCbCopyA, RtlStringCchCopyA
strcat, wscat, strncat, wcsncat	RtlStringCbCatW, RtlStringCchCatW	RtlStringCbCatA, RtlStringCchCatA
sprintf, swprintf, _snprintf, _snwprintf	RtlStringCbPrintfW, RtlStringCchPrintfW	RtlStringCbPrintfA, RtlStringCchPrintfA
vsprintf, vswprintf, vsnprintf, vswnprintf	RtlStringCbVPrintfW, RtlStringCchVPrintfW	RtlStringCbVPrintfA, RtlStringCchVPrintfA
strlen, wcslen	RtlStringCbLengthW, RtlStringCchLengthW	RtlStringCbLengthA, RtlStringCchLengthA

Cb: byte count **Cch**: character count

Null-Terminated String Manipulation

ntstrsafe.h

Standard function (deprecated)	Safe UNICODE Alternative	Safe ANSI Alternative
strcpy, wcsncpy, strncpy, wcsncpy	RtlStringCbCopyW, RtlStringCchCopyW	RtlStringCbCopyA, RtlStringCchCopyA
strcat, wscat, strncat, wcsncat	RtlStringCbCatW, RtlStringCchCatW	RtlStringCbCatA, RtlStringCchCatA
sprintf, swprintf, _snprintf, _snwprintf	RtlStringCbPrintfW, RtlStringCchPrintfW	RtlStringCbPrintfA, RtlStringCchPrintfA
vsprintf, vswprintf, vsnprintf, vswnprintf	RtlStringCbVPrintfW, RtlStringCchVPrintfW	RtlStringCbVPrintfA, RtlStringCchVPrintfA
strlen, wcslen	RtlStringCbLengthW, RtlStringCchLengthW	RtlStringCbLengthA, RtlStringCchLengthA

Cb: byte count **Cch**: character count

ntstrsafe.h

Null-Terminated String Manipulation

Standard function (deprecated)	Safe UNICODE Alternative	Safe ANSI Alternative
strcpy, wcsncpy, strncpy, wcsncpy	RtlStringCbCopyW, RtlStringCchCopyW	RtlStringCbCopyA, RtlStringCchCopyA
strcat, wscat, strcat, wscat	RtlStringCbCatW, RtlStringCchCatW	RtlStringCbCatA, RtlStringCchCatA
sprintf, vsprintf, _snprintf, _snwprintf	RtlStringCbPrintfW, RtlStringCchPrintfW	RtlStringCbPrintfA, RtlStringCchPrintfA
vsprintf, vsnwprintf, vsprintf, vsnwprintf	RtlStringCbVPrintfW, RtlStringCchVPrintfW	RtlStringCbVPrintfA, RtlStringCchVPrintfA
strlen, wcslen	RtlStringCbLengthW, RtlStringCchLengthW	RtlStringCbLengthA, RtlStringCchLengthA

Cb: byte count Cch: character count

Safe String Functions

- Using the **safe string functions** instead of the string manipulation functions provided by C-language runtime libraries, you **protect** your code from **buffer overrun** errors that can make code untrustworthy.
- Each function is available in **two versions**:
 - a **W-suffixed** version that supports double-byte **Unicode** characters
 - an **A-suffixed** version that supports single-byte **ANSI** characters

Data Structures

```

typedef struct _STRING {
    USHORT Length;
    USHORT MaximumLength;
    PCHAR Buffer;
} STRING;

typedef STRING *PSTRING;
typedef STRING ANSI_STRING;

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
} UNICODE_STRING;
  
```

Counted-String Functions

Initialization

```

VOID RtlInitUnicodeString(
    IN OUT PUNICODE_STRING DestinationString,
    IN PCWSTR SourceString
);
  
```

- The **Buffer** member of **DestinationString** is initialized to **point to SourceString**.
- The **length** and **maximum length** for **DestinationString** are initialized to the length of **SourceString**.
- If **SourceString** is **NULL**, the **length** is **zero**.

Initialization

```

VOID RtlInitUnicodeString(
    IN OUT PUNICODE_STRING DestinationString,
    IN PCWSTR SourceString
);
  
```

- Can be running at **IRQL ? DISPATCH_LEVEL** if the **DestinationString** buffer is **nonpageable**.
- Usually, callers run at **IRQL = PASSIVE_LEVEL** because most other **RtlXxxString** routines cannot be called at raised IRQL.

Free a String

Pointer to the Unicode string buffer previously allocated by `RtlAnsiStringToUnicodeString` or `RtlUpcaseUnicodeString`.

```
VOID RtlFreeUnicodeString(
    IN PUNICODE_STRING UnicodeString
);
```

Callers' `IRQL = PASSIVE_LEVEL`.

Special Note

```
UNICODE_STRING foo;
if (bArriving)
    RtlInitUnicodeString(&foo, "Hello, world!");
else
{
    ANSI_STRING bar;
    RtlInitAnsiString(&bar, "Goodbye, cruel world!");
    RtlAnsiStringToUnicodeString(&foo, &bar, TRUE);
}

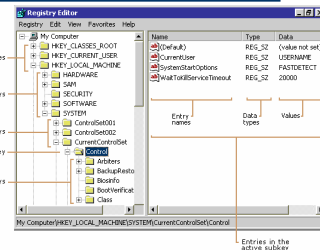
RtlFreeUnicodeString(&foo); // <== don't do this!
```

Basic Knowledge for Writing Driver Programs

Registry Access

The Registry

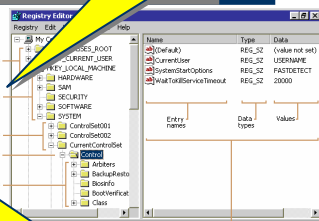
RegEdit



A subtree is a **root**, or **primary division**, of the registry.

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS
- HKEY_CURRENT_CONFIG

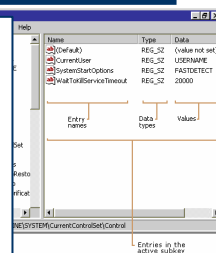
Keys are the **first division** of a subtree. Keys contain **subkeys** and **entries**.



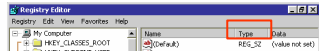
Subkeys are **children** of keys. All registry folders contained in keys are called subkeys. Subkeys can contain entries and other subkeys.

The Registry

- Entries are the **lowest level element** in the registry.
- Each entry consists of an entry **name**, its **data type**, and its **value**.
- Entries store the actual **configuration data** that affects the operating system and programs that run on the system. As such, they are **different from keys and subkeys**.
- Entries are **referenced** by their registry **path and name**.




Data Types ? Registry



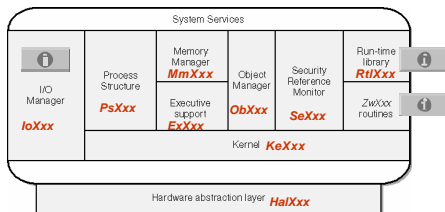
Data Type	Description
REG_BINARY	Binary data in any form
REG_DWORD	A 4-byte numerical value
REG_DWORD_LITTLE_ENDIAN	REG_DWORD with LSB at the lowest address
REG_DWORD_BIG_ENDIAN	REG_DWORD with LSB at the highest address
REG_SZ	A zero-terminated Unicode string
REG_MULTI_SZ	Multiple zero-terminated strings ending with 2 zeros
REG_EXPAND_SZ	A zero-terminated Unicode string, containing unexpanded references to environment variables, such as "%PATH%"
REG_LINK	A Unicode string naming a symbolic link; this type is irrelevant to device and intermediate drivers.
REG_NONE	Data with no particular type

Data Types ? Registry



Data Type	Description
REG_RESOURCE_LIST	A device driver's list of hardware resources, used by the driver or one of the physical devices it controls, in the \ResourceMap tree
REG_RESOURCE_REQUIREMENTS_LIST	A device driver's list of possible hardware resources it or one of the physical devices it controls can use, from which the system writes a subset into the \ResourceMap tree
REG_FULL_RESOURCE_DESCRIPTOR	A list of hardware resources that a physical device is using detected and written into the \HardwareDescription tree by the system

Registry Access Routines



Registry Access ? I/O Manager

Service Function	Description
IoRegisterDeviceInterface	Registers a device functionality. The I/O Manager creates a registry key. Drivers can access the key using IoOpenDeviceInterfaceRegistryKey.
IoSetDeviceInterfaceState	Enables/disables a device interface.
IoGetDeviceProperty	Retrieves device setup information from the registry.
IoOpenDeviceRegistryKey	Opens special key associated with a physical device object (PDO).
IoOpenDeviceInterfaceRegistryKey	Opens a registry key associated with a registered device interface

Callers' IRQL = PASSIVE_LEVEL in the context of a system thread.

Open Device Registry Key

```
NTSTATUS IoOpenDeviceRegistryKey(
    IN PDEVICE_OBJECT DeviceObject,
    IN ULONG DevInstKeyType,
    IN ACCESS_MASK DesiredAccess,
    OUT PHANDLE DevInstRegKey
);
```

Returns a handle to a device-specific or a driver-specific registry key for a particular device instance.

Callers' IRQL = PASSIVE_LEVEL in the context of a system thread.

Open Device Registry Key

Indicates the special key you want to open.

The PDO of the device instance for which the registry key is to be opened.

```
NTSTATUS IoOpenDeviceRegistryKey(
    IN PDEVICE_OBJECT DeviceObject,
    IN ULONG DevInstKeyType,
```

Flag Value	Selected Registry Key
PLUGPLAY_REGKEY_DEVICE	Open a hardware key for storing device-specific information
PLUGPLAY_REGKEY_DRIVER	Open a software key for storing driver-specific information.
PLUGPLAY_REGKEY_CURRENT_HWPROFILE	Open a key relative to the current hardware profile for device or driver information.

Callers' IRQL = **PASSIVE_LEVEL** in the context of a system thread.

Open Device Registry Key

KEY_READ, KEY_WRITE or KEY_ALL_ACCESS

```

NTSTATUS IoOpenDeviceRegistryKey(
    IN PDEVICE_OBJECT DeviceObject,
    IN ULONG DevInstKeyType,
    IN ACCESS_MASK DesiredAccess,
    OUT PHANDLE DevInstRegKey
);

```

Returns a **handle** to a registry key for a particular device instance.

Points to a caller-allocated buffer that, on successful return, contains a **handle** to the requested registry key.

Callers' IRQL = **PASSIVE_LEVEL** in the context of a system thread.

Open Device Interface Registry Key

```

NTSTATUS IoOpenDeviceInterfaceRegistryKey(
    IN PUNICODE_STRING SymbolicLinkName,
    IN ACCESS_MASK DesiredAccess,
    OUT PHANDLE DeviceInterfaceKey
);

```

Returns a **handle** to a registry key for accessing information about a particular device interface instance.

Callers' IRQL = **PASSIVE_LEVEL** in the context of a system thread.

Open Device Interface Registry Key

Points to a string identifying the device interface instance being enabled or disabled. This string was obtained from a previous call to `IoRegisterDeviceInterface` or `IoGetDeviceInterfaces`.

```

NTSTATUS IoOpenDeviceInterfaceRegistryKey(
    IN PUNICODE_STRING SymbolicLinkName,
    IN ACCESS_MASK DesiredAccess,
    OUT PHANDLE DeviceInterfaceKey
);

```

Points to a returned **handle** to the requested registry key if the call is successful.

Points to a caller-allocated buffer that, on successful return, contains a **handle** to the requested registry key.

KEY_READ, KEY_WRITE or KEY_ALL_ACCESS

Registry Access ? Runtime Lib

Service Function	Description
<code>RtlCreateRegistryKey</code>	Creates a key along a relative path.
<code>RtlQueryRegistryValues</code>	Reads several values from the registry.
<code>RtlWriteRegistryValue</code>	Writes a value to the registry.
<code>RtlDeleteRegistryValue</code>	Deletes a registry value.
<code>RtlCheckRegistryKey</code>	Checks the existence of a key.

Registry Access ? Native API

Service Function	Description
<code>ZwOpenKey</code>	Opens a registry key.
<code>ZwClose</code>	Closes handle to a registry key.
<code>ZwCreateKey</code>	Creates a registry key.
<code>ZwDeleteKey</code>	Deletes a registry key.
<code>ZwDeleteValueKey</code>	Deletes a value (Windows 2000 and later).
<code>ZwEnumerateKey</code>	Enumerates subkeys.
<code>ZwEnumerateValueKey</code>	Enumerates values within a registry key.
<code>ZwFlushKey</code>	Commits registry changes to disk.
<code>ZwQueryKey</code>	Gets information about a registry key.
<code>ZwQueryValueKey</code>	Gets a value within a registry key.
<code>ZwSetValueKey</code>	Sets a value within a registry key.

Open/Create a Registry Key

```

NTSTATUS ZwOpenKey(
    OUT PHANDLE KeyHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes
);

NTSTATUS ZwCreateKey(
    OUT PHANDLE KeyHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN ULONG TitleIndex,
    IN PUNICODE_STRING Class OPTIONAL,
    IN ULONG CreateOptions,
    OUT PULONG Disposition OPTIONAL
);

```

OBJECT_ATTRIBUTES Structure

```
typedef struct _OBJECT_ATTRIBUTES {
    ULONG Length;
    HANDLE RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG Attributes;
    PVOID SecurityDescriptor;
    PVOID SecurityQualityOfService;
} OBJECT_ATTRIBUTES;
```

Initialize Object Attributes

```
VOID InitializeObjectAttributes(
    OUT POBJECT_ATTRIBUTES InitializedAttributes,
    IN PUNICODE_STRING ObjectName,
    IN ULONG Attributes,
    IN HANDLE RootDirectory,
    IN PSECURITY_DESCRIPTOR SecurityDescriptor
);
```

Sets up a parameter of type **OBJECT_ATTRIBUTES** for a subsequent call to **ExCreateCallback** or to a **ZwCreateXxx** or **ZwOpenXxx** routine.

Specifies the **OBJECT_ATTRIBUTES** structure to initialize.

Is the full path name for the object.

An initialized **absolute SECURITY_DESCRIPTOR** for the object or **NULL**.

```
VOID InitializeObjectAttributes(
    OUT POBJECT_ATTRIBUTES InitializedAttributes,
    IN PUNICODE_STRING ObjectName,
    IN ULONG Attributes,
    IN HANDLE RootDirectory,
    IN PSECURITY_DESCRIPTOR SecurityDescriptor
);
```

Specifies a **handle** to the root object directory for the **ObjectName**. Specifies **NULL** if **ObjectName** parameter is **fully-qualified**. Use **ZwCreateDirectoryObject** to obtain a handle to an object directory.

Initialize Object Attributes

A combination (**ORed**) of the following flags:

- OBJ_INHERIT
- OBJ_PERMANENT
- OBJ_EXCLUSIVE
- OBJ_CASE_INSENSITIVE
- OBJ_OPENIF
- OBJ_KERNEL_HANDLE

```
VOID InitializeObjectAttributes(
    OUT POBJECT_ATTRIBUTES InitializedAttributes,
    IN PUNICODE_STRING ObjectName,
    IN ULONG Attributes, 0
    IN HANDLE RootDirectory,
    IN PSECURITY_DESCRIPTOR SecurityDescriptor
);
```

Sets up a parameter of type **OBJECT_ATTRIBUTES** for a subsequent call to **ExCreateCallback** or to a **ZwCreateXxx** or **ZwOpenXxx** routine.

Example

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath)
{
    . . . . .
    OBJECT_ATTRIBUTES oa;
    InitializeObjectAttributes(&oa, RegistryPath,
        OBJ_KERNEL_HANDLE OBJ_CASE_INSENSITIVE, NULL, NULL);
    HANDLE hkey;
    status = ZwOpenKey(&hkey, KEY_READ, &oa);
    if (NT_SUCCESS(status)){
        . . . . .
        ZwClose(hkey);
    }
    . . . . .
}
```

Callers' **IRQL = PASSIVE_LEVEL**.

Query Values of Registry Keys

```
NTSTATUS ZwQueryValueKey(
    IN HANDLE KeyHandle,
    IN PUNICODE_STRING ValueName,
    IN KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass,
    OUT PVOID KeyValueInformation,
    IN ULONG Length,
    OUT PULONG ResultLength
);
```

Returns the **value entries** for an open registry key.

Query Value of Registry Keys

IRQL = PASSIVE_LEVEL.

Handle to the key. This handle is created by a successful call to `ZwCreateKey` or `ZwOpenKey`.

The name of the value entry.

```

NTSTATUS ZwQueryValueKey(
    IN HANDLE KeyHandle,
    IN PUNICODE_STRING ValueName,
    IN KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass,
    OUT PVOID KeyValueInformation,
    IN ULONG Length,
    OUT PULONG ResultLength
);

```

Specifies the size, in bytes, of the KeyValue information buffer.

The received size, in bytes, of the key information. Returns `STATUS_SUCCESS`: the actual size of data returned. Returns `STATUS_BUFFER_OVERFLOW` or `STATUS_BUFFER_TOO_SMALL`: the minimum size needed.

Query Value of Registry Keys

IRQL = PASSIVE_LEVEL.

Specifies a `KEY_VALUE_INFORMATION_CLASS` value that determines the type of information returned in the KeyValue information buffer.

```

NTSTATUS ZwQueryValueKey(
    IN HANDLE KeyHandle,
    IN PUNICODE_STRING ValueName,
    IN KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass,
    OUT PVOID KeyValueInformation,
    IN ULONG Length
);

```

Enumerator	Structure of KeyValue information
KeyValueBasicInformation	KEY_VALUE_BASIC_INFORMATION
KeyValueFullInformation	KEY_VALUE_FULL_INFORMATION
KeyValuePartialInformation	KEY_VALUE_PARTIAL_INFORMATION

Using Registry Functions

```

UNICODE_STRING valname;
RtlInitUnicodeString(&valname, L"ImagePath");
size = 0;
status = ZwQueryValueKey(hkey, &valname, KeyValuePartialInformation, NULL, 0, &size);
if (status == STATUS_OBJECT_NAME_NOT_FOUND && size == 0)
    <handle error>;
size = min(size, PAGE_SIZE);
PKEY_VALUE_PARTIAL_INFORMATION vpip =
    PKEY_VALUE_PARTIAL_INFORMATION) ExAllocatePool(PagedPool, size);
if (!vpip)
    <handle error>;
status = ZwQueryValueKey(hkey, &valname, KeyValuePartialInformation, vpip, size, &size);
if (!NT_SUCCESS(status))
    <handle error>;
<do something with vpip->Data>ExFreePool(vpip);

```

Exercise

- Document **Reading** and **Summarization**:
 - What kinds of information are stored in the **main subtrees** of the registry?
 - The **Win32** registry functions.
- Reports using slides

Basic Knowledge for Writing Driver Programs

File Access

Kernel-Mode File I/O

- It's sometimes **useful** to be able to **read** and **write** regular **disk files** from inside a WDM driver, e.g.,
 - download** a large amount of microcode to your hardware,
 - create your own **extensive log** of information for some purpose.
- File access via the **ZwXxx routines** require you be running at **PASSIVE_LEVEL** in a thread that, hence, can safely be suspended.

ZwXxx Routines for File I/O

Service Function	Description
ZwCreateFile	Either creates a new file or directory, or opens an existing file, device, directory, or volume.
ZwOpenFile	Opens an existing file, device, directory, or volume.
ZwClose	Closes object handles
ZwReadFile	Read data from an open file
ZwWriteFile	Write data to an open file
ZwQueryInformationFile	Retrieves various kinds of information about a given file object

Exercise

Programming

- Use the parameter *Registry* passing to the **DriverEntry** routine to retrieve **all values** of the **named entries** of that subkey and **write** them to a **file**.