Understanding IRQL --= Documents =--

# Understanding IRQL

**Download this acticle (zipped pdf format)**.

System: Windows 2000 professional·free · build · 2195 · X86 · Single CPU

## Part I:

In this part, I'll discuss the fundament of IRQL and the differences/relations between IRQL and CPU rings/thread priority/hardware IRQ.

### Definitions:

For a kernel developer, IRQL is not strange. Almost every kernel support routine has a limit of the working IRQL. But what on earth an IRQL stands for ? This article is to disclose it's mysterious veil.

See it's definition from DDK:

Interrupt Request Level (IRQL)

*The priority ranking of an interrupt. A processor has an IRQL setting that threads can raise or lower. Interrupts that occur at or below the processor's IRQL setting are masked and will not interfere with the current operation. Interrupts that occur above the processor's IRQL setting take precedence over the current operation.*

*The particular IRQL at which a piece of kernel-mode code executes determines its hardware priority. Kernel-mode code is always interruptible: an interrupt with a higher IRQL value can occur at any time, thereby causing another piece of kernel-mode code with the system-assigned higher IRQL to be run immediately on that processor. In other words, when a piece of code runs at a given IRQL, the Kernel masks off all interrupt vectors with a lesser or equal IRQL value on the microprocessor.*

It says that higer IRQL interrupt can interrupt other task or interrupt with lower IRQL and all the interrupts with IRQL equal to or below current IRQL will be masked and wait until it get opportunity.

Each processor has an independent IRQL, which descripts the current IRQL of the processor, i.e. the IRQL of the current instructions / codes being executed by this processor.

System support a kernel routine (KeGetCurrentIRQL) to get current processor's IRQL. Let's see the assemble codes:

```
kd> u KeGetCurrentIrql
hal!KeGetCurrentIrql:
80063124 0fb70524f0dfff movzx eax,word ptr [ffdff024]
8006312b c3 ret
```

This routine is very simple. It tells us that current IRQL stores at krenel address 0xffdff024. And it's a WORD (2 bytes), in fact it's only one byte long, the upper 8 bits are zero.

The value of IRQL can be one of the followings.

Software IRQL:

PASSIVE_LEVEL 0 // Passive release level
LOW_LEVEL 0 // Lowest interrupt level
APC_LEVEL 1 // APC interrupt level
DISPATCH_LEVEL 2 // Dispatcher level

Hardware IRQL:

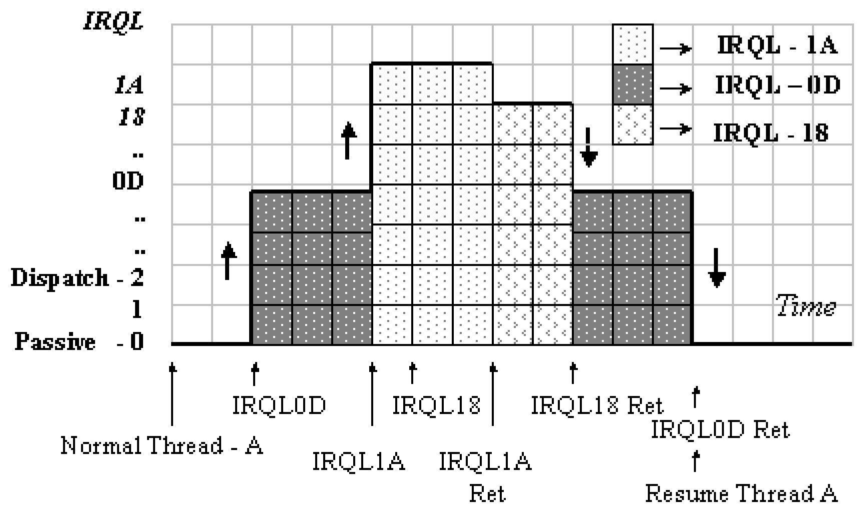
DIRQL: from 3 to 26 for device ISR

PROFILE_LEVEL 27 (0x1B) // timer used for profiling.
CLOCK1_LEVEL 28 (0x1C) // Interval clock 1 level - Not used on x86
CLOCK2_LEVEL 28 (0x1C) // Interval clock 2 level
SYNCH_LEVEL 28 (0x1C) // synchronization level
IPI_LEVEL 29 (0x1D) // Interprocessor interrupt level
POWER_LEVEL 30 (0x1E) // Power failure level
HIGH_LEVEL 31 (0x1F) // Highest interrupt level

The IRQL values are divided into two groups: Software ( 0,1,2 ) / Hardware IRQL ( >= 3). Hardware IRQL is for device ISRs and system, it is similar to (but distinguished with) the level of hardware IRQ, which implemented by i8259, but IRQL is only an action of Windows OS, not hardware's. It's realized by Windows OS. But hardware IRQ level is achieved by 8259A (programmable interrupt controlor). We will detail the hardware IRQ later.

The bigger of it's value, the higher level the IRQL has.

Let us see an example:



Description:

1) Thread A is running at this time.

2) IRQL (0D) interrupt happends, then cpu interrupt the current running thread A and begin to run IRQL (0D).

3) With higher IRQL, IRQL 1A will take over the cpu …
[Here I use this example just to demonstrate the mechanism of IRQL. For a real system, when an hardware IRQ arises, the eflags IF bit will be set to zero, to mask all maskable-interrupts.]

4) When IRQL 1A is running, IRQL 18 arises, but is masked for it's IRQL (18)< Current IRQL (1A).

5) IRQL 1A service finishes, then IRQL 18 will run instead of interrupted IRQL 0D routine.

6) Only after IRQL 18 finishes, IRQL 0D can get the cpu.

7) At last, cpu resume the thread A, when IRQL 0D finishes.

IRQL can be prone to be confused with Thread priority, Cpu Rings, and hardware IRQ. Now let's discuss the differences between them.

**IRQL vs. Thread priority:**

See ddk about Thread priority:

**priority**

An attribute of a thread that determines when and how often the thread is scheduled to run. For a running thread, its priority falls into either of two classes, each class with sixteen levels:

*- Variable priority class has values in the range 0 to 15. This class is used by most threads.*

*Threads with variable priority are always preemptible; that is, they are scheduled to run round-robin with other threads at the same level. In general, the Kernel manages a variable-priority thread as follows: when the thread is interactive with a user, its priority is high (given a boost); otherwise, its priority decays by one level per quantum the thread runs until it reaches its original programmer-defined base priority level.*

*- Real-time priority class has values in the range 16 to 31. This class is used by time-critical threads, making such a thread preemptible only by a thread with higher priority.*

Note that any thread, whatever its priority attribute, is always preemptible by a software or hardware interrupt.

A priority of a thread only affects the decisions of system scheduler:
1) When to execute this thread ?
2) How many will this thread take the time slices ?
The scheuler will decide them based on the priority of the thread.

Generally all the threads (including system threads which runs in ring0, kernel space) run at PASSIVE_LEVEL IRQL: no interrupt vectors are masked. And the scheduler who determinates the state of all the threads runs at DISPATCH_LEVEL.

System realizes it's preemption attribution via thread priority, the executor is the scheduler,. but another attribution "interruptible" of windows nt, is implemented via IRQL.

Windows NT is interruptible, i.e., any codes could be interrupted by higher IRQL interrupt. Every developer should keep it in mind.

**IRQL vs.Cpu Rings:**

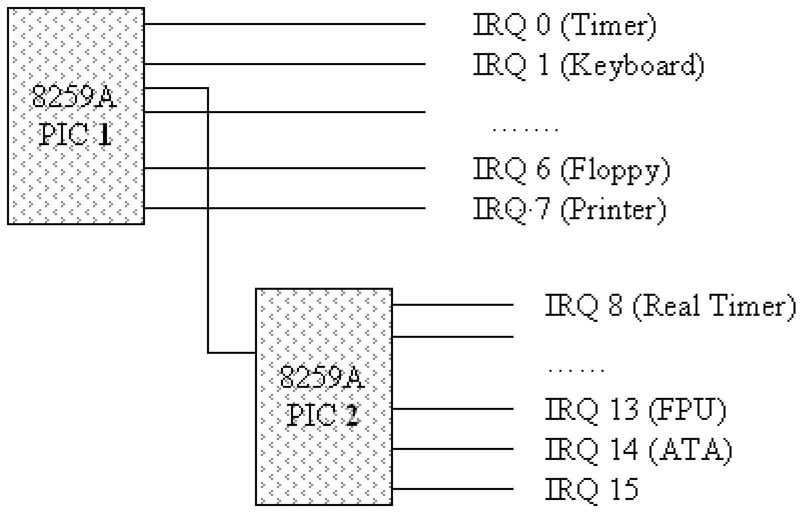
For x86 cpu, it has 4 rings: 0, 1, 2, 3

Every ring defines it's privilege, such as memory page access, io access … Windows only uses ring0 and ring3. Kernel uses ring0, and user routines use ring3.

IRQL only exists in kernel space. For user space, it's meaningless. All user threads are running at PASSIVE_LEVEL, though they can result in a task switch to kernel space and change the IRQL. They could not access the IRQL directly.
And also, from assemble codes of KeGetCurrentIRQL, we know that current IRQL locates at [ffdff024]. Address ffdff024 is in kernel space.

## IRQL vs. Hardware IRQs:

For X86 system, it has two 8259 pics (programmable interrupt controller), each could handle 8 IRQs. But the slave 8259 pic is attached to master 8259 's pin 2 (IRQ2). So there are 15 IRQs available.



In general, IRQ 0 has the highest privilege, the next is IRQ 1, the last is IRQ7. (Irq 8 – Irq 15 have the same privilage with IRQ2 ?).

For each IRQ, we can mask it by zero the correspond bit of IMR (interrupt mask register, port 21h for the master 8259, 0xA1h for the slave.). The IMR register is only writable, it is not readable. So system must store it's current value. For each IRQL, Windows NT maintains a table of IMR at hal!KiI8259MaskTable. (Softice's IRQ command possibly uses this value, I'm not sure.)

The realizaton of IRQ priorities is dependent to hardware (8259 pic). Windows uses hardware independent IRQL to mask all the differences of the various hardwares. IRQL can be looked as an extension of hardware IRQ levels. But IRQL is defined and manipulated by the OS, it's an action of software. The IRQLs of a lower priority IRQ may be a higher level. (See the output result of "intobj" under softice.)

To manager IRQs, windows uses Interrupt Object (KINTERRUPT). The interrupt object is initialized and tied to system interrupt objects chain, when device drivers call IoConnectInterrupt.

Structure Definition of Interrupt object:

```
typedef struct _KINTERRUPT { // Size: 0x1E4
/*000*/ CSHORT Type
/*002*/ USHORT Size
/*004*/ LIST_ENTRY InterruptListEntry
/*00C*/ ULONG ServiceRoutine
/*010*/ ULONG ServiceContext
/*014*/ SpinLock
/*018*/ Spare1
/*01C*/ ActualLock
/*020*/ DispatchAddress
/*024*/ Vector // The tied vector of this IRQ
/*028*/ Irql // Current IRQ's IRQL
/*029*/ SynchronizeIrql // The SynchronizeIRQL of the IRQ (To be detailed later)
/*02A*/ FloatingSave
/*02B*/ Connected
/*02C*/ Number
/*02D*/ ShareVector
/*030*/ Mode
/*034*/ Spare2
/*038*/ Spare3
/*03C*/ DispatchCode
} KINTERRUPT, *PKINTERRUPT;

// List all the interrupt objecits in softice
:intOBJ

Object Service Service Affinity
Address Vector Address Context IRQL Mode Mask Symbol
FF263408 31 F0470A0A FF2AF440 1A Edge 01 i8042prt!.text+070A
FF264D88 33 FC8CEAA0 FF2991D4 18 Edge 01 NDIS!PAGENDSMqN
FF25DA88 37 FC8CEAA0 FF25BBBC 14 Edge 01 NDIS!PAGENDSMqN
FCD6DD88 39 FC999454 FCDB48E8 12 Level 01 ACPI!.text+9134
FF285008 39 F06D2536 FCD615D0 12 Level 01 uhcd!.text+2256
FF2853C8 39 F06D2536 FCD60030 12 Level 01 uhcd!.text+2256
FF274988 39 F06D2536 FCD60AD0 12 Level 01 uhcd!.text+2256
FF299D88 39 FC59AE4A FF2AC0F0 12 Level 01 ltmdmnt!.text+2B2A
FF25C008 39 FC574CE0 FF25E208 12 Level 01 portcls!.text+19C0
FF262D88 3C F0476F00 FF2AA020 0F Edge 01 i8042prt!PAGEMOUCk
FCD6B668 3E FC926E42 FCD52030 0D Edge 01 atapi!.text+5AE2
FCD67B48 3F FC926E42 FCD68030 0C Edge 01 atapi!.text+5AE2
```

E.g., from the output of “intobj”, We see that the IRQL of IRQs. Eg: IRQ 1’s vector is 0x31, it’s IRQL is 0x1AH. And we also know that driver i8042prt connects this IRQ.

Each IRQ is associated with a vector and the system uses vector rather than IRQ. For widows nt, the vector equals (IRQ number + 0x30), See Softice’s output of command “IRQ”:

```
:irq

IRQ Vector Status
00 30 Unmasked
01 31 Unmasked
02 32 Unmasked
03 33 Unmasked
04 34 Masked
05 35 Masked
06 36 Masked
07 37 Unmasked
08 38 Unmasked
09 39 Unmasked
0A 3A Masked
0B 3B Masked
0C 3C Unmasked
0D 3D Masked
0E 3E Unmasked
0F 3F Unmasked
```

Generally, we call IRQ interrupt under the protect mode. It has two types:

1, NMI (NonMaskable Interrupt)

This type of interrupt is reported to cpu via the NMI pin, it can not be masked by zero the Eflags’s IF bit. It’s vector is 02h.

2, INTR (Maskable Interrupt)

This type could be masked by zero. Cpu Eflags IF bit. They are reported to cpu via INTR pin. But cpu needs to access the data bus to get the vector number.
When eflags IF is zero, the INTR will wait until the IF is set to 1.

Besides interrupt, under protect mode, exceptions also use vector numbers.

Exception has three types: (For detail info, see intel cpu manual.)

1, Fault (Eg. Page Fault 0x0E)
2, Trap ( Eg. NTCall, int 0x2E)
2, Abort ( Severe Errors)

The difference of Fault of Trap is that: The instructions which result in the fault will be executived again after finishing the process of the fault. But for a trap, like (int 0x2e), the instructions which result in the trap will be skipped and the next instruction will be executived.

Exceptions use 0x0 – 0x1f (0x02 is excluded) as their vector numbers.

When an interrupt or exception arise, cpu will clear the eflags IF bit automatically. But for a trap, it will not try to modify the IF bit of eflags.

Cpu register IDTR stores the start address of the table of the entries of vectors.
In Softice, to get the entry of vector 31 (IRQ 1):

```
:idt 31
Int Type Sel:Offset Attributes Symbol/Owner
0031 IntG32 0008:FF263444 DPL=0 P
```

So, when IRQ 1 raises, cpu will automatically run the instructions at 0008:FF263444 as response to the interrupt.

For more detail information, please refer Intel CPU Mannuals.

## Part II:

In this part, we'll discuss how the OS realize SpinLocks via IRQL

### Realization of SpinLocks:

Spin locks are very commonly used in drivers to protect data that will be accessed by multiple driver routines running at varying IRQLs. But what's it's realizaton ?
Every book about "Operating System" will tell us that an atomic test-and-set instruction will be adopted. Is windows os uses this way ? The answer is NO. What windows nt adopts is IRQL. That is the spinlock routines will change the IRQL.

There are some kernel routines to acquire/release SpinLocks available for driver developers. These routines are not strange faces, so I'll not introduce their functionalities here.

```
VOID
KeAcquireSpinLock(
IN PKSPIN_LOCK SpinLock,
OUT PKIRQL OldIrql
);

VOID
KeReleaseSpinLock(
IN PKSPIN_LOCK SpinLock,
IN KIRQL NewIrql
);

VOID
KeAcquireSpinLockAtDpcLevel(
IN PKSPIN_LOCK SpinLock
);

VOID
KeReleaseSpinLockFromDpcLevel(
IN PKSPIN_LOCK SpinLock
);
```

DDk says that “Callers of KeAcquireSpinLock must be running at IRQL <= DISPATCH_LEVEL”, for dispatch_level routines, .it would be better to use KeAcuireSpinLockAtDpcLevel, and that callers of KeReleaseSpinLock are running at IRQL DISPATCH_LEVEL, but why ?

The following codes tell us the answer:

```
kd> u Hal!KeAcquireSpinLock
hal!KeAcquireSpinLock:
80066806 8b4c2404 mov ecx,[esp+0x4]
8006680a e849c7ffff call hal!KfAcquireSpinLock (80062f58)
8006680f 8b4c2408 mov ecx,[esp+0x8]
80066813 8801 mov [ecx],al
80066815 c20800 ret 0x8

hal!KfAcquireSpinLock:
80062f58 33c0 xor eax,eax

// Save current IRQL to al
80062f5a a024f0dfff mov al, [ffdff024]

// Change Current IRQL to Dispatch Level
80062f5f c60524f0dfff02 mov byte ptr [ffdff024],0x2
80062f66 c3 ret
```

Then we get the result: The acquiration of spin lock is just only improve current IRQL to DISPATCH_LEVEL. So the article “A Catalog of NT Synchronization Mechanisms” (refer. 2) says “Always relying on spin locks to protect access to shared data may be overkill”. Because after the SpinLock is acquired , the current IRQL will be DISPATCH_LEVEL and then the NT dispatcher (scheduler) preemption will be disabled.

But for the routine which is already running at DISPATCH_LEVEL, they are advised .to use KeAcquireSpinLockAtDpcLevel instead. We can image what KeAcquireSpinLockAtDpcLevel do?

```
kd> u KeAcquireSpinLockAtDpcLevel
nt!KeAcquireSpinLockAtDpcLevel:
804022e4 c20400 ret 0x4

nt!KeReleaseSpinLockFromDpcLevel:
804022f4 c20400 ret 0x4
```

These two routines do nothing, and just return. As all DISPATCH_LEVEL all the routines will be executed synchronously, they can not interrupt each other, i.e. they are alreary synchronized.

When current IRQL > DISPATCH_LEVEL, we are warned never to call spin lock routines , or we’ll get BSOD. Here we get the reason: KeAcquireSpinLock will try to lower the current IRQL, which is not permitted by NT.

## Part III:

I’ll discuss how the OS realize IRQL with two examples: The process of ISR Synchronize Lock and interrupt service routine.

### 1, ISR Synchronize Lock

SpinLocks have three types:

1, standard spin locks
2, ISR synchronization spin locks. Each type has its own IRQL associations
3, default ISR (Interrupt Service Request) spin locks

For “standard spin locks “, we’ve analyzed it at Part II. Now let’s anaylize the left two locks.

Whne developing a video miniport driver, I’ve ever met such an case: to protect some shared data between StartIO and ISR. As we all know, IRQ runs at DIRQL, and we can not call the stand spinlocks.

I noticed that videoport supported an routine VideoPortAcquireDeviceLock. The ddk does not say more about the limits. So I got BSOD when calling this routine in the ISR.

I disassembled it and found it that it used a dispatch synchronize object (Mutex). See the assembles codes below,

```
:u VideoPortAcquireDeviceLock

0008:EB095392 XOR EAX,EAX
0008:EB095394 PUSH EAX
0008:EB095395 PUSH EAX
0008:EB095396 PUSH EAX
0008:EB095397 PUSH EAX
0008:EB095398 MOV EAX,[ESP+14] //EAX = HwDeviceExtension
0008:EB09539C MOV EAX,[EAX-0228] // The Mutex maintained by VideoPort
0008:EB0953A2 ADD EAX,30
0008:EB0953A5 PUSH EAX
0008:EB0953A6 CALL [__imp__KeWaitForSingleObject]
0008:EB0953AC RET 0004

:u VideoPortReleaseDeviceLock

0008:EB0953B0 MOV EAX,[ESP+04]
0008:EB0953B4 PUSH 00
0008:EB0953B6 MOV EAX,[EAX-0228]
0008:EB0953BC ADD EAX,30
0008:EB0953BF PUSH EAX
0008:EB0953C0 CALL [__imp__KeReleaseMutex]
0008:EB0953C6 RET 0004
```

Luckly the videoport supports another mechanism of VideoPortSynchronizeExecution, which will call KeSynchronizeExecution. So let us analyze KeSynchronizeExecution.

```
BOOLEAN
KeSynchronizeExecution(
IN PKINTERRUPT Interrupt,
IN PKSYNCHRONIZE_ROUTINE SynchronizeRoutine,
IN PVOID SynchronizeContext
);
```

SynchronizeRoutine:

```
BOOLEAN
(*PKSYNCHRONIZE_ROUTINE) ( IN PVOID SynchronizeContext );
```

KeSynchronizeExecution will call KfRaiseIRQL to raise current processor’s IRQL to the InterruptObjects’s SynchronzieIRQL and mask all the IRQs below the SynchronizeIRQL, then execute the SynchronizeRoutine which will operate shared sensitive data, and call KfLowerIrql to do the restoring.at the end.

At the time SynchronizeRoutine is called, the interrupt specified by the interrupt object: interrupt will be masked. The the protection of the access of the shared data is achieved.

Followings are the asm codes, from WinDbg.

```
kd> u nt!KeSynchronizeExecution
nt!KeSynchronizeExecution:
80468a70 55 push ebp
```

```
80468a71 8bec mov ebp,esp
80468a73 83ec04 sub esp,0x4
80468a76 53 push ebx
80468a77 56 push esi

//Raise Current IRQL to Interrupt->SynchronizeIrql
80468a78 8b5d08 mov ebx,[ebp+0x8] //ebx = Interrupt
80468a7b 8b4b29 mov ecx,[ebx+0x29] //cl = Interrupt->SynchronizeIrql
80468a7e ff15d8054080 call dword ptr [nt!_imp_KfRaiseIrql (804005d8)]

80468a84 8845fc mov [ebp-0x4],al // al = Old IRQL
80468a87 8b731c mov esi,[ebx+0x1c]
80468a8a 8b4510 mov eax,[ebp+0x10]

// Now call our SynchronizeRoutine
80468a8d 50 push eax
80468a8e ff550c call dword ptr [ebp+0xc] // SynchronizeRoutine

// Resotring …
80468a91 8bd8 mov ebx,eax
80468a93 8b4dfc mov ecx,[ebp-0x4] // Restore Old IRQL
80468a96 ff15dc054080 call dword ptr [nt!_imp_KfLowerIrql (804005dc)]
80468a9c 8bc3 mov eax,ebx
80468a9e 5e pop esi
80468a9f 5b pop ebx
80468aa0 c9 leave
80468aa1 c20c00 ret 0xc


kd> u hal!kfRaiseIrql
hal!KfRaiseIrql:
80062ea0 33c0 xor eax,eax
80062ea2 a024f0dfff mov al,[ffdff024] // Current IRQL
80062ea7 0fb6c9 movzx ecx,cl
80062eaa 80f902 cmp cl,0x2 // DISPATCH_LEVEL = 0x02
80062ead 7625 jbe hal!KfRaiseIrql+0x34 (80062ed4)
80062eaf 8bd0 mov edx,eax
80062eb1 9c pushfd
80062eb2 fa cli

// Change IRQL to new
80062eb3 880d24f0dfff mov [ffdff024],cl

// Besides changing current IRQL, it also mask all the IRQs below the SynchronizeIRQL
80062eb9 8b048dcc890680 mov eax,[hal!KiI8259MaskTable (800689cc)+ecx*4]
80062ec0 0b0530f0dfff or eax,[ffdff030]
80062ec6 e621 out 21,al // Mask I8259 – 1 (IMR1)
80062ec8 c1e808 shr eax,0x8
80062ecb e6a1 out 0xA1, al // Mask I8259 – 2 (IMR2)
80062ecd 9d popfd
80062ece 8bc2 mov eax,edx
80062ed0 c3 ret
80062ed1 8d4900 lea ecx,[ecx]
80062ed4 880d24f0dfff mov [ffdff024],cl
80062eda c3 ret

hal!KfLowerIrql:
80062f10 9c pushfd
80062f11 0fb6c9 movzx ecx,cl

80062f14 803d24f0dfff02 cmp byte ptr [ffdff024],0x2

80062f1b fa cli
80062f1c 7614 jbe hal!KfLowerIrql+0x22 (80062f32)
```

```
// Only need running when Current IRQL > DISPATCH_LEVEL
// Restore the 8259 settings
80062f1e 8b048dcc890680 mov eax,[hal!KiI8259MaskTable (800689cc)+ecx*4]
80062f25 0b0530f0dfff or eax,[ffdff030]
80062f2b e621 out 21,al
80062f2d c1e808 shr eax,0x8
80062f30 e6a1 out a1,al

// Restore current IRQL
80062f32 880d24f0dfff mov [ffdff024],cl

80062f38 a128f0dfff mov eax,[ffdff028]
80062f3d 8a80648a0680 mov al,[eax+0x80068a64]
80062f43 38c8 cmp al,cl
80062f45 7705 ja hal!KfLowerIrql+0x3c (80062f4c)
80062f47 9d popfd
80062f48 c3 ret
80062f49 8d4900 lea ecx,[ecx]
80062f4c ff14854c8a0680 call dword ptr [hal!SWInterruptHandlerTable (80068a4c)+eax*4]
80062f53 9d popfd
80062f54 c3 ret
```

## 2, Default ISR (Interrupt Service Request) spin locks

When an interrupt raises, cpu will be notified by a signal via the INTR pin, then the cpu will read the vector from the data bus. After saving the current executing envrioment, cpu will get the entry of the interrupt from IDTR and execute the ISR routine.

Here the ISR routine is not just the service routine supported by the specific driver when connecting the IRQ. It's hooked by Windows. Windows will decide whether the interrupt request is from the device. If it is surely form a device, KiInterruptDispatch will be called to process the interrupt request.

KiInterruptDispatch will first call HalBeginSystemInterrupt to raise current IRQL and mask 8259A. Then call the service routine at KINTERRUPT object offset 0x0C, which do the really work to perform the interrupt request. Then KiInterruptDispatch calls HalDisableSystemInterrupt to do the restoring work. Then the response of an interrupt request ends.

Here we will analyze the process of Vector 0x31 as example.

```
// Get the entry address of Vector 31 (IRQ 1)
:idt 31
Int Type Sel:Offset Attributes Symbol/Owner
0031 IntG32 0008:FF263444 DPL=0 P


kd> u 0008:FF263444
ff263444 54 push esp
ff263445 55 push ebp
ff263446 53 push ebx
ff263447 56 push esi
ff263448 57 push edi
ff263449 83ec54 sub esp,0x54
ff26344c 8bec mov ebp,esp
ff26344e 89442444 mov [esp+0x44],eax
ff263452 894c2440 mov [esp+0x40],ecx
ff263456 8954243c mov [esp+0x3c],edx

// Current Stack:
// ESP + 0: db * 0x54
// ESP + 54: edi
// ESP + 58: esi
// ESP + 5C: ebx
// ESP + 60: ebp
// ESP + 64: esp
// ESP + 68: Old EIP
// ESP + 6C: Old CS
// ESP + 70: Old Eflags

// Eflags bit 17 is VM bit, if VM = 1, it shows that the caller is from V86 mode,
// If the caller is from V86 mode, jump to ff26357b
ff26345a f744247000000200 test dword ptr [esp+0x70],0x20000
ff263462 0f8513010000 jne ff26357b

// For kernel space, CS = 0x08, it need not save the segments
// For user space (?), it will change current segments registers

ff263468 66837c246c08 cmp word ptr [esp+0x6c],0x8
ff26346e 7423 jz ff263493
ff263470 8c642450 mov [esp+0x50],fs
ff263474 8c5c2438 mov [esp+0x38],ds
ff263478 8c442434 mov [esp+0x34],es
ff26347c 8c6c2430 mov [esp+0x30],gs
ff263480 bb30000000 mov ebx,0x30
ff263485 b823000000 mov eax,0x23
ff26348a 668ee3 mov fs,bx
ff26348d 668ed8 mov ds,ax
ff263490 668ec0 mov es,ax

// ? Modify the exception record structure
ff263493 648b1d00000000 mov ebx,fs:[00000000]
ff26349a 64c70500000000ffffffff mov dword ptr fs:[00000000],0xffffffff

ff2634a5 895c244c mov [esp+0x4c],ebx
ff2634a9 81fc00000100 cmp esp,0x10000
ff2634af 0f829e000000 jb ff263553
ff2634b5 c744246400000000 mov dword ptr [esp+0x64],0x0
ff2634bd fc cld
ff2634be f60550f0dfffff test byte ptr [ffdff050],0xff
ff2634c5 750c jnz ff2634d3

ff2634c7 bf083426ff mov edi,0xff263408 // IntOBJ
ff2634cc e9cf562081 jmp nt!KiInterruptDispatch (80468ba0)

ff2634d1 8bff mov edi,edi
ff2634d3 f7457000000200 test dword ptr [ebp+0x70],0x20000
ff2634da 7509 jnz ff2634e5
ff2634dc f7456c01000000 test dword ptr [ebp+0x6c],0x1
ff2634e3 74e2 jz ff2634c7
ff2634e5 0f21c3 mov ebx,dr0

…

kd> u KiInterruptDispatch
nt!KiInterruptDispatch:
80468ba0 ff0560f5dfff inc dword ptr [ffdff560]
80468ba6 8bec mov ebp,esp
80468ba8 8b4724 mov eax,[edi+0x24]
80468bab 8b4f29 mov ecx,[edi+0x29]
80468bae 50 push eax
80468baf 83ec04 sub esp,0x4

// Initialize Current IRQL & 8259A
80468bb2 54 push esp
80468bb3 50 push eax
80468bb4 51 push ecx
80468bb5 ff1580054080 call dword ptr [nt!_imp__HalBeginSystemInterrupt (80400580)]
80468bbb 0bc0 or eax,eax
80468bbd 741a jz nt!KiInterruptDispatch+0x39 (80468bd9)
```

```
80468bbf 8b771c mov esi,[edi+0x1c]
80468bc2 8b4710 mov eax,[edi+0x10]

// Now call IntObj - ISR
80468bc5 50 push eax
80468bc6 57 push edi
80468bc7 ff570c call dword ptr [edi+0xc]
80468bca fa cli

// Restore …
80468bcb ff1584054080 call dword ptr [nt!_imp__HalEndSystemInterrupt (80400584)]

// Finish the process of the interrupt
80468bd1 e9e0c7ffff jmp nt!Kei386EoiHelper (804653b6)
80468bd6 83c408 add esp,0x8
80468bd9 83c408 add esp,0x8
80468bdc e9d5c7ffff jmp nt!Kei386EoiHelper (804653b6)

kd> u hal!HalBeginSystemInterrupt

hal!HalBeginSystemInterrupt:
80067ab8 0fb65c2408 movzx ebx,byte ptr [esp+0x8]

// Ebx = Vector Number, Ebx – 0x30 = IRQ number
80067abd 83eb30 sub ebx,0x30

//hal!HalpSpecialDismissTable = 80068a6c
80067ac0 ff249d6c8a0680 jmp dword ptr [hal!HalpSpecialDismissTable+ebx*4]


hal!HalBeginSystemInterrupt+3b:
// Entry of IRQ 1
80067af3 8b44240c mov eax,[esp+0xc]

// Save current IRQL, and change to new value from CL
80067af7 0fb70d24f0dfff movzx ecx,word ptr [ffdff024]
80067afe 8808 mov [eax],cl
80067b00 0fb6442404 movzx eax,byte ptr [esp+0x4]
80067b05 a224f0dfff mov [ffdff024],al // al = 0x1a

// Mask 8259A IRQs
80067b0a 8b0485cc890680 mov eax,[hal!KiI8259MaskTable (800689cc)+eax*4]
80067b11 0b0530f0dfff or eax,[ffdff030]

// eax = fffffefa [ffdff030] = 70 2c ff ff | Result: fffffefa
// Mask 8259A (IMR 1/2 (with Timer / Real Time exculed))

80067b17 e621 out 21,al
80067b19 c1e808 shr eax,0x8
80067b1c e6a1 out a1,al

80067b1e 8bc3 mov eax,ebx
80067b20 83f808 cmp eax,0x8
80067b23 7306 jnb hal!HalBeginSystemInterrupt+0x73 (80067b2b)
80067b25 0c60 or al,0x60
80067b27 e620 out 20,al
80067b29 eb08 jmp hal!HalBeginSystemInterrupt+0x7b (80067b33)
80067b2b b020 mov al,0x20
80067b2d e6a0 out a0,al

80067b2f b062 mov al,0x62
80067b31 e620 out 20,al
80067b33 e421 in al,21

80067b35 fb sti
```

```
80067b36 b801000000 mov eax,0x1
80067b3b c20c00 ret 0xc


// HalDisableSystemInterrupt is just reverse to HalBeginSystemInterrupt

hal!HalDisableSystemInterrupt:
80067b40 0fb64c2404 movzx ecx,byte ptr [esp+0x4]
80067b45 83e930 sub ecx,0x30
80067b48 ba01000000 mov edx,0x1
80067b4d d3e2 shl edx,cl
80067b4f fa cli
…
800630b7 e621 out 21,al
800630b9 c1e808 shr eax,0x8
800630bc e6a1 out a1,al
800630be 880d24f0dfff mov [ffdff024],cl
800630c4 a128f0dfff mov eax,[ffdff028]
800630c9 8a80648a0680 mov al,[eax+0x80068a64]
800630cf 38c8 cmp al,cl
800630d1 7703 ja hal!HalEndSystemInterrupt+0x3a (800630d6)
800630d3 c20800 ret 0x8
…
```

【References】

1, www.osr.com Nt Insider 1997: “A Catalog of NT Synchronization Mechanisms”
2, “practice of 80386/486 system programming” by XiaoQing Lieu (Chinese)

Any questions or errors, just mail to me.

Thanks!

Matt

http://sys.xiloo.com
mattwu@163.com

Written at Apr. 23 2002
Updated at Apr. 28 2002

.. **--= End =--**

Last Modified by Matt at Saturday 10/21/2006

IE & 1024x768

Home | Projects | Documents | Resources | History| Contact Me