Using Interrupts on Trizeps 4

Documentation 1.0

This document describes how to use IRQ's to map Interrupt-Sources to an Interrupt-Service-Routine (ISR) or an Interrupt-Service-Thread (IST).

1.0 Introduction

Real-time applications use interrupts to respond to external events in a timely manner. The use of interrupts requires that an operating system (OS) balance performance against ease of use. Microsoft Windows CE balances these two factors by splitting interrupt processing into two steps: an interrupt service routine (ISR) and an interrupt service thread (IST).

Each interrupt request (IRQ) is associated with an ISR; an ISR can respond to multiple IRQ sources. When interrupts are enabled and an interrupt occurs, the kernel calls the registered ISR for that interrupt. Once finished, the ISR returns an interrupt identifier. The kernel examines the returned interrupt identifier and sets the associated event. When the kernel sets the event, the IST starts processing.

The exception handler is the primary target of all interrupts. When an interrupt occurs, the microprocessor transfers control to an exception handler in the kernel. The exception handler then calls the ISR registered to handle the current interrupt. The ISR is responsible for translating the interrupt into a logical interrupt identifier, a SYSINTR, which it passes to the kernel as its return value. The kernel sets an event associated with the logical interrupt, which causes an interrupt service thread (IST) to be scheduled. Code in the IST is responsible for servicing the device interrupt. The IST runs in the context of a thread in Device Manager and is essentially a typical application thread running at a high priority.

(Taken from Windows CE 5.0 Platform-Builder Help „Interrupt Handling Process Overview“)

2.0 IRQ's availlable on the Trizeps4

IRQs's are used to assign every interrupt-source a unique number. If an interrupt-event occurs, the processor will stop execution immediatly and jump to the exception-handler. The exception-handler will check which interrupt occured and will use the irq-number of this interrupt to determine which ISR to call. The ISR will return a SysIntr-Value, which may trigger an event to an IST.

There are two main IRQ-number-spaces.

1. System IRQ's, which are assigned to the main PXA27x-components.
2. GPIO IRQ's, which are assigned to the General-Purpose-IO-Pins.

2.1 System IRQ-Numbers

View the Intel PXA27x Developer Manual on details regarding these interrupts.

TABLE 1. System IRQ-Numbers (defined in .\tr4conxs\src\inc\oalintr.h):

IRQ	**Description**
0	Synchronous Serial Port 3
1	Mobile Scalable Link Interface (Baseband)
2	USB-Host (non OHCI)
3	USB-Host (OHCI)
4	Keypad
5	MemoryStick-Host Controller
6	Power I²C
7	Timer (currently not supported, internal system timer)
8	GPIO 0
9	GPIO 1
10	GPIO (not GPIO 0 or 1) (use special GPIO-IRQ's instead)
11	USB Function Controller
12	Performance Monitor
13	Inter-IC Sound (I²S)
14	AC97
15	Universal Subscriber ID Interface (USIM)
16	Synchronous Serial Port 2
17	LCD Controller
18	I²C
19	Infrared Communication Port (ICP)
20	Standard-UART (COM3)
21	Bluetooth-UART (COM2)
22	Full-Function-UART (COM1)
23	SD/MMC Controller
24	Synchronous Serial Port 1
25	DMA Controller (Audio-driver)
26	Timer 0 (internal High-Res-Counter)
27	Timer 1 (Touch-Sample-Timer)
28	Timer 2 (CPLD-Matrix-Keyboard)
29	Timer 3
30	Real-Time Clock (1 Hz Tick)
31	Real-Time Clock (Alarm)
33	Camera Interface

2.2 GPIO IRQ-Numbers

General-Purpose-Pins on the PXA27x-processor can be configured to cause interrupts on edge-triggered events. The GPIO-interrupt-handler also enables the use of level-triggered interrupts through software.

GPIO IRQ-Numbers are coded values, which contain the gpio-number, the events causing the interrupt and some additional flags (view .\tr4conxs\src\inc\oalintr.h).

```
#define IRQ_TYPE                0xFF000000
#define IRQ_SYSTEM              0x00000000
#define IRQ_GPIO                0x01000000

// Defines for IRQ_GPIO
#define IRQ_GPIO_NUMBER         0x0000007F
#define IRQ_GPIO_FLAGS          0x00FFFF00
#define IRQ_GPIO_RISING         0x00000100
#define IRQ_GPIO_FALLING        0x00000200
#define IRQ_GPIO_HIGH           0x00000400
#define IRQ_GPIO_LOW            0x00000800

// For special cases, where you don't want to clear
// interrupt-detection during Interrupt and InterruptDone():
#define IRQ_NOCLEARFALLING   0x00001000
#define IRQ_NOCLEARRISING    0x00002000
```

Example:

IRQ = 0x0100020C

The upper 8 bits code the IRQ-Type: 0x01 => IRQ_GPIO.

The lower 7 bits code the GPIO-Number: 0x0C => GPIO 12.

The other bits code the interrupt-behaviour: 0x0002 => Interrupt on falling edge.

The special-flags IRQ_NOCLEARFALLING and IRQ_NOCLEARRISING may be used to prevent the GPIO-interrupt-handler from disabling the interrupts for falling or rising edge interrupts. Normaly interrupts for a gpio are disabled in the interrupt-handler and will be enabled again after a call to InterruptDone() (view Platform-Builder Help).

3.0 Mapping IRQ's to Sysintr-Values

Before using an IRQ in your software, you have to map it to a sysintr-value. There are 64 sysintr-values availlabe in WinCE 5.0. About 25 are already mapped to drivers.

A little sample-application named „irq_sample“ is availlable for download.

3.1 IOCTL_HAL_REQUEST_SYSINTR

IRQ's are mapped to a sysintr-value using the kernel-ioctl: IOCTL_HAL_REQUEST_SYSINTR.

Example:

```
DWORD dwIrq      = <IRQ-Number to map>;
DWORD dwSysIntr = <Variable receiving the mapped SysIntr-Value);
KernelIoControl( IOCTL_HAL_REQUEST_SYSINTR,
                  &dwIrq,
                  sizeof(DWORD),
                 &dwSysIntr,
                 sizeof(DWORD),
                 NULL);
```

The KernelIoControl returns TRUE on success. FALSE indicates failure.

3.2 IOCTL_HAL_RELEASE_SYSINTR

If you should no longer need a sysintr-value previously assigned with IOCTL_HAL_REQUEST_SYSINTR, call the kernel-ioctl: IOCTL_HAL_RELEASE_SYSINTR.

Example:

```
DWORD dwSysIntr = <Variable containing the mapped SysIntr-Value);
KernelIoControl(  IOCTL_HAL_RELEASE_SYSINTR,
                  &dwSysIntr,
                  sizeof(DWORD),
                  NULL,
                  0,
                  NULL);
```

4.0 Catching an Interrupt with an Interrupt-Service-Thread (IST)

The simplest way of catching interrupt-events in your application or in a driver is to use an interrupt-service-thread. The IST is a normal thread, which waits for the interrupt-event to occur through a call to WaitForSingleObject(...).

Unlike an ISR, which gets executed immidiatly after the interrupt occurs, the execution time of the IST is controlled through the scheduler, using a priority-sheme. You can set the priority of your IST with a call to CeSetThreadPriority(..). Applications run at a priority between 248 and 255, most drivers run at a priority around 100. Avoid using a high priority (<100), especially if you poll for events or do large data-processing. Threads running at low priority only get executed, if higher-priority-threads are suspended (Sleep(..), WaitForSingleObject(..), WaitForMultipleObject(..),...) !!

4.1 Initialize the Interrupt-Event

Before using interrupts, you have to create an interrupt-event and assign it to a sys-intr-value. Events are used to inform threads and the scheduler, that things have happened.

1. Creating an interrupt-event:
   HANDLE hInterrupt = <Handle to the Interrupt-Event>;
   hInterrupt = CreateEvent(NULL, FALSE, FALSE, NULL);
2. Assigning a sysintr-value to an interrupt-event:
   DWORD dwSysIntr = <Variable containing the Sysintr-Value>
   InterruptInitialize(dwSysIntr, hInterrupt, NULL, NULL);

InterruptInitialize(..) will also enable the interrupt. From now on, the system will signal interrupts to hInterrupt if they occur.

4.2 Waiting for the Interrupt-Event

You wait for an interrupt-event with a call to

WaitForSingleObject(hInterrupt, INFINITE);

WaitForSingleObject suspends the thread and will tell the scheduler to only wake it, if hInterrupt gets signaled or on timeout. In this example the timeout-value is set to INFINITE. You may also set it to a time in milliseconds. The return-value is WAIT_OBJECT_0 if an interrupt occured, WAIT_TIMEOUT if the specified time-out has been reached, or WAIT_FAILED if there was any kind of error.

4.3 Disabling the Interrupt-Event

If you no longer need to watch for the interrupt, you should call:

InterruptDisable(dwSysIntr);
CloseHandle(hInterrupt);

5.0 Running an Interrupt-Service-Routine (ISR) on Interrupts

As already mentioned in Chapter 4, Interrupt-Service-Routines get called directly after an interrupt-event occured. Any other processing is stopped, interrupts are turned off and the only thing running will be your ISR. This also means, that those routines should only do short, time-critical processing!

An ISR is a Dll which gets loaded into kernel-process-space through a call to Load-IntChainHandler(..). This Dll can use the whole virtual-process-space of the kernel (view .\tr4conxs\inc\memmap.inc). It is not allowed to be linked to other Dll's and must export following functions:

- CreateInstance (called on every call to LoadIntChainHandler).
- DestroyInstance (called on a call to FreeIntChainHandler).
- IOControl (called on a call to KernelLibIoControl).
- ISRHandler (called on every interrupt associated with this ISR).

To check wether your ISR exports the above functions, or uses other dll's, you can use depends.exe, which comes with Embedded Visual C++ (.\Programs\Microsoft eMbedded C++ 4.0\Common\Tools\depends.exe)

A sample „isr_sample" is availllable for download. It contains an application to load an ISR and the ISR itself.

5.1 Writing an Interrupt-Service-Routine

View Platform-Builder-Help on details how to write an ISR.

If you write an ISR, don't forget that you're responsible for turning interrupts on and off!

5.1.1 System-IRQ ISR

After a System-IRQ takes place, the interrupt-handler will run the ISR's connected to this IRQ (Yes, you may connect multiple ISR's to one IRQ).

Your ISR may return these values:

- SYSINTR_CHAIN: Call the next ISR in the list.
- SYSINTR_NOP: Signal „no interrupt occured" to the system (No more ISR's get called).
- SYSINTR_xxx: Signal an interrupt event, which can be catched through an IST (No more ISR's get called).

The default behaviour (;all your ISR's return SYSINTR_CHAIN) is, that the interrupt to this peripheral is disabled. It will be reenabled on a call to InterruptDone().

You may use the macro INTC_DISABLE(x) and INTC_DISABLE2(x) (defined in INTCBits.h) to disable the interrupts in you ISR. Usage:

- INTC_DISABLE(1 << irq) for irq-numbers 0..31.
- INTC_DISBALE2(2) for irq 33 (camera interface).

5.1.2 GPIO-IRQ ISR

With exception of GPIO 0 and 1 (which both are treated as System-IRQ), all other gpio's use a special interrupt-handler. This is what the handler does on interrupt at IRQ 10 (GPIO 2 to 120 share one System-IRQ):

1. Look for the GPIO, which caused the Interrupt.
2. Clear the responsible bit off the Edge-Detect-Status-Register (GEDR)
3. Clear Rising-Edge-Detect-Enable-Register (GRER) if IRQ_NOCLEARRISING is not set.
4. Clear Falling-Edge-Detect-Enable-Register (GFER) if IRQ_NOCLEARFALLING is not set.
5. Translate IRQ to SysIntr.
6. Call ISR's.

If rising- or falling-edge-detect is cleared, you must reenable edge-detection in your ISR or IST. View the isr_sample for details.

5.2 Loading an Interrupt-Service-Routine

An ISR is loaded with a simple call to LoadIntChainHandler(..).

Example:

```
HANDLE hIsrHandle = <Handle to the ISR; used for calls to KernelLibIoControl(.)
                    and FreeIntChainHandler(.)>
DWORD dwSysIntr   = <Number of the SysIntr-Value associated to the IRQ>

hIsrHandle = LoadIntChainHandler( TEXT("myisr.dll"),
                                  TEXT("ISRHandler"),
                                  (BYTE) dwSysIntr);
```

In this example, myisr.dll is the name of the ISR-Dll to load and ISRHandler is the name of the function to call on interrupt.

The third parameter (dwSysIntr) is not used as described in the Microsoft-Platform-Builder-Help. In the help it states, that the third parameter should be bIRQ, the IRQ-number of the interrupt. In our Board-Support-Package, the third value is the sysintr-value you retrieve when calling IOCTL_HAL_REQUEST_SYSINTR.

5.3 Communication with an ISR

You can communicate with an ISR through KernelLibIoControl(..)-calls.

Example:

```
HANDLE hIsrHandle = <Handle returned by LoadIntChainHandler(..)>.
int Inbuf[ 10]    = <Buffer containing data to ISR>
int Outbuf[ 10]   = <Buffer getting data from ISR>

KernelLibIoControl(  hIsrHandle ,
                     IOCTL_xxx,
                     InBuf,
                    sizeof(InBuf),
                    OutBuf,
                    sizeof(OutBuf),
                    lpBytesReturned)
```

5.4 Unloading an ISR

If you no longer use the ISR for an IRQ, you may free the instance with a call to FreeIntChainHandler.

Example:

```
HANDLE hIsrHandle = <Handle returned by LoadIntChainHandler(..)>.

FreeIntChainHandler( hIsrHandle);
```

This function will call DestroyInstance(..) from your ISR-Dll.

6.0 Using Interrupts to Wake the Module from Sleep

Some interrupts may be used to wake the Trizeps4-module from sleep.

You can set or remove a wakeup-source through KernelIoControls:

- IOCTL_HAL_ENABLE_WAKE
- IOCTL_HAL_DISABLE_WAKE

Example:

```
DWORD dwSysIntr = < SysIntr-value returned by
                    IOCTL_HAL_REQUEST_SYSINTR; this is not the IRQ-
                    value >

KernelIoControl(    IOCTL_HAL_ENABLE_WAKE,
                   &dwSysIntr,  sizeof(dwSysIntr), NULL, 0, NULL);
KernelIoControl(    IOCTL_HAL_DISABLE_WAKE,
                   &dwSysIntr,  sizeof(dwSysIntr), NULL, 0, NULL)
```

TABLE 2. Wakeup-Sources for Trizeps4 (view .\tr4conxs\src\inc\oalintr.h):

IRQ-Number	**Description**
IRQ_USBOHCI	USB-Host (OHCI)
IRQ_USBNONOHCI	USB-Host (non-OHCI)
IRQ_RTCALARM	Real-Time-Clock-Alarm
IRQ_USBFN	USB-Slave
IRQ_GPIO0	GPIO 0
IRQ_GPIO1	GPIO 1
IRQ_GPIOXX_2 (If GPIO's are listed in one line, only one of them may serve as wakeup-source; view PXA27x-Developer-Manual for details).	
	GPIO 3
	GPIO 4
	GPIO 9
	GPIO 10
	GPIO 11
	GPIO 12
	GPIO 13
	GPIO14
	GPIO 15
	GPIO 36, 38, 40, 53
	GPIO 31, 113
	GPIO 116 <USIM Card detect>
	GPIO 35
	GPIO 83 <MSL port>