Using Timers on Trizeps 4

Documentation 1.0

This documents describes how to use Operating System Timers of the Trizeps4 under WinCE.

1.0 Introduction

The PXA27x processor has two timer channels. The first set, which provides one counter and four match registers, is reserved for use by the Windows CE kernel. The second set provides eight counters and eight match registers, which can be clocked from either the 32.768-kHz timer clock, a 13-MHz clock, or an externally supplied clock, allowing a wide range of timer resolutions. Only the first counter (Timer4) of this set is used by the kernel, leaving up to 7 timers for your application. Timer4 is configured to be the systems millisecond timer, which counts the milliseconds since the device booted. You may get the value of Timer4 through a call to GetTickCount().

All timers may be configured to have a resolution of:

- 1/32768 of a second
- 1 microsecond
- 1 millisecond
- 1 second
- External supplied clock.

Timers 8/10 and 9/11 provide addtional clock-inputs and modes (; view the PXA27x Processors Family Developers Manual for details).

Before using one of the free timers, they must be assigned to a free system-interrupt-value. This is done through an interrupt-request-mechanism (,see the application note: „Using Interrupts on Trizeps 4“). Once assigned, you may register an Interrupt-Service-Routine (ISR) or use a simple Interrupt-Service-Thread (IST) to control the timer usage.

A sample named „timer_sample“ is availlable for download.

The timer functionality will be availlable as of BSP_TR4CONXS_2005Q4.

2.0 Getting a free Timer

As already mentioned, you may get a free timer through the interrupt-request-mechanism. Each timer has an individual IRQ-number. The IRQ-number consist of the number of the timer stored at the lower 8bits or'd with a flag (0x02000000), which identifies this IRQ to be a timer-irq.

TABLE 1. Timer IRQ's

IRQ	**Timer**
0x02000004	Timer 4 (reserved)
0x02000005	Timer 5
0x02000006	Timer 6
0x02000007	Timer 7
0x02000008	Timer 8
0x02000009	Timer 9
0x0200000A	Timer 10
0x0200000B	Timer 11
0x020000FF	Get a free Timer.

To request a timer you must make a call to the kernel-ioctl: IOCTL_HAL_REQUEST_SYSINTR.

Example:

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

If the wanted timer has already been allocated by another application, the above KernelIoControl(..) will return FALSE.

If you use 0x020000FF to get a free timer, you should also make a call to the kernel-ioctl: IOCTL_HAL_TRANSLATE_SYSINTR, to retrieve its number.

Example:

```
KernelIoControl( IOCTL_HAL_TRANSLATE_SYSINTR,
            &dwSysIntr,
            sizeof(DWORD),
            &dwIrq,
            sizeof(DWORD),
            NULL);
```

This time dwSysIntr is the input-value and dwIrq is the output-value, which will receive the timer-number (or’d with 0x02000000).

3.0 Setting up the Timer

Now that you have a timer, we can go on and configure it. The timers are configured through internal registers of the PXA27x.

Each one has a:

- OMCR (OS Match Control Register), which controls what timer resolution and what kind of synchronisation to use.
- OSCR (OS Timer Count Register), which holds the current count-value of the timer.
- OSMR (OS Timer Match Register), which holds the match-value to be compared against the OSCR.

All timers share following registers:

- OSSR (OS Timer Status Register), which contains a bit for each timer, indicating if a match has occured since the last clear.
- OIER (OS Timer Interrupt Enable Register), which lets you enable/disable interrupts for your timer.

For details on these registers view the „PXA27x Processor Family Developer’s Manual“.

The procedure to set up a timer is as follows:

1. Write OMCR with desired configuration.
2. Write OSMR with a value to match against.
3. Write OSCR with any starting value (i.e. 0). This will start the timer.

4.0 What to do on Interrupts

This depends on what approach you have chosen to use, ISR or IST.

4.1 What to do, if you use an IST

If you use an IST, the kernel takes care of disabling (through OIER) and clearing (through OSSR) the timer interrupt. After this is done, it signals the System-Interrupt previously assigned with IOCTL_HAL_REQUEST_SYSINTR.

In the IST you may change the match-register or reconfigure your timer completly. After you’re done with that, you should make a call to InterruptDone(..) to reenable the timer-interrupt.

Note:
Extreme care should be taken if you may want to change OIER. The IST may be interrupted by other timer interrupts, which change OIER. Use InterruptDone(..)

and InterruptDisable(..) to enable/disable interrupts. You have to call InterruptInitialize(..) to reenable interrupts after InterruptDisable(..).

4.2 What to do, if you use an ISR

If you use an ISR, the kernel calls your Interrupt-Service-Routine directly (,with little overhead of code) after the interrupt occured. This is why I would recommend using an ISR instead of an IST. It could take some milliseconds before an IST gets processed.

The kernel does not change any of the timer-registers, to let the ISR be as flexible as possible. This also means, that you have to take care of clearing and disabling/ enabling the interrupt.

Procedure to be done in the ISR:

1. Write a 1 to the bit-position of your timer in the OSSR. This will clear the interrupt-flag.
2. Optionally disable the timer-interrupt through OIER. This might be useful, if you want to call an IST, which reenables the timer through a call to Interrupt-Done(..) or if you just don't want timer-interrupts anymore.
3. Optionally reprogram OSMR or other timer-registers.
4. And do what ever you like to be done on every timer interrupt. Note that an ISR is the only thing running while it is beeing processed. All interrupts are turned off and loops or bad code might hang the system.

5.0 Cleaning up

If you don't need a timer anymore, you should free the timer-ressource, so that other applications may use it.

To do this, make a call to InterruptDisable(..) to disable the timer-interrupt and call the kernel-ioctl IOCTL_HAL_RELEASE_SYSINTR to release the ressource.

Example:

```
DWORD dwSysIntr = <Variable containing SysIntr-Value, received through
IOCTL_HAL_REQUEST_SYSINTR>;

InterruptDisable( dwSysIntr);
KernelIoControl(  IOCTL_HAL_RELEASE_SYSINTR,
                  &dwSysIntr,
                  sizeof(DWORD),
                  NULL,
                  0,
                  NULL);
```