UT80CRH196KD Interrupt Service Routines

Interrupt service routines (ISRs) are some of the most important tasks facing the software engineer during system development. The software engineer must always consider ISR efficiency, and robustness. ISR efficiency is very important because the servicing of an interrupt requires the microcontroller or microprocessor to jump out of its normal program flow and begin operating on instructions that can be considered overhead to the system. In other words, the microcontroller has to stop working toward its main objective to begin working on menial side tasks. If the ISR is not efficient in the way that it handles this overhead workload, it may adversely affect system performance because the microcontroller will not be able to acheive its main objectives in time.

The second major ISR issue confronting the software engineer is handling other interrupts that occur while the microcontroller is in the middle of servicing an interrupt. Depending on the priority of an incoming interrupt, the microcontroller may jump out of one ISR, and start another one; effectively nesting interrupt services. A common problem associated with interrupt servicing is the way that the interrupt mask and pending registers are modified when an ISR is started. These modifications must be done in an appropriate order, and efficiently in order to decrease the risk of missing an interrupt that may occur during these register modifications.

This application note offers some interrupt handling hints to help you write ISRs for the UT80CRH196KD with little risk of loosing an interrupt. The first part of this application note will discuss the initialization of interrupts following initial power-up for the UT80CRH196KD. The second section will cover the modification of pending interrupt bits, the enabling, and the disabling of interrupts from inside the ISR. Additionally, a few skeleton interrupt service routines will be provided to help clarify the correct way to modify the pending interrupt bits.

Initializing Interrupts:

During the system initialization, and following the system reset, all interrupts are disabled, and no interrupts will be pending. Furthermore, the peripheral transaction server will be disabled, and will not be configured to service any interrupts. Therefore, it is up to the system initialization routine to configure the interrupt mask, interrupt pending, interrupt priority, PTS service, PTS select, and PSW registers to handle interrupts appropriately.

The interrupt initialization should be performed according to the flow chart in Figure 1. You should note that any interrupts that you want to have serviced by the PTS must also be enabled by setting the correct bit in the interrupt mask register and then setting the global PTS enable via the “epts” instruction. Additionally, ther may be an equivalent standard interrupt service routine for each interrupt that will be serviced by the PTS. The reason for this standard interrupt service routine is to reconfigure the PTS to begin servicing the corresponding interrupt whenever the PTS count has reached zero, and the PTS disables itself from serving the respective interrupt.

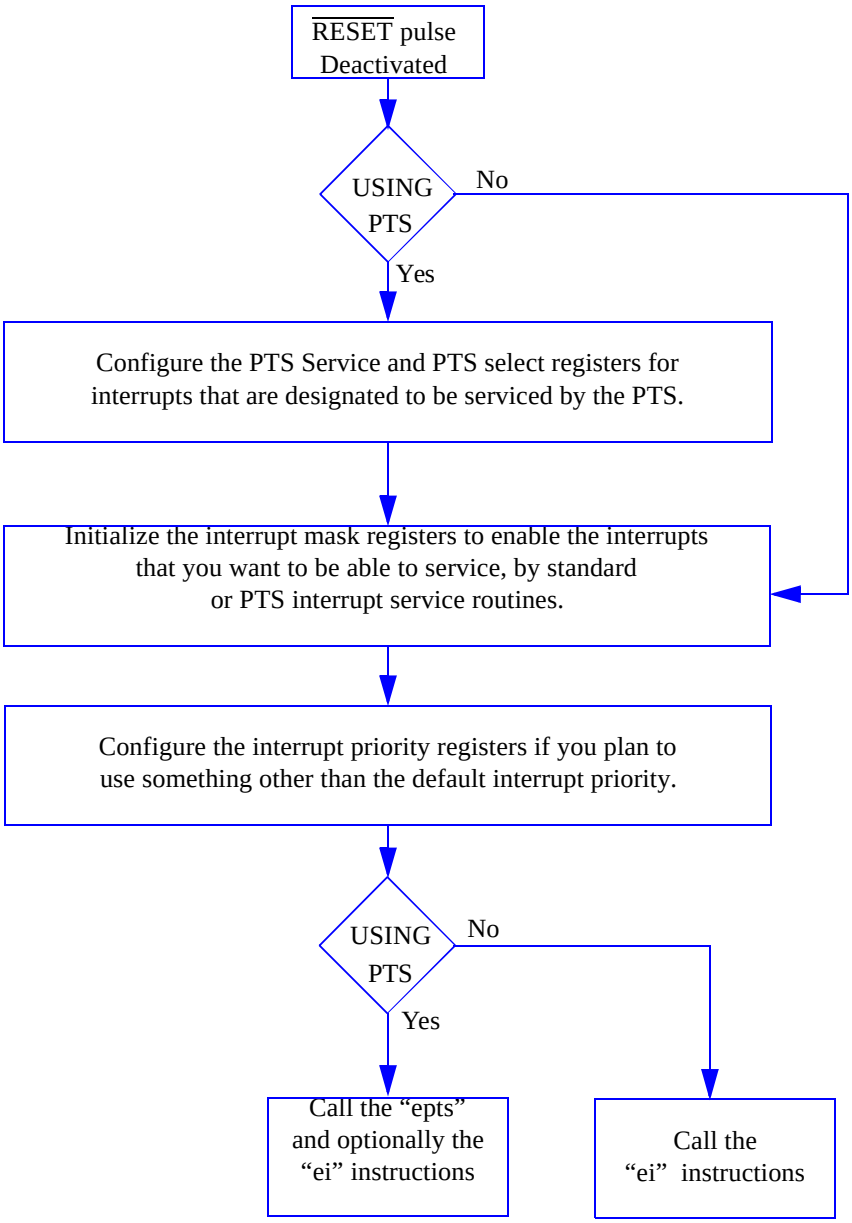

Figure 1. Interrupt Initialization Flow Chart

Clearing and Setting Interrupt Pending Bits:

Whenever an interrupt is serviced, its pending interrupt bit is cleared by the UT80CRH196KD. However, there may be other bits still set in the respective pending interrupt register. These other pending interrupt bits can be modified via software, but, it is highly recommended that you disable all interrupts prior to the instruction that modifies the interrupt pending register. The interrupt disable can be performed by executing one of the following 3 instructions:

1. PUSHA (Pushes the PSW, INT_MASK, INT_MASK1, and WSR registers onto the stack,

and clears each register)

2. PUSHF (Pushes the PSW onto the top of the stack, and clears it)
3. DI (Disable interrupts command, clears the interrupt enable bit in the PSW)

After the microcontroller has executed one of the above 3 instructions, you can change the contents of the interrupt pending registers by using a 2 operand read/modify/write instruction. See Example 1, and 2 for examples of a 2 operand read/modify/write instruction.

Any interrupts that occur during the read/modify/write instruction will be held active until completion of the instruction. This behavior guarantees that any incoming interrupts will not be lost while the pending interrupt registers are being modified. After the modification process is complete, you can restore the interrupts to their original configuration by executing one of the following 3 instructions which correspond to their respective disabling instructions above:

1. POPA (Pops the PSW, INT_MASK, INT_MASK1, and WSR off of the stack)
2. POPF (Pops the PSW off of the stack)
3. EI (Sets the Interrupt Enable bit in the PSW)

Example 1: Clearing Pending Interrupt Bits in an ISR

```
    ISR_LABEL:

    PUSHF                          ;Pushes the PSW onto the stack clearing the
                                   ;Interrupt Enable Bit

    ANDB   INT_PEND, AND_MASK      ;Zeroes located in the AND_MASK register
                                   ;will clear the respective bits in the
                                   ;INT_PEND register

    POPF                           ;Pops the PSW from the stack
                                   ;enabling interrupts

    RET                            ;Return from ISR
```

Example 2: Setting Pending Interrupt Bits in an ISR

```
    ISR_LABEL:
```

```
PUSHF                          ;Pushes the PSW onto the stack clearing the
                               ;Interrupt Enable Bit

ORB     INT_PEND, OR_MASK      ;Ones located in the OR_MASK register
                               ;will set the respective bits in the
                               ;INT_PEND register

POPF                           ;Pops the PSW from the stack
                               ;enabling interrupts

RET                            ;Return from ISR
```

Note: The identifiers in *italics* found in examples 1, and 2 represent user defined identifiers.

Conclusion:

The importance of an efficient, robust interrupt service routine can not be stressed enough. If you follow the recommendations above, you should be confident that your interrupt service routines will be at a low risk of missing interrupts that occur while another interrupt is being serviced.

Please visit the UTMC web site to get the latest revisions of application notes, and software code examples. To help you develop your interrupt service routines, there is currently an ISR, on the UTMC web site, written in C and assembly language that services interrupts from the UT80CRH196KD Error Detection and Correction (EDAC) engine.