Microcontrollers

ApNote

AP1609

☐ additional file
APXXXX01.EXE available

How to make Instruction Sequences uninterruptable

Under certain conditions (e.g inter-task communication, initialization of global system resources, etc.), it may be desirable to protect a critical sequence of instructions from being interrupted by CPU interrupts, PEC transfers, or exception traps, including the external NMI.

C. Meinold / Siemens HL MCB PD

AP1609 ApNote - Revision History		
Actual Revision : Rel.01 Previous Revison: Rel. none		
Page of actual Rel.	Page of prev. Rel.	Subjects changes since last release)

1

1 How to make Instruction Sequences uninterruptable

Under certain conditions (e.g inter-task communication, initialization of global system resources, etc.), it may be desirable to protect a critical sequence of instructions from being interrupted by CPU interrupts, PEC transfers, or exception traps, including the external NMI.

The methods which are discussed in this application note can not only be applied to sequences of instructions, but also to single multiply and divide instructions, which are implemented as interruptable multi-cycle instructions in the SAB 80C166.

For this study, the potential sources of interruption of a program sequence can be divided into two classes as follows:

i) Interrupts, including PEC transfers, which are maskable and which have a programmable priority.

   When an application is not using the external NMI, sequences which should be uninterruptable must only be protected against interrupts, which might occur at any time during execution of a critical sequence.

ii) Exception traps, which are non-maskable and always cause immediate system reaction.

   All hardware traps in the SAB 80C166, except for the external NMI trap, are caused by instructions. This means that except by the NMI, an instruction sequence will never be interrupted by an exception trap unless an instruction within this sequence itself causes an exception condition.

This classification leads to two basic approaches of protecting critical instruction sequences against interruption:

1. Disabling the interrupt system before entry into the critical sequence.
2. Execution of a critical sequence on the class A trap level.

In the following, these methods will be discussed in detail with respect to software overhead and the effects on the interrupt response time.

1.1 Disabling the Interrupt System before Entry into the Critical Sequence

In applications where the external NMI trap is not used, critical instruction sequences which should not be interrupted must only be protected against interrupts. In the SAB 80C166, there are basically two ways to disable interrupt requests from being acknowledged by the CPU:

1. by clearing the IEN bit in the PSW (see Example 1)
2. by setting the CPU priority in bit field ILVL of the PSW to level 15 (0Fh) (see Examples 2, 3).

1.1.1 Example 1: Using `BCLR IEN` to disable Interrupts

```
Main Program:

     BCLR    IEN           ; globally disable interrupts instr N-1
                           ; any instruction which does not belong to
             the critical sequence [1]
First:       instr N       ; first instruction of uninterruptable
             sequence
     ...
Last:        instr N+x     ; last instruction of uninterruptable sequence
     BSET    IEN           ; re-enable interrupt system [2]
```

[1] **Note:** Software modifications of the PSW are performed in the execute phase of an instruction. However, in order to maintain fast interrupt response, modifications of IEN or ILVL are not considered for the current round of interrupt prioritization, but in the next round. This means that an interrupt may potentially be acknowledged after the instruction which disables interrupts via IEN or ILVL in the PSW, or after the instruction following this instruction. **Therefore, one instruction must be placed between the instruction which disables the acceptance of interrupts and the first instruction of the critical sequence to ensure that this sequence is not interrupted.**

[2] **Note:** Any pending interrupt will not be acknowledged until the second instruction cycle after the instruction which re-enables the interrupt system.

When `BCLR IEN` is used to globally disable interrupts, but an interrupt is still acknowledged before the start of the critical sequence, the status of the IEN flag is = 0 upon entry and during execution of the interrupt service routine. This means that all further interrupts are disabled during execution of the interrupt service routine. Assuming that an interrupt request of a source x which is acknowledged after `BCLR IEN` and before the start of the critical sequence has a relatively low priority, this may block higher priority interrupts, which would otherwise have been serviced, for the whole duration of the interrupt service routine of source x until interrupts are re-enabled at the end of the critical sequence:

```
     BCLR    IEN
             ----->        if interrupt occurs here
             instr N-1
             ----->        or here, PSW.IEN = 0, i.e.
                           the associated interrupt service routine
                           is NOT interruptable by higher priority
     interrupts !
First:       instr N
```

The **Main Program Overhead** of the method so far described is just 2 single word instructions to disable/enable the interrupt system. However, the **Interrupt Latency Overhead** caused by instructions which do not directly belong to the critical sequence may be as long as the longest interrupt service routine of the application.

If this is not tolerable, interrupts may be re-enabled e.g. by a `BSET IEN` instruction at the beginning of the interrupt service routine. Note that in this case the `BSET IEN` instruction must be included in each interrupt service routine (task procedure) of the application:

```
Task Procedure:

     BSET    IEN          ; re-enable interrupts at beginning of interrupt
service routine
     ...                  ; code for interrupt service routine
     BCLR    IEN          ; disable interrupts before RETI 3)
     RETI
```

3) **Note:** In order to maintain fast interrupt response, implicit modifications of IEN or ILVL in the PSW by the RETI instruction are not considered in the current round of interrupt prioritization which is performed in parallel with instruction execution. This may lead to a situation where a critical instruction sequence can be interrupted after the first instruction under the following sequence of conditions:

- an interrupt was acknowledged immediately before entry into a critical sequence,
- the interrupt system was not globally disabled during execution of the following interrupt service routine,
- an interrupt request of higher priority than the routine just terminated by the RETI instruction has been generated while the instruction following RETI (= first instruction of critical sequence) is fetched.

This problem can be avoided by placing an instruction which disables the interrupt system immediately before the RETI instruction of each interrupt service routine.

In this case, the **Interrupt Latency Overhead** is reduced to 2 instruction cycles, but an additional **Task Procedure Overhead** of 2 single word instructions for each task procedure of the application is required.

1.1.2 Example 2: Using `BFLDH PSW, #0F0h, #0F0h` to disable Interrupts

```
Main Program:

     BFLDH    PSW, #0F0h, #0F0h ; set ILVL field of PSW to highest priority
              instr N-1         ; any instruction which does not belong to the
                                  critical
                                ; sequence (see Note 1))
First:        instr N           ; first instruction of uninterruptable sequence
     ...
Last:         instr N+x         ; last instruction of uninterruptable sequence
     BFLDH    PSW, #0F0h, #70h  ; restore previous priority of PSW.ILVL field
                                ; (assuming original task priority was 7h) (see
                                  Note 2))

Task Procedure:

     ...                           ; code for interrupt service routine
     BFLDH PSW, #0F0h, #0F0h    ; disable interrupts before RETI (see Note 3))
     RETI
```

When `BFLDH PSW, #0F0h, #0F0h` is used to disable interrupts, but an interrupt is acknowledged before the start of the critical sequence, the ILVL field in the PSW is updated with the priority of this interrupt request upon entry into the interrupt service routine. This automatically allows higher priority interrupts to be acknowledged during the execution of this interrupt service routine. In this case, the real-time behaviour of the application is not affected:

```
     BFLDH    PSW, #0F0h, #0F0h
     ----->                     if interrupt occurs here
              instr N-1
     ----->                     or here, PSW.ILVL = priority of
     interrupting task, i.e.
                                ALL interrupts are serviced according to
     their original priority !
First:        instr N
```

This method requires a **Main Program Overhead** of 2 double word instructions per execution of a critical sequence, and a **Task Procedure Overhead** of 1 double word instruction. The **Interrupt Latency Overhead** (2 BFLDH instructions) is negligible.

1.1.3 Example 3: Using SCXT PSW, #0F800h to disable Interrupts

```
Main Program:

     SCXT     PSW, #0F800h       ; push PSW, set ILVL field of PSW to highest
priority
              instr N-1          ; any instruction which does not belong to the
                                   critical
                                 ; sequence (see Note 1))
First:        instr N            ; first instruction of uninterruptable sequence
     ...
Last:         instr N+x          ; last instruction of uninterruptable sequence
     POP      PSW                ; restore PSW, re-enable interrupt system (see
                                   Note 2))

Task Procedure:

     ...                         ; code for interrupt service routine
     BFLDH    PSW, #0F0h, #0F0h ; disable interrupts before RETI (see Note 3))
     RETI
```

When `SCXT PSW, #0F800h` is used to disable interrupts, but an interrupt is acknowledged before the start of the critical sequence, the ILVL field in the PSW is updated with the priority of this interrupt request. This automatically allows higher priority interrupts to be acknowledged and processed according to their priority, without affecting the real-time behaviour of the application.

This method requires a **Main Program Overhead** of 1 double and 1 single word instruction, and a **Task Procedure Overhead** of 1 double word instruction. Additionally, a **Stack Overhead** of 1 word is required. As for the method described in section 1.1.2, the **Interrupt Latency Overhead** (3 instructions) is negligible.

1.1.4 Notes for C Programmers

The corresponding instructions to disable/enable the interrupt system may be inserted into the source code using the inline assembly facility via #pragma asm/#pragma endasm.

In order to insert the `BFLDH PSW, #0F0h, #0F0h` instruction, the built in function _bfld `(PSW, 0xF000, 0xF000)` may be used.

Note that the compiler may generate additional code to support context switching at the end of an interrupt service routine. This code is placed after the last statement in the interrupt service routine and before the RETI instruction.

1.1.5 Pipeline Diagrams

The following pipeline diagrams show the function of the discussed methods to make instruction sequences uninterruptable.

1.1.5.1 Using `BCLR IEN` to disable Interrupts

Interrupt Possible at End of Cycle	yes [1]	yes [1]	no	no	no	no	yes
Pipeline Stage							
Decode	`BCLR IEN`	`N-1`	`N`	`...`	`BSET IEN`	`next`	`next+1`
Execute	`...`	`BCLR IEN`	`N-1`	`...`	`N+x`	`BSET IEN`	`next`
Write Back	`...`	`...`	`BCLR IEN`	`...`	`...`	`N+x`	`BSET IEN`

[1] **Note:** If an interrupt occurs after this cycle, the associated service routine will not be interruptable!

1.1.5.2 Using `BFLDH PSW, #0F0h, #0F0h` to disable Interrupts

Relevant CPU Priority during Prioritization	p < 15	p < 15	p = 15	p=15	p = 15	p = 15	p < 15
Interrupt Possible at End of Cycle	yes [2]	yes [2]	no	no	no	no	yes
Pipeline Stage							
Decode	`BFLDH...`	`N-1`	`N`	`...`	`RETI(1)`	`RETI(2)`	`next`
Execute	`...`	`BFLDH...`	`N-1`	`...`	`N+x`	`RETI(1)`	`RETI(2)`
Write Back	`...`	`...`	`BFLDH...`	`...`		`N+x`	`RETI(1)`

[2] **Note:** If an interrupt occurs after this cycle, the associated service routine will be interruptable by higher priority interrupts!

1.2 Example 4: Execution of Critical Sequences on the Class A Trap Level

In the SAB 80C166, there are two classes of exception traps, class A and class B traps. The external NMI trap belongs to the class A hardware traps. Class A trap service routines can not be interrupted by class B traps or other class A traps. This means that any instruction sequence which is executed on the class A trap level can not be interrupted, neither by the NMI nor by CPU interrupts.

A class A trap can be forced by software by setting one of the trap flags NMI, STKOF, STKUF in register TFR. In case that all class A traps are already used in an application, an additional flag must be defined which indicates to the trap service routine whether an uninterruptable sequence or an exception trap is to be processed. An example how uninterruptable sequences (named AtomicProc1..n) may be executed on the class A trap level is shown in the following. It is assumed that both the exception trap routine and the uninterruptable sequence use the same registerbank:

```
Main Program:

  DataSection  SECTION BIT 1)

               Atomic DBIT            ; flag for uninterruptable sequence
                                        GLOBAL Atomic

                                      ; may be used by different task
                                        procedures

  DataSection  ENDS

  MainRB       REGDEF R2-R3 COMMON = Para, R0-R1 PRIVATE

  CodeSection  SECTION CODE
  MainProc     PROC TASK INTNO = 0    ; task which is executed after reset

    ...
    MOV        R3, #SOF AtomicProc1   ; segment offset of AtomicProc1
                                      ; parameter for trap routine
    BSET       STKOF                  ; set STKOF flag to force trap
    BSET       Atomic                 ; flag uninterruptable sequence 1)
    ...
```

[1]**Note:** When trap flags are set by software, 1 (at least) or 2 (at most) instructions following the instruction which set the trap flag may be executed before the trap is entered.

```
Trap Service Routine:

  EXTERN      Atomic:BIT 1)

  StackOvRB   REGDEF R0-R1 COMMON = Para, R2-R3 PRIVATE

  CodeSec     SECTION CODE
  StackOvTrap PROC Task INTNO = 4             ; stack overflow task

    SCXT      CP, #StackOvRB                  ; switch registerbank
    BCLR      STKOF                           ; clear trap flag
    JBC      Atomic, SHORT AtomicService     ; test and clear Atomic
             flag 1)

  ExceptionTrapService:
    ...                                      ; code for exception trap service
    ...                                      ;
    JMP      Continue                        ; end of exception trap
             service 1)

  AtomicService:
    CALLI    cc_UC, [R1]                     ; execute AtomicProc1..n
                                               specified by R1

  Continue:
    POP      CP                              ; restore previous
                                               registerbank
    RETI                                     ; return from trap service
             routine

  StackOvTrap ENDP
```

[1)] **Note:** instruction/directive only required when trap routine is shared between uninterruptable sequence and exception trap processing

Note: Since the SAB 80C166 instruction set only provides an intra-segment indirect subroutine call instruction (CALLI cc, [Rwn]), it is most efficient when all uninterruptable sequences (AtomicProc1..n) are located within segment 0. Otherwise, #SEG AtomicProc1..n (segment number) must additionally be passed to the trap routine, #SEG/#SOF Continue and #SEG/#SOF AtomicProc1..n must be pushed on the stack, and a RETS instruction must be executed in order to perform an indirect inter-segment call to AtomicProc1..n.

The **advantages** of processing uninterruptable instruction sequences on the class A trap level are the immediate response and the constant interrupt latency overhead. In addition, other task procedures/interrupt service routines of the application are not affected and must not be modified, as it may be required in the case of some of the methods described in section 1.1.

The **overhead** which is required by this method of executing uninterruptable instruction sequences (including the RETurn from subroutine call to AtomicProc1..n) is as follows:

Trap Routine Overhead: 7 words of code are required when the trap routine is exclusively used for processing uninterruptable instruction sequences, and 5 words when the trap routine is shared with exception trap handling

Main Program Overhead: for each occurrence of an uninterruptable sequence, 3 words of code are required in the main program when the trap routine is exclusively used for processing uninterruptable instruction sequences, and 4 words when the trap routine is shared with exception trap handling.

Stack Overhead: 1 word when the trap routine is shared, and 4 (5) words when segmentation is disabled (enabled) and the trap routine is used exclusively.

Execution Time Overhead: 1.5 us (trap routine exclusive) or 1.8 us (trap routine shared) per invocation of an uninterruptable sequence for a 16-bit non-multiplexed data bus with 0 waitstates.

Interrupt Latency Overhead: 1.2 us (trap routine exclusive) or 1.4 us (trap routine shared) for a 16-bit non-multiplexed data bus with 0 waitstates.