Freescale Semiconductor
Application Note

Document Number: AN3496
Rev.0, 07/2007

A Software Approach to Using Nested Interrupts in HCS08

by: Kenny Ji
Asia & Pacific Operation Microcontroller Division

In the HCS08 family of microcontrollers, interrupts provide a way to save the current CPU status and registers, execute an interrupt service routine (ISR), and then restore the CPU status so processing resumes where it left off before the interrupt.

Before an ISR is completed, the global interrupt mask (I bit) in the condition code register (CCR) is set to mask further interrupts. This mechanism ensures that the ISR is not interrupted during execution. However, the disadvantage of this mechanism is that a high-priority interrupt cannot interrupt a low-priority ISR execution.

This document provides a solution to yield a prioritized interrupt mechanism in software. This benefits anyone who wants more powerful and flexible applications without a task-based real-time operation system (RTOS) support.

Contents

1 Non-Nested Interrupts 2
1.1 Interrupts 2
1.2 Interrupt Stack Frame 3
1.3 Inhibiting Interrupts 3
2 Nested Interrupt Mechanism 4
2.1 Requirements Of Nested Interrupts. 5
2.2 Software Nested Interrupt Scheduler 5
2.3 Implementation 10
3 Port User Program with the Scheduler 11
3.1 Variable and Macro Definitions 11
3.2 Interrupt Service Routine Definitions 12
3.3 Initialization 13
4 Performance 13
4.1 Flash Memory Consumption 13
4.2 RAM Consumption 14
4.3 Time Consumption 14
5 Miscellaneous Topics 15
5.1 Use ISR Not Supporting Scheduler. 15
5.2 Use Scheduler in the Main Loop 15
Appendix AScheduler Code Lists 16

freescale™ semiconductor

1 Non-Nested Interrupts

1.1 Interrupts

If an event occurs in an interrupt enabled source, an associated read-only status flag is set, but the CPU does not respond unless these two conditions are both met.

- Local interrupt mask is a logic 1 to enable the interrupt
- The I bit in the condition code register (CCR) is logic 0 to allow interrupts

The global interrupt mask (I bit) in the CCR is initially set after reset. It screens all maskable interrupt sources. This allows the user program to initialize the stack pointer and perform other system setup before clearing the I bit to allow the CPU to respond to interrupts. Figure 1 shows the I bit in CCR.

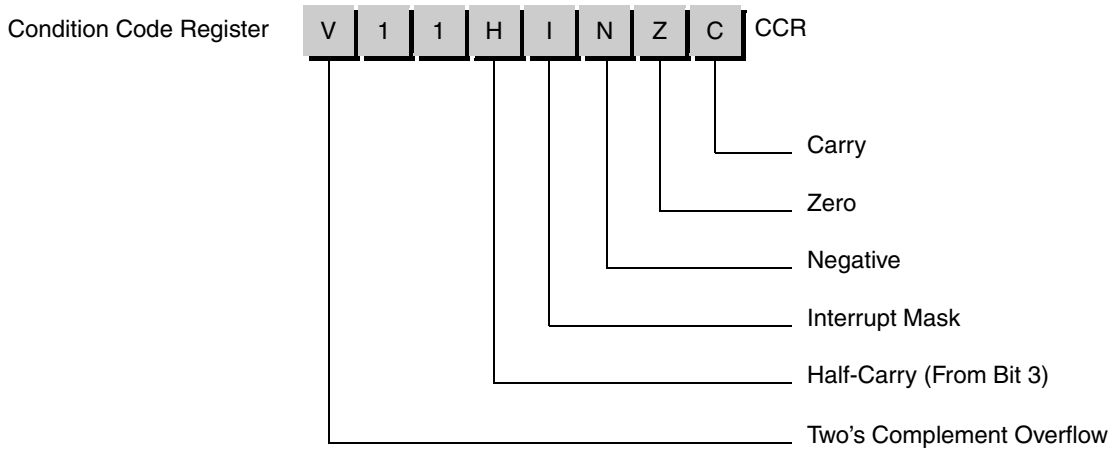


Figure 1. Condition Code Register

When the CPU receives a qualified interrupt request, it completes the current instruction before responding to the interrupt. The interrupt processing process is:

1. Saves the CPU registers to the stack.
2. Sets the I bit in the CCR to mask new interrupts.
3. Fetches the interrupt vector for the highest priority interrupt that is currently pending.
4. Fills the instruction queue with the first three bytes of program information starting from the address fetched from the interrupt vector locations.

While the CPU responds to the interrupt, the I bit is automatically set to avoid another interrupt from interrupting the ISR (this is called nesting of interrupts). Normally, the I bit is restored to 0 when the CCR is restored from the value that was stacked on entry to the ISR.

In rare cases, the I bit may be cleared in an ISR (after clearing the status flag that generated the interrupt) so other interrupts can be serviced before the first service routine is finished. This practice is not recommended because it leads to subtle program errors that are difficult to debug.

1.2 Interrupt Stack Frame

Figure 2 shows the contents and organization of a stack frame. Before the interrupt occurs, the stack pointer (SP) points at the location of the next available byte on the stack. The current values of CPU registers are stored on the stack starting with the low-order byte of the program counter (PCL) and ending with the condition code register (CCR). After stacking, the SP points at the next available location on the stack. This location is the address that is one less than the address where the CCR was saved. The stacked PC value is the address of the instruction in the main program that will be executed next if the interrupt does not occur.

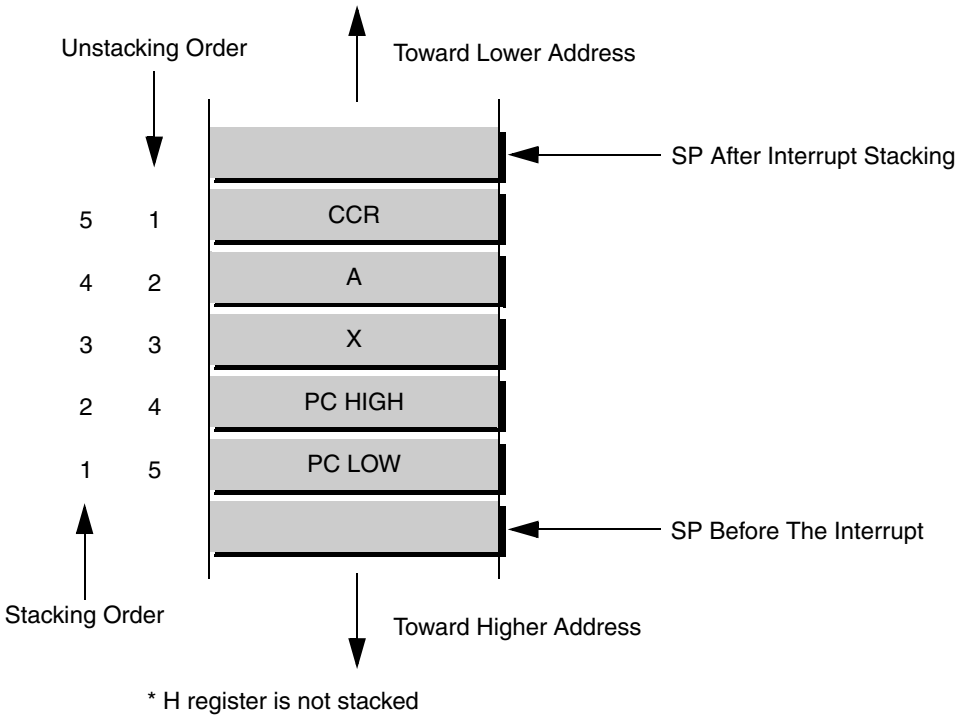


Figure 2. Interrupt Stack Frame

When an RTI instruction is executed, these values are recovered from the stack in reverse order. As part of the RTI sequence, the CPU fills the instruction pipeline by reading three bytes of program information, which start from the PC address that is just recovered from the stack.

1.3 Inhibiting Interrupts

The restriction of the current interrupt mechanism is that nested interrupts are not allowed. This means a high-priority ISR must be executed after a low-priority ISR completes if a low-priority interrupt occurs before a high-priority interrupt. For example, there is a timer has higher priority than keyboard interrupt. If a timer overflow interrupt occurs when the keyboard ISR is in execution, though it is more emergent, it is pended until the keyboard ISR returns. In some cases, this latency is too long and causes problems. Figure 3 shows a typical non-nested interrupt schedule.

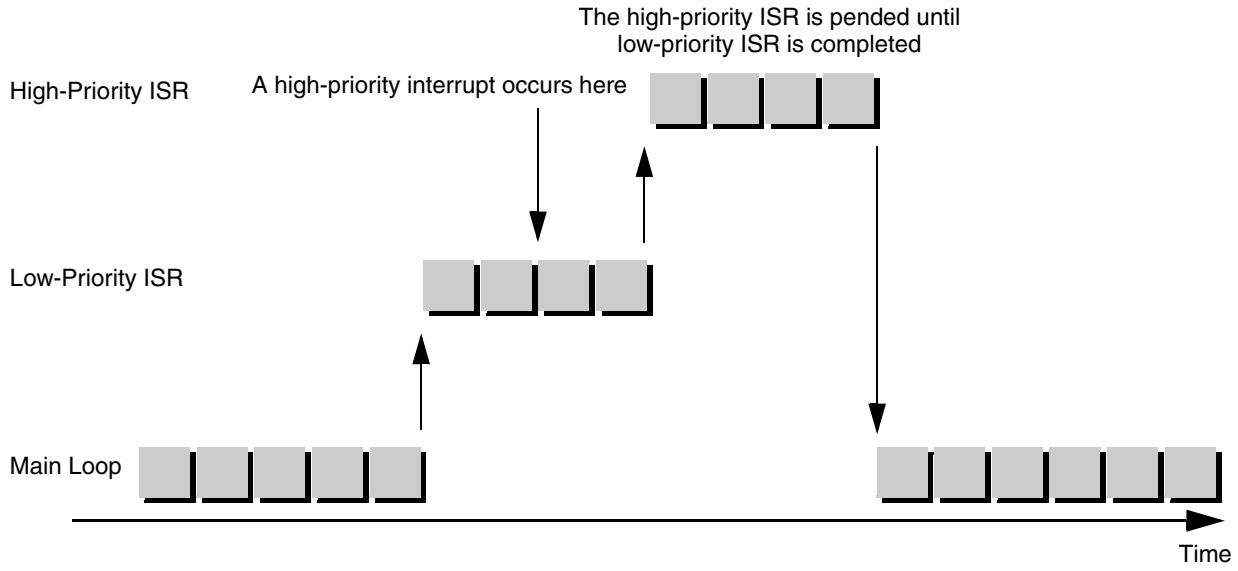


Figure 3. Non-Nested Interrupt Schedule

Therefore, nested interrupts can be available in applications.

One solution is to open the interrupt enabled function during ISR. As mentioned, this practice is not recommended because it can lead to subtle program errors that are difficult to debug. Moreover, a significant disadvantage of this solution is that high-priority ISR can be interrupted by low-priority ISR, which is not allowed in most applications.

An advanced approach is to use RTOS in applications. However, this solution requires large memories and huge CPU consumption. Most RTOS, especially open source RTOS, is task based or thread based. They need special memory allocation mechanism to manage a certain task's or thread's memory when a task or thread switching occurs. For example, uC/OS II, the most popular open source RTOS, needs approximately 384 bytes of RAM to support a four-task system in MC68HC908GP32. Another consideration is the time performance. Most interrupt-based systems have much better performance than tasked-based systems. The interrupt mechanism still has the best performance to deal with real-time works.

2 Nested Interrupt Mechanism

This section introduces a software approach that provides a nested interrupt mechanism in the S08 family of microcontrollers. It extends the capability and flexibility of an interrupt mechanism efficiently. It also keeps low time and space consumptions in the application system.

2.1 Requirements Of Nested Interrupts

In a nested interrupt system, an interrupt is allowed to anytime and anywhere even an ISR is being executed. But, only the highest priority ISR will be executed immediately. The second highest priority ISR will be executed after the highest one is completed.

The rules of a nested interrupt system are:

- All interrupts must be prioritized.
- After initialization, any interrupt is allowed to occur anytime and anywhere.
- If a low-priority ISR is interrupted by a high-priority interrupt, the high-priority ISR is executed.
- If a high-priority ISR is interrupted by a low-priority interrupt, the high-priority ISR continues executing.
- The same priority ISRs must be executed by time order.

Figure 4 shows a typical nested interrupt schedule.

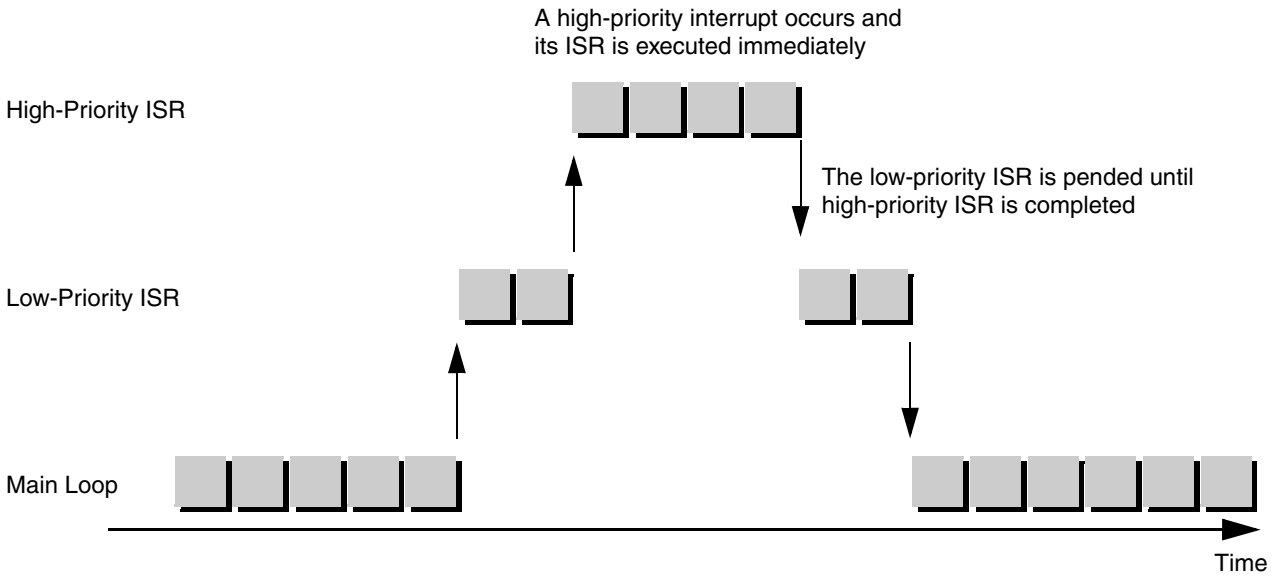


Figure 4. Nested Interrupt Schedule

2.2 Software Nested Interrupt Scheduler

2.2.1 Scheduler Model

In this section, a software nested interrupt scheduler is introduced. As shown in Figure 5, this scheduler includes a preemptive scheduler, a priority arbiter, and a series of priority queues. These queues are prioritized.

When an interrupt occurs, the priority arbiter will put the interrupt in queue according to its priority. This operation is called entering queue. Then the preemptive scheduler goes to work. It scans the queues to find out the highest priority queue available and bring the highest one to work.

In Figure 5, a series of interrupts occurs by order 2, 1, 4, 1, 3, and 2. These interrupts are sorted by the scheduler and executed in the order of 1, 1, 2, 2, 3, and 4 via the scheduler.

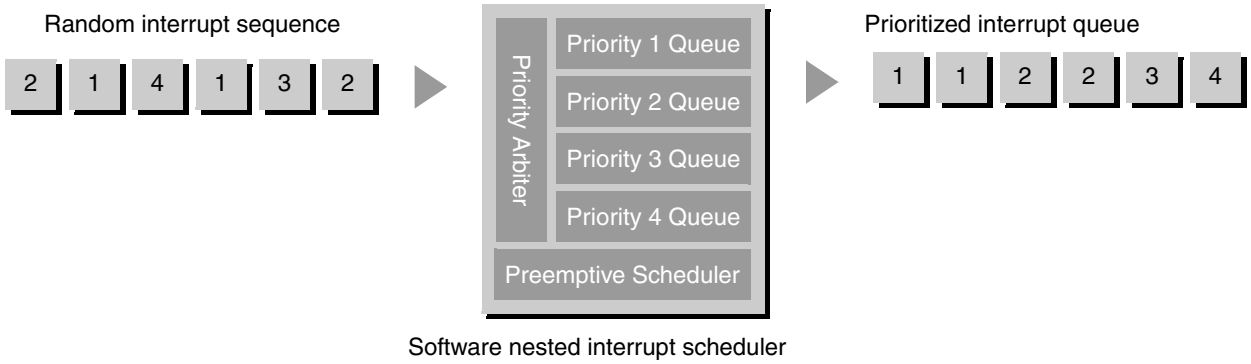


Figure 5. Software Nested Interrupt Scheduler

If a high-priority interrupt occurs when a low-priority ISR is in execution, the scheduler lets the high-priority ISR preempt the CPU resource except a higher priority interrupt occurs during scheduling. The low-priority ISR will be executed after the high-priority ISR is completed. This action ensures that more emergent interrupt is executed prior to the lesser one and allows the preemption operation; therefore, this is called preemption system. If an OS has this mechanism, it is called preemption OS.

If a low-priority interrupt occurs when a high-priority ISR is in execution, the scheduler will continue executing the high-priority ISR. The low-priority interrupt will be pended and kept in the queue until the high-priority ISR is completed. This mechanism ensures that high-priority ISR will be executed earlier than the low-priority interrupt.

2.2.2 Data Structure

2.2.2.1 Interrupt Object

Every interrupt has its own interrupt objects in the scheduler. Each object is implemented by a structure in C programming. it comprises of link, lock, mask, and func. Figure 6 shows the structure of interrupt object.

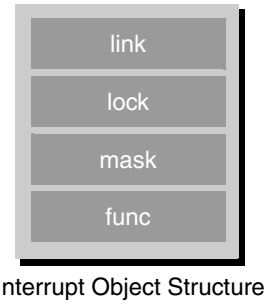


Figure 6. Interrupt Object

The member link is a pointer pointing to the next interrupt object. Because this is a queue system, a single link list is used to make queue accessible. In the single link list, every node has a pointer pointing to the

next object, by which the scheduler can use a first-in-first-out (FIFO) system easily. Furthermore, the pointer points to the interrupt object with the same priority.

The member lock is a flag to identify if the interrupt object is put in the queue or not. After initialization, this flag is set as 0xff. When this object is put into the queue, the flag is set as 0x00. When the object leaves the queue, the flag is set as 0xff again.

The member mask is a variable to identify the priority of the interrupt object. It equals to priority powers of number two. This means the mask will be set by 1 if the priority is 0. It is used to make a comparison in scheduling.

The member func is function pointer to the dispatch function. If an interrupt object is selected to be executed, the scheduler brings the program to a certain address by this function pointer. This function has no input parameters or return value.

2.2.2.2 FIFO Queues

The software nested interrupt scheduler has a set of FIFO queues, which contain all available pending interrupts. Every queue has a queue head. As shown in Figure 7, the head set is an array of structures. The quantity of FIFO queues is equal to that of total priorities levels. Each head of FIFO queue is an element that controls a certain priority queue and contains two pointers, one points to the first pending interrupt in the FIFO queue and the other points to the last pending interrupt in the same FIFO queue.

For example, in Figure 7, the priority X queue has N pending interrupts. The first is pointed by the head. The second is pointed by the first. The last is pointed by both the N-1 and the head. This constructs a small FIFO system. You can get the first object from the queue by inquiring the pending interrupt table and add new coming objects in.

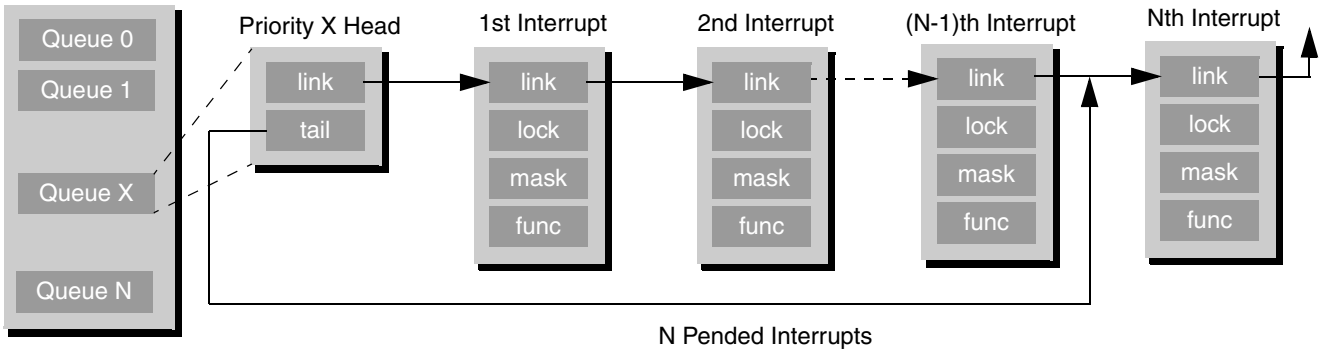


Figure 7. Pended Interrupt Table

A FIFO operation can be divided into a set of simple operations.

- To add a new object at the end of queue, this new object is pointed by both the last object and the tail pointer of head.
- To remove a object, the first object will be skipped by the link pointer of head and its pointer will be set to NULL as a safe operation at the same time.

The FIFO operation also meets the requirement to make a choice when two interrupts with the same priority occurs. In this condition, the second ISR is pended until the first one is completed.

2.2.2.3 Other Schedule Data

The FIFO queues provide an excellent solution to sort the interrupt objects with the same priority. The first-in interrupt object is executed prior to others of the same priority. But to the interrupt objects with different priorities, a series of prioritized queues must be compared to decide which level will be executed prior to others.

The queue head sets help to make a choice. As shown in Figure 8, the head sets are contained in INT_rdytab. During execution, the scheduler checks the queue head one by one. If the highest priority head is not empty, the corresponding queue is selected to execute immediately. To accelerate the seeking, a date INT_sets is used to indicate if the queue is empty or not. The head of the high-priority queue is mapped in MSB and that of the low one is mapped in LSB. After an interrupt object is put in a certain FIFO queue, the corresponding bit in INT_sets are set. After the last interrupt is removed from the FIFO queue, the corresponding bit in INT_set is cleared.

INT_mask is the data that used to store the interrupt object priority level being executed. If the ISR in execution is interrupted by an interrupt, the scheduler will compare the priorities between the new interrupt object in queues and the one in execution after the new interrupt object is put in a queue. This helps to find out which interrupt service routine will be executed after scheduling.

INT_lock is the data that used to solve the scheduler nested scheduling

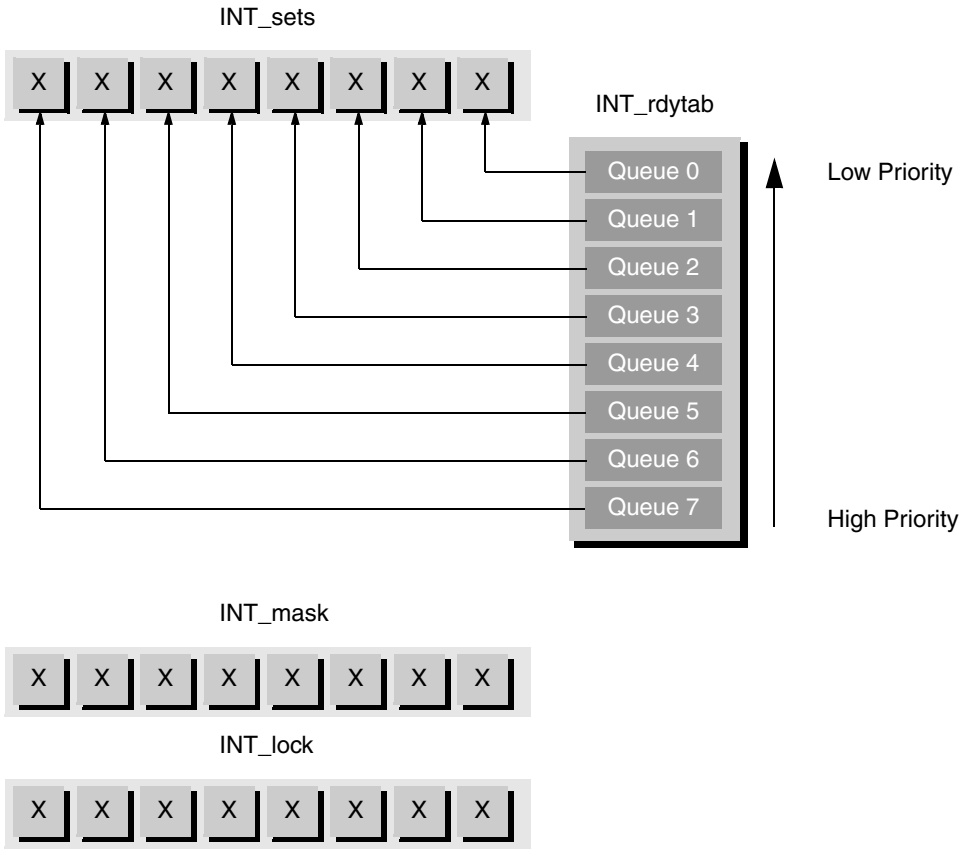
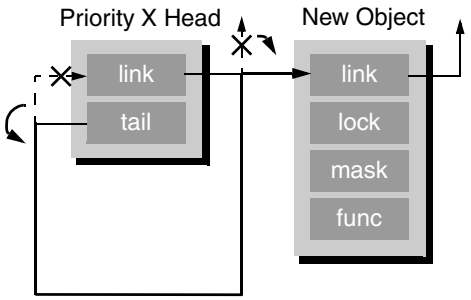


Figure 8. Queue Head Set

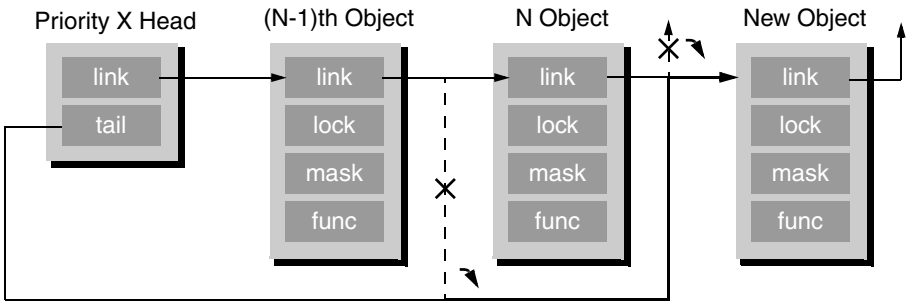
2.2.3 FIFO Operation

There are two FIFO operations, i.e. input operation and output operation. The input operation is a typical addition operation at the end of link list. The output operation is a typical remove operation at the beginning of link list.

As shown in Figure 9, in adding operation the new interrupt object is added to the end of the FIFO queue. If there is at least one interrupt object in the queue, the new interrupt object is pointed by the link pointer of last interrupt object. Otherwise, if there is no existing interrupt object in the FIFO queue, it is pointed by the link pointer of the queue head. The tail pointer of head will be set to the new interrupt object. The link pointer of the new interrupt object is set to NULL.



First Pended Interrupt Addition Operation



Non-First Pended Interrupt Addition Operation

Figure 9. FIFO Queue Addition Operation

As shown in Figure 10, in removing operation, the first interrupt object is removed from the link list. The link pointer of head is set to the link of the removed interrupt object. If the removed interrupt object is the last interrupt object in the FIFO queue, the head will be set to initialization state, in which the link pointer will point to NULL and the tail pointer will point to link.

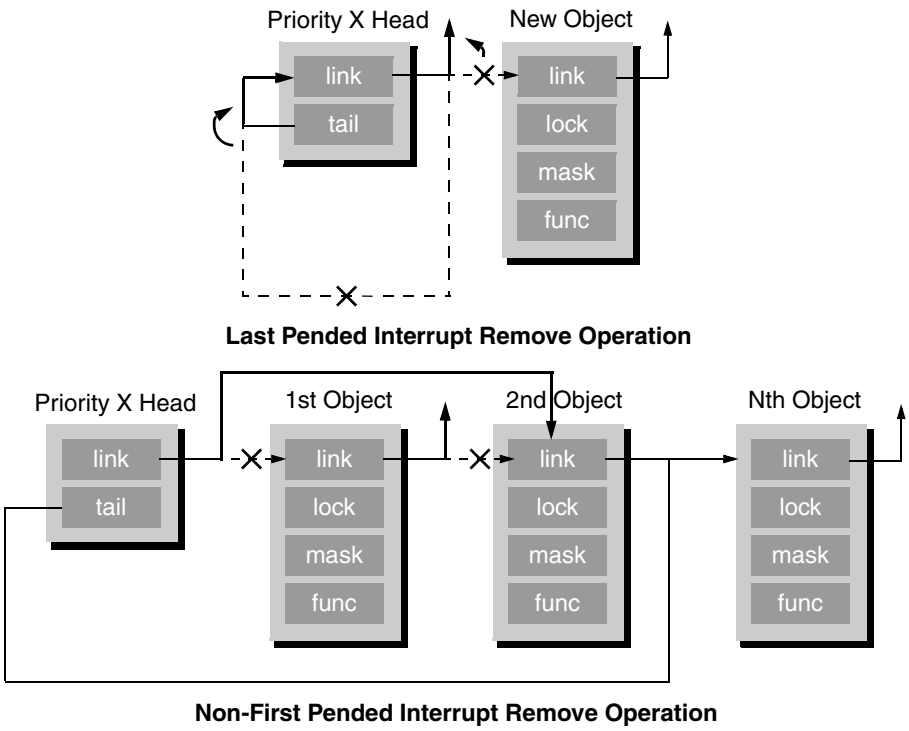
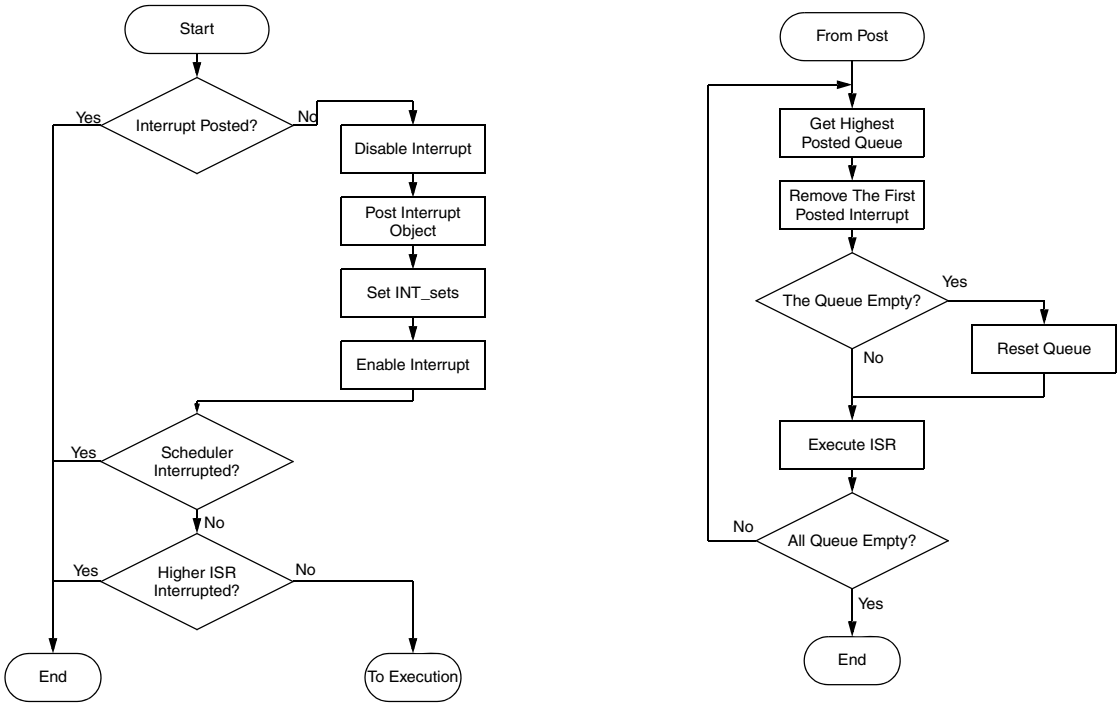


Figure 10. FIFO Queue Remove Operation

2.3 Implementation

The scheduler implementation code is divided into two parts, interrupt post code and interrupt execution code. The former implements the First-In operation of FIFO queue and the latter implements the first-out operation of FIFO queue. Figure 11 shows the scheduler work flow. The left is post procedure work flow and the right is execution procedure work flow.



Post Procedure Work Flow **Execution Procedure Work Flow**

Figure 11. Scheduler Work Flow

Please see Appendix A, "Scheduler Code Lists" for more detailed code implementation.

3 Port User Program with the Scheduler

3.1 Variable and Macro Definitions

Customers must define some variables and macros in program before using this scheduler. These variables and macros include all information used in this scheduler.

The macro MAX_INT_TAB defines the maximum interrupt object quantity used in the program. For example, if there are five interrupt objects used, the macro must be defined equal to or bigger than 5. The following is one of the examples.

Example 1. Define MAX_INT_TAB

```
MAX_INT_TAB     EQU 5
```

The variable INT_objtab is a structure table containing all interrupt objects' information. It must be defined in short addressing mode scope for quick access. The defined macro INT_OBJ supports a shortcut to define the table. The following code defines five interrupt objects.

Example 2. Define Interrupt Objects

```
      _DATA_ZEROPAGE: SECTION

      ...

      INT_objtab:

          INT_OBJ KBI_obj

          INT_OBJ TPM_obj

          INT_OBJ ADC_obj

          INT_OBJ TIM_obj

          INT_OBJ SCI_obj

INT_OBJEND:
```

The constant INT_inittab is a structure table that contains the initial map of interrupt objects. It must be defined in flash memory because the initialization routine copies this information to interrupt objects in RAM. The macro INT_INI supports a shortcut to define this table. This macro needs these three parameters:

- The interrupt object name
- The priority of interrupt object
- The service routine of interrupt object

Regarding to this example, the following code presents an approach to define the table,

Example 3. Define Interrupt Object Initial Information

```
      CONST:          SECTION
      INT_inittab:

          INT_INI KBI_obj,2,KBI_isr       ; KBI object

          INT_INI TPM_obj,1,TPM_isr       ; TPM object

          INT_INI ADC_obj,1,ADC_isr       ; ADC object

          INT_INI TIM_obj,1,TIM_isr       ; MTIM object

          INT_INI SCI_obj,1,SCI_isr       ; SCI object

      INTTABEND:
```

3.2 Interrupt Service Routine Definitions

If a scheduler is used, the interrupt service routine needs to be modified as follows:

- Writing all jobs in a routine pointed by the member func in interrupt object
- Writing a short interrupt service routine only containing the function INT_post

Example 4 shows a simple way to use the interrupt service routine for a KBI interrupt job.

Example 4. Interrupt Service Routine Definition

```
; The real interrupt routine, but only INT_dispatch called to trigger the scheduler
KBI_int:
     INT_dispatch KBI_obj
     RTI ; use return interrupt here

; All jobs are written here, this routine will be called by the scheduler
; after FIFO operations

KBI_isr:
     ; write the indeed jobs here
     ...
     ; return
     RTS ; use return subroutine here
```

3.3 Initialization

All variables used in the scheduler must be initialized before use. The software package supports a function named INT_init to complete initialization. This function has no parameters or return value. Example 5 shows how to use this function in the program.

Example 5. Scheduler Initialization

```
_Startup:
     JSR     INT_init        ; Interrupt Scheduler Initialized
```

4 Performance

The performance is the most important point to the scheduler. Three points restricts the real-time performance of a tiny system, flash memory, RAM, and time.

4.1 Flash Memory Consumption

The flash memory consumption in the schedule are:

- The real scheduler takes 327 bytes flash memory, which is very low even to a tiny system
- Calling the initialization routine takes 3 bytes
- The initial information takes 5 bytes for every interrupt object
- Calling the post function takes 8 bytes for every interrupt object

Totally, 330 + 13 × N bytes flash memory will be used in the scheduler. N is the number of interrupt objects used. Table 1 shows the flash memory consumption for typical cases. The flash memory consumption is less than 0.5K bytes to a system constructed with eight interrupt objects.

Table 1. Flash Memory Consumptions

Interrupt Object Number	Interrupt Object Flash Memory Consumptions (Bytes)	Scheduler Flash Memory Consumptions (Bytes)	Total Flash Memory Consumptions (Bytes)
1	13	330	343
2	26	330	356
3	39	330	369
4	52	330	382
5	65	330	395
6	78	330	408
7	91	330	421
8	104	330	434

4.2 RAM Consumption

The scheduler needs 18 bytes at least and every interrupt object takes 5 bytes. Therefore, the RAM consumption is 18 + 5 × N bytes. N is the interrupt object number. As shown in Table 2, an 8 interrupt object system uses 58 bytes. This is a small consumption to most applications.

Table 2. RAM Consumptions

Interrupt Object Number	Interrupt Object RAM Consumptions (Bytes)	Scheduler RAM Consumptions (Bytes)	Total RAM Consumptions (Bytes)
1	5	18	23
2	10	18	28
3	15	18	33
4	25	18	38
5	30	18	43
6	35	18	48
7	40	18	53
8	45	18	58

4.3 Time Consumption

The time consumptions in a scheduler are:

- If you attempt to post a posted interrupt object, the scheduler denies it. This action takes 6 CPU cycles.
- If you post a non-higher priority interrupt object, the scheduler puts the object in queue and then continues the interrupted higher priority interrupt object. This action takes 90 CPU cycles.
- If you post an interrupt object interrupting another scheduler operation, the scheduler puts the object in queue and returns to the previous scheduler. This action takes 95 CPU cycles.
- If you post a top-priority interrupt object, the scheduler puts the interrupt object in the queue and removes it from the queue immediately to execute. This action takes 205 CPU cycles.

The longest latency of the scheduling is 205 CPU cycles. As a 20 MHz CPU clock system, it produces about 10 μs latency. This is short enough for a real-time system. However, what we can get is to reduce the high-priority interrupt latency when a low-priority interrupt routine is being executed. This is a big improvement in many real-time based systems.

Table 3. Time Consumption

Action	CPU Cycles	Time at 20 MHz CPU Clock (μs)
Post a posted interrupt object	6	0.3
Post a non-highest priority interrupt object	90	4.5
Interrupt another scheduler operation	95	4.75
Post a top priority interrupt object	205	10.25

5 Miscellaneous Topics

5.1 Use ISR Not Supporting Scheduler

Programmers can still use pure ISRs that do not support the scheduler in their program. But the pure ISRs are executed prior to the routines controlled by the scheduler. It is not recommended to enable the interrupt in the pure ISRs, because it causes trouble.

5.2 Use Scheduler in the Main Loop

The interrupt object can also be posted in the main loop and the scheduler executes the interrupt according to the schedule. This means you can post the scheduler as a software interrupt.

Appendix A Scheduler Code Lists

The following code implements a code with five interrupt objects.

```
;*****************************************************************
;* This stationery serves as the framework for a user application. *
;* For a more comprehensive program that demonstrates the more     *
;* advanced functionality of this processor, please see the        *
;* demonstration applications, located in the examples             *
;* subdirectory of the "Freescale CodeWarrior for HC08" program    *
;* directory.                                                      *
;*****************************************************************

; ****************************************************************
; * THIS CODE IS ONLY INTENDED AS AN EXAMPLE OF CODE FOR THE *
; * CODEWARRIOR COMPILER AND HAS ONLY BEEN GIVEN A MIMIMUM *
; * LEVEL OF TEST. IT IS PROVIDED 'AS SEEN' WITH NO GUARANTEES *
; * AND NO PROMISE OF SUPPORT. *
; ****************************************************************

;*****************************************************************
;*
;* Freescale reserves the right to make changes without further notice to any
;* product herein to improve reliability, function, or design. Freescale does
;* not assume any liability arising out of the application or use of any
;* product, circuit, or software described herein; neither does it convey
;* any license under its patent rights nor the rights of others. Freescale
;* products are not designed, intended, or authorized for use as components
;* in systems intended for surgical implant into the body, or other
;* applications intended to support life, or for any other application in
;* which the failure of the Freescale product could create a situation where
;* personal injury or death may occur. Should Buyer purchase or use Freescale
;* products for any such intended or unauthorized application, Buyer shall
;* indemnify and hold Freescale and its officers, employees, subsidiaries,
;* affiliates, and distributors harmless against all claims costs, damages,
;* and expenses, and reasonable attorney fees arising out of, directly or
;* indirectly, any claim of personal injury or death associated with such
;* unintended or unauthorized use, even if such claim alleges that Freescale
;* was negligent regarding the design or manufacture of the part. Freescale
;* and the Freescale logo* are registered trademarks of Freescale Ltd.
;*
;*****************************************************************

    ; Include derivative-specific definitions
    INCLUDE 'derivative.inc'

    ; export symbols
    XDEF _Startup, main
    ; we export both '_Startup' and 'main' as symbols. Either can
    ; be referenced in the linker .prm file or from C/C++ later on

    XREF __SEG_END_SSTACK   ; symbol defined by the linker for the end of the stack

;*****************************************************************
;*
```

```
;* Macro Definition
;*
;*****************************************************************

MAX_INT_TAB     EQU 5

INT_OBJ_SIZE    EQU 5               ; 5 bytes for one object

MAX_RDY_TAB     EQU 7               ; for there are 7 bit available in mask

RDY_OBJ_SIZE    EQU 2               ; 2 bytes ready objects

IDLE            EQU     $01         ; default priority = 1

READY           EQU     $FF         ; -1 : not post
POST            EQU     $00         ; 0 : post

; interrupt object member offset

LINK            EQU     0           ; link : offset 0
LOCK            EQU     1           ; lock : offset 1
MASK            EQU     2           ; mask : offset 2
FUNC            EQU     3           ; func : offset 3

; ready object member offset

HEAD            EQU     0           ; head : offset 0
TAIL            EQU     1           ; head : offset 1


;
; INT_OBJ
; Usage:
;   INT_OBJ name,priority,function
; For example:
;       INT_OBJ KBI_OBJ,3,KBI_isr
;

INT_OBJ         MACRO
; XDEF    \1      ; able to be accessed externally
\1:
    DS.B    INT_OBJ_SIZE
    ENDM

INT_INI         MACRO
; XREF    \3      ; use function externally
\1_:
    DCB.B   1,0         ; NULL          -> link
    DCB.B   1,-1        ; -1            -> lock
    DCB.B   1,1 << \2   ; -1            -> mask
    DCB.W   1,\3        ; function      -> func
    ENDM

;
; INT_DISPATCH
; Usage:
;   INT_DISPATCH object
```

```
;

INT_dispatch    MACRO
    PSHH                ; H -> stack
    PSHX                ; X -> stack

    LDHX    #\1         ; object -> H:X
    JSR     INT_post    ; goto INT_post

    PULX                ; stack -> X
    PULH                ; stack -> H

    ENDM

;
; Left Most Bit Detection , 20 cycles for every calling
;   1xxxxxxx -> 7
;   01xxxxxx -> 6
;   001xxxxx -> 5
;   0001xxxx -> 4
;   00001xxx -> 3
;   000001xx -> 2
;   0000001x -> 1
;   00000001 -> 0
;

LMBD            MACRO
\@LMBD:
    CMPA    #$10                ; #2 , a ?>= 16
    BGE     \@LMBD_7654         ; #3 , goto 7654
\@LMBD_3210:    ; x x x x 3 2 1 0
    CMPA    #$04                ; #2 , a ?>= 4
    BGE     \@LMBD_32           ; #3 , goto 32
\@LMBD_10:      ; x x x x x x 1 0
    CMPA    #$02                ; #2 , a ?>= 2
    BGE     \@LMBD_1            ; #3 , goto 1
\@LMBD_0:       ; x x x x x x x 0
    CLRA                        ; #1 , 0 -> A
    BRA     \@LMBD_exit         ; #3 , exit
\@LMBD_1:       ; x x x x x x 1 x
    LDA     #$01                ; #2 , 1 -> A
    BRA     \@LMBD_exit         ; #3 , exit
\@LMBD_32:      ; x x x x 3 2 x x
    CMPA    #$08                ; #2 , a ?>= 8
    BGE     \@LMBD_3            ; #3 , goto 3
\@LMBD_2:       ; x x x x x 2 x x
    LDA     #$02                ; #2 , 2 -> A
    BRA     \@LMBD_exit         ; #3 , exit
\@LMBD_3:       ; x x x x 3 x x x
    LDA     #$03                ; #2 , 3 -> A
    BRA     \@LMBD_exit         ; #3 , exit
\@LMBD_7654:    ; 7 6 5 4 x x x x
    CMPA    #$40                ; #2 , a ?>= 64
    BGE     \@LMBD_76           ; #3 , goto 76
\@LMBD_54:      ; x x 5 4 x x x x
    CMPA    #$20                ; #2 , a ?>= 32
    BGE     \@LMBD_5            ; #3 , goto 5
```

```
\@LMBD_4:       ; x x x 4 x x x x
    LDA     #$04                ; #2 , 4 -> A
    BRA     \@LMBD_exit         ; #3 , exit
\@LMBD_5:       ; x x 5 x x x x x
    LDA     #$05                ; #2 , 5 -> A
    BRA     \@LMBD_exit         ; #3 , exit
\@LMBD_76:      ; 7 6 x x x x x x
    CMPA    #$80                ; #2 , a ?>= 128
    BGE     \@LMBD_7            ; #3 , goto 7
\@LMBD_6:       ; x 6 x x x x x x
    LDA     #$06                ; #2 , 6 -> A
    BRA     \@LMBD_exit         ; #3 , exit
\@LMBD_7:       ; 7 x x x x x x x
    LDA     #$07                ; #2 , 7 -> A
    \@LMBD_exit:
    ENDM

;*********************************************************************
;*
;* Variable Definition
;*
;*********************************************************************

_DATA_ZEROPAGE: SECTION SHORT

; Global Control Data

INT_lock:       DS.B    1                   ; global lock; -1 : unlock ; others : lock
INT_sets:       DS.B    1                   ; global running sets
INT_mask:       DS.B    1                   ; global mask;
INT_resv:       DS.B    1                   ; global reserved

; Global Ready Data

INT_RDYBEG:

INT_rdytab:     DS.B  RDY_OBJ_SIZE * MAX_RDY_TAB

INT_RDYEND:

INT_OBJBEG:

INT_objtab:

    INT_OBJ KBI_obj

    INT_OBJ TPM_obj

    INT_OBJ ADC_obj

    INT_OBJ TIM_obj

    INT_OBJ SCI_obj

INT_OBJEND:

;*********************************************************************
```

```
;*
;* Constant Definition
;*
;*********************************************************************

CONST:          SECTION

INTTABBEG:

INT_inittab:

    INT_INI KBI_obj,2,KBI_isr       ; KBI object

    INT_INI TPM_obj,1,TPM_isr       ; TPM object

    INT_INI ADC_obj,1,ADC_isr       ; ADC object

    INT_INI TIM_obj,1,TIM_isr       ; MTIM object

    INT_INI SCI_obj,1,SCI_isr       ; SCI object

INTTABEND:


;*********************************************************************
;*
;* Hardware Interrupt Entry Definition
;*
;*********************************************************************

ENTRANCE:

KBI_int:
    INT_dispatch KBI_obj
    RTI

TPM_int:
    INT_dispatch TPM_obj
    RTI

ADC_int:
    INT_dispatch ADC_obj
    RTI

TIM_int:
    INT_dispatch TIM_obj
    RTI

SCI_int:
    INT_dispatch SCI_obj
    RTI


;*********************************************************************
;*
;* Vector Table Definition
;*
```

```
;*****************************************************************
VECTOR_ROM:     SECTION

    ORG $FFD8

    DCB.W   1,ADC_int               ; Vector 19

    ORG $FFDA

    DCB.W   1,KBI_int               ; Vector 18

    ORG $FFE0

    DCB.W   1,SCI_int               ; Vector 15

    ORG $FFE6

    DCB.W   1,TIM_int               ; Vector 12

    ORG $FFF0

    DCB.W   1,TPM_int               ; Vector 07

;*****************************************************************
;*
;* Main Entry
;*
;*****************************************************************

; variable/data section
MY_ZEROPAGE: SECTION  SHORT         ; Insert here your data definition

; code section
MyCode:     SECTION
main:
_Startup:
    LDHX   #__SEG_END_SSTACK ; initialize the stack pointer
    TXS
    CLI                     ; enable interrupts

    JSR     INT_init        ; Interrupt Scheduler Initialized

mainLoop:
    ; Insert your code here
    NOP

    feed_watchdog
    BRA    mainLoop

;*****************************************************************
;*
;* ISR definitions
;*
;*****************************************************************

CODE:           SECTION
```

```
KBI_isr:
    ; input KBI jobs here
    RTS

; TPM interrupt service routine

TPM_isr:
    ; input TPM jobs here
    RTS

; ADC interrupt service routine

ADC_isr:
    ; input ADC jobs here
    RTS

; TIM interrupt service routine

TIM_isr:
    ; input TIM jobs here
    RTS

; SCI interrupt service routine

SCI_isr:
    ; input SCI jobs here
    RTS

;*********************************************************************
;*
;* void INT_init(void)
;*
;*********************************************************************

CODE:           SECTION

INT_init:
    JSR Global_init         ; Gloabal Initialize

    JSR Ready_init          ; Ready List Initialize

    JSR Object_init         ; Object Initialize

    RTS                     ; return

Global_init:

;               INT_mask = 1;

    MOV #IDLE, INT_mask     ; 1 -> INT_mask

;               INT_sets = 0;

    CLR INT_sets            ; 0 -> INT_sets

;               INT_lock = -1;
```

```
    MOV #READY, INT_lock    ; -1 -> INT_lock

    RTS                     ; return;

Ready_init:

;
;               for ( i = 0 ; i < MAX_RDY_TAB ; i++ )
;               {
;                   INT_rdytab[i].link = 0;
;                   INT_rdytab[i].tail = INT_rdytab[i];
;               }
;

    LDHX #INT_rdytab        ; @INT_rdytab -> H:X

Ready_loop:

; INT_rdytab[i].link = 0;

    TXA                     ; INT_rdytab[i] -> A

    CLR LINK,X              ; INT_rdytab[i].link = 0;

    STA TAIL,X              ; @INT_rdytab[i] -> INT_rdytab[i].tail

    INCX                    ; @INT_rdytab[i].tail -> X

    INCX                    ; @INT_rdytab[i + 1].link -> X

    CPHX #INT_RDYEND        ; i ?> MAX_RDY_TAB

    BNE Ready_loop          ; next loop

    RTS                     ; return;

; End of Loop

Object_init:

;               for ( i = 0 ; i < MAX_INT_TAB ; i++ )
;               {
;                   INT_objtab[i] = INT_inittab[i];
;               }

; initialize stack
;
; high address
;   INT_objtab low    4
;   INT_objtab high   3
;   INT_inittab low   2
;   INT_inittab high  1
; low address
;

    AIS #-4                 ; SP - 4 -> SP
```

```
    LDHX #INT_objtab        ; INT_objtab -> H:X
    STHX 3,SP               ; INT_objtab -> stack
    LDHX #INT_inittab       ; INT_inittab -> H:X
    STHX 1,SP               ; INT_inittab -> stack
; INT_inittab
Object_loop:
; INT_inittab[i] -> A
    LDHX 1,SP               ; @INT_inittab[i] -> H:X
    LDA ,X                  ; INT_inittab[i] -> H:X
    AIX #1                  ; i++
    STHX 1,SP               ; i++
; A -> INT_inittab[i]
    LDHX 3,SP               ; @INT_objtab[i] -> H:X
    STA ,X                  ; INT_inittab[i] -> INT_objtab[i]
    AIX #1                  ; i++
    STHX 3,SP               ; i++
; next loop
    CPHX #INT_OBJEND        ; i ?> MAX_INT_TAB
    BNE Object_loop         ; next loop
; uninitialize stack
    AIS #4                  ; pop stack
;               return;
    RTS                     ; return;
;*****************************************************************
;*
;*  void INT_post(swi)
;*  {
;*      if (swi->lock == 0)
;*      {
;*          return;
;*      }
;*
;*      swi->lock = 0;
```

```
;*
;*      INT_lock++;
;*
;*      INT_mask |= swi->mask;
;*
;*      ready->tail.link = swi;
;*
;*      ready->tail = swi;
;*
;*      if ((INT_mask > INT_sets) || INT_lock)
;*      {
;*          INT_lock--;
;*
;*          return;
;*      }
;*
;*      INT_exec();
;*
;*      return;
;*  }
;*
;*********************************************************************

;*********************************************************************
;
;   If swi posted, 6 cycles
;   If swi not posted but not executed, 95 cycles
;   If swi not posted and executed, 233 cycles
;
;*********************************************************************

CODE:           SECTION

INT_post:

;               if (swi->lock == 0)
;               9 cycles

    LDA LOCK,X              ; swi->lock -> X                    #3
    BNE post                ; if (swi->lock != 0) post          #3

;               {
;                   return;
;               }
;               6 cycles

    RTS                     ; interrupt return                  #6 , total 6 cycles

post:

;               swi->lock == 0;
;               5 cycles

    CLR LOCK,X              ; 0 -> swi->lock                    #5

;               INT_lock++;
;               5 cycles
```

```
    INC INT_lock            ; INT_lock++                        #5

;               Disable Interrupts
;               1 cycle

    SEI                     ; disable interrupt                 #1

;               INT_sets |= swi->mask;
;               9 cycles

    LDA MASK,X              ; swi->mask -> A                    #3
    ORA INT_sets            ; INT_sets | mask -> A              #3
    STA INT_sets            ; INT_sets | mask -> INT_sets       #3

;               priority = lmbd(mask);
;               23 cycles

    LDA MASK,X              ; swi->mask -> A                    #3
    LMBD                    ; lmbd(swi->mask) -> A              #20

;               ready = INT_rdytab[priority];
;               3 cycles

    ASLA                    ; sizeof(INT_Obj) * priority -> A   #1
    ADD #INT_rdytab         ; ready-> A                         #2

;               push(swi);
;               2 cycles

    PSHX                    ; swi -> stack                      #2

;               push(ready);
;               2 cycles

    PSHA                    ; ready -> stack                    #2

;               ready->tail->link = swi;
;               ready->tail = swi;
;               18 cycles

    TAX                     ; ready -> X                        #1
    LDA 2,SP                ; swi -> A                          #4
    LDX TAIL,X              ; ready->tail -> X                  #3
    STA ,X                  ; swi -> ready->tail->link          #2
    PULX                    ; ready -> X , SP -> swi            #3
    STA TAIL,X              ; swi -> ready->tail                #3
    AIS #1                  ; SP + 1 -> SP , SP -> xx           #2

;               Enable Interrupts
;               1 cycle

    CLI                     ; enable interrupt                  #1

;               if ((INT_mask > INT_sets) || INT_lock)
;               12 cycles
```

```
    LDA INT_mask            ; INT_mask -> A                     #3
    CMP INT_sets            ; INT_mask ?> INT_sets -> A         #3
    BGT unlock              ; if (INT_mask > INT_sets) unlock   #3
    LDA INT_lock            ; INT_lock -> A                     #3

;               INT_exec();
;               3 cycles

    BEQ INT_exec            ; if (INT_mask > INT_sets) goto unlock; #3 , total 90cycles
unlock:
;               {
;                   INT_lock--;
;                   return;
;               }
;               11 cycles

    DEC INT_lock            ; INT_lock--                        #5
    RTS                     ; interrupt return                  #6 , total 95 cycles

;*****************************************************************
;*
;*  void INT_exec(void)
;*  {
;*      for (;;)
;*      {
;*
;*          Push(INT_mask);
;*
;*          priority = lmbd(1,INT_sets);
;*
;*          rdytab = &SWI_D_rdytab[priority];
;*
;*          swi = (INT_Obj *)ready->link;
;*
;*          swi->lock = -1;
;*
;*          rdytab->link = swi->link;
;*
;*          if(swi->link == NULL)
;*          {
;*              SWI_D_curset ^= swi->mask;
;*
;*              rdytab->tail = (PUINT)rdytab;
;*
;*              rdytab->link = (PUINT)NULL;
;*
;*          }
;*          else
;*          {
;*              swi->link = NULL;
;*          }
;*
;*          INT_mask <<= 1;
;*
;*          INT_lock = -1;
;*
;*          (*func)();
```

```
;*
;*          INT_lock = 0;
;*
;*          INT_mask = Pop();
;*
;*          if (INT_mask > INT_sets)
;*          {
;*              break;
;*          }
;*      }
;*
;*      return;
;*  }
;*
;*********************************************************************

;               INT_exec();

    INT_exec:

;               INT_lock++;
;               5

    INC INT_lock            ; INT_lock + 1 -> INT_lock          #5

;               INT_run();

INT_run:

;               push(INT_mask);
;               5 cycles

    LDA INT_mask            ; INT_mask -> A                     #3
    PSHA                    ; INT_mask -> stack                 #2

;               priority = lmbd(INT_sets);
;               23 cycles

    LDA INT_sets            ; INT_sets -> A                     #3
    LMBD                    ; lmbd(INT_sets) -> A               #20

;               swi = (INT_Obj *)ready->link;
;               5 cycles

    ASLA                    ; sizeof(INT_Obj) * priority -> A   #1
    ADD  #INT_rdytab        ; INT_rdytab[priority] -> A         #2
    PSHA                    ; ready -> stack                    #2

;               swi = (INT_Obj *)ready->link;
;               8 cycles

    TAX                     ; ready -> X                        #1
    LDX LINK,X              ; swi -> X                          #5
    PSHX                    ; swi -> stack                      #2

;               swi->lock = -1;
;               5 cycles
```

```
    DEC LOCK,X              ; -1 -> swi->lock                   #5

;               ready->link = swi->link;
;               9 cycles

    LDA LINK,X              ; swi->link -> A                    #3
    LDX 2,SP                ; ready -> X                        #3
    STA LINK,X              ; swi->link -> ready->link          #3

;               if (ready->link == 0)
;               {
;                   ready->tail = (PUCHAR)ready;
;                   ready->link = (PUCHAR)0;
;                   INT_sets ^= swi->mask;
;               }
;               26 cycles

    BNE not_empty           ; if (ready->head != 0) goto L4     #3
    STX TAIL,X              ; ready -> ready->tail              #3
    CLR LINK,X              ; 0 -> ready->link                  #5
    PULX                    ; swi -> X                          #3
    LDA MASK,X              ; swi->mask -> A                    #3
    EOR INT_sets            ; INT_sets ^ mask -> A              #3
    STA INT_sets            ; INT_sets ^ mask -> INT_sets       #3
    BRA dispatch            ; goto dispatch                     #3

;               else
;               {
;                   swi->link = 0;
;               }
;               8 cycles

not_empty:

    PULX                    ; swi -> X                          #3
    CLR LINK,X              ; 0 -> swi->link                    #5

dispatch:

;               pop();
;               2 cycles

    AIS #1                  ; clear stack                       #2

;               INT_mask = swi->mask << 1;
;               7 cycles

    LDA MASK,X              ; swi->mask -> A                    #3
    ASLA                    ; swi->mask << 1 -> A               #1
    STA INT_mask            ; swi->mask << 1 -> INT_mask        #3

;               INT_lock = -1;
;               4 cycles

    MOV #READY,INT_lock     ; -1 -> INT_lock                    #4
```

```
;               (* func)();
;               20 cycles

    PSHH                    ; push(H)                           #2
    PSHX                    ; push(X)                           #2
    LDHX FUNC,X             ; swi->func -> H:X                  #5
    JSR ,X                  ; (* func)();                       #5
    PULX                    ; X = pop();                        #3
    PULH                    ; H = pop();                        #3

;               INT_lock = 0;
;               4 cycles

    MOV #POST,INT_lock      ; 0 -> INT_lock                     #4

;               if (INT_mask > INT_sets)
;               9 cycles

    PULA                    ; INT_mask -> A , SP -> xx          #3
    STA INT_mask            ; INT_mask -> INT_mask              #3
    CMP INT_sets            ; INT_mask ?> INT_sets              #3

;               {
;                   break;
;               }
;               3 cycles

    BLE INT_run             ; if (INT_mask > INT_sets) break;   #3

;               INT_lock--;
;               5 cycles

    DEC INT_lock            ; INT_lock - 1 -> INT_lock;         #5

;               return;
;               6 cycles

INT_post_exit:
    RTS                     ; interrupt return                  #6 , total 143 cycles

;*****************************************************************
;*  End Of File
;*****************************************************************
```

THIS PAGE IS INTENTIONALLY BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3496
Rev.0
07/2007

freescale™
semiconductor