AN205548

F²MC-16FX Family, Interrupts

This application note describes the functionality of the internal Interrupts and gives some examples

Contents

1 Introduction

This application note describes the functionality of the internal Interrupts and gives some examples.

1.1 Key Features

- Each Resource uses one Interrupt (no Channel Sharing)
- 7 Interrupt Priority Levels selectable (7 = Interrupts Disabled)
- Interrupt Vector Table Relocatable
- Software and Hardware Interrupts
- Software and Hardware Exceptions
- Non-Maskable Interrupt (NMI)

2 Interrupt Types

The Basic Functionality of Internal Interrupts

2.1 Available Interrupts / Exceptions

There are four different types of Interrupts / exceptions available.

2.1.1 Hardware Interrupts

This kind of Interrupts is generated by internal resources. An Interrupt is generated if the corresponding Interrupt enable bit of the resource is set, the level of the vector is equal or less than the global Interrupt level, Interrupts are globally enabled and an Interrupt cause has occurred. While execution of interrupt service routine (ISR), the System Stack Pointer is enabled (`CCR:S` = 1). These interrupts are serviced in user mode only (`CCR:P` = 1). Interrupt level of hardware interrupts is configured by `IL` (Interrupt Level) bits of `ICR` (Interrupt Control Register),

2.1.2 Software Interrupts

Software Interrupts can be requested by executing the INT or INTP instructions. This type of interrupt has no Interrupt level. It also does not have interrupt request or enable flag. While execution of ISR, global Hardware Interrupts are disabled (CCR:I = 0), hence all the hardware interrupts are suspended until INT ISR execution has finished. However INT ISR can be interrupted by hardware exception. The System Stack Pointer is enabled (CCR:S = 1).

2.1.3 Hardware Exceptions

Hardware exceptions are external events which are not maskable by any software instruction. They are processed like Interrupts. They have their own vectors. The following hardware exceptions can occur:

NMI

NMI provides external hardware exception handling. NMI has a fixed interrupt level unlike hardware interrupts. It is level P4 i.e. Privileged Mode – Level 4. That means while execution of NMI ISR, the mode is changed to Privileged Mode (CCR:P = 0) and the interrupt level mask is configured to 4 (PS:ILM = 4). Hence all interrupts/exceptions are suspended until NMI ISR execution has finished. System Stack Pointer is enabled (CCR:S = 1). The P flag and ILM are restored at execution of the RETI instruction. Hence user mode would be restored if the same mode was active at the time of NMI.

HW-INT9 (Embedded Debug Support)

HW-INT9 is used by address match detection function. With that function embedded debug support (operand address break or data value break) or a simple memory protection can be provided. HW-INT9 has a fixed interrupt level unlike hardware interrupts. It is level P6 i.e. Privileged Mode – Level 6. That means while execution of HW-INT9 ISR, the mode is changed to Privileged Mode (CCR:P = 0) and the interrupt level mask is configured to 6 (PS:ILM = 6). Hence all interrupts / exceptions are suspended until HW-INT9 ISR execution has finished, except NMI. System Stack Pointer is enabled (CCR:S = 1). The P flag and ILM are restored at execution of the RETI instruction. Hence user mode would be restored if the same mode was active at the time of HW-INT9.

For detailed information please refer the Memory Patch application note AN204770.

2.1.4 Software Exceptionsl

Software exceptions are always accepted. Same as software interrupts, software exceptions disable any hardware interrupt acceptance. The following software exceptions can occur:

Undefined Instruction

All codes that are not defined in the instruction map are handled as undefined instructions. When an undefined instruction is executed, the ISR whose starting address is stored at interrupt vector INT 10 is executed. While storing the CPU status, PC value saved in the stack is the address at which the undefined instruction is stored. While execution of ISR, global Hardware Interrupts are disabled (CCR:I = 0) hence all the all hardware interrupts are suspended until undefined instruction ISR execution has finished. However undefined instruction ISR can be interrupted by hardware exception. The System Stack Pointer is enabled (CCR:S).

INT9 Instruction

INT9 instruction branches to ISR indicated by interrupt vector INT 9. While execution of ISR, global Hardware Interrupts are disabled (CCR:I = 0) hence all hardware interrupts are suspended until INT9 ISR execution has finished. However INT9 ISR can be interrupted by hardware exception. The System Stack Pointer is enabled (CCR:S).

2.2 Direct Memory Access (DMA)

DMA transfers are accepted regardless of the status of the I flag and the interrupt level.

For detailed information please refer to the DMA application note AN204769.

2.3 Interrupt Acceptance, Levels and Modes

2.3.1 User mode

If the P bit of CCR register is set to 1, it indicates “User Mode”. In the user mode the hardware interrupts are serviced depending on the ILM.

2.3.2 Privileged Mode

If the P bit of CCR register is set to 0, it indicates “Privileged Mode”. This (CCR:P = 0) happens in the event of hardware exceptions such as NMI and HW-INT9. If the P flag is cleared, ILM of PS defines system interrupt levels of the privileged mode (P0 to P7). These interrupt levels **always** have **higher** priority than any ILM setting in user mode (U0 to U7). Hence in the privileged mode all hardware interrupts are suspended. User mode would be restored after the execution of RETI instruction (in the hardware exception ISR), if the same mode was active at the time of hardware exception.

The following table explains various interrupts/exceptions, corresponding levels, acceptance conditions etc.

Table 1. Interrupt Acceptance and Levels - I

Interrupt/ Exception	Description	Interrupt/ Vector Number	Interrupt Level	Acceptance Condition	Action after Acceptance
Hardware Exception	NMI	INT 11	Always Fixed, P4 (Privileged Mode - Level 4)	Current instruction execution is finished String instruction is Interrupted If ILM (Interrupt Level Mask) of PS (Processor Status) register is greater than 4 or P (Privileged Mode) flag of CCR (Code Condition Register) is 1	Save CPU status to system stack S = 1 (use system stack) P = 0 (Privileged Mode) ILM = 4 i.e. all interrupts/exceptions suspended until NMI ISR execution Branch to interrupt vector
Hardware Exception	HW-INT9	INT 9	Always Fixed, P6	Current instruction execution is finished String instruction is Interrupted If ILM of PS register is greater than 6 or P flag of CCR is 1	Save CPU status to system stack S = 1 (use system stack) P = 0 (Privileged Mode) ILM = 6 i.e. all interrupts/exceptions suspended until HW-INT9 ISR execution except NMI Branch to interrupt vector
Software Exception	INT9 Instruction	INT 9, Shared with HW-INT9 Vector	No Level	Always Accepted	Save CPU status to system stack S = 1 (use system stack) I = 0 i.e. all hardware interrupts are suspended until INT9 ISR execution, INT9 ISR can be interrupted by hardware exception Branch to interrupt vector
Software Exception	Undefined Instruction	INT 10	No Level	Always Accepted	Save CPU status to system stack S = 1 (use system stack) I = 0 i.e. all hardware interrupts are suspended until Undefined Instruction ISR execution, Undefined Instruction ISR can be interrupted by hardware exception Branch to interrupt vector

Interrupt/ Exception	Description	Interrupt/ Vector Number	Interrupt Level	Acceptance Condition	Action after Acceptance
Software Interrupt	INT Instruction	INT 0 to 255, As specified by operand	No Level	Always Accepted	Save CPU status to system stack S = 1 (use system stack) I = 0 i.e. all hardware interrupts are suspended until INT ISR execution, INT ISR can be interrupted by hardware exception Branch to interrupt vector

Table 2. Interrupt Acceptance and Levels - II

Interrupt/ Exception	Description	Vector }Number	Interrupt Level	Acceptance Condition	Action after Acceptance
Software Interrupt	INTP Instruction	24-Bit physical address as specified by operand	No Level	Always Accepted	Save CPU status to system stack S = 1 (use system stack) I = 0 i.e. all hardware interrupts are suspended until INT ISR execution, INT ISR can be interrupted by hardware exception Branch to interrupt vector
Hardware Interrupt	Resource / Peripheral Interrupt	INT 13 onwards	As configured by IL (Interrupt Level) bits of ICR (Interrupt Control Register), Between U0 to U7 (User Mode - Level 0 to 7)	Current instruction execution is finished String instruction is Interrupted If ILM of PS register is greater than IL of the peripheral and P flag of CCR is 1 and I flag of CCR is 1 For multiple requests with same IL, smallest interrupt number is accepted.	Save CPU status to system stack S = 1 (use system stack) ILM = IL i.e. peripheral ISR can be interrupted by hardware exception, software exception, software interrupt and hardware interrupt with lower value of IL (i.e. with higher priority) Branch to interrupt vector
Hardware Interrupt	Delayed Interrupt	INT 12	As configured by IL bits of ICR, Between U0 to U7	Current instruction execution is finished String instruction is Interrupted If ILM of PS register is greater than IL of the peripheral and P flag of CCR is 1 and I flag of CCR is 1 No peripheral interrupts pending with same IL as of delayed interrupt	Save CPU status to system stack S = 1 (use system stack) ILM = IL i.e. peripheral ISR can be interrupted by hardware exception, software exception, software interrupt and hardware interrupt with lower value of IL (i.e. with higher priority) Branch to interrupt vector

2.4 Interrupt Latency

2.4.1 Context Saving / Restoring

Once the interrupt is generated and if it is enabled, “normally” the following series of steps are performed:

1. CPU finishes current instruction execution.
2. It stores the current status to stack.
3. It fetches the starting address of the ISR from the corresponding interrupt vector.
4. And Branches to the ISR.

Steps 2 to 4 are also termed as “Context Saving”.

Once the ISR execution is finished, while the execution of RETI instruction the following step is performed:

1. The status is retrieved from the stack.
2. CPU starts executing the code which it was executing at the time of interrupt.

Steps 1 and 2 are also termed as “Context Restoring”.

The time taken for the Context Saving and Context Restoring is dependent on:

- Location of Stack (Internal RAM / External RAM)
- Location of Interrupt Vector (Internal Flash / External Flash)
- Location of Interrupt Service Routine (Internal Flash / External Flash)
- Read Wait States in case of internal flash
- Address indicated by stack pointer

If we consider that the internal RAM is used for stack, internal Flash is used for vector as well as routines and internal flash wait state is 0 then the cycles required for context saving/restoring are:

Table 3. Cycles Required for Context Saving / Restoring

Cycle Required	Address Indicated by Stack Pointer	
	Even Numbered Address	Odd Numbered Address
Context Saving	10	12
Context Restoring	9	11

These timing gets worsened if the stack / interrupt vector / ISRs are located in the external memory. This is because the wait cycles for external bus transfer get added to the above mentioned cycles.

2.4.2 Interrupt Deferring Instructions / Prefix Codes

Other than the above mentioned factors interrupt latency is also dependent on the currently executing instructions or prefix codes. Some of the instructions and all of the prefix codes defer or delay interrupts during their execution. This means that even if the valid interrupt arrives while such instructions / prefix code are getting executed, then such (single / series of) instructions / prefix codes would continue getting executed. After their execution a normal instruction (which does fall into the interrupt deferring category) would also be executed and then only the interrupt would be serviced. Such interrupt deferring instructions / prefix codes are:

Table 4. Interrupt Deferring Instructions / Prefix Codes

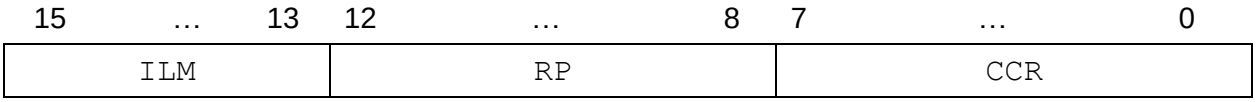
MOV ILM, #imm8	AND CCR, #imm8	OR CCR, #imm8	POPW PS
PCB	ADB	DTB	SPB
NCC	CMR		

2.5 Registers

2.5.1 Processor Status (PS)

The Processor Status contains three sub sections.

Table 5. Processor Status



For Interrupts the Interrupt Level Mask (ILM) and the I-Bit of the Condition Code Register (CCR) are important.

Interrupt Level Mask (ILM)

The three bits of the ILM are “0” after reset. Different settings are described in the following table:

Table 6. Interrupt Level Mask

Bit No. 15	Bit No. 14	Bit No. 13	Level value	Levels of accepted Interrupts
ILM2	ILM1	ILM0		
0	0	0	0	Interrupts disabled
0	0	1	1	0
0	1	0	2	1 and below
0	1	1	3	2 and below
1	0	0	4	3 and below
1	0	1	5	4 and below
1	1	0	6	5 and below
1	1	1	7	6 and below

The Level can be set in C with the language extension directive `__set_il(n)`, where *n* is the level. The machine instruction for this is `MOV ILM,#n`.

Condition Code Register (CCR)

The CCR consists of the following bits:

Table 7. Condition Code Register

Bit No.	Bit Name	Value after Boot ROM execution	Description
7	P[1]	1	Privileged Mode Flag. This flag is set by Boot-ROM. 1 = User Mode, 0 = Privileged Mode*
6	I	0	Global Interrupt Enable Flag
5	S	1	System/User Stack Flag. 1 = System Stack, 0 = User Stack. This bit is set to "1" after execution of INT, INT9 or INTP instruction and also in case of hardware interrupts (before execution of interrupt service routine).
4	T	x	Sticky Bit Flag. This bit is used by logical/arithmetic right shift operations
3	N	x	Negative Flag
2	Z	x	Zero Flag
1	V	x	Overflow Flag
0	C	x	Carry Flag

Interrupts can be enabled globally by the C language extension __EI() and disabled by __DI().

There is no direct bit access to the CCR in assembler, but bits can be set indirectly with logical instructions: Setting the I-Bit: OR CCR,#40 and clearing it: AND CCR,#BF.

Please note, that __DI() and __EI() cannot be set consecutively. Please set at least one instruction in-between, such as a NOP.

Wrong	Correct

__DI();
__EI();
	__DI();
	__wait_nop;
	__EI();

__EI();
__DI();
	__EI();
	__wait_nop();
	__DI();

[1] This bit remains "1" if it was set to "1" before. It would be cleared only while execution of hardware exception ISR.

2.5.2 Interrupt Vector Table Base Register (TBR)

The TBR contains the base address for the Interrupt vector table. It is 16-bit wide, where the 2 least significant Bits are “0”. The upper byte stands for the program bank of the 24-bit address space.

TBR:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TBRH								TBRL						0	0

Address Space:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Program Bank								Upper 16-Bit Address								Lower 16-Bit Address							

The Interrupt vector table has a size of 1024 Bytes. So, if the TBR for example has the value 0xAA24 the vector table starts from 0xAA2400 and ends at 0xAA27FF.

2.5.3 Interrupt Vector Table

The Interrupt Vector Table is a set of 256 quadruple Bytes, which contains the 24-bit Interrupt Service Routine address. The most significant Byte of the quadruple is undefined.

Note, that Reset, INT9, Undefined Instruction, and NMI are also contained in this table.

In general the table is built as follows:

Table 8. Interrupt Vector Table

Interrupt Vector Number	Vector Address	Index of Level Register in ICR	Hardware Interrupt / Interrupt Cause
INT0	TB + 0x3FC	-	-
…	…		
INT7	TB + 0x3E0		
INT8	TB + 0x3DC	-	Reset
INT9	TB + 0x3D8	-	INT9 Instruction/HW-INT9
INT10	TB + 0x3D4	-	Undefined Instruction
INT11	TB + 0x3D0	-	NMI
INT12	TB + 0x3CC	IL12	Delayed Interrupt
INT13	TB + 0x3C8	IL13	RC Clock Timer
INT14	TB + 0x3C4	IL14	Main Clock Timer
INT15	TB + 0x3C0	IL15	Sub Clock Timer
INT16	TB + 0x3BC	IL16	*Reserved*
INT17	TB + 0x3B8	IL17	Device specific Peripheral Vectors
INT18	TB + 0x3B4	IL18	
INT19	TB + 0x3B0	IL19	
...	...	...	
INT254	TB + 0x004	-	-
INT255	TB + 0x000	-	-
TB = Interrupt Vector Table Base Address			

Please note that INT0 till INT7 can collide with the vectored call subroutine table, if the actual program bank (value of PCB) is same as TBRH. Therefore please use CALLV only if the actual program bank is the same in which the Vectored Call Table is located.

2.5.4 Interrupt Control Register (ICR)

Using this register interrupt levels of hardware peripheral interrupts can be configured. For all other vectors ICR:IX contains the vector number and ICR:IL the level. The level of a vector can be read out via ICR:IL by writing the vector number to ICR:IX.

Table 9. Interrupt Control Register

Bit No.	Bit Name	Initial Value	Description
15 … 8	IX7 … IX0	0	Index of the Interrupt Level to be accessed
7 … 3	-	X	*Undefined Bits*
2 … 0	IL2 … IL0	1	Interrupt Level of Index specified in IX[7:0]

To set an Interrupt level please always access ICR via word writing.

For reading out configured interrupt level of a particular hardware peripheral do the following:

- Write the corresponding index to the Interrupt Index Register - IDX (same as IX[7:0] bits of ICR) (essentially a byte-write).
- Read out the entire ICR (essentially a word-read).
- Confirm that the read back value and the written value of IX[7:0] match and then IL[2:0] value is considered to be correct.

2.5.5 Non Maskable Interrupt (NMI)

The NMI is disabled after Reset. Once it is enabled in an application, it cannot be disabled. Only another Reset can disable this feature.

The NMI has a control register as follows:

Table 10. Non Maskable Interrupt

Bit No.	Bit Name	Initial Value	Description
15 … 11	-	X	*Undefined Bits*
10	LEV	1	Level Bit. 1 = NMI Pin high active, 0 = NMI Pin low active.
9	EN	0	Enable Bit. Writing “1” enables NMI feature until Reset. Writing “0” has no effect. Write “0” to FLAG before enabling NMI. Also set the correct level using the LEV bit before enabling NMI.
8	FLAG	X	NMI Flag. 1 = NMI occurred. Writing “0” clears Flag. Write “0” to this Bit before enabling NMI.

The Non Maskable Interrupt functionality is available on either of two pins: NMI and NMI_R. The bit 1 NMI_R of Peripheral resource relocation register 7 configures the same. If this bit is 0 then Pin NMI is used as Non Maskable input pin and if this bit is 1 then Pin NMI_R is used as Non Maskable input pin. The relocation cannot be changed after the NMI is enabled.

Note that the corresponding Port Input Enable Register (PIER) has to be set also.

3 Interrupt Recommendations and Examples

Recommendations and Examples for the Interrupt Usage

3.1 Interrupt Vector Definition

By using the `#pragma intvect` directive, an interrupt vector is defined. It is recommended to use our standard template project, which contains a file called *vectors.c* which performs all interrupt settings. The user may use or copy and modify this file for own projects.

Please make sure to always define the complete interrupt vector table in just one C module as shown in below sample code and do not split it.

```
#pragma intvect My_IRQHandler_1   12    /* Delayed Interrupt           */
#pragma intvect DefaultIRQHandler 13    /* RC Clock Timer              */
#pragma intvect My_IRQHandler_2   14    /* Main Clock Timer            */
#pragma intvect My_IRQHandler_3   15    /* Sub Clock Timer             */
#pragma intvect DefaultIRQHandler 16    /* Reserved                    */
#pragma intvect My_IRQHandler_4   17    /* EXT0                        */
#pragma intvect My_IRQHandler_5   18    /* EXT1                        */
#pragma intvect My_IRQHandler_6   19    /* EXT2                        */

    . . .
```

Please note, that if the Interrupt service functions are located in a different C module, their prototypes have to be defined also for the vector definition.

3.2 Interrupt Level Predefinition

It is possible to predefine all interrupt levels with a default level by a loop. The following sample code shows how to do this.

```
#define MIN_ICR   12       /* Interrupts with levels begin here   */
#define MAX_ICR  120       /* Device specific Interrupt Level end */

#define DEFAULT_ILM_MASK 7

void InitIrqLevels(void)
{
       unsigned char irq;

       for (irq = MIN_ICR; irq <= MAX_ICR; irq++)
       {
              ICR = (irq << 8) | DEFAULT_ILM_MASK;
       }
}
```

3.3 Setting an Interrupt Level

Assume you want to assign the level 2 to the interrupt vector 12. With the following formula in sample code below, the correct word is already calculated by the pre-processor.

```
void InitIrqLevels(void)
{
  ICR = (12 << 8) | 2;
}
```

3.4 Reading an Interrupt Level

If the currently configured level of the peripheral whose vector number is 12 can be read as sample code below:

```
unsigned int icr;
unsigned char idx;

void readIrqLevel(void)
{
  IDX = 12;
  icr = ICR;
  idx = (unsigned char)(icr >> 8);
  if (idx == 12)
  {
       /* IL[2:0] is correct level of interrupt whose vector is 12 */
  }
}
```

3.5 Interrupt Service Routine

The type qualifier `__interrupt` shows the complier, that by entering the following function several registers have to be saved and the function has to be finished with the `RETI` instruction. This function is always of the type of `void` and has no arguments.

Please note, that the Interrupt cause bit always has to be cleared in the service routine; otherwise the service routine will be entered again after execution. Most of the resources have a special Interrupt clear bit, but some have an auto-clear by accessing a special register (e. g. UART read buffer).

The following sample code shows a typical Interrupt service routine.

```
__interrupt void My_Interrupt_Service_Routine_1(void)
{
       Resource_Interrupt_Clear_Bit = 0;       /* clear Interrupt cause */

       /* do something here */
}
```

3.6 Interrupt Service Routine without Register Saving

If the Interrupt service routine does not use any variables and just accesses some resource registers, it is not needed to save any registers before entering the service routine and restoring them at the end as shown in below sample code. The type qualifier `__nosavereg` signalizes this to the compiler. It should stand before the `__interrupt` type qualifier.

```
__nosavereg __interrupt void My_Interrupt_Service_Routine_2(void)
{
       Resource_Interrupt_Clear_Bit = 0;       /* clear Interrupt cause */

       /* only access to resource registers here */
       /* no variables allowed to use here       */
}
```

3.7 Setting global Interrupt Level

To set the global Interrupt Level via the `ILM` register, the language extension `__set_il(n)` exists, where *n* is the global level. Please also see Interrupt Level Mask (`ILM`) in section 2.5.1.

Assume the level 3 is desired. The following sample code shows how to access the `ILM` register from C code.

```
__set_il(3);
```

3.8 Enabling and Disabling Interrupts globally

To enable Interrupts globally use the `__EI()` language extension. `__DI()` disables Interrupts globally. Both extensions access the `I`-Bit in the Condition Code Register as shown in the sample code below.

Please also see Condition Code Register (`CCR`) in section 2.5.1.

```
    . . .

__EI();   /* Enable Interrupts globally */

    . . .

__DI();   /* Disable Interrupts globally */

    . . .
```

3.9 Order of Initialization

For the Interrupt initialization the order of the steps has to be done like in the following example sample code.

```
#define MIN_ICR   12       /* Interrupts with levels begin here   */
#define MAX_ICR  120       /* Device specific Interrupt Level end */

#define DEFAULT_ILM_MASK 7

void InitIrqLevels(void)
{
       unsigned char irq;

       for (irq = MIN_ICR; irq <= MAX_ICR; irq++)
       {
              ICR = (irq << 8) | DEFAULT_ILM_MASK;
       }

       ICR = (12 << 8) | 2;        /* Example: ICR10 has level 2 */
       ICR = (13 << 8) | 3;        /* Example: ICR11 has level 3 */
}

       . . .

void main(void)
{
       InitIrqLevels();    /* First initialize all Interrupt Levels */
       __set_il(7);        /* Set global Interrupt Level to 7       */
       __EI();             /* Enable Interrupt globally             */

       . . .

}
```

Note, that in this example only the initialization flow is shown. Neither the vector definition nor the Interrupt service routines are shown here.

3.10 NMI Initialization

Because the FLAG-Bit of the NMI register is undefined after power-on, it has to be set to “0” before enabling the NMI feature. The following sample code gives an example to do the NMI initialization on MB96340 Series. Note that also the Port Input Enable Register (PIER) needs to be enabled.

```
void Init_NMI(void)
{
    PIER07_IE0 = 1;         /* P07_0 input enable for NMI of MB9634x series */
    NMI_LEV = 1;            /* 1 = high active, 0 = low active              */
    NMI_FLAG = 0;           /* Clear NMI cause flag prophylactic            */
    NMI_EN = 1;             /* NMI now enabled, disable only by reset       */
}
```

3.11 NMI Pin Relocation

The Non Maskable Interrupt functionality is available on either of two pins: NMI and NMI_R. The following example sample code shows how to configure NMI_R pin as an input pin instead of NMI pin on MB96340 Series. Note that also the Port Input Enable Register (PIER) needs to be enabled.

```
void Relocate_NMI(void)
{
    PIER05_IE5 = 1;         /* P05_5 input enable for NMI_R of MB9634x series */
    PRRR7_NMI_R = 1;        /* Pin NMI_R is used as input pin                 */
}
```

3.12 Interrupt Vector Relocation

As discussed in the section 2.5.2, the interrupt vector table can be relocated to any memory location in steps of 1 KB using the TBR.

This may be required when the original vector table is inaccessible at some point of time and still the interrupts need to be serviced. Such need may arise in an application when some sector of the Main Flash needs to be erased or programmed and while it is happening, some interrupts such as CAN receive or UART receive interrupts need to be attended.

The following are the preconditions for the relocation discussed above for MB96340 Series:

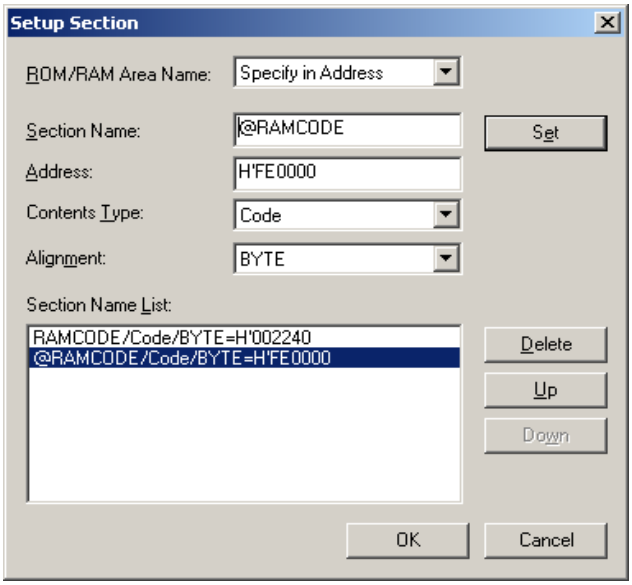
1. The entire application including the interrupt vector is available in the Main Flash.
2. The space equivalent to the entire vector table or the required vectors is reserved in the memory other than Main Flash. It can be Satellite Flash or RAM. For ease of understanding, we would consider that space in RAM would be reserved for the required vectors.
    a. The below sample code indicates that space for relocated CAN0 vector is reserved at RAM address 0x007F78 (vector number 33, offset from TBR = 0x378) and space for relocated UART0-RX vector is reserved at the RAM address 0x007EC0 (vector number 33, offset from TBR = 0x2C0). This is considering later the vector table is relocated to RAM starting from address 0x7C00.

```
#pragma segment DATA=INTVECT2_CAN,locate=0x007F78
__interrupt void (*CAN0_ptr)(void);
#pragma segment DATA=DATA

#pragma segment DATA=INTVECT2_UART,locate=0x007EC0
__interrupt void (*UART0_RX_ptr)(void);
#pragma segment DATA=DATA
```

3. The routine or the function which actually erases / programs the sector of the flash should be available in the memory other than the Main Flash. It can be Satellite Flash or RAM. For ease of understanding, we would consider that the routine is mapped to RAM.
    a. In order to achieve the above, RAMCODE section needs to be defined as below:

Figure 1. CONST Section Setting



The RAM and the Flash area reserved would be dependent on the particular processor which the application uses.

The above dialog box can be reached as mentioned below…

Project -> Setup Project -> Linker -> Disposition Connection -> Set Section

b. In order to link the function `EraseSector()` to RAM, the `#pragma section` directive needs to be used in sample code as follows:

```
#pragma section FAR_CODE=RAMCODE
unsigned char EraseSector (int sec_num)
{
       . . .

       . . .
}
```

4. The interrupt service routines those have to be accessed while the sector of Main Flash is erased are also mapped to RAM. We consider that CAN0 and UART0 Receive interrupt needs to be serviced.

   a. The following sample code gives the interrupt level and vector configuration.

```
void InitIrqLevels(void)
{
  ICR = (33 << 8) | 2;                  // CAN 0 of MB9634x Series
  ICR = (79 << 8) | 3;                  // UART-RX 0 of MB9634x Series
}
   . . .

__interrupt void CAN0_ISR(void);        // Prototype
__interrupt void UART0_Rx_ISR(void);    // Prototype

   . . .

#pragma intvect CAN0_ISR      33        // CAN 0 of MB9634x Series
#pragma intvect UART0_Rx_ISR  79        // UART-RX 0 of MB9634x Series
```

   b. The following sample code links the ISRs to RAM.

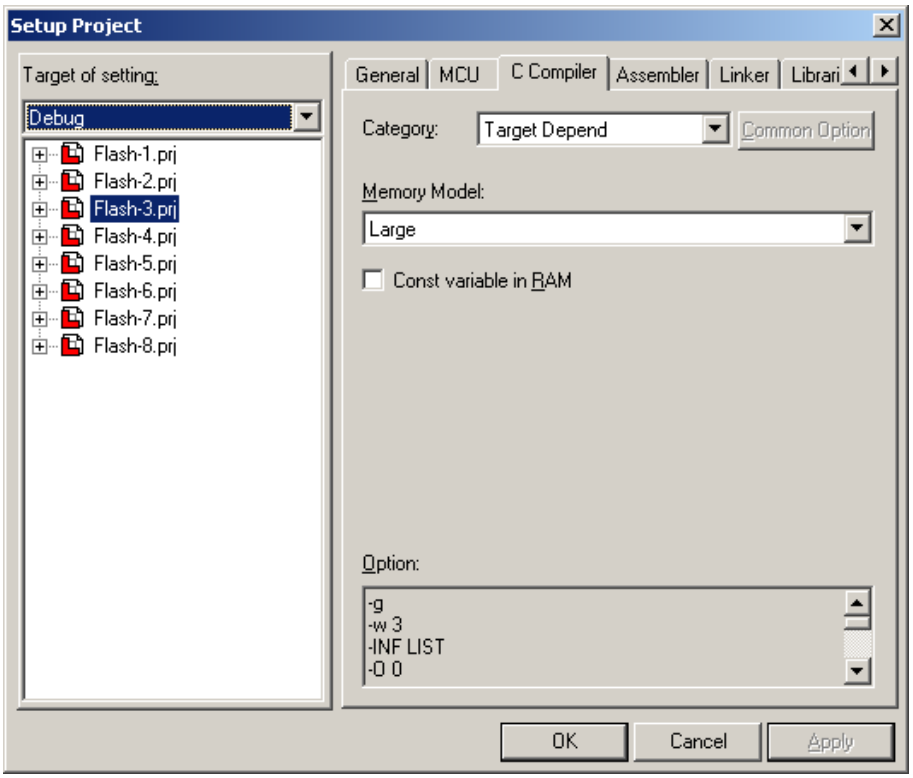
```
#pragma section FAR_CODE=RAMCODE
__interrupt
void CAN0_ISR (void)
{
       . . .

       . . .
}
__interrupt
void UART0_Rx_ISR (void)
{
       . . .

       . . .
}
```

5. The Memory Model for the project should be selected as “Large”.

Figure 2. Memory Model Setting



The above dialog box can be reached as mentioned below…

Project -> Setup Project -> C Compiler -> Category -> Target Depend -> Memory Model

Once the above preconditions are met then the actual relocation would be carried out in the following steps for MB96340 Series:

1. Copy the starting address of CAN0 ISR to relocated CAN0 vector in RAM as reserved in preconditions→ step 2 above.
2. Copy the starting address of UART0 Receive ISR to relocated UART0 Receive vector in RAM as reserved in preconditions→ step 2 above.
3. Store the original TBR settings.
4. Configure the TBR to the new value value.
5. Call the function which erases the sector of Main Flash.
6. Restore the original TBR value.

The below sample code demonstrates the actual relocation of vector table.

```
void Main(void)
{
    unsigned int tbr;

    InitIrqLevels(); /* First initialize all Interrupt Levels */
    __set_il(7);     /* Set global Interrupt Level to 7       */
    __EI();          /* Enable Interrupt globally             */

    . . .

    CAN0_ptr = CAN0_ISR;          /* Store CAN0 ISR address to relocated vector*/
    UART0_ptr = UART0_Rx_ISR;     /* Store UART0-Rx ISR address to relocated
                                     vector */
    tbr = TBR;                    /* Save original TBR */
    TBR = 0x007C;                 /* Configure TBR to point to vector table in
                                     RAM at the base address 0x7C00 */
    EraseSector(0xF0);            /* Erase Falsh sector 0xF0 */
    TBR = tbr;                    /* Restore original TBR value */

        . . .
}
```

4 Additional Information

Information about Cypress Microcontrollers can be found on the following Internet page:

http://www.cypress.com/cypress-microcontrollers

The software example related to this application note is:

96340_intvect

It can be found on the following Internet page:

http://www.cypress.com/16lx

Document History

Document Title: AN205548 - F²MC-16FX Family, Interrupts

Document Number: 002-05548

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	MKEA	06/23/2006 12/28/2006 02/21/2007 08/10/2007 08/15/2007 07/08/2008	Initial Release Reviewed the document and updated with review findings Updated with re-review findings Restructure application note, add sections for priority, latency Corrected alignment of paragraphs Add information on PIER for NMI, update initialization of NMI
*A	5072990	MKEA	02/29/2016	Migrated Spansion Application Note from MCU-AN-300210-E-V15 to Cypress format
*B	5865629	AESATP12	08/31/2017	Updated logo and copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709