The following document contains information on Cypress products. The document has the series name, product name, and ordering part numbering with the prefix “MB”. However, Cypress will offer these products to new and existing customers with the series name, product name, and ordering part number with the prefix “CY”.

How to Check the Ordering Part Number

1. Go to www.cypress.com/pcn.
2. Enter the keyword (for example, ordering part number) in the **SEARCH PCNS** field and click **Apply**.
3. Click the corresponding title from the search results.
4. Download the Affected Parts List file, which has details of all changes

For More Information

Please contact your local sales office for additional information about Cypress products and solutions.

About Cypress

Cypress is the leader in advanced embedded system solutions for the world's most innovative automotive, industrial, smart home appliances, consumer electronics and medical products. Cypress' microcontrollers, analog ICs, wireless and USB-based connectivity solutions and reliable, high-performance memories help engineers design differentiated products and get them to market first. Cypress is committed to providing customers with the best support and development resources on the planet enabling them to disrupt markets by creating new product categories in record time. To learn more, go to www.cypress.com.

AN205263

F²MC-FR, MB91460, 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 Vector (no Channel Sharing)
- 16 Interrupt Priority Levels selectable (31 = Interrupts Disabled)
- Interrupt priority level shared between two peripherals
- Interrupt Vector Table Re-locatable
- Peripheral Interrupts
- Exceptions such as Undefined Instruction
- Traps such as INT instruction, Step Trace
- Non-Maskable Interrupt (NMI)
- DMA can be used with most of the interrupts

2. Interrupt Types

The basic functionality of internal interrupts

2.1 Available Exceptions / Interrupts / Traps

There are different types of Interrupts / exceptions / traps available.

2.1.1 Interrupts

2.1.1.1 Peripheral Interrupts

These kinds of Interrupts are generated by internal resources hence the may also referred to as "external" since they originate outside the CPU. 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 = 0). Interrupt level of peripheral interrupts is configured by ICRxx[1] (Interrupt Control Register).

It should be noted that one ICR configures (same) interrupt level for 2 peripheral resources. For more details please refer the hardware manual.

2.1.1.2 Non-Maskable Interrupts

Non-Maskable interrupts (NMI) are interrupts that cannot be masked by software**.** This is because it is not possible to write a value less than 16 using a software instruction.

NMI has a fixed interrupt level unlike peripheral interrupts. It is level 15. That means while execution of NMI ISR, the interrupt level mask is configured to 15 (PS:ILM = 15). Hence all the other interrupts (whose level is above 15) are suspended until NMI ISR execution has finished. System Stack Pointer is enabled (CCR:S = 0). The ILM is restored at execution of the RETI instruction.

2.1.2 Exceptions

Exceptions originate from within the instruction sequence. Exceptions are processed by first saving the necessary information to resume the currently executing instruction, and then starting the processing routine corresponding to the type of exception that has occurred.

2.1.2.1 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 14 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 interrupt flag is cleared (CCR:I = 0), hence all the all peripheral interrupts are suspended until undefined instruction ISR execution has finished. Undefined instruction ISR can not also be interrupted by NMI and Traps. The System Stack Pointer is enabled (CCR:S = 0).

2.1.3 Traps

Traps originate from within the instruction sequence. Traps are processed by first saving the necessary information to resume processing from the next instruction in the sequence, and then starting the processing routine corresponding to the type of trap that has occurred.

2.1.3.1 INT Instruction

“INT #u8” instruction branches to ISR indicated by interrupt vector u8. While execution of ISR global interrupt flag is cleared (CCR:I = 0), hence all the all peripheral interrupts are suspended until INT ISR execution has finished. However INT ISR can be interrupted by NMI and other Traps. The System Stack Pointer is enabled (CCR:S = 0).

2.1.3.2 INTE Instruction

The INTE instruction is used to create a software trap for debugging. The INTE instruction cannot be used in user programs involving debugging with an emulator.

[1] xx: 00 to 63

2.1.3.3 Step Trace Traps

Step trace traps are traps used by debuggers. This type of trap can be created for each individual instruction in a sequence by setting the T flag in the system condition code register (SCR) in the program status (PS). Step trace traps cannot be used in user programs involving debugging with an emulator.

2.1.3.4 Coprocessor Not Found

Coprocessor not found traps are generated when coprocessor instructions are executed such as COPOP/COPLD/COPST/COPSV while the coprocessor is absent. While execution of ISR, global interrupt flag is cleared (CCR:I = 0) hence all the all peripheral interrupts are suspended until INT ISR execution has finished. However this ISR can be interrupted by NMI and other Traps. The System Stack Pointer is enabled (CCR:S = 0).

2.1.3.5 Coprocessor Error

Coprocessor error trap is generated when an error has occurred in a coprocessor operation and the CPU executes another coprocessor instruction (such as COPOP/COPLD/COPST) involving the same coprocessor. While execution of ISR, global interrupt flag is cleared (CCR:I = 0) hence all the all peripheral interrupts are suspended until INT ISR execution has finished. However this ISR can be interrupted by NMI and other Traps. The System Stack Pointer is enabled (CCR:S = 0).

2.2 Direct Memory Access (DMA)

Most of the peripheral interrupts (except few peripheral interrupt such as CAN, I2C, External Interrupts etc.) can be used to initiate DMA transfers regardless of the status of the I flag and the interrupt level.

For detailed information please refer to the DMA application note MCU-AN-300059.

2.3 Interrupt Acceptance, Levels and Modes

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

Table 1. Interrupt Acceptance and Levels

Exception/ Interrupt/ Trap	Description	Interrupt/ Vector Number	Interrupt Level	Acceptance Condition	Action after Acceptance
Exception	Undefined Instruction	14	No Level	Always Accepted	Save CPU status to system stack S = 0 (use system stack) I = 0 i.e. all peripheral interrupts are suspended until Undefined Instruction ISR execution, Undefined Instruction ISR can not be interrupted Branch to interrupt vector
Trap	INTE Instruction	9	Always Fixed, 4	Always Accepted	Save CPU status to system stack S = 0 (use system stack) ILM = 4 i.e. all traps/interrupts/exceptions suspended until INTE ISR execution Branch to interrupt vector
Trap	INT Instruction	0 to 255, As specified by operand	No Level	Always Accepted	Save CPU status to system stack S = 0 (use system stack) I = 0 i.e. all peripheral interrupts are suspended until INT ISR execution, INT ISR can be interrupted by NMI, Undefined Instruction, Step Trace Trap and INTE Instruction Branch to interrupt vector

Exception/ Interrupt/ Trap	Description	Interrupt/ Vector Number	Interrupt Level	Acceptance Condition	Action after Acceptance
Interrupt	NMI	15	Always Fixed, 15	Current instruction execution is finished If ILM (Interrupt Level Mask) of PS (Processor Status) register is greater than 15	Save CPU status to system stack S = 0 (use system stack) ILM = 15 i.e. all peripheral interrupts are suspended until NMI ISR execution, NMI ISR can be interrupted by Undefined Instruction, Step Trace Trap and INTE Instruction Branch to interrupt vector
Trap	Step Trace Trap	12	Always Fixed, 4	Always Accepted	Save CPU status to system stack S = 0 (use system stack) ILM = 4 i.e. all traps/interrupts/exceptions suspended until Step Trace ISR execution Branch to interrupt vector
Trap	Coprocessor Exception	7 & 8	No Level		Save CPU status to system stack S = 0 (use system stack) I = 0 i.e. all peripheral interrupts are suspended until Coprocessor Exception ISR execution, Coprocessor ISR can be interrupted by NMI, Undefined Instruction, Step Trace Trap and INTE Instruction Branch to interrupt vector
Interrupt	User Interrupt	16onwards	As configured by the corresponding ICR (Interrupt Control Register), Between Level 16 (Highest) to 31 (Disabled)	Current instruction execution is finished String instruction is Interrupted If ILM of PS register is greater than interrupt level of the peripheral configured by ICR and I flag of CCR is 1 For multiple requests with same interrupt level, smallest interrupt number is accepted.	Save CPU status to system stack S = 0 (use system stack) ILM = ICR i.e. peripheral ISR can be interrupted by all EITs 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 4 then the cycles required for context saving/restoring are:

Table 2. Cycles Required for Context Saving / Restoring

Cycle Required	Address Indicated by Stack Pointer	
	4bytes alignment	2 Byte alignment
Context Saving	19 CLKB cycles	20 CLKB cycles
Context Restoring	9 CLKB cycles	9 CLKB cycles

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.

2.5 Registers

2.5.1 Processor Status (PS)

The Processor Status contains three sub sections.

Table 3. Processor Status

31 … 21	20 … 16	17 … 11	10 … 8	7 … 0
---	ILM	---	SCR	CCR

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

2.5.1.1 Interrupt Level Mask (ILM)

After reset the ILM is set to “01111”. It should be noted that when the original value of ILM is between 16 to 31 then the ILM can be only set with the values between 16 to 31 (both inclusive). In such case if the value is between 0 to 15 is attempted to be written, then the ILM would be set to the specified value + 16. Where as when the original value of ILM is between 0 to 15, then any value between 0 to 31 can be set.

Table 4. Interrupt Level Mask

Value of ILM	Remarks
0 to 14	ILM is set to this value in case of exceptions and traps.
15	In case of NMI, ILM is set to this value. All the peripheral interrupt are disabled
16 to 30	Peripheral interrupts enabled those have their ICR value less then ILM.
31[2]	All interrupts are enabled.

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*.

2.5.1.2 Condition Code Register (CCR)

The CCR consists of the following bits:

Table 5. Condition Code Register

Bit No.	Bit Name	Initial Value	Description
7	-	-	-
6	SV	-	Supervisor Mode Flag. 1 = User Mode, 0 = Supervisor Mode
5	S	0	System/User Stack Flag. 0 = System Stack, 1 = User Stack. This bit is set to “1” after in case of all EITs
4	I	0	Global Interrupt Enable Flag
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().

[2] Peripheral Interrupts disabled if the ICR is set to this value.

There is no direct bit access to the CCR in assembler, but bits can be set indirectly with logical instructions: Setting the I-Bit: ORCCR #10 and clearing it: ANDCCR #EF.

Wrong	Correct

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

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

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

2.5.1.3 System Condition Code Register (SCR)

The SCR consists of the following bits:

Table 6. System Condition Code Register

Bit No.	Bit Name	Initial Value	Description
10	D1	x	Step Division Flag. These bits hold intermediate data during the execution of step division.
9	D0	x	
8	T	0	Step Trace Trap Flag. Setting this bit enables step trace trap. Its used by the emulator.

2.5.2 Table Base Register (TBR)

The TBR contains the start address of the vector table to be used during EIT processing. Its initial value after reset is 0x000FFC00.

It is possible to relocate the interrupt vector table to the required address during run time by configuring the TBR accordingly. Both the mode and reset vector have a fixed addresses 0x000FFFF8 and 0x000FFFFC respectively. These addresses remains fix, although the TBR register is rewritten because its initial value after reset is 0x000FFC00.

It should be noted that the TBR register should not be assigned values greater than 0xFFFFFC00.

2.5.2.1 Interrupt Vector Table

The Interrupt Vector Table is a set of 256 words, which contains the 32-bit starting address of corresponding Interrupt Service Routines.

The vector address is calculated is as follows:

Vector address = TBR + Offset value = TBR + 0x03FC - 4 x Vector number

First 16 interrupts vectors are for special purpose EITs where as the later vectors are for peripheral interrupts. For more information please refer the hardware manual.

2.5.3 Interrupt Control Register (ICR00-63)

Using these registers interrupt levels of peripheral interrupts can be configured.

Table 7. Interrupt Control Register

Bit No.	Bit Name	Initial Value	Description
7-5	-	-	-
4	ICR4	1	This bit is read-only and the write to this bit is ignored. Since the power-on reset value of this bit is 1, if the value of 0x0F is attempted to be written to ICR register then the actual ICR value would become 0x10.
3-0	ICR3-0	1111	These bits along with ICR4 configure the interrupt level of the corresponding peripheral.

It should be noted that there is single such register for two peripherals, hence the interrupt level for such two peripherals are SAME. For example ICR00 configures the same interrupt level for External Interrupt 0 as well as External Interrupt 1. If a value of 31 is written to such register then the corresponding peripheral interrupt is disabled.

2.5.4 Hold Request Cancel Level Register (HRCL)

The DMA controller can request the CPU for the D-Bus by asserting D-Bus hold request (DHREQ), the CPU in turn would respond to this DMA request by D-Bus hold acknowledge (DHACK) granting the access of the D-Bus to the DMA.

In such situation NMI or peripheral interrupt can cancel the hold request (depending upon configuration of LVL bits if HRCL register) and gain the access of the bus while corresponding ISR executes. Upon execution of RETI instruction the DMA would gain access to the D-Bus again (provided the MHALTI flag and the corresponding peripheral interrupt flag is cleared in the ISR).

Bit No.	Bit Name	Initial Value	Description
7	MHALTI	0	This bit is set to 1 in case of Non-maskable Interrupt. It is also set to 1 in case of a peripheral interrupt with the higher priority than specified with LVL bits.
6-5	-	-	-
4	LVL4	1	These bits define the interrupt level at which a hold request cancel request is generated for the bus master. For e.g. if the LVL bits are configured as 20 (b'10100) then all the peripherals whose interrupt level is configured from 16 to 19 and the NMI can generate hold request cancel request. It should be noted that LVL4 bit is read-only. Since the power-on reset value of this bit is 1, if the value of 0x0F is attempted to be written to LVL bits then the actual LVL value would become 0x10.
4-0	LVL4-0	11111	

2.6 Non-Maskable Interrupt (NMI)

Non-Maskable interrupt is available in few derivatives. If the signal on the NMIX pin is at LOW level for 1 CLKP cycle then NMI is generated. In order to clear the NMI the MHALTI flag of HRCL register needs to be cleared after the level of the signal appearing on the NMIX pin changes to HIGH.

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 and do not split it.

```
/*                              SAMPLE CODE                                    */
/*-----------------------------------------------------------------------------*/

#pragma intvect DefaultIRQHandler 15    /* Non Maskable Interrupt       */
#pragma intvect My_IRQHandler_1   16    /* External Interrupt 0         */
#pragma intvect DefaultIRQHandler 17    /* External Interrupt 1         */
#pragma intvect My_IRQHandler_2   18    /* External Interrupt 2         */
#pragma intvect My_IRQHandler_3   19    /* External Interrupt 3         */
#pragma intvect DefaultIRQHandler 20    /* External Interrupt 4         */
#pragma intvect DefaultIRQHandler 21    /* External Interrupt 5         */
#pragma intvect My_IRQHandler_4   22    /* External Interrupt 6         */
#pragma intvect My_IRQHandler_5   23    /* External Interrupt 7         */
#pragma intvect My_IRQHandler_6   24    /* External Interrupt 8         */
#pragma intvect DefaultIRQHandler 25    /* External Interrupt 9
```

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 Setting an Interrupt Level

The following code snippet assigns the level 24 to reload timer 0 and 1.

```
/*                            SAMPLE CODE                                      */
/*-----------------------------------------------------------------------------*/
void InitIrqLevels(void)
{
 . . .

   ICR08 = 24;     /* Reload Timer 0               */
                       /* Reload Timer 1               */
        . . .

        . . .
    }
```

3.3 Reading an Interrupt Level

If the currently configured level of the peripheral can be read by reading out the corresponding ICR.

```
/*                          SAMPLE CODE                                      */
/*---------------------------------------------------------------------------*/

unsigned int icr_rlt01;

void readIrqLevel(void)
{
  icr_rlt01 = ICR;
}
```

3.4 Interrupt Service Routine

The type qualifier __interrupt notifies the complier that 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 flag should be always cleared in the interrupt 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 code shows a typical Interrupt service routine.

```
/*                          SAMPLE CODE                                      */
/*---------------------------------------------------------------------------*/

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

 /* do something here */
}
```

3.5 Interrupt Service Routine without Register Saving

If the application requires that after entering certain interrupt service routine if the stack should not be affected (by pushing CPU internal registers) then such ISR should be written in assembly.

3.6 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 2.5.1.1.

Assume the level 20 is desired. The following code shows how to access the ILM register from C code.

```
/*                          SAMPLE CODE                                      */
/*---------------------------------------------------------------------------*/

__set_il(20);
```

3.7 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.

Please also see 2.5.1.2.

```
/*                                  SAMPLE CODE                                  */
/*-------------------------------------------------------------------------------*/
    . . .
__EI();   /* Enable Interrupts globally */
    . . .
__DI();   /* Disable Interrupts globally */
    . . .
```

3.8 Order of Initialization

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

```
/*                                  SAMPLE CODE                                  */
/*-------------------------------------------------------------------------------*/
void InitIrqLevels(void)
{
    . . .

    ICR08 = 24;     /* Reload Timer 0               */
                        /* Reload Timer 1               */
    . . .

    . . .
}


    . . .

void main(void)
{
    InitIrqLevels(); /* First initialize all Interrupt Levels */
    __set_il(31);    /* Set global Interrupt Level to 31       */
    __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.

3.9 Interrupt Vector Relocation

As discussed in the section 2.5.2 , the interrupt vector table can be relocated to any word-aligned (32-bit) memory location in steps of 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 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.

Here it is demonstrated to relocate the CAN0 and UART0 Receive vectors and not the entire vector table but it is HIGHLY recommended that the entire vector table is relocated to RAM. This is because if in case any unhandled interrupt occurs then the software would crash.

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 Flash.
2. The space equivalent to the entire vector table or the required vectors is reserved in the memory other than Flash. For ease of understanding, we would consider that space in RAM would be reserved for the required vectors.

   a) The below code indicates that space for relocated CAN0 vector is reserved at RAM address 0x0003233C (vector number 48, offset from TBR = 0x33C) and space for relocated UART0-RX vector is reserved at the RAM address 0x00032324 (vector number 54, offset from TBR = 0x324). This is considering later the vector table is relocated to RAM starting from address 0x00032000.

```
/*                        SAMPLE CODE                                          */
/*-------------------------------------------------------------------------*/
#pragma segment DATA=INTVECT2_CAN,locate=0x0003233C
__interrupt void (*CAN0_ptr)(void);
#pragma segment DATA=DATA

#pragma segment DATA=INTVECT2_UART,locate=0x00032324
__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. For ease of understanding, we would consider that the routine is mapped to RAM.

   a) In order to achieve the above, section `IRAM` can be used (which is defined in the standard project template *91460_template_91467d*).

Figure 1. IRAM Section Setting

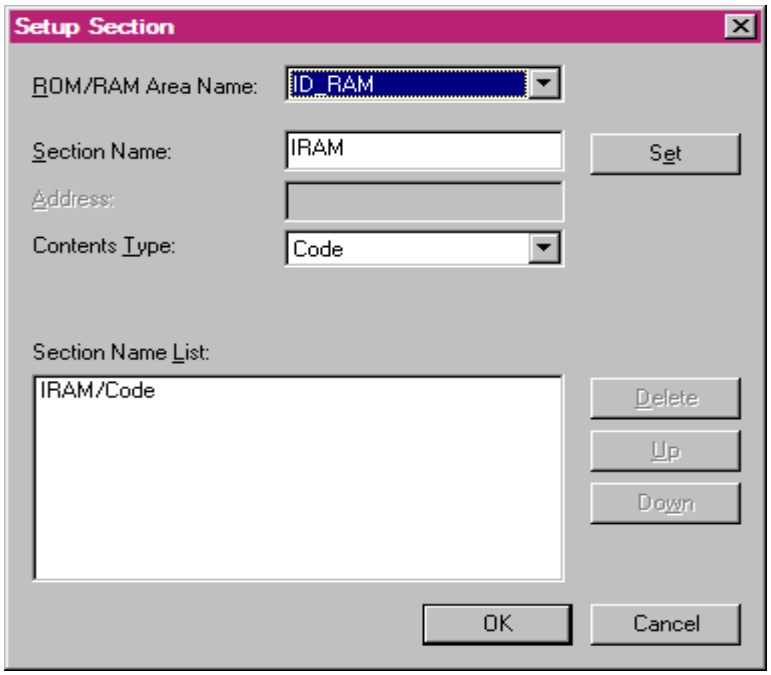
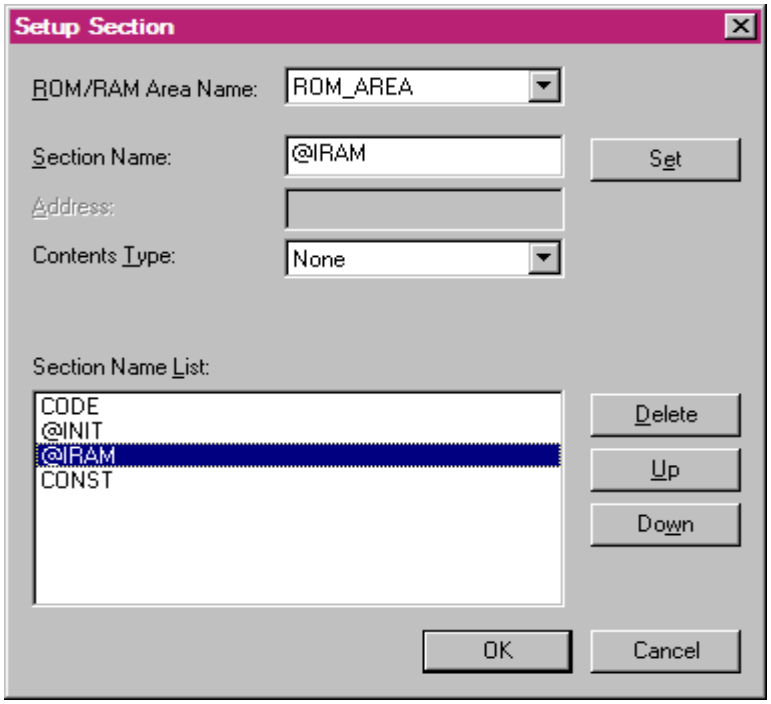


Figure 2. @IRAM Section Setting



The above dialog box can be reached as mentioned below…

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

b) In order to copy the section from Flash to RAM, `I_RAM` define in the *Start91460.asm* should be set to `ON`.

```
/*                                  SAMPLE CODE                                 */
/*------------------------------------------------------------------------------*/
;================================================================================
; 4.4  Copy code from Flash to I-RAM
;================================================================================
;
#set    I_RAM           ON       ; <<< select  if  code  in  section IRAM
;                                    should be copied
```

c) In order to link the function EraseSector() to RAM, the #pragma section directive needs to be used as follows:

```
/*                                SAMPLE CODE                                   */
/*------------------------------------------------------------------------------*/
#pragma section CODE=IRAM
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 code does the interrupt level configuration and vector configuration.

```
/*                            SAMPLE CODE                                       */
/*------------------------------------------------------------------------------*/
void InitIrqLevels(void)
{
. . .

ICR16 = 20;       /* CAN 0                          */
                      /* CAN 1                          */
ICR19 = 31;       /* USART (LIN) 0 RX               */
                         /* USART (LIN) 0 TX                */
. . .
}
   . . .

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

   . . .

#pragma intvect CAN0_ISR          48 /* CAN 0                    */
#pragma intvect UART0_Rx_ISR       54 /* USART (LIN) 0 RX         */
```

b) The following code links the ISRs to RAM.

```
/*                              SAMPLE CODE                                    */
/*-----------------------------------------------------------------------------*/
#pragma section CODE=IRAM
__interrupt void CAN0_ISR (void)
{
. . .

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

. . .
}
```

Once the above preconditions are met then the actual relocation would be carried out in the following steps for MB91460 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 code demonstrates the actual relocation of vector table

```
/*                                SAMPLE CODE                                  */
/*-----------------------------------------------------------------------------*/
unsigned int tbr;

void Main(void)
{
 InitIrqLevels(); /* First initialize all Interrupt Levels */
 __set_il(7);     /* Set global Interrupt Level to 7       */
 __EI();       /* Enable Interrupt globally             */
PORTEN = 0x3;       /* Enable I/O Ports */
 . . .

CAN0_ptr = CAN0_ISR;     /* Store CAN0 ISR address to relocated vector */
UART0_ptr = UART0_Rx_ISR;/* Store UART0Rx ISR address to relocated
                    vector */
__DI();                    /* Disable interrupts */
__asm(" STM0 (R7,R6) ");   /* Store R7 & R6 */
__asm(" MOV TBR, R7 ");  /* Save original TBR */
__asm(" LDI #_tbr, R6 ");
__asm(" ST R7, @R6 ");
__asm(" LDI #32000h, R7 ");/* Configure TBR to point to vector table in */
__asm(" MOV R7, TBR ");    /* RAM at the base address 0x00032000 */
__asm(" LDM0 (R7,R6) ");   /* Restore R7 & R6 */
__EI();                    /* enable interrupts */

EraseSector(0x09);             /* Erase Flash sector 0x09 */

__DI();                    /* Disable interrupts */
__asm(" STM0 (R6) ");      /* Store R6 */
__asm(" LDI #_tbr, R6 ");  /* Load tbr address to R6 */
__asm(" LD @R6, R6 ");   /* Restore original TBR value */
__asm(" MOV R6, TBR ");
__asm(" LDM0 (R6) ");      /* Restore R6 */
__EI();                    /* Enable interrupts */

           . . .
```

4. Additional Information

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

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

The software example related to this application note is:

91460_intvect

It can be found on the following Internet page:

http://www.cypress.com/cypress-mcu-product-softwareexamples

Document History

Document Title: AN205263 - F²MC-FR, MB91460, Interrupts

Document Number: 002-05263

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	NOFL	02/06/2008	First Version; MPi
*A	5085492	NOFL	04/13/2016	Converted Spansion Application Note “MCU-AN-300055-E-V10” to Cypress format
*B	5873469	AESATMP9	09/05/2017	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