Microblaze Interrupts and the EDK

The following is an excerpt from the Microblaze datasheet regarding interrupts.

Interrupt
MicroBlaze supports one external interrupt source (connected to the Interrupt input port). The processor only reacts to interrupts if the Interrupt Enable (IE) bit in the Machine Status Register (MSR) is set to 1. On an interrupt, the instruction in the execution stage completes while the instruction in the decode stage is replaced by a branch to the interrupt vector (address 0x10). The interrupt return address (the PC associated with the instruction in the decode stage at the time of the interrupt) is automatically loaded into general purpose register R14. In addition, the processor also disables future interrupts by clearing the IE bit in the MSR. The IE bit is automatically set again when executing the RTID instruction.

Interrupts are ignored by the processor if either of the break in progress (BIP) or exception in progress (EIP) bits in the MSR are set to 1.

Simply put, there is only one interrupt and thus, only one interrupt vector (0x10). Working in ‘C’, we need basically two things:

1. Some way to let the linker know to place the address of our ISR routine in the interrupt vector.
2. Some way for the compiler to know to compile our code as an ISR (with proper stack and register handling, and an RTID at the end).

This is done by simply with the following code:

```c
void myISR( void ) __attribute__((interrupt_handler));
```

The attribute fulfills both of the requirements stated above. Without it, the code compiles as follows:

```c
void myISR( void );

void myISR( void )
{
}
```

Compiles to:

```c
myISR:
 .frame r1,0,r15    # vars= 0, regs= 0, args= 0
 .mask 0x00000000
$LM2:
 .stabn 68,0,55,$LM2-myISR
 rtsd r15,8
 nop          # Unfilled delay slot
 .end myISR
```
With the attribute:

```c
void myISR( void ) __attribute__ ((interrupt_handler));

void myISR( void )
{
}
```

```assembly
myISR:
_interrupt_handler:
  .frame r1,20,r15  # vars= 0, regs= 3, args= 0
  .mask 0x00060800
  addik r1,r1,-20
  swi r15,r1,0
  swi r11,r1,8
  swi r17,r1,12
  mfs r11,rmsr  #mfs
  swi r18,r1,16
  swi r11,r1,4
$LM2:
  .stabn  68,0,55,$LM2-myISR
  lwi r15,r1,0
  lwi r11,r1,4
  mts rmsr,r11  #mts
  lwi r11,r1,8
  lwi r17,r1,12
  lwi r18,r1,16
  rtid r14,0
  addik r1,r1,20
.end _interrupt_handler
```

As can be seen, the attribute adds the appropriate code to handle an interrupt. Also added is the label `_interrupt_handler` which seems to be the default label that is loaded into the address at the interrupt vector (as ‘main’ is for the reset vector). This label can be changed by modifying the MSS file, but this adds unnecessary complexity to the process.

The EDK has an interrupt handler already in place (if you do not override it) that can be used. Again, this adds unnecessary complexity, but for completeness sake the code is included below.:

```c
/************************** Function Prototypes ******************************/
void __interrupt_handler () __attribute__ ((interrupt_handler));

/************************** Variable Definitions *****************************/
extern MB_InterruptVectorTableEntry MB_InterruptVectorTable;
/*****************************************************************************/
* This function is the standard interrupt handler used by the MicroBlaze processor.
* It saves all volatile registers, calls the users top level interrupt handler.
```
* When this returns, it restores all registers, and returns using a rtid instruction.
* The compiler saves all volatiles and the MSR */
void __interrupt_handler(void)
{
    /* The compiler saves all volatiles and the MSR */
    MB_InterruptVectorTable.Handler(MB_InterruptVectorTable.CallBackRef);
    /* The compiler restores all volatiles and MSR, and returns from interrupt */
}

******************************************************************************
******************************************************************************
******************************************************************************
******************************************************************************
******************************************************************************
******************************************************************************

/**
* Registers a top-level interrupt handler for the MicroBlaze. The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.
******************************************************************************
******************************************************************************
******************************************************************************
******************************************************************************
******************************************************************************
******************************************************************************

void microblaze_register_handler(XInterruptHandler Handler, void *DataPtr)
{
    MB_InterruptVectorTable.Handler = Handler;
    MB_InterruptVectorTable.CallBackRef = DataPtr;
}

The microblaze_register_handler() function is then required if the standard ISR is used.

Regardless of which ISR is used (the default microblaze or your own), interrupts have to enabled globally before any will be handled by the software:

    microblaze_enable_interrupts();

Using the Interrupt Controller for Multiple Interrupt Sources

This example uses interrupts from the following sources:
- opb_timer
- opb_gpio
- opb_uartlite (Attached in hardware, but unused in software)
- opb_spi

These are connected to the interrupt controller (opb_intc) which is connected to the microblaze as shown in the following diagram:
The ‘Intr’ line is used to make the multiple interrupt connections to the opb_intc. This can be seen in the following picture:
With this hardware configuration, the following code snippets are used to implement the design.

```c
// ISR Code****************************************************************
#define TIMER_INT XPAR_TIMER_INTERRUPT_MASK
#define BTN_INT XPAR_BUTTONS_IP2INTC_IRPT_MASK
#define SPI_INT XPAR_SPI_IP2INTC_IRPT_MASK
#define INTC_IPR (*((volatile unsigned long *)( XPAR_INT_CTRL_BASEADDR + 0x04)))
#define INTC_IER (*((volatile unsigned long *)( XPAR_INT_CTRL_BASEADDR + 0x08)))
#define INTC_IAR (*((volatile unsigned long *)( XPAR_INT_CTRL_BASEADDR + 0x1C)))
#define INTC_MER (*((volatile unsigned long *)( XPAR_INT_CTRL_BASEADDR + 0x1C)))

void myISR( void ) __attribute__ ((interrupt_handler));

void myISR( void )
{
    if( INTC_IPR & TIMER_INT )  // Timer Interrupt Is Pending
        timer_ISR();
    if( INTC_IPR & BTN_INT )  // Button interrupt is pending
        button_ISR();
    if( INTC_IPR & SPI_INT )  // SPI interrupt is pending
        spi_ISR();

    INTC_IAR = INTC_IPR;   // Acknowledge Interrupts
}

// *************************************************************************

// Timer Specific Code *****************************************************
#define TCSR0 (*((volatile unsigned long *)( XPAR_TIMER_BASEADDR + 0x00)))
#define TLR0 (*((volatile unsigned long *)( XPAR_TIMER_BASEADDR + 0x04)))

void timer_ISR( void )
{
    // Do Stuff Here
    TCSR0 = TCSR0;  // Acknowledge Interrupt In Timer (Clear pending bit)
}

// *************************************************************************

// GPIO (Connected to Buttons) Specific Code ******************************
#define BTNS (*((volatile unsigned long *)( XPAR_BUTTONS_BASEADDR)))
#define BTN_OE (*((volatile unsigned long *)( XPAR_BUTTONS_BASEADDR + 0x04)))
#define BTN_GIE (*((volatile unsigned long *)( XPAR_BUTTONS_BASEADDR + 0x11C)))
#define BTN_IER (*((volatile unsigned long *)( XPAR_BUTTONS_BASEADDR + 0x128)))
#define BTN_ISR (*((volatile unsigned long *)( XPAR_BUTTONS_BASEADDR + 0x120)))

void button_ISR( void )
{
    // Do Stuff Here
    BTN_ISR = BTN_ISR; // Clear any pending button interrupts
}

// *************************************************************************
```

// SPI Specific Code *******************************************************
#define SPI_GIE (*((volatile unsigned long *)( XPAR_SPI_BASEADDR+0x1C)))
#define SPI_ISR (*((volatile unsigned long *)( XPAR_SPI_BASEADDR+0x20)))
#define SPI_IER (*((volatile unsigned long *)( XPAR_SPI_BASEADDR+0x28)))

void spi_ISR( void )
{
    // Do Stuff Here
    SPI_ISR = SPI_ISR; // Clear pending interrupts
}

// ************************************************************************

void main( void )
{
    // ...
    TCSR0 = 0x000007F6;  // Timer Load and Clear any Pending Ints
    TCSR0 = 0x000007D6;  // Timer Clear Load Bit
    BTN_OE = ~0x00;  // Buttons are inputs
    BTN_IER = BTN_CHNL1;  // Enable Interrupts for all 3 buttons;
    BTN_GIE = ~0x00;  // Enable Interrupts for Button GPIO
    SPI_IER = SPI_TxEMPTY;  // Enable Interrupt for empty
    SPI_GIE = ~0x00000000;  // Global SPI Interrupt Enable

    // Enable Timer and Button Interrupt in IntC
    INTC_IER = TIMER_INT | BTN_INT | SPI_INT;
    INTC_MER = 0x03;  // Int Controller Master Enable
    microblaze_enable_interrupts();
    //...
}