



XAPP778 (v1.0) January 11, 2005

# Using and Creating Interrupt-Based Systems

Author: Paul Glover

## Summary

This application note describes how to properly set up external and internal interrupts in an embedded hardware system. Use of an interrupt controller to manage more than one interrupt is also included. The application note discusses the software use model, including initializing the interrupt controller and peripherals, registering the interrupt handlers, and enabling interrupts. Differences between the MicroBlaze™ and PowerPC™ processors are addressed.

## PowerPC Hardware Exceptions/Interrupts Management

### Definitions

#### Definition of Exceptions

*Exceptions* are events detected by the processor that often require action by system software. Most exceptions are unexpected and are the result of error conditions. A few exceptions can be programmed to occur through the use of exception-causing instructions. Some exceptions are generated by external devices and communicated to the processor using external signaling. Still other exceptions can occur when the processor recognizes pre-programmed conditions.

#### Definition of Interrupts

*Interrupts* are automatic control transfers that occur as a result of an exception. An interrupt occurs when the processor suspends execution of a program after detecting an exception. The processor saves the suspended-program machine state and a return address into the suspended program. This information is stored in a pair of special registers, called *save/restore registers*. A predefined machine state is loaded by the processor, which transfers control to an *interrupt handler*. An interrupt handler is a system-software routine that responds to the interrupt, often by correcting the condition causing the exception. System software places interrupt handlers at predefined addresses in physical memory and the interrupt mechanism automatically transfers control to the appropriate handler based on the exception condition.

Throughout this Application Note, the terms exception and interrupts can be used interchangeably.

#### Description of PowerPC Exceptions

[Table 1](#) lists the exceptions supported by the Virtex-II Pro and Virtex-4 FX PPC405. Included is the exception-vector offset into the interrupt-handler table, the exception classification, and a brief description of the cause. Gray-shaded rows indicate exceptions that are not supported by the Virtex-II Pro PPC405 but can occur on the Virtex-4 FX PowerPC processor. Refer to

© 2005 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and further disclaimers are as listed at <http://www.xilinx.com/legal.htm>. All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

NOTICE OF DISCLAIMER: Xilinx is providing this design, code, or information "as is." By providing the design, code, or information as one possible implementation of this feature, application, or standard, Xilinx makes no representation that this implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of the implementation, including but not limited to any warranties or representations that this implementation is free from claims of infringement and any implied warranties of merchantability or fitness for a particular purpose.

**Interrupt Reference**, page 206 of the **PowerPC Reference Manual**, for a detailed description of each exception and its resulting interrupt.

**Table 1: Virtex-II Pro and Virtex-4 FX PPC405 Exceptions**

Exception	Vector Offset	Classification	Cause
Critical Input	0x0100	Critical	External critical-interrupt signal
Machine Check	0x0200	Critical	External bus error
Data Storage	0x0300	Noncritical	Data-access violation
Instruction Storage	0x0400	Noncritical	Instruction-access violation
External	0x0500	Noncritical	External noncritical-interrupt signal
Alignment	0x0600	Noncritical	Unaligned operand of <b>dcread</b> , <b>lwarx</b> , <b>stwcx</b> . Use of <b>dcbz</b> to noncacheable or writethrough memory.
Program	0x0700	Noncritical	Improper or illegal instruction execution. Execution of trap instructions.
FPU Unavailable	0x0800	Noncritical	Attempt to execute an FPU instruction when FPU is disabled.
System Call	0x0C00	Noncritical	Execution of <b>sc</b> instruction.
APU Unavailable	0x0F20	Noncritical	Attempt to execute an APU instruction when APU is disabled.
Programmable-Interval Timer	0x1000	Noncritical	Time-out on the programmable interval timer.
Fixed-Interval Timer	0x1010	Noncritical	Time-out on the fixed-interval timer.
Watchdog Timer	0x1020	Critical	Time-out on the watchdog timer.
Data TLB Miss	0x1100	Noncritical	No data-page translation found.
Instruction TLB Miss	0x1200	Noncritical	No instruction-page translation found.
Debug	0x2000	Critical	Occurrence of a debug event.

### Simultaneous Exceptions and Interrupt Priority

The PPC405 interrupt mechanism responds to exceptions serially. If multiple exceptions are pending simultaneously, the associated interrupts occur in a consistent and predictable order. Even though critical and noncritical exceptions use different save/restore register pairs, simultaneous occurrences of these exceptions are also processed serially. The PPC405 uses the interrupt priority shown in PowerPC Reference Guide, Table 7-2, for handling simultaneous exceptions.

## Description of PPC External Interrupts

### *EICC405CRITINPUTIRQ (Input)*

When asserted, this signal indicates that the external interrupt controller (EIC) is requesting that the processor block respond to an external critical interrupt. The EIC could be a pin, peripheral, or interrupt controller. When deasserted, no request exists. The EIC is responsible for collecting critical interrupt requests from other peripherals and presenting them as a single request to the processor block. Once asserted, this signal remains asserted by the EIC until software deasserts the request (this is typically done by writing to a DCR in the EIC).

### *EICC405EXTINPUTIRQ (Input)*

When asserted, this signal indicates the EIC is requesting that the processor block respond to an external noncritical interrupt. When deasserted, no request exists. The EIC is responsible for collecting noncritical interrupt requests from other peripherals and presenting them as a single request to the processor block. Once asserted, this signal remains asserted by the EIC until software deasserts the request (this is typically done by writing to a device control register (DCR) in the EIC).

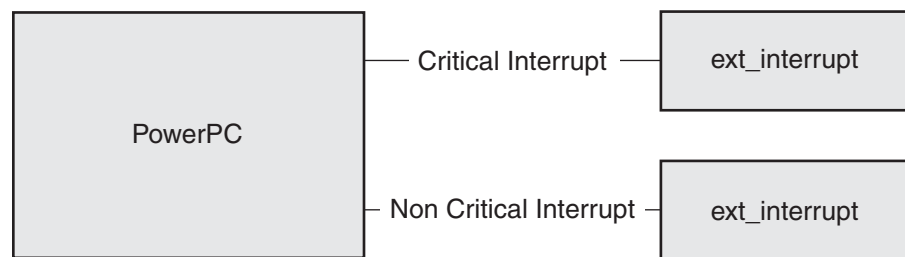
## Implementations

There are two types of interrupts in embedded processor systems: those generated outside the embedded system and those generated by a peripheral, which could be an interrupt controller, in the embedded system. The type of interrupt generated and the number of interrupts generated controls the configuration of the embedded system. The following sections describe two different interrupt configurations.

### Direct Interrupt Connection to the Embedded Processor

The PowerPC core supports two interrupt pins, EICC405CRITINPUTIRQ and EICC405EXTINPUTIRQ, which are active high.

Figure 1 illustrates the direct connection methodology for the PowerPC processor.



XAPP778\_01\_112404

Figure 1: Direct Connect Method for Interrupts

There are two methods to connect Interrupts in Xilinx Platform Studio (XPS). The first method illustrates the connections made in the MHS file. The second method utilizes the Add/Edit Hardware Platform Specifications dialog. This method will be illustrated in the Interrupt Controller section.

### ***MHS Examples***

To serve as an example of direct connections, the following MHS file examples illustrate:

1. External Interrupt connected directly to the PowerPC core.
2. Peripheral Interrupt connected directly to the PowerPC core.

When declaring an external interrupt there are two keywords used to define the interrupt. The first is the SIGIS = INTERRUPT keyword. This defines the port as an interrupt signal. The second is the SENSITIVITY keyword. This keyword is used to define the type of interrupt driving this port. There are four possible values for this keyword:

1. EDGE\_FALLING
2. EDGE\_RISING
3. LEVEL\_HIGH
4. LEVEL\_LOW

For additional information on these keywords, refer to the Platform Specification Format Reference Manual.

#### ***Example 1 - The MHS file specification used to connect an external interrupt to the PowerPC system.***

```
PORT ext_interrupt = ext_interrupt, DIR = INPUT, SIGIS = Interrupt,  
SENSITIVITY = LEVEL_HIGH
```

```
BEGIN ppc405  
  
    PARAMETER INSTANCE = ppc405_0  
    PARAMETER HW_VER = 2.00.c  
    BUS_INTERFACE JTAGPPC = jtagppc_0_0  
    BUS_INTERFACE IPLB = plb  
    BUS_INTERFACE DPLB = plb  
    PORT PLBCLK = sys_clk_s  
    PORT C405RSTCHIPRESETREQ = C405RSTCHIPRESETREQ  
    PORT C405RSTCORERESETREQ = C405RSTCORERESETREQ  
    PORT C405RSTSYSRESETREQ = C405RSTSYSRESETREQ  
    PORT RSTC405RESETCHIP = RSTC405RESETCHIP  
    PORT RSTC405RESETCORE = RSTC405RESETCORE  
    PORT RSTC405RESETSYS = RSTC405RESETSYS  
    PORT EICC405EXTINPUTIRQ = ext_interrupt  
    PORT CPMC405CLOCK = proc_clk_s  
  
END
```

***Example 2 - The MHS file specification used to connect a single internal peripheral to the PowerPC system.***

```

BEGIN opb_uartlite
    PARAMETER INSTANCE = RS232_Uart
    PARAMETER HW_VER = 1.00.b
    PARAMETER C_BAUDRATE = 115200
    PARAMETER C_BASEADDR = 0x7fffc500
    PARAMETER C_HIGHADDR = 0x7fffc5ff
    BUS_INTERFACE SOPB = opb
    PORT OPB_Clk = sys_clk_s
    PORT Interrupt = RS232_Uart_Interrupt
    PORT RX = fpga_0_RS232_Uart_RX
    PORT TX = fpga_0_RS232_Uart_TX
END

BEGIN ppc405
    PARAMETER INSTANCE = ppc405_0
    PARAMETER HW_VER = 2.00.c
    BUS_INTERFACE JTAGPPC = jtagppc_0_0
    BUS_INTERFACE IPLB = plb
    BUS_INTERFACE DPLB = plb
    PORT PLBCLK = sys_clk_s
    PORT C405RSTCHIPPRESETREQ = C405RSTCHIPPRESETREQ
    PORT C405RSTCORERESETREQ = C405RSTCORERESETREQ
    PORT C405RSTSYSRESETREQ = C405RSTSYSRESETREQ
    PORT RSTC405RESETCORE = RSTC405RESETCORE
    PORT RSTC405RESETCORE = RSTC405RESETCORE
    PORT RSTC405RESETCORE = RSTC405RESETCORE
    PORT RSTC405RESETCORE = RSTC405RESETCORE
    PORT RSTC405RESETCORE = RSTC405RESETCORE
    PORT EICC405EXTINPUTIRQ = RS232_Uart_Interrupt
    PORT CPMC405CLOCK = proc_clk_s
END

```

### **Usage of Interrupt Controller in the Embedded System**

Because the PowerPC core supports only two external interrupts, designs that require more interrupts must include an Interrupt Controller. Two different interrupt controllers are available in EDK, the difference being the bus interface to the controller. They are an OPB Interrupt Controller (OPB\_INTC) and a DCR Interrupt Controller (DCR\_INTC).

The OPB\_INTC connects to the OPB bus in the embedded system. In order to access this peripheral, the processor must utilize the PLB and OPB bus. The DCR\_INTC controller has a direct connection to the processor removing this overhead.

The PowerPC system interrupt ports are level sensitive only. If you have an edge-sensitive interrupt, an interrupt controller must be used. Figure 2 illustrates an OPB\_INTC-based PowerPC system. Figure 5 illustrates a DCR\_INTC-based PowerPC system.

**OPB INTC**

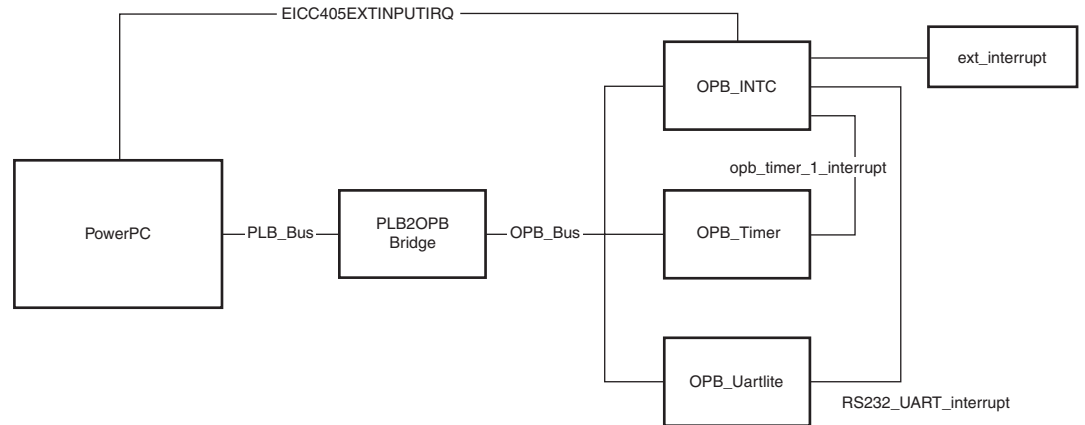
Interrupt controllers are used to expand the number of interrupt inputs a computer system has available to the CPU and provides a priority-encoding scheme.

Priority between interrupt requests is determined by vector position. The least significant bit (LSB, in this case bit 0) has the highest priority. This will be illustrated in the MHS example.

For addition information on the OPB\_INTC controller including control registers refer to the OPB\_INTC data sheet.

**MHS Example**

A MHS example of a PowerPC system using the OPB\_INTC.



XAPP778\_04\_112404

Figure 2: PowerPC with OPB INTC

### XPS - Add/Edit Hardware Platform Specifications

In the Add/Edit Hardware Platform Specifications dialog, select the Ports tab. In the Internal Ports Connection section, click the button in the net name field for the Intr port. This is illustrated in [Figure 3](#).

Instance	Port Name	Net Name	Pol...	Range
plb_bram_if_cntr_2	PLB_Clk	sys_clk_s	I	
opb_timer_1	OPB_Clk	sys_clk_s	I	
opb_timer_1	Interrupt	opb_timer_1_I...	O	
opb_intc_0	Irq	EICC405EXTIN...	O	
opb_intc_0	Intr	RS232... ..	I	[C_NU..
dcm_0	CLKIN	dcm_clk_s	I	
dcm_0	CLK0	sys_clk_s	O	
dcm_0	CLKFX	proc_clk_s	O	
dcm_0	CLKFB	sys_clk_s	I	

Figure 3: Add/Edit Hardware Platform Specifications

Selecting the “...” button, the Connect and Prioritize Interrupts dialog, allows the user to connect the interrupts, external or internal, to the interrupt controller. The interrupts can be prioritized by adjusting the order of the interrupts displayed in the dialog as shown in [Figure 4](#).

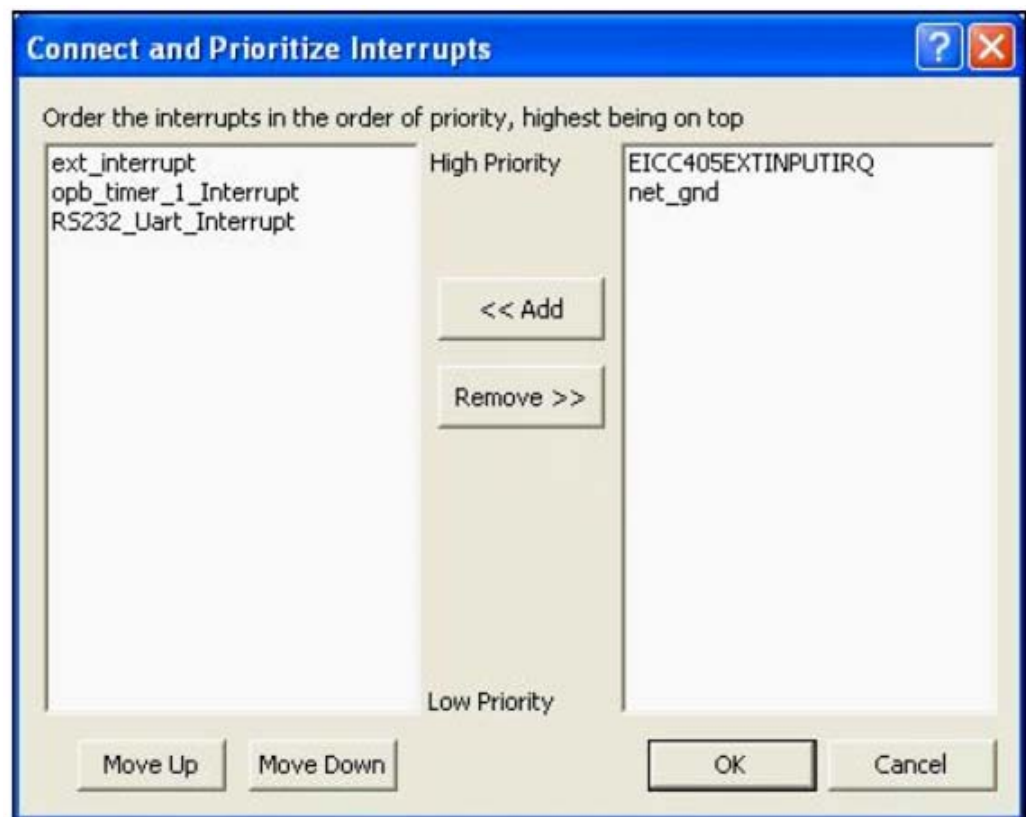


Figure 4: Connect and Prioritize Interrupts

**Example 1 - The MHS file specification used to connect multiple internal peripherals to the PowerPC core.**

```
PORT ext_interrupt = ext_interrupt, DIR = INPUT, SIGIS = Interrupt,  
SENSITIVITY = LEVEL_HIGH
```

```
BEGIN ppc405  
  
  PARAMETER INSTANCE = ppc405_0  
  PARAMETER HW_VER = 2.00.c  
  BUS_INTERFACE JTAGPPC = jtagppc_0_0  
  BUS_INTERFACE IPLB = plb  
  BUS_INTERFACE DPLB = plb  
  PORT PLBCLK = sys_clk_s  
  PORT C405RSTCHIPPRESETREQ = C405RSTCHIPPRESETREQ  
  PORT C405RSTCORERESETREQ = C405RSTCORERESETREQ  
  PORT C405RSTSYSRESETREQ = C405RSTSYSRESETREQ  
  PORT RSTC405RESETCCHIP = RSTC405RESETCCHIP  
  PORT RSTC405RESETCORE = RSTC405RESETCORE  
  PORT RSTC405RESETSYS = RSTC405RESETSYS  
  PORT EICC405EXTINPUTIRQ = EICC405EXTINPUTIRQ  
  PORT CPMC405CLOCK = proc_clk_s  
END
```

```
BEGIN opb_intc  
  
  PARAMETER INSTANCE = opb_intc_0  
  PARAMETER HW_VER = 1.00.c  
  PARAMETER C_BASEADDR = 0x7fffc600  
  PARAMETER C_HIGHADDR = 0x7fffc6ff  
  BUS_INTERFACE SOPB = opb  
  PORT Irq = EICC405EXTINPUTIRQ  
  PORT Intr = RS232_Uart_Interrupt & opb_timer_1_Interrupt & ext_interrupt  
END
```

```
BEGIN opb_uartlite  
  
  PARAMETER INSTANCE = RS232_Uart  
  PARAMETER HW_VER = 1.00.b  
  PARAMETER C_BAUDRATE = 115200  
  PARAMETER C_BASEADDR = 0x7fffc500  
  PARAMETER C_HIGHADDR = 0x7fffc5ff
```



```

BUS_INTERFACE SOPB = opb
PORT OPB_Clk = sys_clk_s

PORT Interrupt = RS232_Uart_Interrupt
PORT RX = fpga_0_RS232_Uart_RX
PORT TX = fpga_0_RS232_Uart_TX
END

```

```

BEGIN opb_timer
  PARAMETER INSTANCE = opb_timer_1
  PARAMETER HW_VER = 1.00.b
  PARAMETER C_COUNT_WIDTH = 32
  PARAMETER C_ONE_TIMER_ONLY = 0
  PARAMETER C_BASEADDR = 0x7fffc700
  PARAMETER C_HIGHADDR = 0x7fffc7ff
  BUS_INTERFACE SOPB = opb
  PORT OPB_Clk = sys_clk_s
  PORT Interrupt = opb_timer_1_Interrupt
END

```

The priority of the interrupts is defined by the connection order to the **Intr** port on the OPB\_INTC. As shown above, the Intr port is defined as follows:

**PORT Intr = RS232\_Uart\_Interrupt & opb\_timer\_1\_Interrupt & ext\_interrupt**

*Table 2: Interrupt priority*

Interrupt Name	Priority
ext_interrupt	1 Highest
opb_timer_1_Interrupt	2
RS232_Uart_Interrupt	3 Lowest

### **Limitations**

The OPB\_INTC supports up to 32 interrupt sources. For additional interrupts, OPB\_INTC controllers can be cascaded. It should be noted that most operating systems do not support cascaded interrupt controllers.

**DCR\_INTC**

Interrupt controllers are used to expand the number of interrupt inputs a computer system has available to the CPU and provides a priority-encoding scheme. Priority between interrupt requests is determined by vector position. The least significant bit (LSB, in this case bit 0) has the highest priority. This will be illustrated in the MHS example.

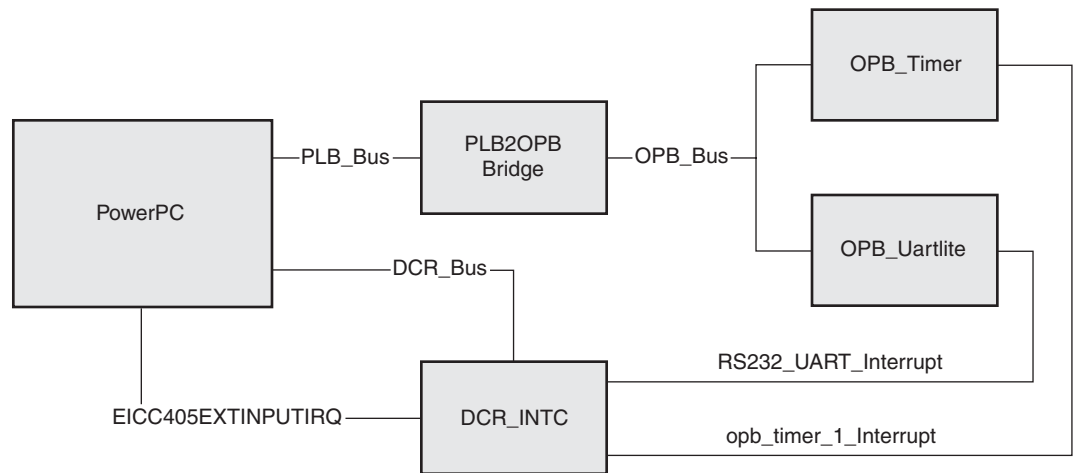
The DCR\_INTC has the following limitations:

1. Software must be run in “privilege mode.”
2. Requires the use of special “C” function headers or inline ASM.
3. Requires an additional 16KB of memory for data storage. This can be removed by customizing the BSP.

For addition information on the DCR\_INTC controller including control registers refer to the DCR\_INTC data sheet.

**MHS Example**

A MHS example of a PowerPC system using the DCR\_INTC.



XAPP778\_05\_112404

Figure 5: PowerPC System with DCR\_INTC

**Example 1- The MHS file specification used to connect multiple internal peripherals to the PowerPC system.**

```

BEGIN ppc405
    PARAMETER INSTANCE = ppc405_0
    PARAMETER HW_VER = 2.00.c
    PARAMETER C_DCR_RESYNC = 1
    BUS_INTERFACE JTAGPPC = jtagppc_0_0
    BUS_INTERFACE IPLB = plb
    BUS_INTERFACE DPLB = plb
    BUS_INTERFACE MDCR = dcr
    PORT PLBCLK = sys_clk_s
    PORT DCRCLK = sys_clk_s
    
```

```
PORT C405RSTCHIPRESETREQ = C405RSTCHIPRESETREQ
PORT C405RSTCORERESETREQ = C405RSTCORERESETREQ
PORT C405RSTSYSRESETREQ = C405RSTSYSRESETREQ
PORT RSTC405RESETCCHIP = RSTC405RESETCCHIP
PORT RSTC405RESETCORE = RSTC405RESETCORE
PORT RSTC405RESETSYS = RSTC405RESETSYS
PORT EICC405EXTINPUTIRQ = EICC405EXTINPUTIRQ
PORT CPMC405CLOCK = proc_clk_s
END

BEGIN dcr_intc
PARAMETER INSTANCE = dcr_intc_0
PARAMETER HW_VER = 1.00.b
PARAMETER C_BASEADDR = 0b0000000000
PARAMETER C_HIGHADDR = 0b0000000111
BUS_INTERFACE SDCR = dcr
PORT DCR_Clk = sys_clk_s
PORT DCR_Rst = sys_bus_reset
PORT Irq = EICC405EXTINPUTIRQ
PORT Intr = RS232_Uart_Interrupt & opb_timer_1_Interrupt
END

BEGIN opb_uartlite
PARAMETER INSTANCE = RS232_Uart
PARAMETER HW_VER = 1.00.b
PARAMETER C_BAUDRATE = 9600
PARAMETER C_DATA_BITS = 8
PARAMETER C_ODD_PARITY = 0
PARAMETER C_USE_PARITY = 0
PARAMETER C_CLK_FREQ = 10000000
PARAMETER C_BASEADDR = 0x80000500
PARAMETER C_HIGHADDR = 0x800005ff
BUS_INTERFACE SOPB = opb
PORT OPB_Clk = sys_clk_s
PORT Interrupt = RS232_Uart_Interrupt
PORT RX = fpga_0_RS232_Uart_RX
PORT TX = fpga_0_RS232_Uart_TX
END
```

```

BEGIN opb_timer

    PARAMETER INSTANCE = opb_timer_1
    PARAMETER HW_VER = 1.00.b
    PARAMETER C_COUNT_WIDTH = 32
    PARAMETER C_ONE_TIMER_ONLY = 0
    PARAMETER C_BASEADDR = 0x80000700
    PARAMETER C_HIGHADDR = 0x800007ff
    BUS_INTERFACE SOPB = opb
    PORT OPB_Clk = sys_clk_s
    PORT Interrupt = opb_timer_1_Interrupt

END
    
```

The priority of the interrupts is defined by the connection order to the **Intr** port on the DCR\_INTC. As shown above, the Intr port is defined as follows:

```
PORT Intr = RS232_Uart_Interrupt & opb_timer_1_Interrupt
```

*Table 3: Interrupt priority*

Interrupt Name	Priority
opb_timer_1_Interrupt	1 Highest
RS232_Uart_Interrupt	2 Lowest

### **Limitations**

Users should be aware that there is no DCR clock input to the processor block of the Virtex-II Pro and Virtex-II ProX devices. When dealing with signals that cross CPU clock domain and DCR clock domain, users may want to add re-synchronization flip-flops to simplify timing constraints, or set up appropriate multi-cycle/false path constraints in the UCF file.

An example for the re-synchronization of DCR interface can be found in the Xilinx Embedded Development Kit (EDK). Please refer to the Virtex-II Pro PowerPC405 wrapper IP in the "Processor IP Reference Guide" for details.

The Virtex-4-FX family does have a DCR clock input and does not have the synchronization issues mentioned here.

## PowerPC Software Exception/ Interrupt Management

A key component of Interrupt management is the software written to control the interrupts. This section will discuss the drivers provided by EDK for the processor cores, interrupt controller, and peripherals with interrupts.

### EDK Provided Drivers

EDK provides drivers for the OPB\_INTC, DCR\_INTC, and PowerPC cores. The drivers allow the software designer to enable interrupts, disable interrupts, register handlers, and unregister handlers. The details for each of these drivers will be discussed below.

#### INTC

There are two levels of abstraction in the INTC driver. Level “0” provides baseline functionality, and Level “1” provides a higher level of abstraction. Each of the driver levels will be discussed. The INTC driver provides multiple functions to utilize and control the OPB and DCR Interrupt controller. This application note will focus on the following Level “0” functions:

- `XIntc_DeviceInterruptHandler`
- `XIntc_mMasterEnable` (macro)
- `XIntc_mEnableIntr` (macro)
- `XIntc_SetIntrSvcOption`
- `XIntc_RegisterHandler`

For additional information regarding the Level “1” functions, refer to the `xintc.c` file or the Driver Functions documentation. The following section provides a brief description of each of these functions.

#### ***XIntc\_DeviceInterruptHandler(void)***

This function is the primary interrupt handler for the driver. It must be connected to the interrupt source such that is called when an interrupt of the interrupt controller is active. It will resolve which interrupts are active and enabled and call the appropriate interrupt handler. It uses the `AckBeforeService` flag in the configuration data to determine when to acknowledge the interrupt. Highest priority interrupts are serviced first. The driver can be configured to service only the highest priority interrupt or all pending interrupts using the Level 1 `XIntc_SetOptions()` function or the Level 0 `XIntc_SetIntrSrvOption()` function.

This function assumes that an interrupt vector table has been previously initialized. It does not verify that entries in the table are valid before calling an interrupt handler.

This function is declared in the `xintc_l.c` file.

#### ***XIntc\_mMasterEnable(Interrupt Controller Base Address)***

Enable all interrupts in the Master Enable register of the interrupt controller. The interrupt controller defaults to all interrupts disabled from reset such that this function must be used to enable interrupts.

This macro is declared in the `Xintc_l.h` file.

#### ***XIntc\_mEnableIntr (BaseAddress, EnableMask)***

Used to enable individual interrupts in the interrupt controller. `EnableMask` is a 32-bit value, where every bit in the mask corresponds to an interrupt input signal that is connected to the interrupt controller. Interrupt `INT0` corresponds to the LSB of `EnableMask`.

This macro is declared in the `Xintc_l.h` file.

#### ***XIntc\_SetIntrSvcOption(Xuint32 BaseAddress, int Option)***

This function is used to set the interrupt service option, which can configure the driver so that it services only a single interrupt at a time when an interrupt occurs, or services all pending

interrupts when an interrupt occurs. The default behavior when using a Level “1” driver is to service only a single interrupt, whereas the default behavior when using a Level “0” driver is to service all outstanding interrupts when an interrupt occurs.

The two options are `XIN_SVC_SGL_ISR_OPTION`, if you want only a single interrupt serviced when an interrupt occurs, or `XIN_SVC_ALL_ISRS_OPTION`, if you want all pending interrupts serviced when an interrupt occurs.

***void XIntc\_RegisterHandler(Xuint32 BaseAddress, int InterruptId, XInterruptHandler Handler, void \*CallbackRef);***

Register a handler function for a specific interrupt ID. The vector table of the interrupt controller is updated, overwriting any previous handler. The handler function will be called when an interrupt occurs for the given interrupt ID.

This function can also be used to remove a handler from the vector table by passing in the `XIntc_DefaultHandler()` as the handler and `XNULL` as the callback reference.

*Table 4: XIntc\_RegisterHandler function parameters*

<b>Function Parameters</b>	<b>Descriptions</b>
BaseAddress	The base address of the interrupt controller whose vector table will be modified.
InterruptId	The interrupt ID to be associated with the input handler. This ID can be found in the <code>xparameters.h</code> file written by LibGen.
Handler	The function pointer that will be added to the vector table for the given interrupt ID. It adheres to the <code>XInterruptHandler</code> signature found in <code>xbasic_types.h</code> .
CallbackRef	The argument that will be passed to the new handler function when it is called. This is user-specific.

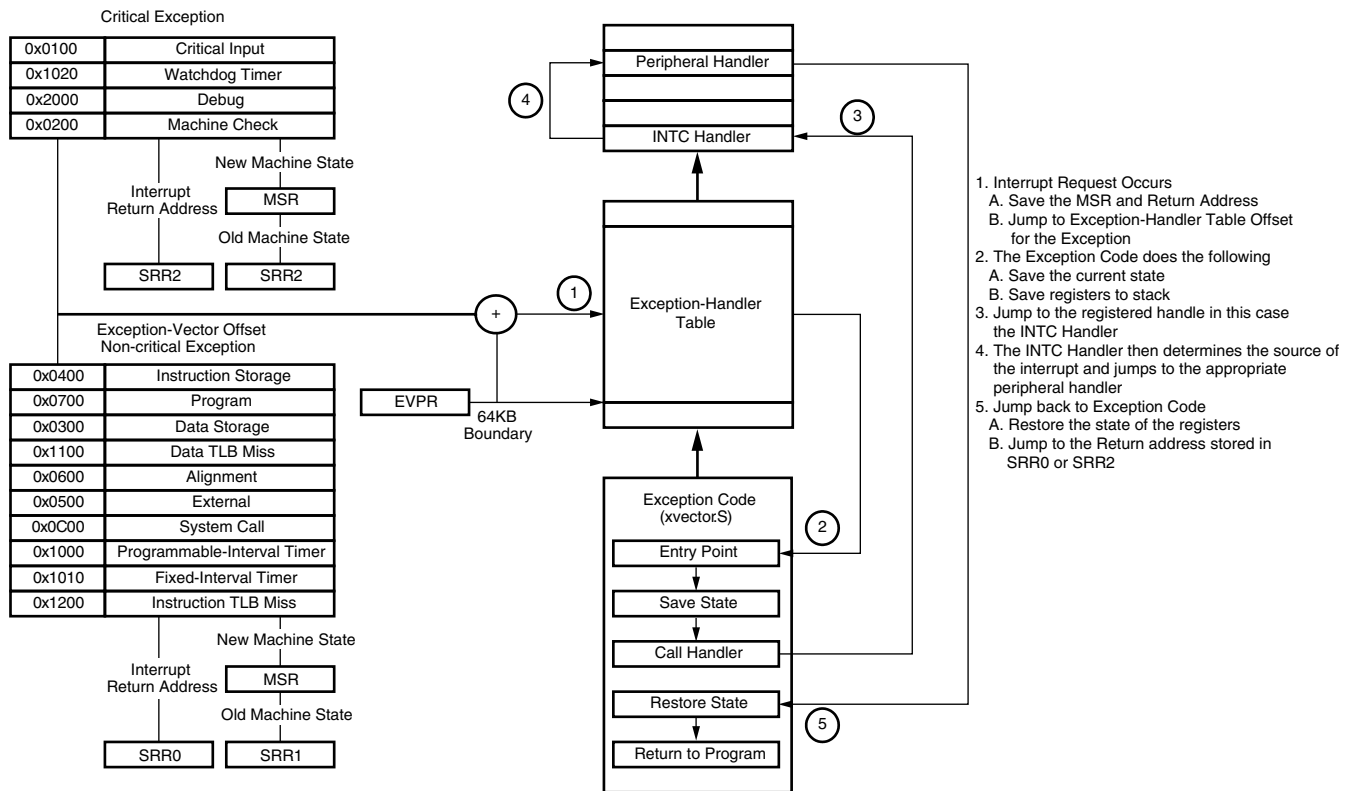
These functions can be utilized for both the `OPB_INTC` and the `DCR_INTC`.

### **PowerPC BSP**

The standalone BSP provided with the PowerPC processor provides several functions used to initialize exceptions/interrupts, register the exception/interrupt handlers, and enable/disable exceptions/interrupts. The following section will discuss these functions.

When an exception or interrupt occurs, the PowerPC system jumps to the Interrupt-Handler Table. The base address of the Interrupt Handler Table is contained in the `EVPR` register. Since the `EVPR` register is only 16 bits, the base address of the Interrupt-Handler Table must be defined on a 64KB boundary. The Exception/Interrupt Handler is registered in the Interrupt

Handler Table at the proper offset for the associated Exception/interrupt. Figure 6 illustrates this flow.



XAPP778\_06\_112404

Figure 6: PowerPC Interrupt Flow

## Functions used to manage PowerPC Exceptions

The PowerPC BSP provides the following functions to manage exceptions.

### ***void XExc\_Init(void)***

Initialize exception handling for the PowerPC system. The exception vector table is setup with the stub handler for all exceptions.

It also stores the baseaddress of the exception handling code (xvectors.S) in the EVPR register (Exception Vector Prefix Register). Uses inline assembly defined in "xpseudo\_asm.h."

### ***void XExc\_RegisterHandler(Xuint8 ExceptionId, XExceptionHandler Handler, void \*DataPtr)***

Makes the connection between the ID of the exception source and the associated handler that is to run when the exception is recognized. The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.

*Table 5: Exception IDs*

<b>Exception</b>	<b>Exception ID</b>
CRITICAL INPUT	XEXC_ID_CRITICAL_INT
MACHINE CHECK	XEXC_ID_MACHINE_CHECK
DATA STORAGE	XEXC_ID_DATA_STORAGE_INT
INSTRUCTION STORAGE	XEXC_ID_INSTRUCTION_STORAGE_INT
EXTERNAL	XEXC_ID_NON_CRITICAL_INT
ALIGNMENT	XEXC_ID_ALIGNMENT_INT
PROGRAM	XEXC_ID_PROGRAM_INT
SYSTEM CALL	XEXC_ID_SYSTEM_CALL
PROGRAMABLE-INTERVAL TIMER	XEXC_ID_PIT_INT
FIXED-INTERVAL TIMER	XEXC_ID_FIT_INT
WATCHDOG_TIMER	XEXC_ID_WATCHDOG_TIMER_INT
DATA_TLB_MISS	XEXC_ID_DATA_TLB_MISS_INT
INSTRUCTION_TLB_MISS	XEXC_ID_INSTRUCTION_TLB_MISS_INT
DEBUG	XEXC_ID_DEBUG_INT

***void XExc\_RemoveHandler(Xuint8 ExceptionId)***

Removes the handler for a specific exception ID. The stub handler is then registered for this exception ID. The stub handler has no functionality; it simply returns program access back to previously running code.

***XExc\_mEnableExceptions(EnableMask)***

Enables exceptions in the PowerPC processor.

***XExc\_mDisableExceptions(DisableMask)***

Disables exceptions in the PowerPC processor.



Table 6: Masks That Can be Specified When Enabling/Disabling Exceptions

Mask	Functionality
XEXC_CRITICAL	Enables all critical exceptions
XEXC_NON_CRITICAL	Enables all non-critical exceptions
XEXC_ALL	Enables all exceptions

## LibGen

With respect to interrupts, the Library Generator (LibGen) does two critical things:

1. Registers peripheral interrupt handlers
2. Generates the system include file `xparameters.h`. This file defines base addresses of the peripherals in the system, **#defines** needed by drivers, OS's, libraries and user programs, as well as function prototypes. This information is used in conjunction with the drivers.

LibGen is run in XPS by selecting **Tools -> Generate Libraries and BSPs**.

In order to simplify the registration of peripheral interrupt handlers, handler information can be placed in the MSS file. LibGen then registers these handlers removing the requirement to register the interrupt handlers in the application code. For example, if the embedded processor design contains an OPB\_UARTLITE peripheral with an interrupt, the handler is assigned in the MSS file as shown below:

```
BEGIN DRIVER

PARAMETER DRIVER_NAME = uartlite

PARAMETER DRIVER_VER = 1.00.b

PARAMETER HW_INSTANCE = RS232_Uart

PARAMETER int_handler = uart_int_handler, int_port = Interrupt

END
```

For external port interrupts, the following line must be included in the MSS file.

```
PARAMETER int_handler = ext_int_handler, int_port = ext_interrupt
```

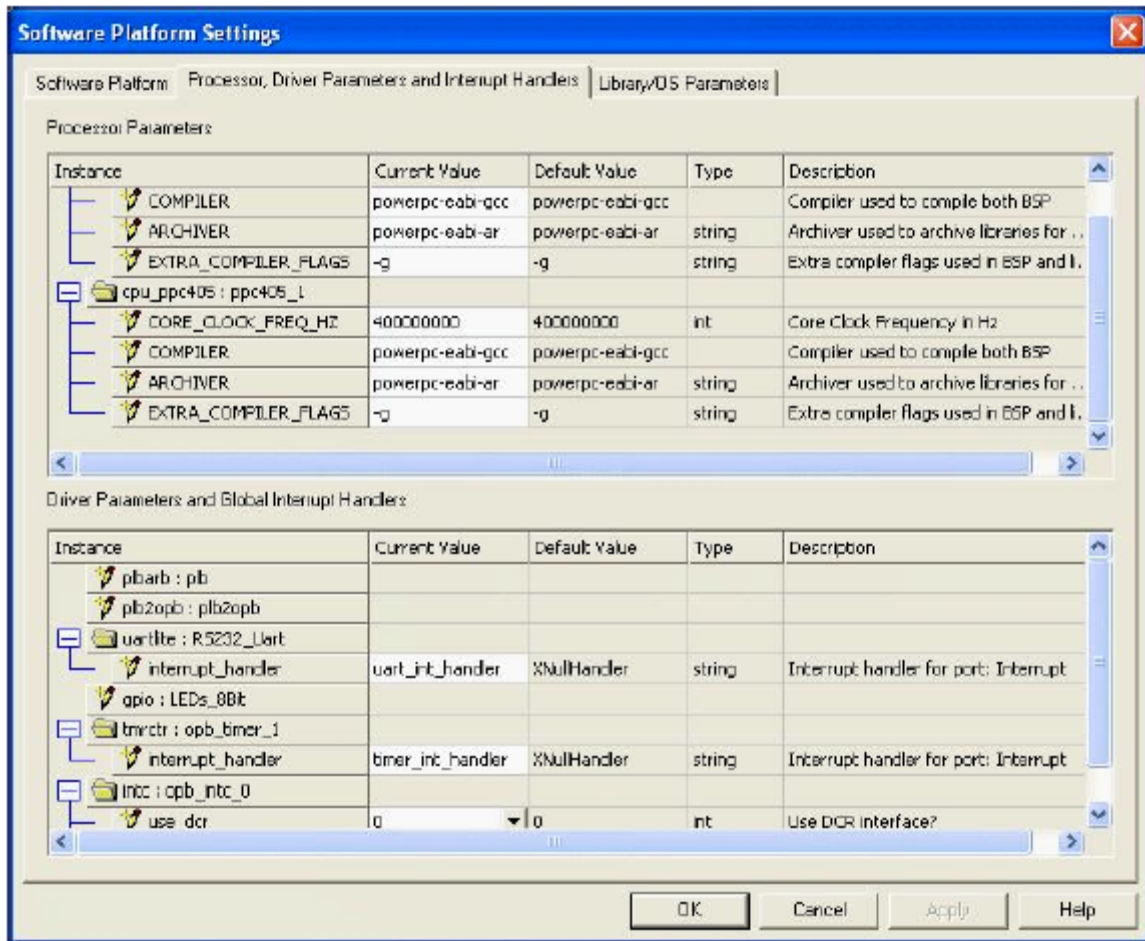


Figure 7: Software Platform Settings Dialog

If the OPB/DCR\_INTCC peripheral is included in the design, LibGen writes the xintc\_g.c file, which is used to register the handlers. The xintc\_g.c file can be found in the <project\_directory>\<ppc405\_0>\libsrc\intcc\_v1\_00\_c\src directory.

Where ppc405\_0 is the processor instance in the embedded processor design.

### XPS Registration of External Interrupts

XPS provides the ability to write these parameters to the MSS file via the Software Platform Settings. In XPS, select **Project -> Software Platform Settings** to open this dialog. Figure 7 illustrates Processor, Driver Parameters and Interrupt Handlers tab.

The Interrupt Handlers for each peripheral can be specified. This information is then written in the MSS file.

## Example Code

The following code snippet illustrates the main software loop used to correctly setup exceptions/interrupts for a PowerPC system design including an OPB\_INTC.

```
/* Initialize exception-handler table */
XExc_Init();

/* Register the default interrupt handler with the non-critical interrupt
pin */
XExc_RegisterHandler(XEXEC_ID_NON_CRITICAL_INT,
(XExceptionHandler)XIntc_DeviceInterruptHandler, (void *)0);

/* Register PIT Exception handler */
XExc_RegisterHandler(XEXEC_ID_PIT_INT,
(XExceptionHandler)pit_timer_int_handler, (void *)0);

/* Register UART interrupt handler */
XIntc_RegisterHandler(XPAR_OPB_INTC_0_BASEADDR,
XPAR_OPB_INTC_0_RS232_UART_INTERRUPT_INTR, uart_int_handler, (void *)0);

/* Initialise and enable the PIT timer */
XTime_PITSetInterval( 0xffffffff00 );
XTime_PITEnableAutoReload();

/* Enable the OPB_INTC to generate interrupts*/
XIntc_mMasterEnable(XPAR_OPB_INTC_0_BASEADDR);

/* Set the number of cycles the timer counts before interrupting */
XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_1_BASEADDR, 0,
(timer_count*timer_count+1) * 800000000);

XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_1_BASEADDR, 1,
(timer_count*timer_count+1) * 100000000);

/* reset the timers, and clear any currently pending interrupts */
XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK | XTC_CSR_AUTO_RELOAD_MASK
| XTC_CSR_DOWN_COUNT_MASK );
```

```

    XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 1,
    XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK | XTC_CSR_AUTO_RELOAD_MASK
    | XTC_CSR_DOWN_COUNT_MASK );

    /* Enable timer and uart interrupts in the interrupt controller */
    XIntc_mEnableIntr(XPAR_OPB_INTC_0_BASEADDR, XPAR_OPB_TIMER_1_INTERRUPT_MASK
    | XPAR_RS232_UART_INTERRUPT_MASK);

    /* Enable uart interrupts */
    XUartLite_mEnableIntr(XPAR_RS232_UART_BASEADDR);

    /* Enable pit interrupt */
    XTime_PITEnableInterrupt();

    /* Enable PPC non-critical interrupts */
    XExc_mEnableExceptions(XEXC_NON_CRITICAL);

```

The following code snippet illustrates the main software loop used to correctly setup exceptions/interrupts for a PowerPC system design including a DCR\_INTC. Changes from OPB\_INTC code are highlighted in bold.

```

    /* Initialize exception handling */
    XExc_Init();

    /* Register external interrupt handler */
    XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
    (XExceptionHandler)XIntc_DeviceInterruptHandler, (void *)0);

    /* Register UART interrupt handler */
    XIntc_RegisterHandler(XPAR_DCR_INTC_0_BASEADDR,
    XPAR_DCR_INTC_0_RS232_UART_INTERRUPT_INTR, uart_int_handler, (void *)0);

    /* Register PIT interrupt handler */
    XExc_RegisterHandler(XEXC_ID_PIT_INT,
    (XExceptionHandler)pit_timer_int_handler, (void *)0);

    /* Initialise and enable the PIT timer */
    XTime_PITSetInterval( 0xffffffff00 );
    XTime_PITEnableAutoReload();

```

```
/* Start the interrupt controller */
XIntc_mMasterEnable(XPAR_DCR_INTC_0_BASEADDR);

/* Set the gpio as output on high 4 bits (LEDs)*/
XGpio_mSetDataDirection(XPAR_LEDS_8BIT_BASEADDR, 1, 0x00);

/* Set the number of cycles the timer counts before interrupting */
XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_1_BASEADDR, 0,
(timer_count*timer_count+1) * 80000000);

XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_1_BASEADDR, 1,
(timer_count*timer_count+1) * 100000000);

/* reset the timers, and clear interrupts */
XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK );

XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 1,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK );

/* Enable timer and uart interrupts in the interrupt controller */
XIntc_mEnableIntr(XPAR_DCR_INTC_0_BASEADDR,
XPAR_OPB_TIMER_1_INTERRUPT_MASK | XPAR_RS232_UART_INTERRUPT_MASK);

/* Enable uart interrupts */
XUartLite_mEnableIntr(XPAR_RS232_UART_BASEADDR);

/* Enable pit interrupt */
XTime_PITEnableInterrupt();

/* Enable PPC non-critical interrupts */
XExc_mEnableExceptions(XEXC_NON_CRITICAL);
```

## MicroBlaze Hardware Exceptions/ Interrupts Management

### Description of MB Exceptions

All versions of the MicroBlaze processor support reset, interrupt, user exception and break. Starting with version 3.00a, the MicroBlaze processor can also be configured to support hardware exceptions. The following section describes the execution flow associated with each of these events. [Table 7](#) describes each of the exceptions and interrupts.

Table 7: Exception and Interrupt Descriptions

Exception	Description
Reset	When a Reset or Debug_Rst occurs, the MicroBlaze processor will flush the pipeline and start fetching instructions from the reset vector (address 0x0).
User Vector (Exception)	The user exception vector is located at address 0x8. Inserting a BRAILD instruction in the software application causes a user exception.
Interrupt	The MicroBlaze processor supports one external interrupt source (connecting to the <code>Interrupt</code> input port). The processor will only react to interrupts if the interrupt enable (IE) bit in the machine status register (MSR) is set to 1. The processor ignores interrupts if the break in progress (BIP) bit in the MSR register is set to 1.
Break: Non-maskable Hardware	<p>Hardware breaks are performed by asserting the external break signal (i.e. the <code>Ext_BRK</code> input ports). On a break the instruction in the execution stage will complete, while the instruction in the decode stage is replaced by a branch to the break vector (address 0x18).</p> <p>The break return address (the PC associated with the instruction in the decode stage at the time of the break) is automatically loaded into general-purpose register R16. The MicroBlaze processor also sets the Break In Progress (BIP) flag in the Machine Status Register (MSR).</p> <p>A nonmaskable break (i.e., the <code>Ext_NM_BRK</code> input port) will always be handled immediately.</p>
Break: Hardware Break	<p>Hardware breaks are performed by asserting the external break signal (i.e. the <code>Ext_BRK</code> input ports). On a break the instruction in the execution stage will complete, while the instruction in the decode stage is replaced by a branch to the break vector (address 0x18).</p> <p>The break return address (the PC associated with the instruction in the decode stage at the time of the break) is automatically loaded into general-purpose register R16. The MicroBlaze processor also sets the Break In Progress (BIP) flag in the Machine Status Register (MSR).</p> <p>A normal hardware break (i.e. the <code>Ext_BRK</code> input port) is only handled when there is no break in progress (i.e. <code>MSR[BIP]</code> is set to 0). The Break In Progress flag also disables interrupts and exceptions.</p>

Table 7: Exception and Interrupt Descriptions (Continued)

Exception	Description
Break: Software Break	To perform a software break, use the <code>brk</code> and <code>brki</code> instructions.
Hardware Exception	<p>A hardware exception in the MicroBlaze processor will flush the pipeline and branch to the hardware exception vector (address 0x20). The exception will also load the decode stage program counter value into the general-purpose register R17. The execution stage instruction in the exception cycle is not executed.</p> <p>The following will result in a hardware exception:</p> <ul style="list-style-type: none"> <li>◆ Unaligned data access</li> <li>◆ Illegal op-code</li> <li>◆ Instruction bus error</li> <li>◆ Data bus error</li> <li>◆ Divide by zero</li> </ul>

The relative priority of each of the exceptions and interrupts starting with the highest is:

1. Reset
2. Hardware Exception
3. Non-maskable Break
4. Break
5. Interrupt
6. User Vector (Exception)

Table 8 defines the memory address locations of the associated vectors and the hardware enforced register file locations for return address. Each vector allocates two addresses to allow full address range branching (requires an IMM followed by a BRAI instruction).

Table 8: Vectors and Return Address Register File Location

Event	Vector Address	Register File Return Address
Reset	0x00000000 - 0x00000004	-
User Vector (Exception)	0x00000008 - 0x0000000C	-
Interrupt	0x00000010 - 0x00000014	R14
Break: Non-maskable Hardware	0x00000018 - 0x0000001C	R16
Break: Hardware Break		
Break: Software Break		
Hardware Exception	0x00000020 - 0x00000024	R17

### Description of MB Interrupt

The MicroBlaze processor supports one external interrupt source via a connection to the *Interrupt* input port. The processor will only react 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 will complete, 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. In order for an Interrupt to interrupt the currently executing Interrupt, the interrupt handler code must re-enable interrupts. If an OPB\_INTC controller has been utilized, the INTC driver code must be modified to re-enable interrupts.

The processor ignores interrupts, if the break in progress (BIP) bit in the MSR register is set to 1.

The *Interrupt* input port can be configured to be Level or Edge sensitive using the parameters shown in [Table 9](#).

Table 9: MicroBlaze Interrupt Port Configuration Parameters

Parameter Name	Feature/Description	Allowable Values	Default Value	VHDL Type
C_INTERRUPT_IS_EDGE	Level/Edge Interrupt	0, 1	0	Integer
C_EDGE_IS_POSITIVE	Negative/Positive Edge Interrupt	0, 1	1	Integer

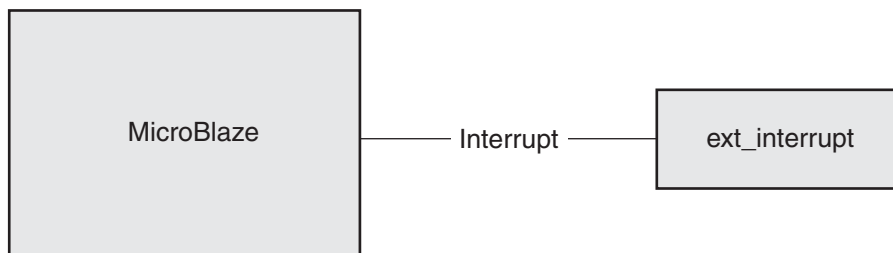
### Implementations

There are three types of interrupts in embedded processor systems, those generated outside the embedded system, those generated by a peripheral in the embedded system, and those generated by an Interrupt Controller included in the embedded system. The type of interrupt generated and the number of interrupts generated controls the configuration of the embedded system. The following sections describe two different interrupt configurations.

#### Direct Interrupt Connection to the Embedded Processor

The PowerPC core supports two interrupt pins, EICC405CRITINPUTIRQ and EICC405EXTINPUTIRQ, which are active high. The MicroBlaze processor INTERRUPT pin can be configured to match the interrupt source using parameters found on the MicroBlaze core.

[Figure 8](#) illustrates the direct connection methodology for the MicroBlaze processor.



XAPP778\_08\_112404

Figure 8: Direct Connect Method for Interrupts



There are two methods to connect Interrupts in XPS. The first method utilizes illustrates the MHS file connections. The second method illustrates the Add/Edit Hardware Platform Specifications dialog.

### ***MHS Examples***

To serve as an example of direct connections, the following MHS file examples illustrate:

1. External Interrupt connected directly to the MicroBlaze core.
2. Peripheral Interrupt connected directly to the MicroBlaze core.

When declaring an external interrupt there are two keywords used to define the interrupt. The first is the SIGIS = INTERRUPT keyword. This defines the port as an interrupt signal. The second is the SENSITIVITY keyword. This keyword is used to define the type of interrupt driving this port. There are four possible values for this keyword:

1. EDGE\_FALLING
2. EDGE\_RISING
3. LEVEL\_HIGH
4. LEVEL\_LOW

For additional information on these keywords, refer to the Platform Specification Format Reference Manual.

### ***Example 1 - The MHS file specification used to connect an external interrupt to the MicroBlaze system.***

```
PORT ext_interrupt = ext_interrupt, DIR = INPUT, SIDGIS = INTERRUPT,  
SENSITIVITY = LEVEL_HIGH
```

```
BEGIN microblaze  
  
  PARAMETER INSTANCE = microblaze_0  
  PARAMETER HW_VER = 3.00.a  
  PARAMETER C_DEBUG_ENABLED = 1  
  PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 1  
  PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 1  
  PARAMETER C_INTERRUPT_IS_EDGE = 0  
  PARAMETER C_EDGE_IS_POSITIVE = 1  
  BUS_INTERFACE DOPB = mb_opb  
  BUS_INTERFACE IOPB = mb_opb  
  BUS_INTERFACE DLMB = dlmb  
  BUS_INTERFACE ILMB = ilmb  
  PORT CLK = sys_clk_s  
  PORT DBG_CAPTURE = DBG_CAPTURE_s  
  PORT DBG_CLK = DBG_CLK_s  
  PORT DBG_REG_EN = DBG_REG_EN_s  
  PORT DBG_TDI = DBG_TDI_s  
  PORT DBG_TDO = DBG_TDO_s  
  PORT DBG_UPDATE = DBG_UPDATE_s
```

```
PORT Interrupt = ext_interrupt
END
```

**Example 2 - The MHS file specification used to connect a single internal peripheral to the MicroBlaze system.**

```
BEGIN opb_uartlite
  PARAMETER INSTANCE = RS232
  PARAMETER HW_VER = 1.00.b
  PARAMETER C_BAUDRATE = 115200
  PARAMETER C_USE_PARITY = 0
  PARAMETER C_CLK_FREQ = 50000000
  PARAMETER C_BASEADDR = 0x80200600
  PARAMETER C_HIGHADDR = 0x802006ff
  BUS_INTERFACE SOPB = mb_opb
  PORT OPB_Clk = sys_clk_s
  PORT Interrupt = RS232_Interrupt
  PORT RX = fpga_0_RS232_RX
  PORT TX = fpga_0_RS232_TX
END

BEGIN microblaze
  PARAMETER INSTANCE = microblaze_0
  PARAMETER HW_VER = 3.00.a
  PARAMETER C_DEBUG_ENABLED = 1
  PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 1
  PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 1
  BUS_INTERFACE DOPB = mb_opb
  BUS_INTERFACE IOPB = mb_opb
  BUS_INTERFACE DLMB = dlmb
  BUS_INTERFACE ILMB = ilmb
  PORT CLK = sys_clk_s
  PORT DBG_CAPTURE = DBG_CAPTURE_s
  PORT DBG_CLK = DBG_CLK_s
  PORT DBG_REG_EN = DBG_REG_EN_s
  PORT DBG_TDI = DBG_TDI_s
  PORT DBG_TDO = DBG_TDO_s
  PORT DBG_UPDATE = DBG_UPDATE_s
  PORT Interrupt = RS232_Interrupt
END
```

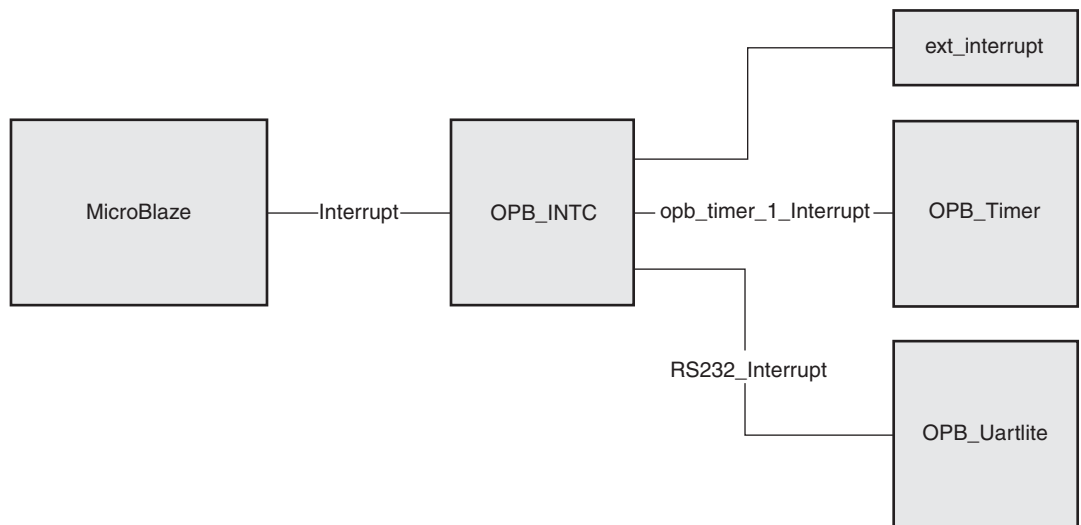
### Usage of Interrupt Controller in the Embedded System

Because the MicroBlaze core only supports one external interrupt, designs which require more than one interrupt, must include an OPB Interrupt Controller (OPB\_INTC). Figure 10 illustrates an OPB\_INTC-based MicroBlaze system.

#### OPB INTC

Interrupt controllers are used to expand the number of interrupt inputs a computer system has available to the CPU and provides a priority-encoding scheme. Priority between interrupt requests is determined by vector position. The least significant bit (LSB, in this case bit 0) has the highest priority. This will be illustrated in the MHS example.

For additional information on the OPB\_INTC controller, including control registers, refer to the OPB\_INTC data sheet.



XAPP778\_11\_112404

Figure 9: MicroBlaze Processor with OPB\_INTC

#### XPS - Add/Edit Hardware Platform Specifications

In the Add/Edit Hardware Platform Specifications dialog, select the Ports tab. In the Internal Ports Connection section, click the button in the net name field for the Intr port. This is illustrated in Figure 10.

Instance	Port Name	Net Name	Pol...	Range
SRAM_256Kx32	Mem_BEN	fpga_0_SRAM...	O	[0:C_...
opb_timer_1	OPB_Clk	sys_clk_s	I	
opb_timer_1	Interrupt	opb_timer_1_I...	O	
opb_intc_0	Irq	Interrupt	O	
opb_intc_0	Intr	RS232...	I	[C_NU..
SRAM_256Kx32_...	Sig	fpga_0_SRAM...	I	[0:C_...
SRAM_256Kx32_...	Out1	fpga_0_SRAM...	O	[C_LE...
mb_opb	SYS_Rst	sys_rst_s	I	
mb_opb	OPB_Clk	sys_clk_s	I	

Figure 10: Add/Edit Hardware Platform Specifications

Selecting the “...” button, the Connect and Prioritize Interrupts dialog, allows the user to connect the interrupts, external or internal, to the interrupt controller. The interrupts can be prioritized by adjusting the order of the interrupts displayed in the dialog as shown in Figure 11.

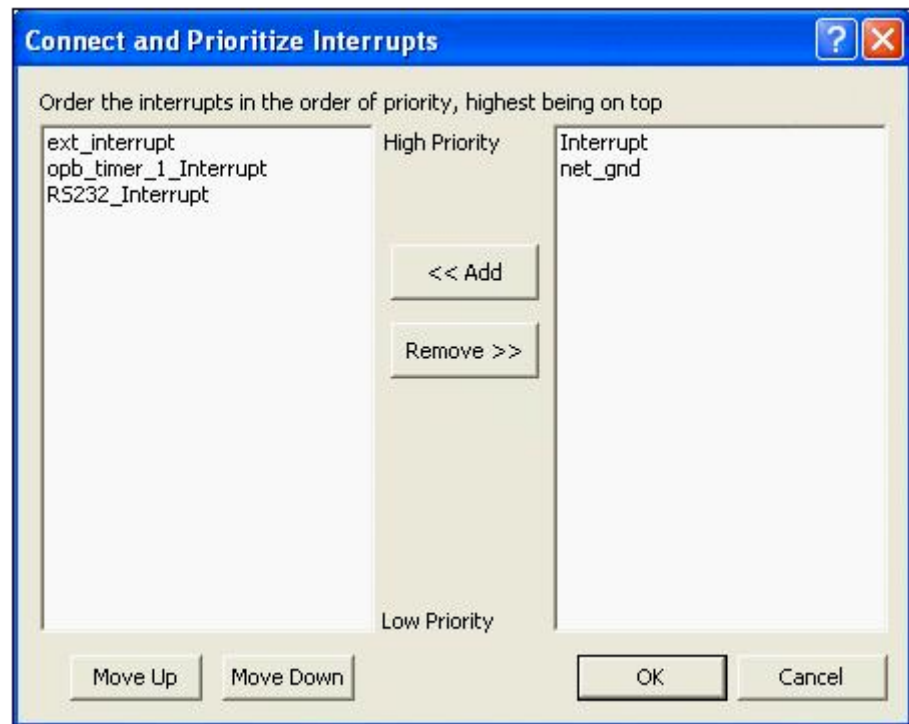


Figure 11: Connect and Prioritize Interrupts

### MHS Examples

The following MHS example illustrates a MicroBlaze system with OPB\_INTC

**Example 1- The MHS file specification used to connect multiple internal peripherals to the MicroBlaze system.**

```
BEGIN microblaze
  PARAMETER INSTANCE = microblaze_0
  PARAMETER HW_VER = 3.00.a
  PARAMETER C_DEBUG_ENABLED = 1
  PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 1
  PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 1
  BUS_INTERFACE DOPB = mb_opb
  BUS_INTERFACE IOPB = mb_opb
  BUS_INTERFACE DLMB = dlmb
  BUS_INTERFACE ILMB = ilmb
  PORT CLK = sys_clk_s
  PORT DBG_CAPTURE = DBG_CAPTURE_s
  PORT DBG_CLK = DBG_CLK_s
```

```
PORT DBG_REG_EN = DBG_REG_EN_s
PORT DBG_TDI = DBG_TDI_s
PORT DBG_TDO = DBG_TDO_s
PORT DBG_UPDATE = DBG_UPDATE_s
PORT Interrupt = Interrupt
END

BEGIN opb_intc
PARAMETER INSTANCE = opb_intc_0
PARAMETER HW_VER = 1.00.c
PARAMETER C_BASEADDR = 0x80200200
PARAMETER C_HIGHADDR = 0x802002ff
BUS_INTERFACE SOPB = mb_opb
PORT Irq = Interrupt
PORT Intr = RS232_Interrupt & opb_timer_1_Interrupt & ext_interrupt
END

BEGIN opb_timer
PARAMETER INSTANCE = opb_timer_1
PARAMETER HW_VER = 1.00.b
PARAMETER C_COUNT_WIDTH = 32
PARAMETER C_ONE_TIMER_ONLY = 0
PARAMETER C_BASEADDR = 0x80200000
PARAMETER C_HIGHADDR = 0x802000ff
BUS_INTERFACE SOPB = mb_opb
PORT OPB_Clk = sys_clk_s
PORT Interrupt = opb_timer_1_Interrupt
END

BEGIN opb_uartlite
PARAMETER INSTANCE = RS232
PARAMETER HW_VER = 1.00.b
PARAMETER C_BAUDRATE = 115200
PARAMETER C_DATA_BITS = 8
PARAMETER C_ODD_PARITY = 0
PARAMETER C_USE_PARITY = 0
PARAMETER C_CLK_FREQ = 50000000
PARAMETER C_BASEADDR = 0x80200600
```

```

PARAMETER C_HIGHADDR = 0x802006ff
BUS_INTERFACE SOPB = mb_opb
PORT OPB_Clk = sys_clk_s
PORT Interrupt = RS232_Interrupt
PORT RX = fpga_0_RS232_RX
PORT TX = fpga_0_RS232_TX

END
    
```

The priority of the interrupts is defined by the connection order to the **Intr** port on the OPB\_INTC. Table 10 illustrates the priority of these interrupts. The Intr port is defined as follows:

***PORT Intr = RS232\_Interrupt & opb\_timer\_1\_Interrupt & ext\_interrupt***

Table 10: MicroBlaze System Interrupt Priority

Interrupt Name	Priority
ext_interrupt	1 Highest
opb_timer_1_Interrupt	2
RS232_Interrupt	3 Lowest

## MicroBlaze Software Exception/ Interrupt Management

A key component of Interrupt management is the software written to control the interrupts. This section will discuss the drivers provided by EDK for the processor cores, interrupt controller, and peripherals with interrupts.

### EDK Provided Drivers

EDK provides drivers for the OPB\_INTC, MicroBlaze cores. The drivers allow the software designer to enable interrupts, disable interrupts, register handlers, and unregister handlers. The details for each of these drivers will be discussed below.

### INTC

There are two levels of abstraction in the INTC driver. Level “0” provides baseline functionality, while Level “1” provides a higher level of abstraction. Each of the driver levels will be discussed. The INTC driver provides multiple functions to utilize and control the OPB Interrupt controller. This application note will focus on the following Level “0” functions:

- XIntc\_DeviceInterruptHandler
- XIntc\_mMasterEnable
- XIntc\_mEnableIntr
- XIntc\_SetIntrSvcOption

For additional information regarding the Level “1” functions, refer to the xintc.c file or the Driver Functions documentation. The following section provides a brief description of each of these functions.

***XIntc\_DeviceInterruptHandler(void)***

This function is the primary interrupt handler for the driver. It must be connected to the interrupt source such that is called when an interrupt of the interrupt controller is active. It will resolve which interrupts are active and enabled and call the appropriate interrupt handler. It uses the AckBeforeService flag in the configuration data to determine when to acknowledge the interrupt. Highest priority interrupts are serviced first. The driver can be configured to service only the highest priority interrupt or all pending interrupts using the Level 1 `XIntc_SetOptions()` function or the Level 0 `XIntc_SetIntrSrvOption()` function.

This function assumes that an interrupt vector table has been previously initialized. It does not verify that entries in the table are valid before calling an interrupt handler. This function is declared in the `xintc_l.c` file.

***XIntc\_mMasterEnable(Interrupt Controller Base Address)***

Enable all interrupts in the Master Enable register of the interrupt controller. The interrupt controller defaults to all interrupts disabled from reset such that this function must be used to enable interrupts. This macro is declared in the `Xintc_l.h` file.

***XIntc\_mEnableIntr (BaseAddress, EnableMask)***

Used to enable individual interrupts in the interrupt controller. `EnableMask` is a 32-bit value, where every bit in the mask corresponds to an interrupt input signal that is connected to the interrupt controller. Interrupt INT0 corresponds to the LSB of `EnableMask`. This macro is declared in the `Xintc_l.h` file.

***XIntc\_SetIntrSvcOption(Xuint32 BaseAddress, int Option)***

This function is used to set the interrupt service option, which can configure the driver so that it services only a single interrupt at a time when an interrupt occurs, or services all pending interrupts when an interrupt occurs. The default behavior when using a Level "1" driver is to service only a single interrupt, whereas the default behavior when using a Level "0" driver is to service all outstanding interrupts when an interrupt occurs.

The two options are `XIN_SVC_SGL_ISR_OPTION`, if you want only a single interrupt serviced when an interrupt occurs, or `XIN_SVC_ALL_ISRS_OPTION`, if you want all pending interrupts serviced when an interrupt occurs.

***XIntc\_RegisterHandler(Xuint32 BaseAddress, int InterruptId, XInterruptHandler Handler, void \*CallbackRef);***

Register a handler function for a specific interrupt ID. The vector table of the interrupt controller is updated, overwriting any previous handler. The handler function will be called when an interrupt occurs for the given interrupt ID.

This function can also be used to remove a handler from the vector table by passing in the `XIntc_DefaultHandler()` as the handler and `XNULL` as the callback reference.

**Table 11: XIntc\_RegisterHandler function parameter descriptions**

Function Parameters	Descriptions
BaseAddress	The base address of the interrupt controller whose vector table will be modified.
InterruptId	The interrupt ID to be associated with the input handler. This ID can be found in the <code>xparameters.h</code> file written by LibGen.
Handler	The function pointer that will be added to the vector table for the given interrupt ID. It adheres to the <code>XInterruptHandler</code> signature found in <code>xbasic_types.h</code> .
CallBackRef	The argument that will be passed to the new handler function when it is called. This is user-specific.

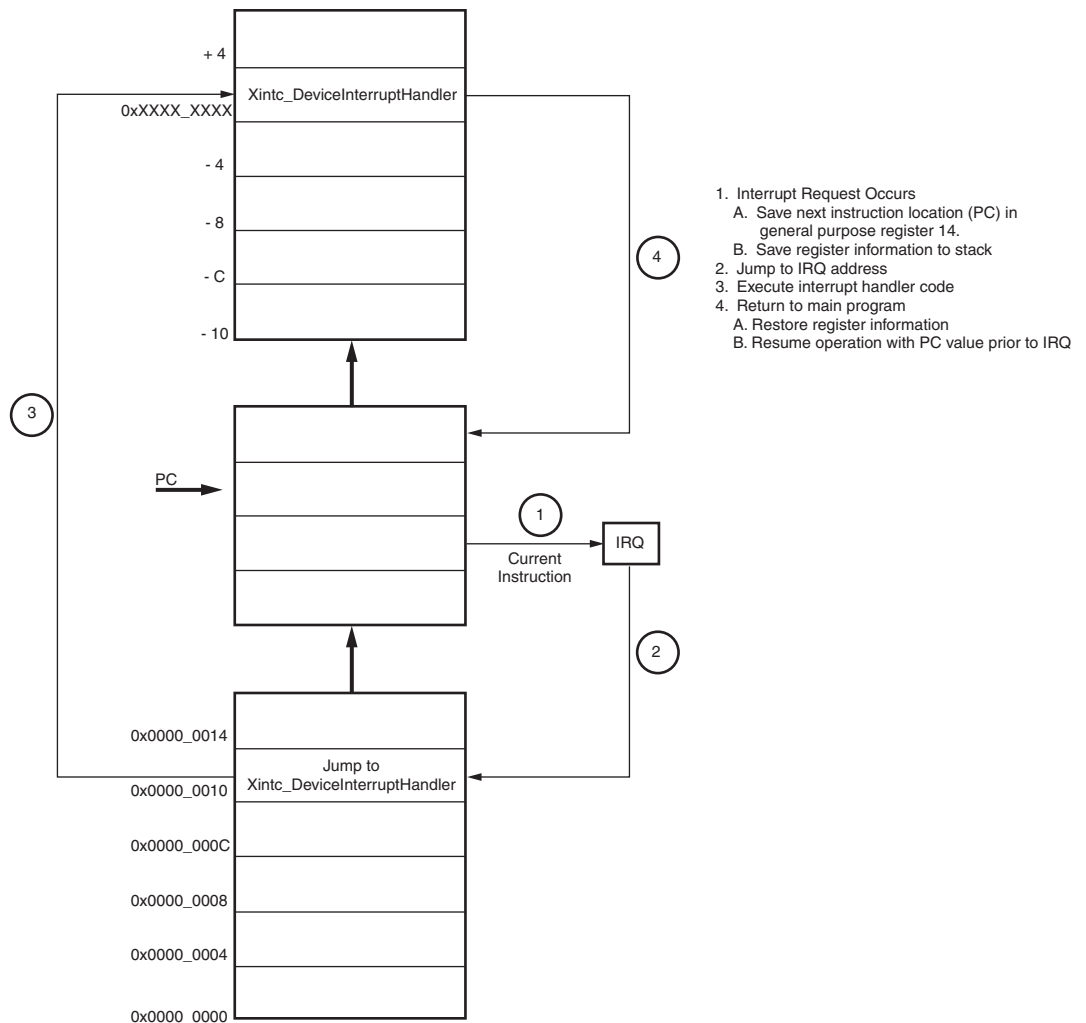
These functions can be utilized for the `OPB_INTC`.

### MicroBlaze BSP

The standalone BSP provided with the MicroBlaze system provides several functions used to initialize exceptions/interrupts, register the exception/interrupt handlers, and enable/disable exceptions/interrupts. The following section will discuss these functions.

When an interrupt occurs, the MicroBlaze system jumps to address `0x10`. The branch instruction written to address `0x10`, jumps to the interrupt handler for the interrupt connected to the MicroBlaze interrupt port. For example, if the `OPB_INTC` interrupt is connected to the MicroBlaze interrupt port, the address specified for the branch instruction points to the default `OPB_INTC` handler, **`XIntc_DeviceInterruptHandler`**. This process is illustrated in [Figure 11](#). The process is similar for exceptions, except the MicroBlaze processor jumps to a specific address based on the exception taken. [Table 1](#) contains the exceptions and the associated jump address.





XAPP778\_12\_112404

Figure 12: MicroBlaze Interrupt Flow

### Functions used to manage MicroBlaze Interrupts

#### ***void microblaze\_enable\_interrupts(void)***

This function enables interrupts on the MicroBlaze system. When the MicroBlaze processor starts up, interrupts are disabled. Interrupts must be explicitly turned on using this function.

#### ***void microblaze\_disable\_interrupts(void)***

This function disables interrupts on the MicroBlaze processor. This function may be called when entering a critical section of code where a context switch is undesirable.

#### ***void microblaze\_register\_handler(XInterruptHandler Handler, void \*DataPtr)***

This function allows one to register the interrupt handler for the interrupt on the MicroBlaze processor. This handler will be invoked by the first level interrupt handler that is present in the BSP. The first level interrupt handler takes care of saving and restoring registers, as necessary for interrupt handling and hence the function that you register with this handler can concentrate on the other aspects of interrupt handling, without worrying about saving registers.

## Functions used to manage MicroBlaze Exceptions

This feature and hence the corresponding interfaces are available only on the MicroBlaze v3.00.a processor.

The handlers for the various exceptions can be specified in the parameter *exception\_vectors*, in the XPS software platform settings GUI. Specify the name of the routine that you wish to handle a particular exception, in the field corresponding to the exception type.

### ***void microblaze\_disable\_exceptions(void)***

Disable hardware exceptions from the MicroBlaze processor. This routine clears the appropriate "exceptions enable" bit in the MSR of the processor.

### ***void microblaze\_enable\_exceptions(void)***

Enable hardware exceptions from the MicroBlaze processor. This routine sets the appropriate "exceptions enable" bit in the MSR of the processor.

### ***void microblaze\_register\_exception\_handler (Xuint8 ExceptionId, XExceptionHandler Handler, void \*DataPtr)***

Register a handler for the specified exception type. *Handler* is the function that handles the specified exception. *DataPtr* is a callback data value that is passed to the exception handler at run-time. The valid exception IDs are defined in *microblaze\_exceptions\_i.h*.

Table 12: *microblaze\_register\_exception\_handler* exception ID table explanations

Exception ID	Value	Description
XEXC_ID_UNALIGNED_ACCESS	1	Unaligned access exceptions.
XEXC_ID_IOPB_EXCEPTION	2	Exception due to a timeout from the IOPB bus.
XEXC_ID_ILLEGAL_OPCODE	3	Exception due to an attempt to execute an illegal opcode.
XEXC_ID_DOPB_EXCEPTION	4	Exception due to a timeout on the DOPB bus.
XEXC_ID_DOPB_EXCEPTION	5	Divide by zero exceptions from the hardware divide.

Nested exceptions are allowed by the MicroBlaze system and the exception handler, in its prologue, re-enables exceptions. Thus, exceptions within exception handlers are allowed and handled.

## LibGen

With respect to interrupts, the Library Generator (LibGen) does two critical things:

1. Registers peripheral interrupt handlers
2. Generates the include file *xparameters.h*. This file defines base addresses of the peripherals in the system, **#defines** needed by drivers, OS's, libraries and user programs, as well as function prototypes. This information is used in conjunction with the drivers.

LibGen is run in XPS by selecting **Tools -> Generate Libraries and BSPs**.

In order to simplify the registration of peripheral interrupt handlers, handler information can be placed in the MSS file. LibGen then registers these handlers removing the requirement to register the interrupt handlers in the application code. For example, if the embedded processor design contains an OPB\_UARTLITE peripheral with an interrupt, the handler is assigned in the MSS file as shown below:

```

BEGIN DRIVER

PARAMETER DRIVER_NAME = uartlite

PARAMETER DRIVER_VER = 1.00.b

PARAMETER HW_INSTANCE = RS232_Uart

PARAMETER int_handler = uart_int_handler, int_port = Interrupt

END

```

For external port interrupts, the following line must be included in the MSS file.

```
PARAMETER int_handler = ext_int_handler, int_port = ext_interrupt
```

If the OPB/DCR\_INTC peripheral is included in the design, LibGen writes the xintc\_g.c file, which is used to register the handlers. The xintc\_g.c file can be found in the <project\_directory>\<microblaze\_0>\libsrc\intc\_v1\_00\_c\src directory.

Where microblaze\_0 is the processor instance in the embedded processor design.

### XPS Registration of External Interrupts

XPS provides the ability to write these parameters to the MSS file via the Software Platform Settings. In XPS, select Project -> Software Platform Settings to open the dialog. Figure 13 illustrates Processor, Driver Parameters and Interrupt Handlers tab.

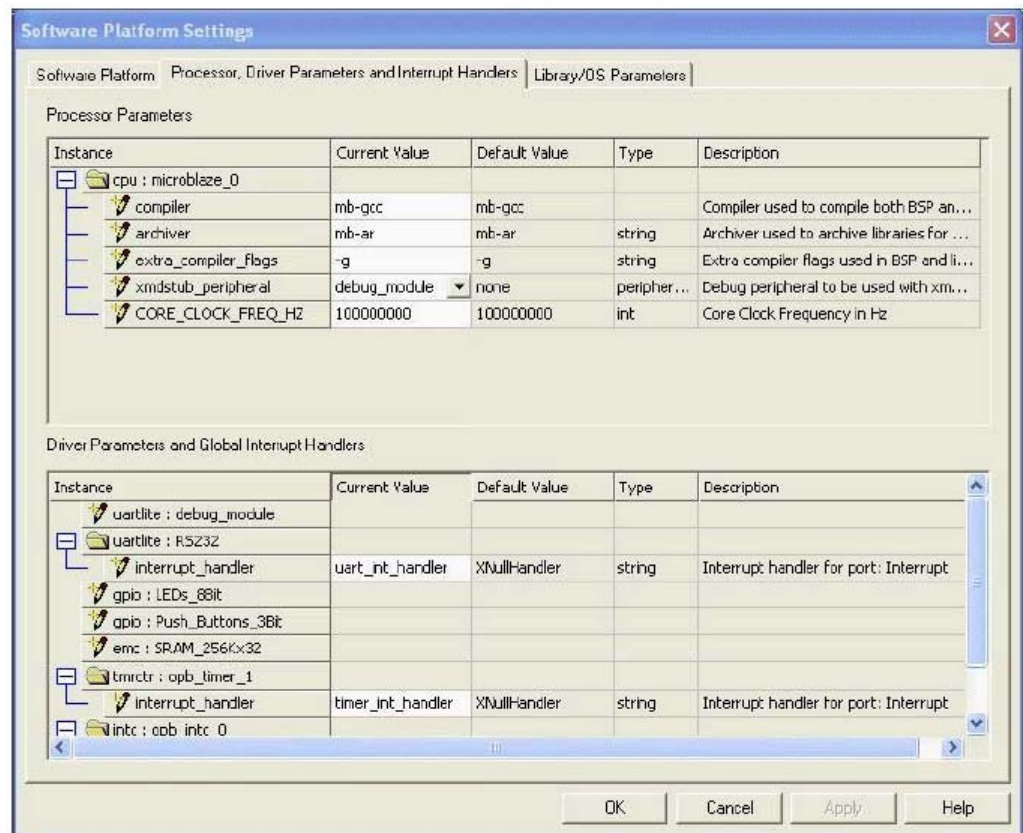


Figure 13: Software Platform Settings Dialog

The Interrupt Handlers for each peripheral can be specified. This information is then written in the MSS file.

## Example Code

The following code snippet illustrates the main software loop used to correctly setup exceptions/interrupts for a MicroBlaze design.

```
/* Register UART interrupt handler */
XIntc_RegisterHandler(XPAR_OPB_INTC_0_BASEADDR,
XPAR_OPB_INTC_0_RS232_INTERRUPT_INTR, uart_int_handler, (void *)0);

/* Register External interrupt handler */
XIntc_RegisterHandler(XPAR_OPB_INTC_0_BASEADDR,
XPAR_OPB_INTC_0_SYSTEM_EXT_INTERRUPT_INTR, ext_int_handler, (void *)0);

/* Start the interrupt controller */
XIntc_mMasterEnable(XPAR_OPB_INTC_0_BASEADDR);

/* Set the gpio as output on high 4 bits (LEDs)*/
XGpio_mSetDataDirection(XPAR_LEDS_8BIT_BASEADDR, 1, 0x00);

/* Set the number of cycles the timer counts before interrupting */
XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_1_BASEADDR, 0,
(timer_count*timer_count+1) * 800000000);

XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_1_BASEADDR, 1,
(timer_count*timer_count+1) * 100000000);

/* Reset the timers, and clear interrupts */
XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 0,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK );

XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, 1,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK | XTC_CSR_AUTO_RELOAD_MASK
| XTC_CSR_DOWN_COUNT_MASK );

/* Enable timer and uart interrupts in the interrupt controller */
XIntc_mEnableIntr(XPAR_OPB_INTC_0_BASEADDR, XPAR_OPB_TIMER_1_INTERRUPT_MASK
| XPAR_RS232_INTERRUPT_MASK);

/* Enable uart interrupts */
XUartLite_mEnableIntr(XPAR_RS232_BASEADDR);
```

```
/* Enable MicroBlaze interrupts */  
microblaze_enable_interrupts();
```

---

---

## Revision History

The following table shows the revision history for this document.

Date	Version	Revision
01/11/05	1.0	Initial Xilinx release.