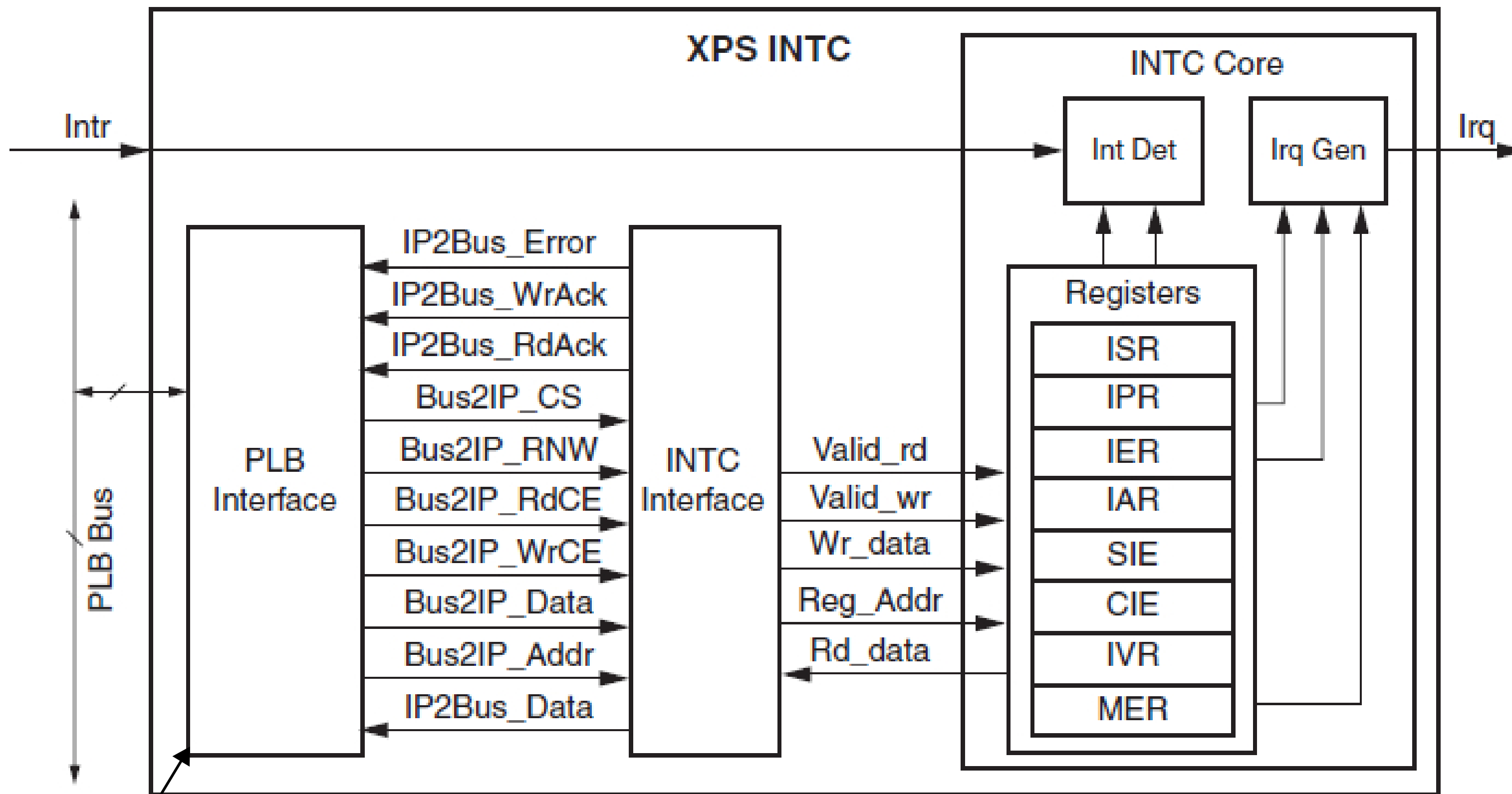
XPS INTC Block Diagram



Note:

Intc Interface: Design module does not exist by this name. This interface is part of top-level XPS INTC.

DS572_02_101405

Slave interface

Figure 2: **XPS INTC Block Diagram**

XPS INTC registers are memory mapped:

- Interrupt Status Register (ISR): Indicate presence or absense of an active interrupt signal. Is writable by software (for testing) until the HIE bit in the MER is set.
- Interrupt Pending Register (IPR): Read-only -- Logical AND of ISR and IER. Bit is set if interrupt is pending.
- Interrupt Enable Register (IER): Read/write register -- Writing a 1 to a bit position enables that signal's ability to generate an interrupt. Note that interrupts are still captured when the bit is 0, they are just not acted on (they are masked). If the bit is later updated to a 1, an interrupt occurs immediately.
- Interrupt Acknowledge Register (IAR): Write-only -- Clears the interrupt request if one exists in the ISR
- Set Interrupt Enables (SIE): Avoids read/modify/write sequence required for IER. Writing a 1 to a bit position sets the corresponding IER bit, while writing 0s does NOTHING.

- Clear Interrupt Enables (CIE): Same except writing a 1 to a bit position clear the IER (writing 0 does NOTHING).
- Interrupt Vector Register (IVR): Read-only -- ordinal value of the highest priority, enabled, active interrupt input. INT0 (always the LSB) is the highest priority interrupt. When no interrupt inputs are active, IVR contains all 1s.
- Master Enable Register (MER): Two-bit, Read/write register (bits in LSB). Least significant bit is the Master Enable, second bit is Hardware Interrupt Enable (HIE) bit. Writing 1 to ME enables IRQ output signal -- 0 masks all interrupts. HIE is a write-once bit. After reset, it is 0 allowing software testing, i.e., it allows software updates to be made to the ISR.
-

XPS GPIO Block Diagram

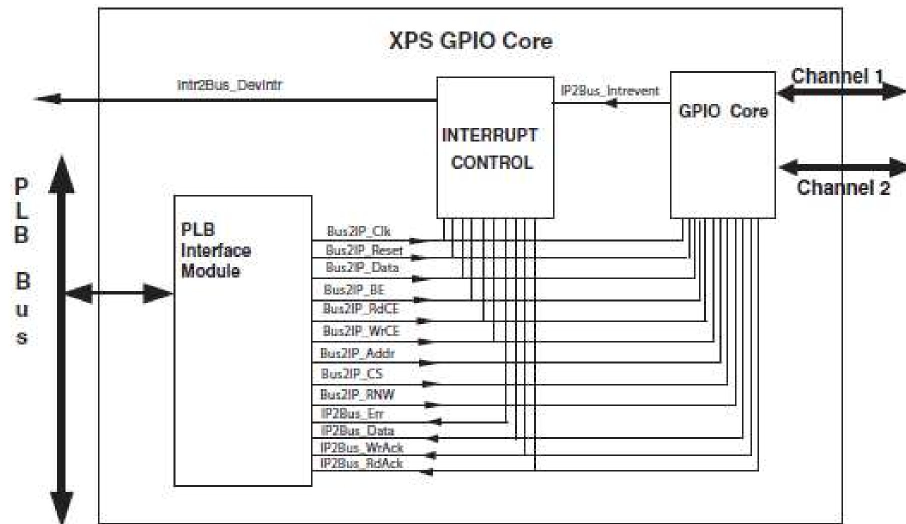
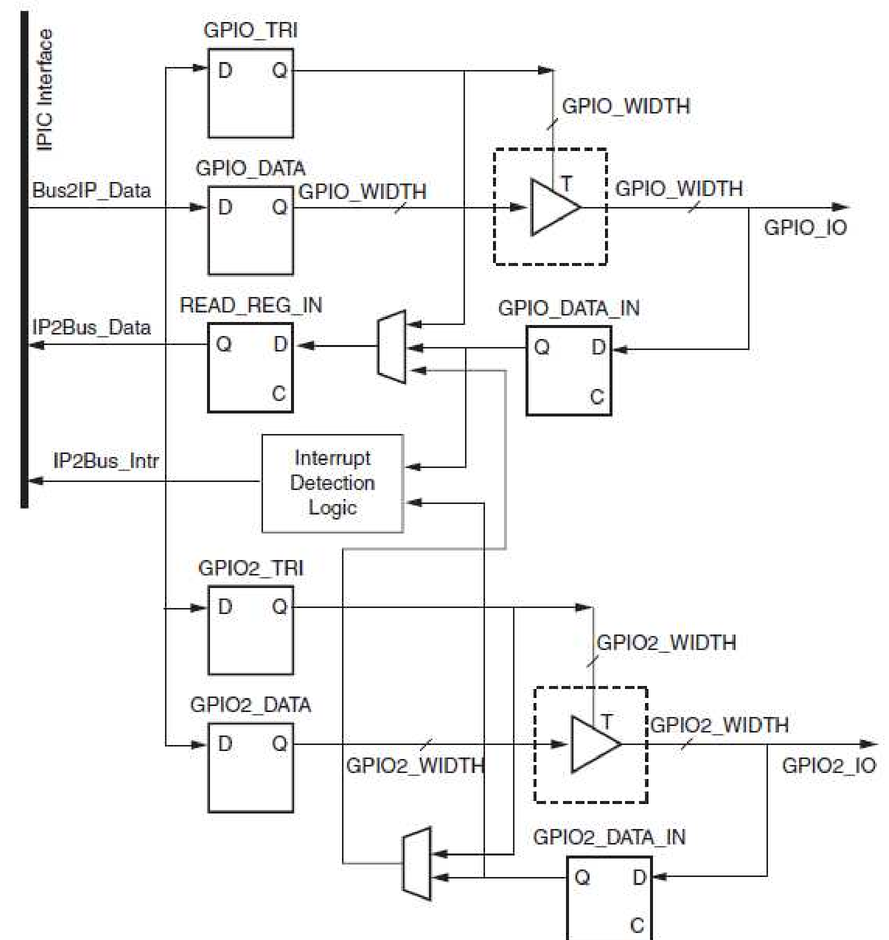


Figure 1: **XPS GPIO Block Diagram**



GPIO core is a 32-bit peripherial that attaches to the PLB, which is configurable as single or dual GPIO channels with or without interrupt control. The ports can be configured dynamically for input or output by enabling or disabling the 3-state buffer, and can be configured to generate an interrupt when a transition on any of their inputs occurs.

There are 4 XPS GPIO registers (if C_IS_DUAL is set to 1):

- GPIO_DATA **Channel 1 XPS GPIO Data Register** C_BASEADDR + 0x00 Read/Write
- GPIO_TRI **Channel 1 XPS GPIO 3-state Register** C_BASEADDR + 0x04 Read/Write
- GPIO2_DATA **Channel 2 XPS GPIO Data register** C_BASEADDR + 0x08 Read/Write
- GPIO2_TRI **Channel 2 XPS GPIO 3-state Register** C_BASEADDR + 0x0C Read/Write

The data registers are used to read the input ports and write the output ports. The XPS GPIO 3-state register is used to configure the ports dynamically as input (1) or output (0).

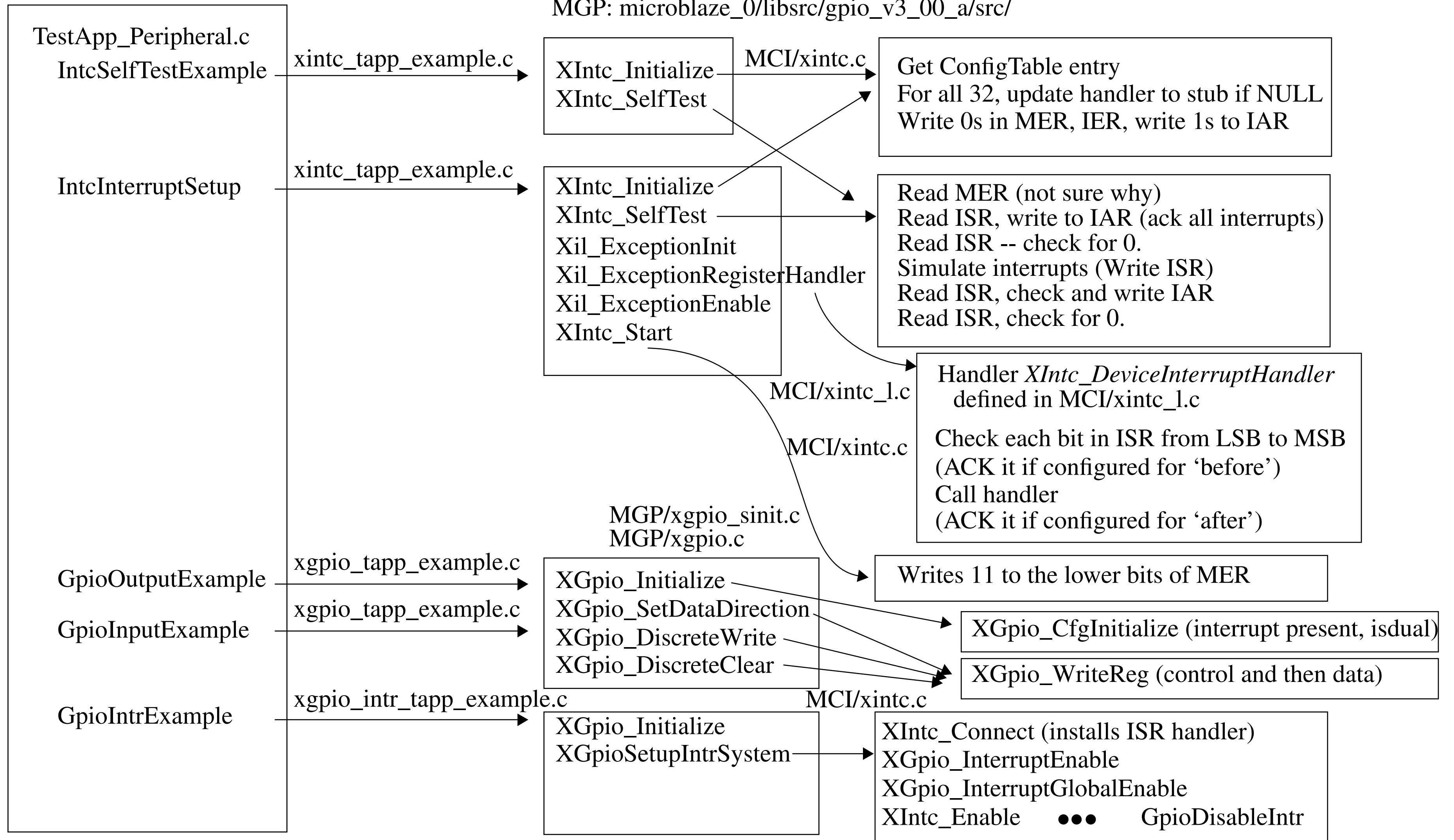
To support interrupt capability for channels, the PLB Interface module implements the following registers:

- Global Interrupt Enable register (GIE) (only bit 0 is valid, writing a '1' enables interrupts from both channels)
- IP Interrupt Enable Register (IP IER) (bit 30 for channel 2, bit 31 for channel 1 -- other bits are reserved)
- IP Interrupt Status Register (IP ISR) (bit 30 for channel 2, bit 31 for channel 1 -- other bits are reserved)

The IP IER implements independent interrupt enable bit for each channel while the Global Interrupt Enable Register provides the master enable/disable for the interrupt output to the processor.

The IP ISR provides Read and **Toggle-On-Write** access. The Toggle-On-Write mechanism allows interrupt service routines to clear one or more ISR bits using a single write transaction. This occurs when a '1' is written to a given bit position, which inverts the corresponding bit in the ISR.

TestAPP_Peripheral.c Software Call Sequences:



General Information

1) The xparameters.h file in microblaze_0/include defines two Gpio devices (#define XPAR_XGPIO_NUM_INSTANCES 2). The LEDs are device ID 0 (#define XPAR_LEDS_8BIT_DEVICE_ID 0) while the pushbuttons are device id 1 (#define XPAR_PUSH_BUTTONS_3BIT_DEVICE_ID 1) -- also defined as GPIO_DEVICE_ID in main()

Required Operations to Enable Interrupts for GPIO

- *XGpio_Initialize*: (microblaze_0/libsrc/gpio_v3_00_a/src/xgpio_sinit.c)

  Instance created in main()

```
XGpio Gpio; /* The Instance of GPIO */
XIntc Intc; /* The Instance of the Interrupt Controller Driver */
```

  Typedef for Gpio given in microblaze_0/libsrc/gpio_v3_00_a/src/xgpio.h

```
typedef struct {
        u32 BaseAddress;          /* Device base address */
        u32 IsReady;              /* Device is initialized and ready */
        int InterruptPresent;     /* Are interrupts supported in h/w */
        int IsDual;               /* Are 2 channels supported in h/w */
} XGpio;
```

  Config structure defined in microblaze_0/libsrc/gpio_v3_00_a/src/xgpio.h

```
typedef struct {
        u16 DeviceId;             /* Unique ID  of device */
        u32 BaseAddress;          /* Device base address */
        int InterruptPresent;     /* Are interrupts supported in h/w */
        int IsDual;               /* Are 2 channels supported in h/w */
} XGpio_Config;
```

  Two elements initialized in microblaze_0/libsrc/gpio_v3_00_a/src/xgpio_g.c

```
XGpio_Config XGpio_ConfigTable[] =
   {
        {
                XPAR_LEDS_8BIT_DEVICE_ID,
                XPAR_LEDS_8BIT_BASEADDR,
                XPAR_LEDS_8BIT_INTERRUPT_PRESENT,
                XPAR_LEDS_8BIT_IS_DUAL
        },
        {
```

```
                XPAR_PUSH_BUTTONS_3BIT_DEVICE_ID,
                XPAR_PUSH_BUTTONS_3BIT_BASEADDR,
                XPAR_PUSH_BUTTONS_3BIT_INTERRUPT_PRESENT,
                XPAR_PUSH_BUTTONS_3BIT_IS_DUAL
        }
    };
```

First thing *XGpio_Initialize* does is search for the device ID (XPAR_PUSH_BUTTONS_3BIT_DEVICE_ID) in the Config-Table. If NULL, return error, otherwise call *XGpio_CfgInitialize* (defined in microblaze_0/libsrc/gpio_v3_00_a/src/xgpio.c). This routine simply copies the remaining three data fields from XGpio_ConfigTable to the instance pointer fields. So it appears that the XGpio_ConfigTable is the 'bible' with regard to devices that are present in the system.

- GlobalIntrMask = GPIO_CHANNEL1

  This statement makes the channel on which the interrupt is generated global so that the ISR can use it to ack the interrupt.

- *XIntc_Initialize* (microblaze_0/libsrc/intc_v2_02_a/src/xintc.c)

  Constants and typedef (microblaze_0/libsrc/intc_v2_02_a/src/xintc_l.h)

```
#define XIN_ISR_OFFSET      0   /* Interrupt Status Register */
#define XIN_IPR_OFFSET      4   /* Interrupt Pending Register */
#define XIN_IER_OFFSET      8   /* Interrupt Enable Register */
#define XIN_IAR_OFFSET      12  /* Interrupt Acknowledge Register */
#define XIN_SIE_OFFSET      16  /* Set Interrupt Enable Register */
#define XIN_CIE_OFFSET      20  /* Clear Interrupt Enable Register */
#define XIN_IVR_OFFSET      24  /* Interrupt Vector Register */
#define XIN_MER_OFFSET      28  /* Master Enable Register */

#define XIN_INT_MASTER_ENABLE_MASK      0x1UL
#define XIN_INT_HARDWARE_ENABLE_MASK    0x2UL   /* once set cannot be cleared */

typedef struct {
        XInterruptHandler Handler;
        void *CallBackRef;
} XIntc_VectorTableEntry;
```

  Typedef for Config array (microblaze_0/libsrc/intc_v2_02_a/src/xintc.h)

```
typedef struct {
        u16 DeviceId;           /**< Unique ID  of device */
        u32 BaseAddress;        /**< Register base address */
        u32 AckBeforeService;   /**< Ack location per interrupt */
        u32 Options;            /**< Device options */
```

```
        XIntc_VectorTableEntry HandlerTable[XPAR_INTC_MAX_NUM_INTR_INPUTS];
} XIntc_Config;
```

Typedef for XIntc instance in main

```
typedef struct {
        u32 BaseAddress;          /**< Base address of registers */
        u32 IsReady;              /**< Device is initialized and ready */
        u32 IsStarted;            /**< Device has been started */
        u32 UnhandledInterrupts; /**< Intc Statistics */
        XIntc_Config *CfgPtr;     /**< Pointer to instance config entry */

} XIntc;
```

In xintc_g.c, this is initialized for the single interrupt controller created with information from xparameters.h:

```
XIntc_Config XIntc_ConfigTable[] =
{
        {
                XPAR_XPS_INTC_0_DEVICE_ID,
                XPAR_XPS_INTC_0_BASEADDR,
                XPAR_XPS_INTC_0_KIND_OF_INTR,
                XIN_SVC_SGL_ISR_OPTION,
                {
                        {
                                XNullHandler,
                                (void *) XNULL
                        }
                }
        }
};
```

XIntc_Initialize first checks the IsStarted field of the instance, and returns without mods if it is started already. It searches table above for match to DeviceId (only one in this case), and updates IsReady and IsStarted fields to 0, and sets CfgPtr to point to table entry. Sets the Options field in the CfgPtr to XIN_SVC_SGL_ISR_OPTION, which indicates to service the highest priority interrupt and then return (don't service them all). (THIS WAS ALREADY SET in the table initialization???). Set the instance BaseAddress to that given in the table element.

Number of interrupts defined in xparameters.h (#define XPAR_INTC_MAX_NUM_INTR_INPUTS 1).

Now, for each interrupt to be handled (only 1 in our case) initialize the the HandlerTable entry to 'StubHandler' if it is NULL. Unconditionally initialize the CallBackRef pointer to the instance.

Update a global var:

```
u32 XIntc_BitPosMask[XPAR_INTC_MAX_NUM_INTR_INPUTS];
```

with the bitmask (2^x value) corresponding to the interrupt number. Set to 1 in our case.
Finally, write 0s to the MIE and IER registers to disable the connection to the processor and all interrupts. Write all 1's to the IAR register, acknowledging all interrupts. Set the instance field IsReady to XIL_COMPONENT_IS_READY

- *XIntc_Connect* (microblaze_0/libsrc/intc_v2_02_a/src/xintc.c)
  This routine sets the Cfg table HandlerTable[INTC_GPIO_INTERRUPT_ID].Handler to *GpioDriverHandler*() and it's CallBackRef field to the instance pointer. (INTC_GPIO_INTERRUPT_ID is set to 0).
  The *GpioDriverHandler* is YOUR handler function.
- *XGpio_InterruptEnable* (microblaze_0/libsrc/intc_v2_02_a/src/xgpio_intr.c)
  Constants defining the register offsets for the GPIO channel (microblaze_0/libsrc/gpio_v3_00_a/src/xgpio_l.h)

```
#define XGPIO_DATA_OFFSET       0x0   /**< Data register for 1st channel */
#define XGPIO_TRI_OFFSET        0x4   /**< I/O direction reg for 1st channel */
#define XGPIO_DATA2_OFFSET      0x8   /**< Data register for 2nd channel */
#define XGPIO_TRI2_OFFSET       0xC   /**< I/O direction reg for 2nd channel */

#define XGPIO_GIE_OFFSET        0x11C /**< Glogal interrupt enable register */
#define XGPIO_ISR_OFFSET        0x120 /**< Interrupt status register */
#define XGPIO_IER_OFFSET        0x128 /**< Interrupt enable register */

#define XGPIO_GIE_GINTR_ENABLE_MASK     0x80000000
```

  Calls *XGpio_ReadReg* on the Gpio IER register and then a *XGpio_WriteReg* to update it with the bitwire OR of the read value with GPIO_CHANNEL1 (enables interrupt on the Gpio channel).
- *XGpio_InterruptGlobalEnable* (microblaze_0/libsrc/gpio_v3_00_a/src/xgpio_intr.c)
  Calls *XGpio_WriteReg* and updates GIE register with XGPIO_GIE_GINTR_ENABLE_MASK
- *XIntc_Enable* (microblaze_0/libsrc/intc_v2_02_a/src/xintc.c)
  Get the 2^ID mask from XIntc_BitPosMask (defined and set above in *XIntc_Initialize*). Read the interrupt controller IER register using *XIntc_In32* and update it using *XIntc_Out32* by bitwire-ORing the current mask with 2^ID mask (1 in our case).
- *Xil_ExceptionInit* (???)
  Initialize the exception table and register the interrupt controller handler with the exception table. Defined in the xil functions (somewhere).
- *Xil_ExceptionRegisterHandler* (???)

Install the *XIntc_InterruptHandler* function (defined in microblaze_0/libsrc/intc_v2_02_a/src/xintc_intr.c but simply calls *XIntc_DeviceInterruptHandler* which is defined in microblaze_0/libsrc/intc_v2_02_a/src/xintc_l.c). This function services all interrupts, calling the handlers for those interrupts that are active and then acknowledging them. The DeviceId of the controller is passwd in as a parameter (only one controller for our implementation). The bitwise AND of the ISR and IER register is obtained using the base address obtained from the appropriate XIntc_ConfigTable element and the bits are parsed from LSB (high priority) to MSB.

- *Xil_ExceptionEnable* (???)
  Enable non-critical interrupts
- *XIntc_Start* (microblaze_0/libsrc/intc_v2_02_a/src/xintc.c)
  Bitwire OR XIN_INT_HARDWARE_ENABLE_MASK with XIN_INT_MASTER_ENABLE_MASK (see definitions above), basically creates a ‘11’, and updates the MER with this value using *XIntc_Out32*. Also updates Intc instance field ‘IsStarted’ with XIL_COMPONENT_IS_STARTED.
- *XGpio_InterruptDisable* (microblaze_0/libsrc/gpio_v3_00_a/src/xgpio_intr.c)
  Disable the interrupts in the Gpio
- *XIntc_Disable* (microblaze_0/libsrc/intc_v2_02_a/src/xintc.c)
  Disable the GPIO interrupt in the controller