Appendix B

Interrupt Management

This appendix describes how to set up interrupts in a Xilinx® embedded hardware system. Also, this appendix describes the software flow of control during interrupts and the software APIs for managing interrupts. To benefit from this description, you need to have an understanding of hardware interrupts and their usefulness.

Hardware Setup

You must first wire the interrupts in your hardware so the processor receives interrupts.

The MicroBlaze™ processor has a single external interrupt port called `Interrupt`. The PowerPC® 405 processor and the PowerPC 440 processor each have two ports for handling interrupts. One port generates a *critical* category external interrupt and the other port generates a *non-critical* category external interrupt, the difference between the two categories being the priority level over other competing interrupts and exceptions in the system. The critical category has the highest priority.

- On the PowerPC 405 processor, the critical and non-critical interrupt ports are named `EICC405CRITINPUTIRQ` and `EICC405EXTINPUTIRQ` respectively.
- On the PowerPC 440 processor, the critical and non-critical interrupt ports are named `EICC440CRITIRQ` and `EICC440EXTIRQ` respectively.

There are two ways to wire interrupts to a processor:

- The interrupt signal from the interrupting peripheral is directly connected to the processor interrupt port. In this configuration, only one peripheral can interrupt the processor.
- The interrupt signal from the interrupting peripheral is connected to an interrupt controller core which in turn generates an interrupt on a signal connected to the interrupt port on the processor. This allows multiple peripherals to send interrupt signals to a processor. This is the more common method as there are usually more than one peripheral on embedded systems that require access to the interrupt function.

Figure B-1, page 236 illustrates the interrupt configurations.

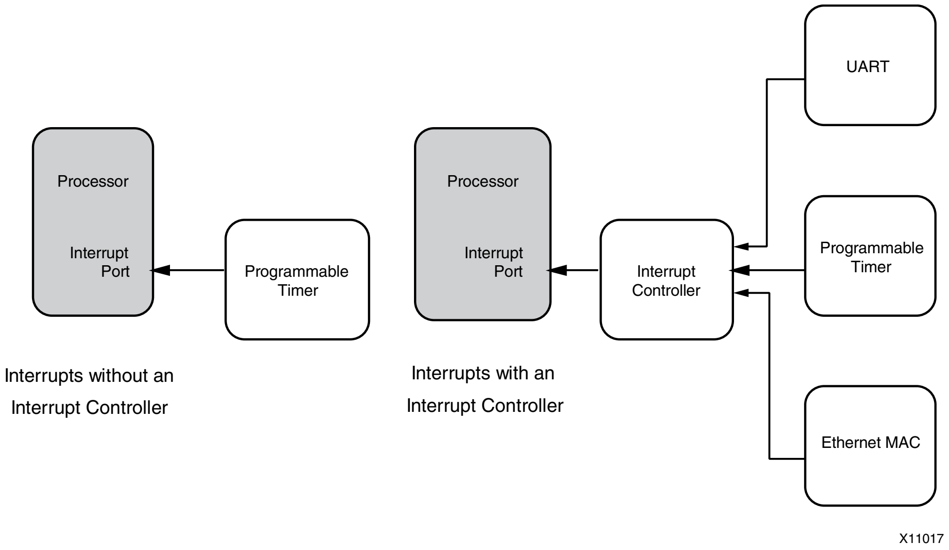

Figure B-1: **Interrupt Configurations**

Software Setup and Interrupt Flow

Interrupts are typically vectored through multiple levels in the software platform before the application interrupt handlers are executed. The Xilinx software platforms (Standalone and XilKernel) follow the interrupt flow shown in Figure B-2.

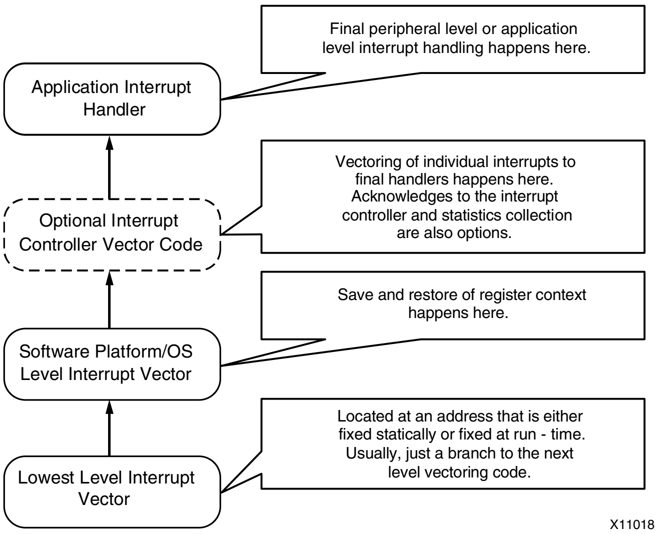
Application Interrupt Handler
Final peripheral level or application level interrupt handling happens here.
Optional Interrupt Controller Vector Code
Vectoring of individual interrupts to final handlers happens here. Acknowledges to the interrupt controller and statistics collection are also options.
Software Platform/OS Level Interrupt Vector
Save and restore of register context happens here.
Lowest Level Interrupt Vector
Located at an address that is either fixed statically or fixed at run - time. Usually, just a branch to the next level vectoring code.
X11018

Figure B-2: **Interrupt Flow**

Interrupt Flow for MicroBlaze Systems

MicroBlaze interrupts go through the following flow:

1. Interrupts have to be enabled on MicroBlaze by setting appropriate bits in the Machine Status Registers (MSR).
2. Upon an external interrupt signal being raised, the processor first disables further interrupts. Then, the processor jumps to an absolute, fixed address `0x0000_0010`.
3. The software platform or OS provides vectoring code at this address which transfers control to the main platform interrupt handler.
4. The platform interrupt handler saves all of the processor registers (that could be clobbered further down) onto the current application stack.
5. The handler then transfers control to the next level handler. Because the next level handler can be dependent on whether there is an interrupt controller in the system or not, the handler consults an internal interrupt vectoring table to determine the function address of the next level handler. It also consults the vectoring table for a callback value that it must pass to the next level handler. Finally, the actual call is made.
    - On systems with an interrupt controller, the next level handler is the handler provided by the interrupt controller driver. This handler queries the interrupt controller for all active interrupts in the system. For each active interrupt, it consults its internal vector table, which contains the user registered handler for each interrupt line. If the user has not registered any handler, a default do-nothing handler is registered. The registered handler for each interrupt gets invoked in turn (in interrupt priority order).
    - On systems without an interrupt controller, the next handler is the final interrupt handler that the application wishes to execute.
6. The final interrupt handler for a particular interrupt typically queries the interrupting peripheral and determines the cause for the interrupt. It does a series of actions that are appropriate for the given peripheral and the cause for the interrupt. The handler is also responsible for acknowledging the interrupt at the interrupting peripheral. After the interrupt handler is finished, it returns back and the interrupt stack gets unwound all the way back to the software platform level interrupt handler.
7. The platform level interrupt handler restores the registers it saved on the stack and returns control back to the Program Counter (PC) location where the interrupt occurred. The return instruction also enables interrupts again on the MicroBlaze processor. The application resumes normal execution at this point.

    Xilinx recommends that interrupt handlers be kept to a short duration and the bulk of the work be left to the application to handle. This prevents long lockouts of other (possibly higher priority) interrupts and is considered good system design.

Figure B-3, page 238 shows a MicroBlaze interrupt flow without an interrupt controller, and Figure B-4, page 238 shows a MicroBlaze interrupt flow with an interrupt controller.

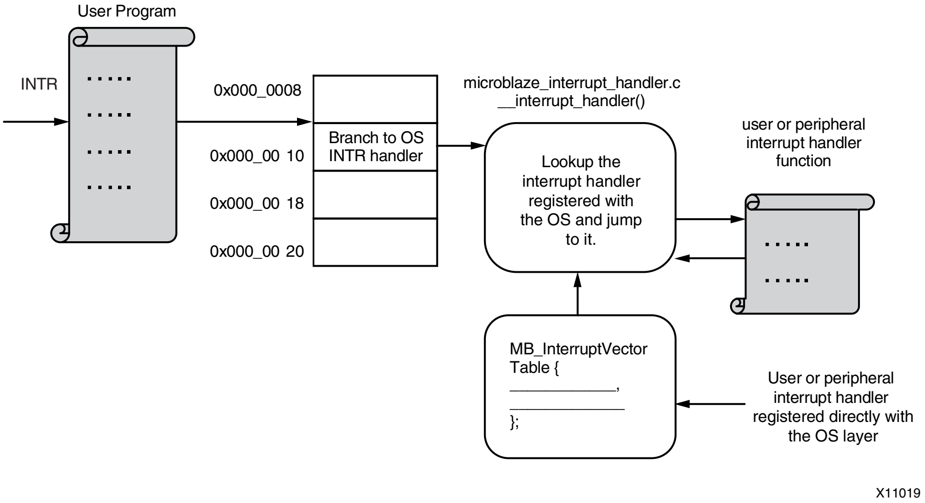
User Program
INTR
0x000_0008
0x000_00 10
0x000_00 18
0x000_00 20
Branch to OS INTR handler
microblaze_interrupt_handler.c
__interrupt_handler()
Lookup the interrupt handler registered with the OS and jump to it.
user or peripheral interrupt handler function
MB_InterruptVector Table {
______________,
____
};
User or peripheral interrupt handler registered directly with the OS layer
X11019

Figure B-3: **MicroBlaze Interrupt Flow without Interrupt Controller**

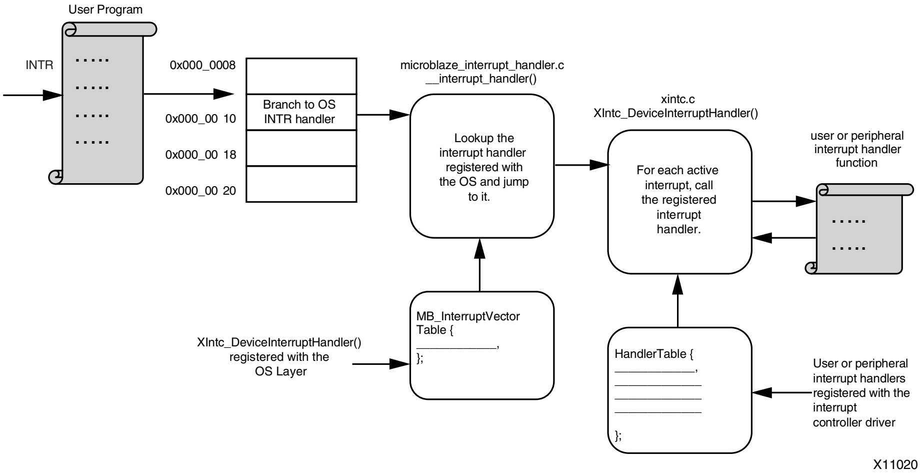
User Program
INTR
0x000_0008
0x000_00 10
0x000_00 18
0x000_00 20
Branch to OS INTR handler
microblaze_interrupt_handler.c
__interrupt_handler()
Lookup the interrupt handler registered with the OS and jump to it.
xintc.c
XIntc_DeviceInterruptHandler()
For each active interrupt, call the registered interrupt handler.
user or peripheral interrupt handler function
MB_InterruptVector Table {
____________,
};
XIntc_DeviceInterruptHandler() registered with the OS Layer
HandlerTable {
____________,
____
____
____
};
User or peripheral interrupt handlers registered with the interrupt controller driver
X11020

Figure B-4: **MicroBlaze Interrupt Flow with Interrupt Controller**

Interrupt Flow for PowerPC Systems

Interrupts on the PowerPC processors go through the following flow:

1. Interrupts must be enabled on the PowerPC processor by setting appropriate bits in the Machine Status Registers (MSR). Depending on whether critical or non-critical (or both) interrupts are being used, appropriate bits must be set.
2. Upon the external interrupt signal being raised, the processor first disables further interrupts. The processor then calculates an address for the interrupt type and jumps to that address. The calculation varies between the PowerPC 405 processor and the PowerPC 440 processor.
    - The PowerPC 405 processor consults the software-set value of the Exception Vector Prefix Register (EVPR) and adds a constant offset to this value (depending on the interrupt type) to determine the final physical address where the vector code is placed.
    - The PowerPC 440 processor has independent offset registers for each interrupt type (labeled `IVOR0-IVOR15`). Each offset register contains a value that is appended to the Interrupt Vector Prefix register (IVPR) to obtain the final physical address of the interrupt vector code.
3. The processor jumps to the calculated interrupt vector code address.
4. Each interrupt vector location contains a platform interrupt handler that is appropriate for the interrupt type:
    - For external critical and non-critical interrupts, the handler saves all of the processor registers (that could be clobbered further down) onto the current application stack.
    - The handler then transfers control to the next level handler. Because this can be dependent on whether there is an interrupt controller in the system, the handler consults an internal interrupt vectoring table to determine the function address of the next level handler.
    - The handler also consults the vectoring table for a callback value that it must pass to the next level handler. Then, the handler makes the actual call.
    - On systems with an interrupt controller, the next level handler is the handler provided by the interrupt controller driver. This handler queries the interrupt controller for all active interrupts in the system. For each active interrupt, it consults its internal vector table, which contains the user-registered handler for each interrupt line.
      If no handler is registered, a default do-nothing handler is registered. The registered handler for each interrupt gets invoked in turn (in interrupt priority order).
    - On systems without an interrupt controller, the next handler is the final interrupt handler that is executed by the application.
5. The final interrupt handler for a particular interrupt typically queries the interrupting peripheral and determines the cause for the interrupt. It usually does a series of actions that are appropriate for the given peripheral and the cause for the interrupt. The handler is also responsible for acknowledging the interrupt at the interrupting peripheral. When the interrupt handler completes its activity, it returns back and the interrupt stack gets unwound back to the software platform level interrupt handler.

The platform level interrupt handler restores the registers that it saved on the stack and returns control back to the Program Counter (PC) location where the interrupt occurred.

The return instruction also enables interrupts again on the PowerPC processor. The application resumes normal execution at this point.

It is recommended that interrupt handlers be of a short duration and that the bulk of the interrupt work be done by application. This prevents long lockouts of other (possibly higher priority) interrupts and is considered good system design.

Figure B-5 shows a PowerPC processor interrupt flow without an interrupt controller.

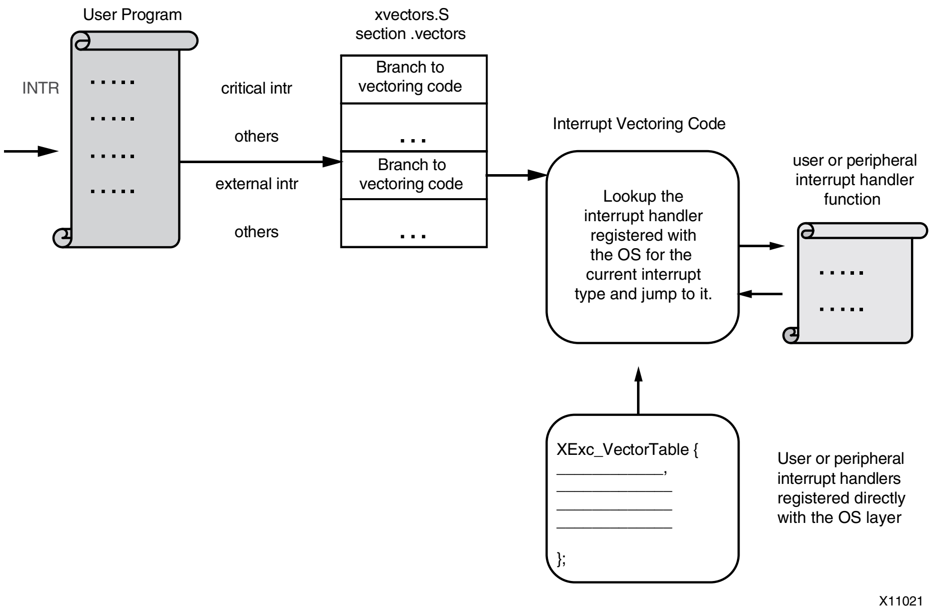
User Program
INTR
critical intr
others
external intr
others
xvectors.S
section .vectors
Branch to vectoring code
...
Branch to vectoring code
...
Interrupt Vectoring Code
Lookup the interrupt handler registered with the OS for the current interrupt type and jump to it.
user or peripheral interrupt handler function
XExc_VectorTable {
};
User or peripheral interrupt handlers registered directly with the OS layer
X11021

Figure B-5: **PowerPC Processor Interrupt Flow without Interrupt Controller**

Figure B-6 shows a PowerPC processor interrupt flow with an interrupt controller.

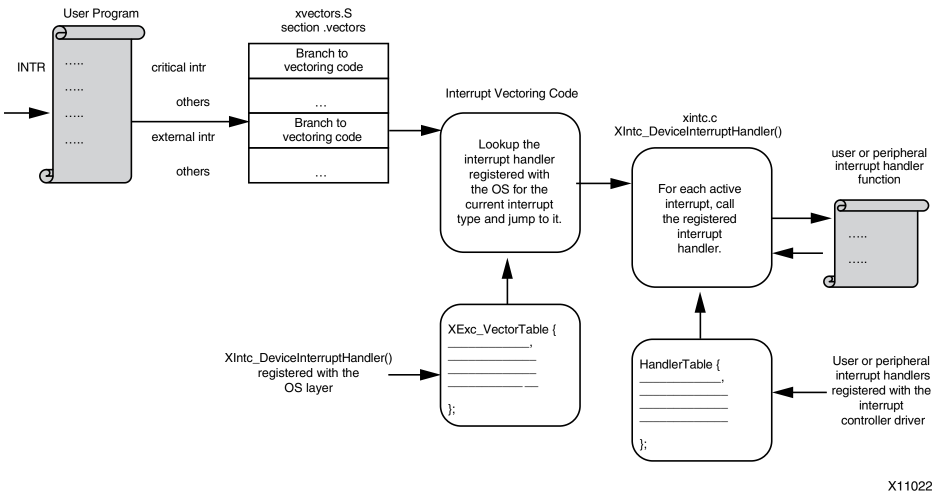
User Program
INTR
critical intr
others
external intr
others
xvectors.S
section .vectors
Branch to vectoring code
...
Branch to vectoring code
...
Interrupt Vectoring Code
Lookup the interrupt handler registered with the OS for the current interrupt type and jump to it.
xintc.c
XIntc_DeviceInterruptHandler()
For each active interrupt, call the registered interrupt handler.
user or peripheral interrupt handler function
XIntc_DeviceInterruptHandler() registered with the OS layer
XExc_VectorTable {
};
HandlerTable {
};
User or peripheral interrupt handlers registered with the interrupt controller driver
X11022

Figure B-6: **PowerPC Processor Interrupt Flow with Interrupt Controller**

Software APIs

This section provides an overview of the software APIs involved in handling and managing interrupts, lists the available Software APIs by processor type, and provides examples of interrupt management code.

Note: This chapter is not meant to cover the APIs comprehensively. Refer to the interrupt controller device driver documentation as well as the reference documentation for the Standalone platform to for all the details of the APIs.

Interrupt Controller Driver

The Xilinx interrupt controller supports the following features:

- Enabling and disabling specific individual interrupts
- Acknowledging specific individual interrupts
- Attaching specific callback function to handle interrupt source
- Enabling and disabling the master
- Sending a single callback per interrupt or handling all pending interrupts for each interrupt of the processor

The acknowledgement of the interrupt within the interrupt controller is selectable, either prior to calling the device handler or after the handler is called. Interrupt signal inputs are either edge or level signal; consequently, support for those inputs is required:

- Edge-driven interrupt signals require that the interrupt is acknowledged prior to the interrupt being serviced to prevent the loss of interrupts which are occurring close together.
- Level-driven interrupt input signals require the interrupt to be acknowledged after servicing the interrupt to ensure that the interrupt only generates a single interrupt condition.

API Descriptions

`int` **`XIntc_Initialize`** `(XIntc *` *`InstancePtr`*`, u16` *`DeviceId`*`)`

Description	Initializes a specific interrupt controller instance or driver. All the fields of the `XIntc` structure and the internal vectoring tables are initialized. All interrupt sources are disabled.
Parameters	*`InstancePtr`* is a pointer to the XIntc instance. *`DeviceId`* is the unique id of the device controlled by this `XIntc` instance (obtained from `xparameters.h`). Passing in a *`DeviceId`* associates the generic `XIntc` instance to a specific device, as chosen by the caller or application developer.

int **XIntc_Connect** (XIntc * *InstancePtr*, u8 *Id*, XInterruptHandler *Handler*, void * *CallBackRef*)

Description	Makes the connection between the *`Id`* of the interrupt source and the associated handler that is to be run when the interrupt occurs. The argument provided in this call as the *`CallBackRef`* is used as the argument for the handler when it is called.
Parameters	*`InstancePtr`* is a pointer to the `XIntc` instance. *`Id`* contains the ID of the interrupt source and should be in the range of 0 to `XPAR_INTC_MAX_NUM_INTR_INPUTS` - 1 with 0 being the highest priority interrupt. *`Handler`* is the handler for that interrupt. *`CallBackRef`* is the callback reference, usually the instance pointer of the connecting driver The handler provided as an argument overwrites any handler that was previously connected.

void **XIntc_Disconnect** (XIntc* *InstancePtr*, u8 *Id*)

Description	Disconnects the `XIntc` instance.
Parameters	*`InstancePtr`* is a pointer to the `XIntc` instance. *`Id`* contains the ID of the interrupt source and should be in the range of 0 to `XPAR_INTC_MAX_NUM_INTR_INPUTS` - 1 with 0 being the highest priority interrupt.

Void **XIntc_Enable** (XIntc * *InstancePtr*, u8 *Id*)

Description	Enables the interrupt source provided as the argument *`Id`*. Any pending interrupt condition for the specified Id occurs after this function is called.
Parameters	*`InstancePtr`* is a pointer to the `XIntc` instance. *`Id`* contains the ID of the interrupt source and should be in the range of 0 to `XPAR_INTC_MAX_NUM_INTR_INPUTS` - 1 with 0 being the highest priority interrupt.

void **XIntc_Disable** (Xintc * *InstancePtr*, u8 *Id*)

Description	Disables the interrupt source provided as the argument *`Id`*, such that the interrupt controller does not cause interrupts for the specified Id. The interrupt controller continues to hold an interrupt condition for the Id, but does not cause an interrupt.
Parameters	*`InstancePtr`* is a pointer to the `XIntc` instance. *`Id`* contains the ID of the interrupt source and should be in the range of 0 to `XPAR_INTC_MAX_NUM_INTR_INPUTS` - 1 with 0 being the highest priority interrupt.

`int` **`XIntc_Start`** `(XIntc *` *`InstancePtr`*`, u8` *`Mode`*`)`

Description	Starts the interrupt controller by enabling the output from the controller to the processor. Interrupts can be generated by the interrupt controller after this function is called.
Parameters	*`InstancePtr`* is a pointer to the `XIntc` instance. *`Mode`* determines if software is allowed to simulate interrupts or if real interrupts are allowed to occur. Modes are mutually exclusive. The interrupt controller hardware resets in a mode that allows software to simulate interrupts until this mode is exited. It cannot be re-entered after it has been exited. Mode is one of the following valued: `XIN_SIMULATION_MODE` enables simulation of interrupts only. `XIN_REAL_MODE` enables hardware interrupts only. This function must be called after `Xintc` initialization is completed.

`void` **`XIntc_Stop`** `(XIntc *` *`InstancePtr`*`)`

Description	Stops the interrupt controller by disabling the output from the controller so that no interrupts are caused by the interrupt controller.
Parameters	*`InstancePtr`* is a pointer to the `XIntc` instance.

Hardware Abstraction Layer APIs

The following is a summary of exception functions, which can run on MicroBlaze, PowerPC 405, and PowerPC 440 processors.

Header File

```
#include "xil_exception.h"
```

Typedef

```
typedef void(* Xil_ExceptionHandler)(void *Data)
```

This typedef is the exception handler function pointer.

`void` **`Xil_ExceptionDisable`** `()`

Description	Disable Exceptions. On PowerPC 405 and PowerPC 440 processors, this function only disables non-critical exceptions.

`void` **`Xil_ExceptionEnable`** `()`

Description	Enable Exceptions. On PowerPC 405 and PowerPC 440 processors, this function only enables non-critical exceptions.

`void Xil_ExceptionInit()`

Description	Initialize exception handling for the processor. The exception vector table is set up with the stub handler for all exceptions.

`void Xil_ExceptionRegisterHandler(u32 Id, Xil_ExceptionHandler Handler,void *Data)`

Description	Make the connection between the ID of the exception source and the associated handler that runs when the exception is recognized. Data is used as the argument when the handler is called.
Parameters	Parameters: *`Id`* contains the identifier (ID) of the exception source. This should be `XIL_EXCEPTION_INT` or be in the range of 0 to `XIL_EXCEPTION_LAST`. Refer to the xil_exception.h file for further information. *`Handler`* is the handler for that exception. *`Data`* is a reference to data that is passed to the handler when it is called.

`void Xil_ExceptionRemoveHandler(u32 Id)`

Description	Remove the handler for a specific exception ID. The stub handler is then registered for this exception ID.
Parameters	*`Id`* contains the ID of the exception source. It should be `XIL_EXCEPTION_INT` or in the range of `0` to `XIL_EXCEPTION_LAST`. Refer to the `xil_exception.h` file for further information.

Interrupt Setup Example

```
/***************************** Include Files *************************/

#include "xparameters.h"
#include "xtmrctr.h"
#include "xintc.h"
#include "xil_exception.h"

/********************** Constant Definitions ************************/
/*
 * The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are only defined here such that a user can
 * easily change all the needed parameters in one place.
 */
#define TMRCTR_DEVICE_IDXPAR_TMRCTR_0_DEVICE_ID
#define INTC_DEVICE_IDXPAR_INTC_0_DEVICE_ID
#define TMRCTR_INTERRUPT_IDXPAR_INTC_0_TMRCTR_0_VEC_ID

/*
 * The following constant determines which timer counter of the device
 * that is used for this example, there are currently 2 timer counters
 * in a device and this example uses the first one, 0, the timer numbers
 * are 0 based
 */
#define TIMER_CNTR_0 0

/*
 * The following constant is used to set the reset value of the timer
 * counter, making this number larger reduces the amount of time this
 * example consumes because it is the value the timer counter is loaded
 * with when it is started
 */
#define RESET_VALUE 0xF0000000

/******************** Function Prototypes ************************/

int TmrCtrIntrExample(XIntc* IntcInstancePtr,
         XTmrCtr* InstancePtr,
         u16 DeviceId,
         u16 IntrId,
         u8 TmrCtrNumber);

void TimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber);

/********************* Variable Definitions ***********************/
XIntc InterruptController;  /* The instance of the Interrupt Controller
*/

XTmrCtr TimerCounterInst;   /* The instance of the Timer Counter */

/*
 * The following variables are shared between non-interrupt processing
 * and interrupt processing such that they must be global.
 */
volatile int TimerExpired;


/*****************************************************************/
/**
* This function is the main function of the Tmrctr example using
* Interrupts.
*
* @paramNone.
*
* @returnXST_SUCCESS to indicate success, else XST_FAILURE to indicate
*   a Failure.
*
* @noteNone.
*
******************************************************************/

int main(void)
{

  int Status;

  /*
   * Run the Timer Counter - Interrupt example.
   */
  Status = TmrCtrIntrExample(&InterruptController,
          &TimerCounterInst,
          TMRCTR_DEVICE_ID,
          TMRCTR_INTERRUPT_ID,
          TIMER_CNTR_0);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  return XST_SUCCESS;

}

/********************************************************************/
/**
* This function does a minimal test on the timer counter device and
* driver as a design example. The purpose of this function is to
* illustrate how to use the XTmrCtr component. It initializes a timer
* counter and then sets it up in compare mode with auto reload such that
* a periodic interrupt is generated.
*
* This function uses interrupt driven mode of the timer counter.
*
* @paramIntcInstancePtr is a pointer to the Interrupt Controller
*   driver Instance
* @paramTmrCtrInstancePtr is a pointer to the XTmrCtr driver Instance
* @paramDeviceId is the XPAR_<TmrCtr_instance>_DEVICE_ID value from
*   xparameters.h
* @paramIntrId is
XPAR_<INTC_instance>_<TmrCtr_instance>_INTERRUPT_INTR
*   value from xparameters.h
* @paramTmrCtrNumber is the number of the timer to which this
*   handler is associated with.
*
* @returnXST_SUCCESS if the Test is successful, otherwise XST_FAILURE
*
* @noteThis function contains an infinite loop such that if interrupts
*   are not working it may never return.
*
*********************************************************************/
int TmrCtrIntrExample(XIntc* IntcInstancePtr,
          XTmrCtr* TmrCtrInstancePtr,
          u16 DeviceId,
          u16 IntrId,
          u8 TmrCtrNumber)
{
  int Status;
  int LastTimerExpired = 0;

  /*
   * Initialize the timer counter so that it's ready to use,
   * specify the device ID that is generated in xparameters.h
   */
  Status = XTmrCtr_Initialize(TmrCtrInstancePtr, DeviceId);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  /*
   * Initialize the interrupt controller driver so that
   * it's ready to use, specify the device ID that is generated in
   * xparameters.h
```

```
 */
Status = XIntc_Initialize(IntcInstancePtr, INTC_DEVICE_ID);
if (Status != XST_SUCCESS) {
  return XST_FAILURE;
}

/*
 * Connect a device driver handler that will be called when an
 * interrupt for the device occurs, the device driver handler performs
 * the specific interrupt processing for the device
 */
Status = XIntc_Connect(IntcInstancePtr, IntrId,
             (XInterruptHandler)XTmrCtr_InterruptHandler,
             (void *)TmrCtrInstancePtr);

if (Status != XST_SUCCESS) {
  return XST_FAILURE;
}

/*
 * Start the interrupt controller such that interrupts are enabled for
 * all devices that cause interrupts, specific real mode so that
 * the timer counter can cause interrupts thru the interrupt
 * controller.
 */
Status = XIntc_Start(IntcInstancePtr, XIN_REAL_MODE);
if (Status != XST_SUCCESS) {
  return XST_FAILURE;
}

/*
 * Enable the interrupt for the timer counter
 */
XIntc_Enable(IntcInstancePtr, IntrId);

/*
 * Initialize the exception table.
 */
Xil_ExceptionInit();

/*
 * Register the interrupt controller handler with the exception table.
 */
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
             (Xil_ExceptionHandler)
             XIntc_InterruptHandler,
             IntcInstancePtr);

/*
 * Enable exceptions.
 */
Xil_ExceptionEnable();
if (Status != XST_SUCCESS) {
  return XST_FAILURE;
}

/*
 * Setup the handler for the timer counter that will be called from the
 * interrupt context when the timer expires, specify a pointer to the
```

```
   * timer counter driver instance as the callback reference so the
   * handler is able to access the instance data
   */
  XTmrCtr_SetHandler(TmrCtrInstancePtr,
        TimerCounterHandler,
        TmrCtrInstancePtr);

  /*
   * Enable the interrupt of the timer counter so interrupts will occur
   * and use auto reload mode such that the timer counter will reload
   * itself automatically and continue repeatedly, without this option
   * it would expire once only
   */
  XTmrCtr_SetOptions(TmrCtrInstancePtr, TmrCtrNumber,
        XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);

  /*
   * Set a reset value for the timer counter such that it will expire
   * earlier than letting it roll over from 0, the reset value is loaded
   * into the timer counter when it is started
   */
  XTmrCtr_SetResetValue(TmrCtrInstancePtr, TmrCtrNumber, RESET_VALUE);

  /*
   * Start the timer counter such that it's incrementing by default,
   * then wait for it to timeout a number of times
   */
  XTmrCtr_Start(TmrCtrInstancePtr, TmrCtrNumber);

  while (1) {
    /*
     * Wait for the first timer counter to expire as indicated by the
     * shared variable which the handler will increment
     */
    while (TimerExpired == LastTimerExpired) {
    }
    LastTimerExpired = TimerExpired;

    /*
     * If it has expired a number of times, then stop the timer counter
     * and stop this example
     */
    if (TimerExpired == 3) {

      XTmrCtr_Stop(TmrCtrInstancePtr, TmrCtrNumber);
      break;
    }
  }

  /*
   * Disable the interrupt for the timer counter
   */
  XIntc_Disable(IntcInstancePtr, DeviceId);

  return XST_SUCCESS;
}

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

```
* This function is the handler which performs processing for the timer
* counter. It is called from an interrupt context such that the amount
* of processing performed should be minimized. It is called when the
* timer counter expires if interrupts are enabled.
*
* This handler provides an example of how to handle timer counter
* interrupts but is application specific.
*
* @paramCallBackRef is a pointer to the callback function
* @paramTmrCtrNumber is the number of the timer to which this
*   handler is associated with.
*
* @returnNone.
*
* @noteNone.
*
*********************************************************************/
void TimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber)
{
  XTmrCtr *InstancePtr = (XTmrCtr *)CallBackRef;

  /*
   * Check if the timer counter has expired, checking is not necessary
   * since that's the reason this function is executed, this just shows
   * how the callback reference can be used as a pointer to the instance
   * of the timer counter that expired, increment a shared variable so
   * the main thread of execution can see the timer expired
   */
  if (XTmrCtr_IsExpired(InstancePtr, TmrCtrNumber)) {
   TimerExpired++;
   if(TimerExpired == 3) {
     XTmrCtr_SetOptions(InstancePtr, TmrCtrNumber, 0);
   }
  }
}
```