Microblaze Software Configurations of Interrupts (draft 2)

Rhys Gretch, Forrest Brewer

October 13, 2021

1 General Interrupt flow

Generally, configuring interrupts in software follows the same flow. The interrupt is initialized, then connected to the handler, which is a custom function you've written. Then the interrupt is started and enabled. The microblaze drivers expose these commands through the following function calls:

1. `XIntc_Initialize(XIntc* InterruptController, uint INTERRUPT_ID)`

   Where `InterruptController` is a user instantiated variable of type `XIntc`, and `INTERRUPT_ID` is the macro for the overall interrupt controller `(XPAR_MICROBLAZE_0_AXI_INTC_DEVICE_ID)`.

   This function initializes all fields of the `InterruptController`, creates an initial vector table with stub function calls, and disables all interrupt sources and outputs.

2. `XIntc_Connect(XIntc* InterruptController, uint DEVICE_MASK,`

   `XInterruptHandler handler_function, void* CallBackRef)`

   Where `InterruptController` is the same as the one from initialize, `DEVICE_MASK` is the macro for the peripheral that is driving the interrupt, `handler_function` is the interrupt handler function you wrote cast as an `XInterruptHandler`, and `CallBackRef` is the peripheral specific controller (The object this should be is described below)

   This function updates the vector table for the ID of the interrupt source with the associated handler to be run when an interrupt is recognized.

3. `XIntc_Start(XIntc* InterruptController, uint MODE)`

   `InterruptController` is the same as before. `MODE` can be one of two macros, `XIN_SIMULATION_MODE` or `XIN_REAL_MODE`. For the purpose of this class you should use the REAL option.

   This function starts the interrupt controller by enabling it to output to the processor. Now Interrupts may be generated.

4. `XIntc_Enable(XIntc* InterruptController, uint DEVICE_MASK)`

   `InterrruptController` and `DEVICE_MASK` are the same as before

   This function enables the interrupt source specified by `DEVICE_MASK`.

Note: while these are not the exact function declarations you need to use your identifiers, and even names change with tool version. To see these functions definitions and source code go to:
`<project name>_bsp/microblaze_0/libsrc/intc_<version #>`

2 Peripheral Specific Interrupts

Once the interrupt has been configured to work in the microblaze environment, you must configure the specific device interrupt. For timers and GPIO this process is slightly different. Also, for every interrupt you configure you will need a separate `XIntc InterruptController` instance, and the function flow from above must be repeated for each instance.

2.1 Timer Interrupts

For timers the interrupt is configured in a general way following the instructions above, then configured using a timer specific flow:

1. `XTmerCtr_Initialize(XTmCtr* timerController, uint INTERRUPT_ID)`

   `timerController` is user initialized and should also be the `CallBackRef` on the `XIntc_Connect` call. `INTERRUPT_ID` should be the same as the `XINTC_Initialize` call

   This function initializes a specific timer/counter instance. It Initializes fields of the XTmrCtr structure, then resets the timer.

2. `XTmrCtr_SetOptions(XTmrCtr* timerController, uint 0, uint OPTION_MACROS)`

   `timerController` is the same as the one from `XTmerCtr_Initialize` call. `OPTION_MACROS` let you configure the timer interrupt's behavior. See `extra.c` for an example of which macros can be used.

3. `XTmrCtr_SetResetValue(XTmrCtr* timerController, uint 0, RESET_VALUE)`

   The `timerController` is the same object that was used above.. The `RESET_VALUE` is what value the counter/timer resets to after it expires.

   In the setup provided by `extra.c` the timer/counter counts up. When it reaches its max value it triggers its interrupt, then resets to the `RESET_VALUE`. Therefore it takes the interrupt `0xFFFFFFFF-RESET_VALUE` clock cycles before it triggers again.

4. `XTmrCtr_Start(XTmrCtr* timerController, uint 0)`

   `timerController` is the same object that was used above.

   This function starts the specified timer counter so that it begins running from `RESET_VALUE`.

Extra.c has a full example of how to set up and configure a timer interrupt. To see these functions definitions and source code go to `<project name>_bsp/microblaze_0/libsrc/tmrctr_<version #>`

2.2 GPIO Interrupts

For GPIO peripherals the general flow follows:

1. `XGpio_Initialize(XGpio* gpioController, uint GPIO_DEVICE_ID)`

   The `gpioController` is user initialized and should also be the `CallBackRef` on the `XIntc_Connect` call. The `GPIO_DEVICE_ID` is a macro specifying the GPIO device and can be found in `Xparameters.h`. `Xparameters.h` is found is the bsp package that is created by default with a new application project.

   This function initializes the `XGpio` object with the device's information.

2. `XGpio_InterruptEnable(XGpio* gpioController, uint 1)`

   `gpioController` is the same object as before.

   This function ensures interrupts are configured for this device then enables the interrupt.

3. `XGpio_InterruptGlobalEnable(XGpio* gpioController)`

   `gpioController` is the same object as before.

   This function allows for interrupt output signals to be passed through to the microblaze.

The `gpioController` must be unique for each gpio peripheral, even if the peripheral is not an interrupt. The instance identifies status and addresses unique to the assigned peripheral. To see these functions definitions and source code go to `<project name>_bsp/microblaze_0/libsrc/gpio_<version #>`

3 Code Examples

3.1 extra.h

```
#include "xtmrctr.h"
#include "xintc.h"
#include "xparameters.h"
#include "xtmrctr_l.h"
#include "xintc_l.h"
#include "mb_interface.h"
#include <xbasic_types.h>
#include <xio.h>

#define INTC_DEVICE_ID                  XPAR_INTC_0_DEVICE_ID
#define RESET_VALUE 1000

void extra_handler();
void extra_disable();
void extra_enable();
int extra_method();
```

3.2 extra.c

```
#include "extra.h"
#include <stdlib.h>

XIntc sys_intc;
XTmrCtr sys_tmrctr;
Xuint32 data;

unsigned int count = 0;

void extra_handler() {
  // This is the interrupt handler function
  // Do not print inside of this function.
  Xuint32 ControlStatusReg;
  /*
   * Read the new Control/Status Register content.
   */
  ControlStatusReg = XTimerCtr_ReadReg(sys_tmrctr.BaseAddress, 0, XTC_TCSR_OFFSET);

  // xil_printf("Timer interrupt occurred. Count= %d\r\n", count);
  // XGpio_DiscreteWrite(&led,1,count);
  count++;      // increment count
  /*
   * Acknowledge the interrupt by clearing the interrupt
   * bit in the timer control status register
   */
  XTmrCtr_WriteReg(sys_tmrctr.BaseAddress, 0, XTC_TCSR_OFFSET, ControlStatusReg

  |XTC_CSR_INT_OCCURED_MASK);
}

void extra_disable() {
  XIntc_Disable(&sys_intc, XPAR_MICROBLAZE_0_AXI_INTC_AXI_TIMER_0_INTERRUPT_INTR);
}

void extra_enable() {
  XIntc_Enable(&sys_intc, XPAR_MICROBLAZE_0_AXI_INTC_AXI_TIMER_0_INTERRUPT_INTR);
}
```

```
int extra_method() {
  //xil_printf("I'm in the main() method\r\n");

  XStatus Status;
  /*
   * Initialize the interrupt controller driver so that
   * it is ready to use, specify the device ID that is generated in
   * xparameters.h
   */
  Status = XST_SUCCESS;
  Status = XIntc_Initialize(&sys_intc, XPAR_MICROBLAZE_0_AXI_INTC_DEVICE_ID);

  if ( Status != XST_SUCCESS ) {
    if( Status == XST_DEVICE_NOT_FOUND ) {
      xil_printf("XST_DEVICE_NOT_FOUND...\r\n");
    }
    else {
      xil_printf("a␣different␣error␣from␣XST_DEVICE_NOT_FOUND...\r\n");
    }
    xil_printf("Interrupt␣controller␣driver␣failed␣to␣be␣initialized...\r\n");
    return XST_FAILURE;
  }
  xil_printf("Interrupt␣controller␣driver␣initialized!\r\n");
  /*
   * Connect the application handler that will be called when an interrupt
   * for the timer occurs
   */
  Status = XIntc_Connect(&sys_intc,
        XPAR_MICROBLAZE_0_AXI_INTC_AXI_TIMER_0_INTERRUPT_INTR,
        (XInterruptHandler)extra_handler, &sys_tmrctr);
  if ( Status != XST_SUCCESS ) {
    xil_printf(
    "Failed␣to␣connect␣the␣application␣handlers␣to␣the␣interrupt␣controller...\r\n");
    return XST_FAILURE;
  }
  xil_printf("Connected␣to␣Interrupt␣Controller!\r\n");
  /*
   * Start the interrupt controller such that interrupts are enabled for
   * all devices that cause interrupts.
   */
  Status = XIntc_Start(&sys_intc, XIN_REAL_MODE);
  if ( Status != XST_SUCCESS ) {
    xil_printf("Interrupt␣controller␣driver␣failed␣to␣start...\r\n");
    return XST_FAILURE;
  }
  xil_printf("Started␣Interrupt␣Controller!\r\n");
  /*
   * Enable the interrupt for the timer counter
   */
  XIntc_Enable(&sys_intc, XPAR_MICROBLAZE_0_AXI_INTC_AXI_TIMER_0_INTERRUPT_INTR);
  /*
   * Initialize the timer counter so that it's ready to use,
   * specify the device ID that is generated in xparameters.h
   */
  Status = XTmrCtr_Initialize(&sys_tmrctr, XPAR_MICROBLAZE_0_AXI_INTC_DEVICE_ID);
  if ( Status != XST_SUCCESS ) {
    xil_printf("Timer␣initialization␣failed...\r\n");
    return XST_FAILURE;
  }
  xil_printf("Initialized␣Timer!\r\n");
  /*
```

```
   * 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(&sys_tmrctr, 0, XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);
  /*
   * Set a reset value for the timer counter such that it will expire
   * eariler than letting it roll over from 0, the reset value is loaded
   * into the timer counter when it is started
   */
  XTmrCtr_SetResetValue(&sys_tmrctr, 0, 0xFFFFFFFF-RESET_VALUE);
  /*
   * 0x17D7840 = 25*10^6 clock cycles @ 50MHz = 500ms
   *
   * Start the timer counter such that it's incrementing by default,
   * then wait for it to timeout a number of times
   */
  XTmrCtr_Start(&sys_tmrctr, 0);
  /*
   * Register the intc device driver's handler with the Standalone
   * software platform's interrupt table
   */
  microblaze_register_handler((XInterruptHandler)XIntc_DeviceInterruptHandler,
      (void*)XPAR_MICROBLAZE_0_AXI_INTC_DEVICE_ID);
//  microblaze_register_handler((XInterruptHandler)XIntc_DeviceInterruptHandler,
//      (void*)PUSHBUTTON_DEVICE_ID);
  //xil_printf("Registers handled!\r\n");
  /*
   * Enable interrupts on MicroBlaze
   */
  microblaze_enable_interrupts();
  xil_printf("Interrupts enabled!\r\n");
  /*
   * At this point, the system is ready to respond to interrupts from the timer
   */
  return XST_SUCCESS;
}
```