```
/* $Id: xintc.c,v 1.1.2.1 2010/09/17 05:26:04 svemula Exp $ */
/******************************************************************************
*
* (c) Copyright 2002-2009 Xilinx, Inc. All rights reserved.
*
* This file contains confidential and proprietary information of Xilinx, Inc.
* and is protected under U.S. and international copyright and other
* intellectual property laws.
*
* DISCLAIMER
* This disclaimer is not a license and does not grant any rights to the
* materials distributed herewith. Except as otherwise provided in a valid
* license issued to you by Xilinx, and to the maximum extent permitted by
* applicable law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND WITH ALL
* FAULTS, AND XILINX HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS,
* IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF
* MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE;
* and (2) Xilinx shall not be liable (whether in contract or tort, including
* negligence, or under any other theory of liability) for any loss or damage
* of any kind or nature related to, arising under or in connection with these
* materials, including for any direct, or any indirect, special, incidental,
* or consequential loss or damage (including loss of data, profits, goodwill,
* or any type of loss or damage suffered as a result of any action brought by
* a third party) even if such damage or loss was reasonably foreseeable or
* Xilinx had been advised of the possibility of the same.
*
* CRITICAL APPLICATIONS
* Xilinx products are not designed or intended to be fail-safe, or for use in
* any application requiring fail-safe performance, such as life-support or
* safety devices or systems, Class III medical devices, nuclear facilities,
* applications related to the deployment of airbags, or any other applications
* that could lead to death, personal injury, or severe property or
* environmental damage (individually and collectively, "Critical
* Applications"). Customer assumes the sole risk and liability of any use of
* Xilinx products in Critical Applications, subject only to applicable laws
* and regulations governing limitations on product liability.
*
* THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS PART OF THIS FILE
* AT ALL TIMES.
*
******************************************************************************/
/*****************************************************************************/
/**
*
* @file xintc.c
*
* Contains required functions for the XIntc driver for the Xilinx Interrupt
* Controller. See xintc.h for a detailed description of the driver.
*
* <pre>
* MODIFICATION HISTORY:
*
* Ver   Who  Date     Changes
* ----- ---- -------- -----------------------------------------------------
* 1.00a ecm  08/16/01 First release
* 1.00b jhl  02/21/02 Repartitioned the driver for smaller files
* 1.00b jhl  04/24/02 Made LookupConfig global and compressed ack before table
*                     in the configuration into a bit mask
* 1.00c rpm  10/17/03 New release. Support the static vector table created
*                     in the xintc_g.c configuration table.
* 1.00c rpm  04/23/04 Removed check in XIntc_Connect for a previously connected
*                     handler. Always overwrite the vector table handler with
*                     the handler provided as an argument.
* 1.10c mta  03/21/07 Updated to new coding style
* 1.11a sv   11/21/07 Updated driver to support access through a DCR bridge
* 2.00a ktn  10/20/09 Updated to use HAL Processor APIs.
* </pre>
*
******************************************************************************/

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

#include "xil_types.h"
#include "xil_assert.h"
#include "xintc.h"
#include "xintc_l.h"
#include "xintc_i.h"
```

```
/************************** Constant Definitions *****************************/


/**************************** Type Definitions *******************************/


/***************** Macros (Inline Functions) Definitions *********************/


/************************** Variable Definitions *****************************/

/* Array of masks associated with the bit position, improves performance
 * in the ISR and acknowledge functions, this table is shared between all
 * instances of the driver, this table is not statically initialized because
 * the size of the table is based upon the maximum used interrupt id */
u32 XIntc_BitPosMask[XPAR_INTC_MAX_NUM_INTR_INPUTS];

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

static void StubHandler(void *CallBackRef);

/*****************************************************************************/
/**
*
* Initialize a specific interrupt controller instance/driver. The initialization entails:
*       - Initialize fields of the XIntc structure
*       - Initial vector table with stub function calls
*       - All interrupt sources are disabled
*       - Interrupt output is disabled
*
* @param        InstancePtr is a pointer to the XIntc instance to be worked on.
* @param        DeviceId is the unique id of the device controlled by this XIntc
*               instance.  Passing in a device id associates the generic XIntc
*               instance to a specific device, as chosen by the caller or
*               application developer.
* @return
*               - XST_SUCCESS if initialization was successful
*               - XST_DEVICE_IS_STARTED if the device has already been started
*               - XST_DEVICE_NOT_FOUND if device configuration information was
*               not found for a device with the supplied device ID.
* @note         None.
*
******************************************************************************/
int XIntc_Initialize(XIntc * InstancePtr, u16 DeviceId)
   {
   u8 Id;
   XIntc_Config *CfgPtr;
   u32 NextBitMask = 1;

   Xil_AssertNonvoid(InstancePtr != NULL);

/* If the device is started, disallow the initialize and return a status indicating it is started.
   This allows the user to stop the device and reinitialize, but prevents a user from inadvertently initializ
ing */
   if (InstancePtr->IsStarted == XIL_COMPONENT_IS_STARTED)
      { return XST_DEVICE_IS_STARTED; }

/* Lookup the device configuration in the CROM table. Use this configuration info down below when initializin
g this component.  */
   CfgPtr = XIntc_LookupConfig(DeviceId);
   if (CfgPtr == NULL)
      { return XST_DEVICE_NOT_FOUND; }

/* Set some default values */
   InstancePtr->IsReady = 0;
   InstancePtr->IsStarted = 0;  /* not started */
   InstancePtr->CfgPtr = CfgPtr;

   InstancePtr->CfgPtr->Options = XIN_SVC_SGL_ISR_OPTION;

/* Save the base address pointer such that the registers of the interrupt can be accessed */
#if (XPAR_XINTC_USE_DCR_BRIDGE != 0)
   InstancePtr->BaseAddress = ((CfgPtr->BaseAddress >> 2)) & 0xFFF;
#else
   InstancePtr->BaseAddress = CfgPtr->BaseAddress;
#endif
```

```
/* Initialize all the data needed to perform interrupt processing for each interrupt ID up to the maximum
   used */
   for (Id = 0; Id < XPAR_INTC_MAX_NUM_INTR_INPUTS; Id++)
      {

/* Initalize the handler to point to a stub to handle an interrupt which has not been connected to a
   handler. Only initialize it if the handler is 0 or XNullHandler, which means it was not initialized
   statically by the tools/user. Set the callback reference to this instance so that unhandled interrupts
   can be tracked.  */
      if ((InstancePtr->CfgPtr->HandlerTable[Id].Handler == 0) ||
         (InstancePtr->CfgPtr->HandlerTable[Id].Handler == XNullHandler))
         { InstancePtr->CfgPtr->HandlerTable[Id].Handler = StubHandler; }
      InstancePtr->CfgPtr->HandlerTable[Id].CallBackRef = InstancePtr;

/* Initialize the bit position mask table such that bit positions are lookups only for each interrupt id, wit
h 0
 * being a special case (XIntc_BitPosMask[] = { 1, 2, 4, 8, ... }) */
      XIntc_BitPosMask[Id] = NextBitMask;
      NextBitMask *= 2;
      }

/* Disable IRQ output signal -- Disable all interrupt sources -- Acknowledge all sources */
   XIntc_Out32(InstancePtr->BaseAddress + XIN_MER_OFFSET, 0);
   XIntc_Out32(InstancePtr->BaseAddress + XIN_IER_OFFSET, 0);
   XIntc_Out32(InstancePtr->BaseAddress + XIN_IAR_OFFSET, 0xFFFFFFFF);

/* Indicate the instance is now ready to use, successfully initialized */
   InstancePtr->IsReady = XIL_COMPONENT_IS_READY;

   return XST_SUCCESS;
   }

/*****************************************************************************/
/**
* Starts the interrupt controller by enabling the output from the controller
* to the processor. Interrupts may be generated by the interrupt controller
* after this function is called.
*
* It is necessary for the caller to connect the interrupt handler of this
* component to the proper interrupt source.
* @param        InstancePtr is a pointer to the XIntc instance to be worked on.
* @param        Mode determines if software is allowed to simulate interrupts or
*               real interrupts are allowed to occur. Note that these 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 reentered once it has been exited.
*
*               One of the following values should be used for the mode.
*               - XIN_SIMULATION_MODE enables simulation of interrupts only
*               - XIN_REAL_MODE enables hardware interrupts only
* @return
*               - XST_SUCCESS if the device was started successfully
*               - XST_FAILURE if simulation mode was specified and it could not
*               be set because real mode has already been entered.
*
* @note         Must be called after XIntc initialization is completed.
*
*****************************************************************************/
int XIntc_Start(XIntc * InstancePtr, u8 Mode)
   {
   u32 MasterEnable = XIN_INT_MASTER_ENABLE_MASK;

/* Assert the arguments */
   Xil_AssertNonvoid(InstancePtr != NULL);
   Xil_AssertNonvoid((Mode == XIN_SIMULATION_MODE) || (Mode == XIN_REAL_MODE))
   Xil_AssertNonvoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

/* Check for simulation mode */
   if (Mode == XIN_SIMULATION_MODE)
      {
      if (MasterEnable & XIN_INT_HARDWARE_ENABLE_MASK)
         { return XST_FAILURE; }
      }
   else
      { MasterEnable |= XIN_INT_HARDWARE_ENABLE_MASK; }
```

```
/* Indicate the instance is ready to be used and is started before we enable the device.  */
   InstancePtr->IsStarted = XIL_COMPONENT_IS_STARTED;

   XIntc_Out32(InstancePtr->BaseAddress + XIN_MER_OFFSET, MasterEnable);

   return XST_SUCCESS;
   }

/****************************************************************************/
/**
* Stops the interrupt controller by disabling the output from the controller
* so that no interrupts will be caused by the interrupt controller.
* @param        InstancePtr is a pointer to the XIntc instance to be worked on.
* @return       None.
* @note         None.
*
*****************************************************************************/
void XIntc_Stop(XIntc * InstancePtr)
   {
/* Assert the arguments */
   Xil_AssertVoid(InstancePtr != NULL);
   Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

/* Stop all interrupts from occurring thru the interrupt controller by disabling all
   interrupts in the MER register */
   XIntc_Out32(InstancePtr->BaseAddress + XIN_MER_OFFSET, 0);

   InstancePtr->IsStarted = 0;
   }

/****************************************************************************/
/**
* Makes the connection between the Id of the interrupt source and the
* associated handler that is to run when the interrupt is recognized. The
* argument provided in this call as the Callbackref is used as the argument
* for the handler when it is called.
* @param        InstancePtr is a pointer to the XIntc instance to be worked on.
* @param        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.
* @param        Handler to the handler for that interrupt.
* @param        CallBackRef is the callback reference, usually the instance
*               pointer of the connecting driver.
* @return
*               - XST_SUCCESS if the handler was connected correctly.
* @note
*
* WARNING: The handler provided as an argument will overwrite any handler
* that was previously connected.
*
*****************************************************************************/
int XIntc_Connect(XIntc * InstancePtr, u8 Id, XInterruptHandler Handler, void *CallBackRef)
   {
/* Assert the arguments */
   Xil_AssertNonvoid(InstancePtr != NULL);
   Xil_AssertNonvoid(Id < XPAR_INTC_MAX_NUM_INTR_INPUTS);
   Xil_AssertNonvoid(Handler != NULL);
   Xil_AssertNonvoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

/* The Id is used as an index into the table to select the proper handler */
   InstancePtr->CfgPtr->HandlerTable[Id].Handler = Handler;
   InstancePtr->CfgPtr->HandlerTable[Id].CallBackRef = CallBackRef;

   return XST_SUCCESS;
   }

/****************************************************************************/
/**
* Updates the interrupt table with the Null Handler and NULL arguments at the
* location pointed at by the Id. This effectively disconnects that interrupt
* source from any handler. The interrupt is disabled also.
* @param        InstancePtr is a pointer to the XIntc instance to be worked on.
* @param        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.
* @return       None.
* @note         None.
```

```
*****************************************************************************/
void XIntc_Disconnect(XIntc * InstancePtr, u8 Id)
   {
   u32 CurrentIER;
   u32 Mask;

/* Assert the arguments */
   Xil_AssertVoid(InstancePtr != NULL);
   Xil_AssertVoid(Id < XPAR_INTC_MAX_NUM_INTR_INPUTS);
   Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

/* Disable the interrupt such that it won't occur while disconnecting
 * the handler, only disable the specified interrupt id without
 * modifying the other interrupt ids */
   CurrentIER = XIntc_In32(InstancePtr->BaseAddress + XIN_IER_OFFSET);

   Mask = XIntc_BitPosMask[Id];/* convert from integer id to bit mask */

   XIntc_Out32(InstancePtr->BaseAddress + XIN_IER_OFFSET, (CurrentIER & ~Mask));

/* Disconnect the handler and connect a stub, the callback reference
 * must be set to this instance to allow unhandled interrupts to be
 * tracked */
   InstancePtr->CfgPtr->HandlerTable[Id].Handler = StubHandler;
   InstancePtr->CfgPtr->HandlerTable[Id].CallBackRef = InstancePtr;
   }

/*****************************************************************************/
/**
* Enables the interrupt source provided as the argument Id. Any pending
* interrupt condition for the specified Id will occur after this function is
* called.
* @param        InstancePtr is a pointer to the XIntc instance to be worked on.
* @param        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.
* @return       None.
* @note         None.
*****************************************************************************/
void XIntc_Enable(XIntc * InstancePtr, u8 Id)
   {
   u32 CurrentIER;
   u32 Mask;

/* Assert the arguments */
   Xil_AssertVoid(InstancePtr != NULL);
   Xil_AssertVoid(Id < XPAR_INTC_MAX_NUM_INTR_INPUTS);
   Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

/* The Id is used to create the appropriate mask for the desired bit position.
   Id currently limited to 0 - 31 */
   Mask = XIntc_BitPosMask[Id];

/* Enable the selected interrupt source by reading the interrupt enable
 * register and then modifying only the specified interrupt id enable */
   CurrentIER = XIntc_In32(InstancePtr->BaseAddress + XIN_IER_OFFSET);
   XIntc_Out32(InstancePtr->BaseAddress + XIN_IER_OFFSET, (CurrentIER | Mask));
   }

/*****************************************************************************/
/**
* Disables the interrupt source provided as the argument Id such that the
* interrupt controller will not cause interrupts for the specified Id. The
* interrupt controller will continue to hold an interrupt condition for the
* Id, but will not cause an interrupt.
* @param        InstancePtr is a pointer to the XIntc instance to be worked on.
* @param        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.
* @return       None.
* @note         None.
*****************************************************************************/
void XIntc_Disable(XIntc * InstancePtr, u8 Id)
   {
   u32 CurrentIER;
   u32 Mask;
```

```
/* Assert the arguments */
   Xil_AssertVoid(InstancePtr != NULL);
   Xil_AssertVoid(Id < XPAR_INTC_MAX_NUM_INTR_INPUTS);
   Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

/* The Id is used to create the appropriate mask for the desired bit position.
   Id currently limited to 0 - 31 */
   Mask = XIntc_BitPosMask[Id];

/* Disable the selected interrupt source by reading the interrupt enable
 * register and then modifying only the specified interrupt id */
   CurrentIER = XIntc_In32(InstancePtr->BaseAddress + XIN_IER_OFFSET);
   XIntc_Out32(InstancePtr->BaseAddress + XIN_IER_OFFSET, (CurrentIER & ~Mask));
   }

/****************************************************************************/
/**
* Acknowledges the interrupt source provided as the argument Id. When the
* interrupt is acknowledged, it causes the interrupt controller to clear its
* interrupt condition.
* @param        InstancePtr is a pointer to the XIntc instance to be worked on.
* @param        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.
* @return       None.
* @note         None.
*****************************************************************************/
void XIntc_Acknowledge(XIntc * InstancePtr, u8 Id)
   {
   u32 Mask;

/* Assert the arguments */
   Xil_AssertVoid(InstancePtr != NULL);
   Xil_AssertVoid(Id < XPAR_INTC_MAX_NUM_INTR_INPUTS);
   Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

/* The Id is used to create the appropriate mask for the desired bit position.
   Id currently limited to 0 - 31 */
   Mask = XIntc_BitPosMask[Id];

/* Acknowledge the selected interrupt source, no read of the acknowledge
 * register is necessary since only the bits set in the mask will be
 * affected by the write */
   XIntc_Out32(InstancePtr->BaseAddress + XIN_IAR_OFFSET, Mask);
   }

/****************************************************************************/
/**
* A stub for the asynchronous callback. The stub is here in case the upper
* layers forget to set the handler.
* @param        CallBackRef is a pointer to the upper layer callback reference
* @return       None.
* @note         None.
*****************************************************************************/
static void StubHandler(void *CallBackRef)
   {
/* Verify that the inputs are valid */
   Xil_AssertVoid(CallBackRef != NULL);

/* Indicate another unhandled interrupt for stats */
   ((XIntc *) CallBackRef)->UnhandledInterrupts++;
   }

/****************************************************************************/
/**
* Looks up the device configuration based on the unique device ID. A table
* contains the configuration info for each device in the system.
* @param        DeviceId is the unique identifier for a device.
* @return       A pointer to the XIntc configuration structure for the specified
*               device, or NULL if the device was not found.
* @note         None.
*****************************************************************************/
XIntc_Config *XIntc_LookupConfig(u16 DeviceId)
   {
   XIntc_Config *CfgPtr = NULL;
   int Index;
```

```
   for (Index = 0; Index < XPAR_XINTC_NUM_INSTANCES; Index++)
      {
      if (XIntc_ConfigTable[Index].DeviceId == DeviceId)
         {
         CfgPtr = &XIntc_ConfigTable[Index];
         break;
         }
      }

   return CfgPtr;
   }
```