```
/* $Id: xintc_l.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_l.c
*
* This file contains low-level driver functions that can be used to access the
* device.  The user should refer to the hardware device specification for more
* details of the device operation.
*
* <pre>
* MODIFICATION HISTORY:
*
* Ver   Who  Date     Changes
* ----- ---- -------- -----------------------------------------------------
* 1.00b jhl  04/24/02 First release
* 1.00c rpm  10/17/03 New release. Support the static vector table created
*                     in the xintc_g.c configuration table.
* 1.00c rpm  04/09/04 Added conditional compilation around the old handler
*                     XIntc_LowLevelInterruptHandler(). This handler will only
*                     be include/compiled if XPAR_INTC_SINGLE_DEVICE_ID is
*                     defined.
* 1.10c mta  03/21/07 Updated to new coding style
* 1.10c ecm  07/09/07 Read the ISR after the Acknowledge in the interrupt
*                     handler to support architectures with posted write bus
*                     access issues.
* 2.00a ktn  10/20/09 Updated t use HAL Processor APIs and  _m is removed
*                     from all the macro definitions.
* </pre>
*
******************************************************************************/


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

#include "xparameters.h"
#include "xil_types.h"
```

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

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


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


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


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

static XIntc_Config *LookupConfigByBaseAddress(u32 BaseAddress);

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

/*****************************************************************************/
/**
* This is the interrupt handler for the driver interface provided in this file
* when there can be no argument passed to the handler. In this case, we just
* use the globally defined device ID for the interrupt controller. This function
* is provided mostly for backward compatibility. The user should use
* XIntc_DeviceInterruptHandler() if possible.
* This function does not support multiple interrupt controller instances to be
* handled.
* The user must connect this function to the interrupt system such that it is
* called whenever the devices which are connected to it cause an interrupt.
* @return       None.
* @note
* The constant XPAR_INTC_SINGLE_DEVICE_ID must be defined for this handler
* to be included in the driver compilation.
******************************************************************************/
#ifdef XPAR_INTC_SINGLE_DEVICE_ID
void XIntc_LowLevelInterruptHandler(void)
   {
/* A level of indirection here because the interrupt handler used with
 * the driver interface given in this file needs to remain void - no
 * arguments.  So we need the globally defined device ID of THE
 * interrupt controller. */
   XIntc_DeviceInterruptHandler((void *) XPAR_INTC_SINGLE_DEVICE_ID);
   }
#endif

/*****************************************************************************/
/**
* 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 {XIntc_SetOptions()} function or
* the {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.
* @param        DeviceId is the zero-based device ID defined in xparameters.h
*               of the interrupting interrupt controller. It is used as a direct
*               index into the configuration data, which contains the vector
*               table for the interrupt controller. Note that even though the
*               argument is a void pointer, the value is not a pointer but the
*               actual device ID.  The void pointer type is necessary to meet
*               the XInterruptHandler typedef for interrupt handlers.
* @return       None.
* @note
* The constant XPAR_INTC_MAX_NUM_INTR_INPUTS must be setup for this to compile.
* Interrupt IDs range from 0 - 31 and correspond to the interrupt input signals
* for the interrupt controller. XPAR_INTC_MAX_NUM_INTR_INPUTS specifies the
* highest numbered interrupt input signal that is used.
*
******************************************************************************/
void XIntc_DeviceInterruptHandler(void *DeviceId)
```

```
   {
   u32 IntrStatus;
   u32 IntrMask = 1;
   int IntrNumber;
   volatile u32 Register;                       /* used as bit bucket */
   XIntc_Config *CfgPtr;

/* Get the configuration data using the device ID */
   CfgPtr = &XIntc_ConfigTable[(u32) DeviceId];

/* Get the interrupts that are waiting to be serviced */
   IntrStatus = XIntc_GetIntrStatus(CfgPtr->BaseAddress);

/* Service each interrupt that is active and enabled by checking each
 * bit in the register from LSB to MSB which corresponds to an interrupt
 * intput signal */
   for (IntrNumber = 0; IntrNumber < XPAR_INTC_MAX_NUM_INTR_INPUTS; IntrNumber++)
      {
      if (IntrStatus & 1)
         {
         XIntc_VectorTableEntry *TablePtr;

/* If the interrupt has been setup to acknowledge it before servicing the interrupt,
   then ack it */
         if (CfgPtr->AckBeforeService & IntrMask)
            { XIntc_AckIntr(CfgPtr->BaseAddress, IntrMask); }

/* The interrupt is active and enabled, call the interrupt handler that was setup with
   the specified parameter */
         TablePtr = &(CfgPtr->HandlerTable[IntrNumber]);
         TablePtr->Handler(TablePtr->CallBackRef);

/* If the interrupt has been setup to acknowledge it after it has been serviced then ack it */
         if ((CfgPtr->AckBeforeService & IntrMask) == 0)
            { XIntc_AckIntr(CfgPtr->BaseAddress, IntrMask); }

/* Read the ISR again to handle architectures with posted write bus access issues.  */
         Register = XIntc_GetIntrStatus(CfgPtr->BaseAddress);

/* If only the highest priority interrupt is to be serviced, exit loop and return after servicing
 * the interrupt */
         if (CfgPtr->Options == XIN_SVC_SGL_ISR_OPTION)
            { return; }
         }

/* Move to the next interrupt to check */
      IntrMask <<= 1;
      IntrStatus >>= 1;

/* If there are no other bits set indicating that all interrupts have been serviced, then exit
   the loop */
      if (IntrStatus == 0)
         { break; }
      }
   }

/****************************************************************************/
/**
* 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 the driver interface given in xintc.h file is to service
* only a single interrupt, whereas the default behavior when using the driver
* interface given in this file is to service all outstanding interrupts when an
* interrupt occurs.
* @param        BaseAddress is the unique identifier for a device.
* @param        Option is 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.
* @return       None.
* @note
* Note that this function has no effect if the input base address is invalid.
*****************************************************************************/
void XIntc_SetIntrSvcOption(u32 BaseAddress, int Option)
   {
   XIntc_Config *CfgPtr;
```

```
   CfgPtr = LookupConfigByBaseAddress(BaseAddress);
   if (CfgPtr != NULL)
      { CfgPtr->Options = Option; }
   }

/****************************************************************************/
/**
* 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 NULL as the
* callback reference.
* @param        BaseAddress is the base address of the interrupt controller
*               whose vector table will be modified.
* @param        InterruptId is the interrupt ID to be associated with the input
*               handler.
* @param        Handler is the function pointer that will be added to
*               the vector table for the given interrupt ID.
* @param        CallBackRef is the argument that will be passed to the new
*               handler function when it is called. This is user-specific.
* @return       None.
* @note
* Note that this function has no effect if the input base address is invalid.
****************************************************************************/
void XIntc_RegisterHandler(u32 BaseAddress, int InterruptId, XInterruptHandler Handler,
   void *CallBackRef)
   {
   XIntc_Config *CfgPtr;

   CfgPtr = LookupConfigByBaseAddress(BaseAddress);
   if (CfgPtr != NULL)
      {
      CfgPtr->HandlerTable[InterruptId].Handler = Handler;
      CfgPtr->HandlerTable[InterruptId].CallBackRef = CallBackRef;
      }
   }

/****************************************************************************/
/**
* Looks up the device configuration based on the base address of the device.
* A table contains the configuration info for each device in the system.
* @param        BaseAddress is the unique identifier for a device.
* @return
* A pointer to the configuration structure for the specified device, or
* NULL if the device was not found.
* @note         None.
****************************************************************************/
static XIntc_Config *LookupConfigByBaseAddress(u32 BaseAddress)
   {
   XIntc_Config *CfgPtr = NULL;
   int Index;

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

   return CfgPtr;
   }
```