

Xilinx Device Drivers

[Driver Summary](#) [Copyright](#)

[Main Page](#) [Data Structures](#) [File List](#) [Data Fields](#) [Globals](#)

Xilinx Device Drivers Documentation

Generated on 24 Jun 2004 for Xilinx Device Drivers

Device Driver Summary

A summary of each device driver is provided below. This includes links to the driver's layer 1, high-level header file and its layer 0, low-level header file. A description of the device driver layers can be found in the Device Driver Programmer Guide. In addition, [building block](#) components are described, followed by a list of layer 2 drivers/adapters available for the [VxWorks](#) Real-Time Operating System (RTOS).

Example usage files for each driver can be found in the driver's directory in the EDK installation area under *sw/XilinxProcessorIPLib/drivers/.../examples*.

ATM Controller

The Asynchronous Transfer Mode (ATM) Controller driver resides in the *atmc* subdirectory. Details of the layer 1 high level driver can be found in the [xatmc.h](#) header file. Details of the layer 0 low level driver can be found in the [xatmc_1.h](#) header file.

DDR ECC

The DDR ECC driver resides in the *ddr* subdirectory. Details of the layer 1 high level driver can be found in the [xddr.h](#) header file. Details of the layer 0 low level driver can be found in the [xddr_1.h](#) header file.

Delta-Sigma ADC

The Delta-Sigma Analog-to-Digital Converter (ADC) driver resides in the *dsadc* subdirectory. Details of the layer 1 high level driver can be found in the [xdsadc.h](#) header file. Details of the layer 0 low level driver can be found in the [xdsadc_1.h](#) header file.

Delta-Sigma DAC

The Delta-Sigma Digital-to-Analog Converter (DAC) driver resides in the *dsdac* subdirectory. Details of the layer 1 high level driver can be found in the [xdsdac.h](#) header file. Details of the layer 0 low level driver can be found in the [xdsdac_1.h](#) header file.

Central DMA

The Central DMA driver resides in the *dmacentral* subdirectory. Details of the layer 1 high

level driver can be found in the [xdmcentral.h](#) header file. Details of the layer 0 low level driver can be found in the [xdmcentral_1.h](#) header file.

Ethernet 10/100 MAC

The Ethernet 10/100 MAC driver resides in the *emac* subdirectory. Details of the layer 1 high level driver can be found in the [xemac.h](#) header file. Details of the layer 0 low level driver can be found in the [xemac_1.h](#) header file.

Ethernet 10/100 MAC Lite

The Ethernet 10/100 MAC Lite driver resides in the *emac_lite* subdirectory. Details of the layer 1 high level driver can be found in the [xemaclite.h](#) header file. Details of the layer 0 low level driver can be found in the [xemaclite_1.h](#) header file.

External Memory Controller

The External Memory Controller driver resides in the *emc* subdirectory. Details of the layer 1 high level driver can be found in the [xemc.h](#) header file. Details of the layer 0 low level driver can be found in the [xemc_1.h](#) header file.

General Purpose I/O

The General Purpose I/O driver resides in the *gpio* subdirectory. Details of the layer 1 high level driver can be found in the [xgpio.h](#) header file. Details of the layer 0 low level driver can be found in the [xgpio_1.h](#) header file.

Gigabit Ethernet MAC

The 1 Gigabit Ethernet MAC driver resides in the *gemac* subdirectory. Details of the layer 1 high level driver can be found in the [xgemac.h](#) header file. Details of the layer 0 low level driver can be found in the [xgemac_1.h](#) header file.

Hardware ICAP

The Hardware ICAP driver resides in the *hwicap* subdirectory. Details of the layer 1 high level driver can be found in the [xhwicap.h](#) header file. Details of the layer 0 low level driver can be found in the [xhwicap_1.h](#) header file.

HDLC

The HDLC driver resides in the *hdlc* subdirectory. Details of the layer 1 high level driver can be found in the [xhdlc.h](#) header file. Details of the layer 0 low level driver can be found in the [xhdlc_1.h](#) header file.

Intel StrataFlash

The Intel StrataFlash driver resides in the *flash* subdirectory. Details of the layer 1 high level driver can be found in the [xflash.h](#) header file. Details of the layer 0 low level driver can be found in the [xflash_intel_1.h](#) header file.

Inter-Integrated Circuit (IIC)

The IIC driver resides in the *iic* subdirectory. Details of the layer 1 high level driver can be found in the [xiic.h](#) header file. Details of the layer 0 low level driver can be found in the [xiic_1.h](#) header file.

Interrupt Controller

The Interrupt Controller driver resides in the *intc* subdirectory. Details of the layer 1 high level driver can be found in the [xintc.h](#) header file. Details of the layer 0 low level driver can be found in the [xintc_1.h](#) header file.

OPB Arbiter

The OPB Arbiter driver resides in the *opbarb* subdirectory. Details of the layer 1 high level driver can be found in the [xopbarb.h](#) header file. Details of the layer 0 low level driver can be found in the [xopbarb_1.h](#) header file.

OPB to PLB Bridge

The OPB to PLB bridge driver resides in the *opb2plb* subdirectory. Details of the layer 1 high level driver can be found in the [xopb2plb.h](#) header file. Details of the layer 0 low level driver can be found in the [xopb2plb_1.h](#) header file.

PCI Bridge

The PCI bridge driver resides in the *pci* subdirectory. Details of the layer 1 high level driver can be found in the [xpci.h](#) header file. Details of the layer 0 low level driver can be found in the

[xpci_1.h](#) header file.

PLB Arbiter

The PLB arbiter driver resides in the *plbarb* subdirectory. Details of the layer 1 high level driver can be found in the [xplbarb.h](#) header file. Details of the layer 0 low level driver can be found in the [xplbarb_1.h](#) header file.

PLB to OPB Bridge

The PLB to OPB bridge driver resides in the *plb2opb* subdirectory. Details of the layer 1 high level driver can be found in the [xplb2opb.h](#) header file. Details of the layer 0 low level driver can be found in the [xplb2opb_1.h](#) header file.

Rapid I/O

The Rapid I/O driver resides in the *rapidio* subdirectory. Details of the layer 0 low level driver can be found in the [xrapidio_1.h](#) header file.

Serial Peripheral Interface (SPI)

The SPI driver resides in the *spi* subdirectory. Details of the layer 1 high level driver can be found in the [xspi.h](#) header file. Details of the layer 0 low level driver can be found in the [xspi_1.h](#) header file.

System ACE

The System ACE driver resides in the *sysace* subdirectory. Details of the layer 1 high level driver can be found in the [xsysace.h](#) header file. Details of the layer 0 low level driver can be found in the [xsysace_1.h](#) header file.

Timer/Counter

The Timer/Counter driver resides in the *tmrctr* subdirectory. Details of the layer 1 high level driver can be found in the [xtmrctr.h](#) header file. Details of the layer 0 low level driver can be found in the [xtmrctr_1.h](#) header file.

UART Lite

The UART Lite driver resides in the *uartlite* subdirectory. Details of the layer 1 high level driver can be found in the [xuartlite.h](#) header file. Details of the layer 0 low level driver can be found in the [xuartlite_1.h](#) header file.

UART 16450/16550

The UART 16450/16550 driver resides in the *uartns550* subdirectory. Details of the layer 1 high level driver can be found in the [xuartns550.h](#) header file. Details of the layer 0 low level driver can be found in the [xuartns550_1.h](#) header file.

Watchdog Timer/Timebase

The Watchdog Timer/Timebase driver resides in the *wdttb* subdirectory. Details of the layer 1 high level driver can be found in the [xwdttb.h](#) header file. Details of the layer 0 low level driver can be found in the [xwdttb_1.h](#) header file.

Building Block Components

Common

Common components reside in the *common* subdirectory and comprise a collection of header files and ".c" files that are commonly used by all device drivers and application code. Included in this collection are: [xstatus.h](#) , which contains the identifiers for Xilinx status codes; [xparameters.h](#) , which contains the identifiers for the driver configurations and memory map; and [xbasic_types.h](#) , which contains identifiers for primitive data types and commonly used constants.

CPU/CPU_PPC405

CPU components reside in the *cpu[_ppc405]* subdirectory and comprise I/O functions specific to a processor. These I/O functions are defined in *xio.h*. These functions are used by drivers and are not intended for external use.

IPIF

IPIF components reside in the *ipif* subdirectory and comprise functions related to the IP Interface (IPIF) interrupt control logic. Since most devices are built with IPIF, drivers utilize this common source code to prevent duplication of code within the drivers. These functions are used by drivers and are not intended for external use.

DMA

DMA components reside in the *dma* subdirectory and comprise functions used for Direct Memory Access (DMA). Simple DMA, scatter-gather DMA, and multi-channel DMA are supported.

Packet FIFO

Packet FIFO components reside in the *packet_fifo* subdirectory and comprise functions used for packet FIFO control. Packet FIFOs are typically used by devices that process and potentially retransmit packets, such as Ethernet and ATM. These functions are used by drivers and are not intended for external use.

Channel FIFO

Channel FIFO components reside in the *channel_fifo* subdirectory and comprise functions used for channelized FIFO control. Channel FIFOs are typically used by devices that support multi-channel processing, such as multi-channel HDLC. These functions are used by drivers and are not intended for external use.

VxWorks Support

VxWorks RTOS adapters (also known as layer 2 device drivers) are provided for the following devices:

- 10/100 Ethernet MAC (Enhanced Network Driver Interface)
- 10/100 Ethernet MAC Lite (Enhanced Network Driver Interface)
- Gigabit Ethernet MAC (Enhanced Network Driver Interface)
- UART 16550/16450 (Serial IO Interface)
- UART Lite (Serial IO Interface)
- System ACE (Block Device Interface)
- PCI (PCI memory access and VxWorks PCI library calls)

Keep in mind that all Xilinx device drivers are available to a VxWorks application. The ones listed above have been seamlessly integrated into a standard VxWorks interface.

atmc/v1_00_c/src/xatmc.h File Reference

Detailed Description

The implementation of the [XAtmc](#) component, which is the driver for the Xilinx ATM controller.

The Xilinx ATM controller supports the following features:

- Simple and scatter-gather DMA operations, as well as simple memory mapped direct I/O interface (FIFOs).
- Independent internal transmit and receive FIFOs
- Internal loopback
- Header error check (HEC) generation and checking
- Cell buffering with or without header/User Defined
- Parity generation and checking
- Header generation for transmit cell payloads
- Physical interface (PHY) data path of 16 bits
- Basic statistics gathering such as long cells, short cells, parity errors, and HEC errors

The driver does not support all of the features listed above. Features not currently supported by the driver are:

- Simple DMA (in polled or interrupt mode)
- Direct I/O (FIFO) operations in interrupt mode (polled mode does use the FIFO directly)

It is the responsibility of the application get the interrupt handler of the ATM controller and connect it to the interrupt source.

The driver services interrupts and passes ATM cells to the upper layer software through callback functions. The upper layer software must register its callback functions during initialization. The driver requires callback functions for received cells, for confirmation of transmitted cells, and for asynchronous errors. The frequency of interrupts can be controlled with the packet threshold and packet wait bound features of the scatter-gather DMA engine.

The callback function which performs processing for scatter-gather DMA is executed in an interrupt context and is designed to allow the processing of the scatter-gather list to be passed to a thread context. The scatter-gather processing can require more processing than desired in an interrupt context. Functions are provided to be called from the callback function or thread context to get cells from the send and receive scatter-gather list.

Some errors that can occur in the device require a device reset. These errors are listed in the `SetErrorHandler` function header. The upper layer's error handler is responsible for resetting the device and re-configuring it based on its needs (the driver does not save the current configuration).

DMA Support

The Xilinx ATMC device is available for both the IBM On-Chip Peripheral Bus (OPB) and Processor Local Bus (PLB). This driver works for both. However, a current limitation of the ATMC device on the PLB is that it does not support DMA. For this reason, the DMA scatter-gather functions (e.g., [XAtmc_SgSend\(\)](#)) of this driver will not function for the PLB version of the ATMC device.

Note:

Xilinx drivers are typically composed of two components, one is the driver and the other is the adapter. The driver is independent of OS and processor and is intended to be highly portable. The adapter is OS-specific and facilitates communication between the driver and the OS.

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	JHL	07/31/01	First release
1.00c	rpm	01/08/03	New release supports v2.00a of packet fifo driver an v1.23b of the IPIF driver

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xdma_channel.h"  
#include "xbasic_types.h"  
#include "xpacket_fifo_v2_00_a.h"
```

Data Structures

```
struct XAtmc  
struct XAtmc_Config  
struct XAtmc_Stats
```

Configuration options

These options are used in [XAtmc_SetOptions\(\)](#) to configure the device.

```
#define XAT_LOOPBACK_OPTION  
#define XAT_POLLED_OPTION  
#define XAT_DISCARD_SHORT_OPTION
```

```

#define XAT_DISCARD_PARITY_OPTION
#define XAT_DISCARD_LONG_OPTION
#define XAT_DISCARD_HEC_OPTION
#define XAT_DISCARD_VXI_OPTION
#define XAT_PAYLOAD_ONLY_OPTION
#define XAT_NO_SEND_PARITY_OPTION

```

Cell status

These constants define the status values for a received cell. The status is available when polling to receive a cell or in the buffer descriptor after a cell is received using DMA scatter-gather.

```

#define XAT_CELL_STATUS_LONG
#define XAT_CELL_STATUS_SHORT
#define XAT_CELL_STATUS_BAD_PARITY
#define XAT_CELL_STATUS_BAD_HEC
#define XAT_CELL_STATUS_VXI_MISMATCH
#define XAT_CELL_STATUS_NO_ERROR

```

Typedefs for callbacks

Callback functions.

```

typedef void(* XAtmc_SgHandler )(void *CallBackRef, Xuint32 CellCount)
typedef void(* XAtmc_ErrorHandler )(void *CallBackRef, XStatus ErrorCode)

```

Functions

```
XStatus XAtmc_Initialize (XAtmc *InstancePtr, Xuint16 DeviceId)
```

```
XStatus XAtmc_Start (XAtmc *InstancePtr)
```

```
XStatus XAtmc_Stop (XAtmc *InstancePtr)
```

```
void XAtmc_Reset (XAtmc *InstancePtr)
```

```
XStatus XAtmc_SelfTest (XAtmc *InstancePtr)
```

```
XAtmc_Config * XAtmc_LookupConfig (Xuint16 DeviceId)
```

```
XStatus XAtmc_SgSend (XAtmc *InstancePtr, XBufDescriptor *BdPtr)
```

```
XStatus XAtmc_SgRecv (XAtmc *InstancePtr, XBufDescriptor *BdPtr)
```

```
XStatus XAtmc_SgGetSendCell (XAtmc *InstancePtr, XBufDescriptor **PtrToBdPtr, int *BdCountPtr)
```

```
XStatus XAtmc_SgGetRecvCell (XAtmc *InstancePtr, XBufDescriptor **PtrToBdPtr, int *BdCountPtr)
```

```
XStatus XAtmc_PollSend (XAtmc *InstancePtr, Xuint8 *BufPtr, Xuint32 ByteCount)
```

```
XStatus XAtmc_PollRecv (XAtmc *InstancePtr, Xuint8 *BufPtr, Xuint32 *ByteCountPtr, Xuint32
*CellStatusPtr)
```

```
XStatus XAtmc_SetOptions (XAtmc *InstancePtr, Xuint32 Options)
```

```
Xuint32 XAtmc_GetOptions (XAtmc *InstancePtr)
```

```
XStatus XAtmc_SetPhyAddress (XAtmc *InstancePtr, Xuint8 Address)
```

Xuint8 XAtmc_GetPhyAddress (**XAtmc** *InstancePtr)
XStatus XAtmc_SetHeader (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint32** Header)
Xuint32 XAtmc_GetHeader (**XAtmc** *InstancePtr, **Xuint32** Direction)
XStatus XAtmc_SetUserDefined (**XAtmc** *InstancePtr, **Xuint8** UserDefined)
Xuint8 XAtmc_GetUserDefined (**XAtmc** *InstancePtr)
XStatus XAtmc_SetPktThreshold (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint8** Threshold)
XStatus XAtmc_GetPktThreshold (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint8** *ThreshPtr)
XStatus XAtmc_SetPktWaitBound (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint32** TimerValue)
XStatus XAtmc_GetPktWaitBound (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint32** *WaitPtr)
void **XAtmc_GetStats** (**XAtmc** *InstancePtr, **XAtmc_Stats** *StatsPtr)
void **XAtmc_ClearStats** (**XAtmc** *InstancePtr)
XStatus XAtmc_SetSgRecvSpace (**XAtmc** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)
XStatus XAtmc_SetSgSendSpace (**XAtmc** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)
void **XAtmc_InterruptHandler** (void *InstancePtr)
void **XAtmc_SetSgRecvHandler** (**XAtmc** *InstancePtr, void *CallBackRef, **XAtmc_SgHandler** FuncPtr)
void **XAtmc_SetSgSendHandler** (**XAtmc** *InstancePtr, void *CallBackRef, **XAtmc_SgHandler** FuncPtr)
void **XAtmc_SetErrorHandler** (**XAtmc** *InstancePtr, void *CallBackRef, **XAtmc_ErrorHandler** FuncPtr)

Define Documentation

#define

XAT_CELL_STATUS_BAD_HEC

XAT_CELL_STATUS_LONG	Cell was too long
XAT_CELL_STATUS_SHORT	Cell was too short
XAT_CELL_STATUS_BAD_PARITY	Cell parity was not correct
XAT_CELL_STATUS_BAD_HEC	Cell HEC was not correct
XAT_CELL_STATUS_VXI_MISMATCH	Cell VPI/VCI fields didn't match the expected header values
XAT_CELL_STATUS_NO_ERROR	Cell received without errors

#define

XAT_CELL_STATUS_BAD_PARITY

XAT_CELL_STATUS_LONG	Cell was too long
XAT_CELL_STATUS_SHORT	Cell was too short
XAT_CELL_STATUS_BAD_PARITY	Cell parity was not correct
XAT_CELL_STATUS_BAD_HEC	Cell HEC was not correct
XAT_CELL_STATUS_VXI_MISMATCH	Cell VPI/VCI fields didn't match the expected header values
XAT_CELL_STATUS_NO_ERROR	Cell received without errors

#define

XAT_CELL_STATUS_LONG

XAT_CELL_STATUS_LONG	Cell was too long
XAT_CELL_STATUS_SHORT	Cell was too short
XAT_CELL_STATUS_BAD_PARITY	Cell parity was not correct
XAT_CELL_STATUS_BAD_HEC	Cell HEC was not correct
XAT_CELL_STATUS_VXI_MISMATCH	Cell VPI/VCI fields didn't match the expected header values
XAT_CELL_STATUS_NO_ERROR	Cell received without errors

#define

XAT_CELL_STATUS_NO_ERROR

XAT_CELL_STATUS_LONG	Cell was too long
XAT_CELL_STATUS_SHORT	Cell was too short
XAT_CELL_STATUS_BAD_PARITY	Cell parity was not correct
XAT_CELL_STATUS_BAD_HEC	Cell HEC was not correct
XAT_CELL_STATUS_VXI_MISMATCH	Cell VPI/VCI fields didn't match the expected header values
XAT_CELL_STATUS_NO_ERROR	Cell received without errors

#define

XAT_CELL_STATUS_SHORT

XAT_CELL_STATUS_LONG	Cell was too long
XAT_CELL_STATUS_SHORT	Cell was too short
XAT_CELL_STATUS_BAD_PARITY	Cell parity was not correct
XAT_CELL_STATUS_BAD_HEC	Cell HEC was not correct
XAT_CELL_STATUS_VXI_MISMATCH	Cell VPI/VCI fields didn't match the expected header values
XAT_CELL_STATUS_NO_ERROR	Cell received without errors

#define**XAT_CELL_STATUS_VXI_MISMATCH**

XAT_CELL_STATUS_LONG	Cell was too long
XAT_CELL_STATUS_SHORT	Cell was too short
XAT_CELL_STATUS_BAD_PARITY	Cell parity was not correct
XAT_CELL_STATUS_BAD_HEC	Cell HEC was not correct
XAT_CELL_STATUS_VXI_MISMATCH	Cell VPI/VCI fields didn't match the expected header values
XAT_CELL_STATUS_NO_ERROR	Cell received without errors

#define**XAT_DISCARD_HEC_OPTION**

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define**XAT_DISCARD_LONG_OPTION**

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define**XAT_DISCARD_PARITY_OPTION**

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define

XAT_DISCARD_SHORT_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define

XAT_DISCARD_VXI_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define

XAT_LOOPBACK_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define

XAT_NO_SEND_PARITY_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define

XAT_PAYLOAD_ONLY_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define

XAT_POLLED_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

Typedef Documentation

```
typedef void(* XAtmc_ErrorHandler)(void *CallbackRef, XStatus
ErrorCode)
```

Callback when data is sent or received with scatter-gather DMA.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

ErrorCode indicates the error that occurred.

```
typedef void(* XAtmc_SgHandler)(void *CallbackRef, Xuint32
CellCount)
```

Callback when data is sent or received with scatter-gather DMA.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

CellCount is the number of cells sent or received.

Function Documentation

```
void XAtmc_ClearStats ( XAtmc * InstancePtr )
```

Clears the **XAtmc_Stats** structure for this driver.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint32 ( XAtmc *  
XAtmc_GetHeader InstancePtr,  
 Xuint32 Direction  
 )
```

Gets the send or receive ATM header in the ATM controller. The ATM controller attaches the send header to cells which are to be sent but contain only the payload.

If the ATM controller is configured appropriately, it will compare the header of received cells against the receive header and discard cells which don't match in the VCI and VPI fields of the header.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates whether we're retrieving the send header or the receive header.

Returns:

The ATM header currently being used by the ATM controller for attachment to transmitted cells or the header which is being compared against received cells. An invalid specified direction will cause this function to return a value of 0.

Note:

None.

```
Xuint32 ( XAtmc *  
XAtmc_GetOptions InstancePtr  
 )
```

Gets Atmc driver/device options. The value returned is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

The 32-bit value of the Atmc options. The value is a bit-mask representing all options that are currently enabled. See **xatmc.h** for a detailed description of the options.

Note:

None.

```
Xuint8 ( XAtmc * )  
XAtmc_GetPhyAddress InstancePtr
```

Gets the PHY address for this driver/device.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

The 5-bit PHY address (0 - 31) currently being used by the ATM controller.

Note:

None.

```
XStatus ( XAtmc *  
XAtmc_GetPktThreshold InstancePtr,  
 Xuint32 Direction,  
 Xuint8 * ThreshPtr  
 )
```

Gets the value of the packet threshold register for this driver/device. The packet threshold is used for interrupt coalescing when the ATM controller is configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

ThreshPtr is a pointer to the byte into which the current value of the packet threshold register will be copied. An output parameter. A value of 0 indicates the use of packet threshold by the hardware is disabled.

Returns:

- XST_SUCCESS if the packet threshold was retrieved successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
XStatus ( XAtmc *  
XAtmc_GetPktWaitBound InstancePtr,  
 Xuint32 Direction,  
 Xuint32 * WaitPtr  
 )
```

Gets the packet wait bound register for this driver/device. The packet wait bound is used for interrupt coalescing when the ATM controller is configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

WaitPtr is a pointer to the byte into which the current value of the packet wait bound register will be copied. An output parameter. Units are in milliseconds in the range 0 - 1023. A value of 0 indicates the packet wait bound timer is disabled.

Returns:

- XST_SUCCESS if the packet wait bound was retrieved successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
void  
XAtmc_GetStats ( XAtmc *  
                InstancePtr,  
                XAtmc_Stats * StatsPtr  
                )
```

Gets a copy of the **XAtmc_Stats** structure, which contains the current statistics for this driver.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None. Although the output parameter will contain a copy of the statistics upon return from this function.

Note:

None.

```
Xuint8  
XAtmc_GetUserDefined ( XAtmc *  
                      InstancePtr
```

Gets the 2nd byte of the User Defined data in the ATM controller for the channel which is sending data. The ATM controller will attach the header to all cells which are being sent and do not have a header. The header of a 16 bit Utopia interface contains the User Defined data which is two bytes. The first byte contains the HEC field and the second byte is available for user data. This function only allows the second byte to be retrieved.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

The second byte of the User Defined data.

Note:

None.

```
XStatus  
XAtmc_Initialize ( XAtmc *  
                  InstancePtr,  
                  Xuint16 DeviceId  
                  )
```

Initializes a specific ATM controller instance/driver. The initialization entails:

- Initialize fields of the **XAtmc** structure
- Clear the ATM statistics for this device
- Initialize the IPIF component with its register base address
- Configure the FIFO components with their register base addresses.
- Configure the DMA channel components with their register base addresses. At some later time, memory pools for the scatter-gather descriptor lists will be passed to the driver.
- Reset the ATM controller

The only driver function that should be called before this Initialize function is called is GetInstance.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XAtmc** instance. Passing in a device id associates the generic **XAtmc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started

Note:

None.

```
void XAtmc_InterruptHandler ( void * InstancePtr )
```

Interrupt handler for the ATM controller driver. It performs the following processing:

- Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: the ATM controller, the send packet FIFO, the receive packet FIFO, the send DMA channel, or the receive DMA channel. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the ATM controller.
- Call the appropriate handler based on the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the ATMC controller instance for the interrupt.

Returns:

None.

Note:

None.

XAtmc_Config* (**Xuint16**)
XAtmc_LookupConfig *DeviceId*

Looks up the device configuration based on the unique device ID. The table `AtmcConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId contains the unique device ID that for the device. This ID is used to lookup the configuration.

Returns:

A pointer to the configuration for the specified device, or `XNULL` if the device could not be found.

Note:

None.

XStatus (**XAtmc** * *InstancePtr*,
XAtmc_PollRecv **Xuint8** * *BufPtr*,
Xuint32 * *ByteCountPtr*,
Xuint32 * *CellStatusPtr*)

Receives an ATM cell in polled mode. The device/driver must be in polled mode before calling this function. The driver receives the cell directly from the ATM controller packet FIFO. This is a non-blocking receive, in that if there is no cell ready to be received at the device, the function returns with an error. The buffer into which the cell will be received must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer into which the received `Atmc` cell will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to the size of the buffer on entry into the function and the size the received cell on return from the function.

CellStatusPtr is both an input and an output parameter. It is a pointer to the status of the cell which is received. It is only valid if the return value indicates success. The status is necessary when cells with errors are not being discarded. This status is a bit mask which may contain one or more of the following values with the exception of `XAT_CELL_STATUS_NO_ERROR` which is mutually exclusive. The status values are:

- `XAT_CELL_STATUS_NO_ERROR` indicates the cell was received without any errors
- `XAT_CELL_STATUS_BAD_PARITY` indicates the cell parity was not correct
- `XAT_CELL_STATUS_BAD_HEC` indicates the cell HEC was not correct
- `XAT_CELL_STATUS_SHORT` indicates the cell was not the correct length

- XAT_CELL_STATUS_VXI_MISMATCH indicates the cell VPI/VCI fields did not match the expected header values

Returns:

- XST_SUCCESS if the cell was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_NO_DATA if there is no cell to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the cell is too small for the cell waiting in the FIFO.

Note:

The input buffer must be big enough to hold the largest ATM cell. The buffer must also be 32-bit aligned.

```

XStatus          ( XAtmc *
XAtmc_PollSend   InstancePtr,
                   Xuint8 * BufPtr,
                   Xuint32 ByteCount
                   )

```

Sends an ATM cell in polled mode. The device/driver must be in polled mode before calling this function. The driver writes the cell directly to the ATM controller packet FIFO, then enters a loop checking the device status for completion or error. The buffer to be sent must be word-aligned.

It is assumed that the upper layer software supplies a correctly formatted ATM cell based upon the configuration of the ATM controller (attaching header or not).

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer containing the ATM cell to be sent.

ByteCount is the size of the ATM cell. An ATM cell for a 16 bit Utopia interface is 54 bytes with a 6 byte header and 48 bytes of payload. This function may be used to send short cells with or without headers depending on the configuration of the ATM controller.

Returns:

- XST_SUCCESS if the cell was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_PFIFO_NO_ROOM if there is no room in the FIFO for this cell
- XST_FIFO_ERROR if the FIFO was overrun or underrun

Note:

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PollSend thread.

The input buffer must be big enough to hold the largest ATM cell. The buffer must also be 32-bit aligned.

```
void ( XAtmc * )  
XAtmc_Reset InstancePtr
```

Resets the ATM controller. It resets the the DMA channels, the FIFOs, and the ATM controller. The reset does not remove any of the buffer descriptors from the scatter-gather list for DMA. Reset must only be called after the driver has been initialized.

The configuration after this reset is as follows:

- Disabled transmitter and receiver
- Default packet threshold and packet wait bound register values for scatter-gather DMA operation
- PHY address of 0

The upper layer software is responsible for re-configuring (if necessary) and restarting the ATM controller after the reset.

When a reset is required due to an internal error, the driver notifies the upper layer software of this need through the ErrorHandler callback and specific status codes. The upper layer software is responsible for calling this Reset function and then re-configuring the device.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

None.

Note:

The reset is accomplished by setting the IPIF reset register. This takes care of resetting all hardware blocks, including the ATM controller.

```
XStatus ( XAtmc * )  
XAtmc_SelfTest InstancePtr
```

Performs a self-test on the ATM controller device. The test includes:

- Run self-test on DMA channel, FIFO, and IPIF components
- Reset the ATM controller device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run.

It should be noted that data caching must be disabled when this function is called because the DMA self-test uses two local buffers (on the stack) for the transfer test.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

- XST_SUCCESS if self-test was successful
- XST_PFIFO_BAD_REG_VALUE if the FIFO failed register self-test
- XST_DMA_TRANSFER_ERROR if DMA failed data transfer self-test
- XST_DMA_RESET_REGISTER_ERROR if DMA control register value was incorrect after a reset
- XST_REGISTER_ERROR if the ATM controller failed register reset test
- XST_LOOPBACK_ERROR if the ATM controller internal loopback failed
- XST_IPIF_REG_WIDTH_ERROR if an invalid register width was passed into the function
- XST_IPIF_RESET_REGISTER_ERROR if the value of a register at reset was invalid
- XST_IPIF_DEVICE_STATUS_ERROR if a write to the device status register did not read back correctly
- XST_IPIF_DEVICE_ACK_ERROR if a bit in the device status register did not reset when acked
- XST_IPIF_DEVICE_ENABLE_ERROR if the device interrupt enable register was not updated correctly by the hardware when other registers were written to
- XST_IPIF_IP_STATUS_ERROR if a write to the IP interrupt status register did not read back correctly
- XST_IPIF_IP_ACK_ERROR if one or more bits in the IP status register did not reset when acked
- XST_IPIF_IP_ENABLE_ERROR if the IP interrupt enable register was not updated correctly when other registers were written to

Note:

Because this test uses the PollSend function for its loopback testing, there is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the self-test thread.

```

void
XAtmc_SetErrorHandler
    ( XAtmc *
      InstancePtr,
      void *
      CallbackRef,
      XAtmc_ErrorHandler FuncPtr
    )

```

Sets the callback function for handling errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback which should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- XST_DMA_ERROR indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- XST_FIFO_ERROR indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- XST_RESET_ERROR indicates an unrecoverable ATM controller error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- XST_ATMC_ERROR_COUNT_MAX indicates the counters of the ATM controller have reached the maximum value and that the statistics of the ATM controller should be cleared.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```

XStatus
XAtmc_SetHeader
    ( XAtmc *
      InstancePtr,
      Xuint32 Direction,
      Xuint32 Header
    )

```

Sets the send or receive ATM header in the ATM controller. If cells with only payloads are given to the controller to be sent, it will attach the header to the cells. If the ATM controller is configured appropriately, it will compare the header of received cells against the receive header and discard cells which don't match in the VCI and VPI fields of the header.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the direction, send(transmit) or receive, for the header to set.

Header contains the ATM header to be attached to each transmitted cell for cells with only payloads or the expected header for cells which are received.

Returns:

- XST_SUCCESS if the PHY address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
XStatus ( XAtmc *  
XAtmc_SetOptions InstancePtr,  
Xuint32 OptionsFlag  
)
```

Set Atmc driver/device options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option. A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off. See [xatmc.h](#) for a detailed description of the available options.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

OptionsFlag is a bit-mask representing the Atmc options to turn on or off

Returns:

- XST_SUCCESS if options were set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

```
XStatus ( XAtmc *  
XAtmc_SetPhyAddress InstancePtr,  
 Xuint8 Address  
 )
```

Sets the PHY address for this driver/device. The address is a 5-bit value. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Address contains the 5-bit PHY address (0 - 31).

Returns:

- XST_SUCCESS if the PHY address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```
XStatus ( XAtmc *  
XAtmc_SetPktThreshold InstancePtr,  
 Xuint32 Direction,  
 Xuint8 Threshold  
 )
```

Sets the packet count threshold register for this driver/device. The device must be stopped before setting the threshold. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

Threshold is the value of the packet threshold count used during interrupt coalescing. A value of 0 disables the use of packet threshold by the hardware.

Returns:

- XST_SUCCESS if the threshold was successfully set
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_DMA_SG_COUNT_EXCEEDED if the threshold must be equal to or less than the number of descriptors in the list
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
XStatus ( XAtmc *  
XAtmc_SetPktWaitBound InstancePtr,  
 Xuint32 Direction,  
 Xuint32 TimerValue  
 )
```

Sets the packet wait bound register for this driver/device. The device must be stopped before setting the timer value. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold.

The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

TimerValue is the value of the packet wait bound used during interrupt coalescing. It is in milliseconds in the range 0 - 1023. A value of 0 disables the packet wait bound timer.

Returns:

- XST_SUCCESS if the packet wait bound was set successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
void ( XAtmc *  
XAtmc_SetSgRecvHandler InstancePtr,  
 void * CallbackRef,  
 XAtmc_SgHandler FuncPtr  
 )
```

Sets the callback function for handling received cells in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of cells, determined by the DMA scatter-gather packet threshold, are received. The number of received cells is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each received cell from the list and should attach a new buffer to each descriptor. It is important that the specified number of cells passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

CallBackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XAtmc_SetSgRecvSpace(XAtmc *InstancePtr,  
Xuint32 *MemoryPtr,  
Xuint32 ByteCount)
```

Gives the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the Atmc driver. The memory space must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the XAtmc_Initialize function because the DMA channel components must be initialized before the memory space is set.

```
void XAtmc_SetSgSendHandler ( XAtmc * InstancePtr,  
                             void * CallbackRef,  
                             XAtmc_SgHandler FuncPtr  
                             )
```

Sets the callback function for handling confirmation of transmitted cells in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of cells, determined by the DMA scatter-gather packet threshold, are sent. The number of sent cells is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each sent cell from the list and should also free the buffers attached to the descriptors if necessary. It is important that the specified number of cells passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XAtmc_SetSgSendSpace ( XAtmc * InstancePtr,  
                              Xuint32 * MemoryPtr,  
                              Xuint32 ByteCount  
                              )
```

Gives the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the Atmc driver. The memory space must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the XAtmc_Initialize function because the DMA channel components must be initialized before the memory space is set.

```
XStatus XAtmc_SetUserDefined ( XAtmc * InstancePtr, Xuint8 UserDefined )
```

Sets the 2nd byte of the User Defined data in the ATM controller for the channel which is sending data. The ATM controller will attach the header to all cells which are being sent and do not have a header. The header of a 16 bit Utopia interface contains the User Defined data which is two bytes. The first byte contains the HEC field and the second byte is available for user data. This function only allows the second byte to be set.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

UserDefined contains the second byte of the User Defined data.

Returns:

- XST_SUCCESS if the user-defined data was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```

XStatus ( XAtmc *
XAtmc_SgGetRecvCell InstancePtr,
XBufDescriptor ** PtrToBdPtr,
int * BdCountPtr
)

```

Gets the first buffer descriptor of the oldest cell which was received by the scatter-gather DMA channel of the ATM controller. This function is provided to be called from a callback function such that the buffer descriptors for received cells can be processed. The function should be called by the application repetitively for the number of cells indicated as an argument in the callback function. This function may also be used when only payloads are being sent and received by the ATM controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the cell. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the cell. This input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the cell pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- XST_SUCCESS if a descriptor was successfully returned to the driver.
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode.
- XST_DMA_SG_NO_LIST if the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```

XStatus ( XAtmc *
XAtmc_SgGetSendCell InstancePtr,
XBufDescriptor ** PtrToBdPtr,
int * BdCountPtr
)

```

Gets the first buffer descriptor of the oldest cell which was sent by the scatter-gather DMA channel of the ATM controller. This function is provided to be called from a callback function such that the buffer descriptors for sent cells can be processed. The function should be called by the application repetitively for the number of cells indicated as an argument in the callback function. This function may also be used when only payloads are being sent and received by the ATM controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the cell. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the cell. this input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the cell pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- XST_SUCCESS if a descriptor was successfully returned to the driver.
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode.
- XST_DMA_SG_NO_LIST if the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```
XStatus ( XAtmc *  
XAtmc_SgRecv InstancePtr,  
XBufDescriptor * BdPtr  
)
```

Adds this descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be word-aligned. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of cells to replace filled buffers with empty buffers. The contents of the specified buffer descriptor are copied into the scatter-gather transmit list. This function can be called when the device is started or stopped.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list

because a locked descriptor exists at the insert point.

- XST_DMA_SG_NOHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

Note:

None.

```
XStatus          ( XAtmc *  
XAtmc_SgSend    InstancePtr,  
                  XBufDescriptor * BdPtr  
                  )
```

Sends an ATM cell using scatter-gather DMA. The caller attaches the cell to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire ATM cell may or may not be contained within one descriptor. The contents of the buffer descriptor are copied into the scatter-gather transmit list. The caller is responsible for providing mutual exclusion to guarantee that a cell is contiguous in the transmit list. The buffer attached to the descriptor must be word-aligned.

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the cell, the inserts are committed, which means the descriptors for this cell are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted ATM cell based upon the configuration of the ATM controller (attaching header or not). The ATM controller must be started before calling this function.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Returns:

- XST_SUCCESS if the buffer was successfully sent
- XST_DEVICE_IS_STOPPED if the ATM controller has not been started yet
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the descriptor list for the DMA channel is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- XST_DMA_SG_NOHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

XStatus (**XAtmc** *)

XAtmc_Start *InstancePtr*

Starts the ATM controller as follows:

- If not in polled mode enable interrupts
- Enable the transmitter
- Enable the receiver
- Start the DMA channels if the descriptor lists are not empty

It is necessary for the caller to connect the interrupt service routine of the ATM controller to the interrupt source, typically an interrupt controller, and enable the interrupt in the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully
- XST_DEVICE_IS_STARTED if the device is already started
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.
- XST_DMA_SG_LIST_EMPTY iff configured for scatter-gather DMA and no buffer descriptors have been put into the list for the receive channel.

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XStatus (**XAtmc** *)

XAtmc_Stop *InstancePtr*

Stops the ATM controller as follows:

- Stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver
- Disable interrupts if not in polled mode

It is the callers responsibility to disconnect the interrupt handler of the ATM controller from the interrupt source, typically an interrupt controller, and disable the interrupt in the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully

- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to stop the device, the user is required to provide protection of this shared data (typically using a semaphore).

Generated on 24 Jun 2004 for Xilinx Device Drivers

XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS

AND

SOLUTIONS FOR XILINX DEVICES. BY PROVIDING THIS DESIGN, CODE,
OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF

INFRINGEMENT,

AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY

REQUIRE

FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY
WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS

OF

INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND

FITNESS

FOR A PARTICULAR PURPOSE.

(c) Copyright 2003-2004 Xilinx Inc.

All rights reserved.

Xilinx Device Drivers Data Structures

Here are the data structures with brief descriptions:

XAtmc	
XAtmc_Config	
XAtmc_Stats	
XDdr	
XDdr_Stats	
XDmaCentral	
XDmaCentral_Config	
XDmaMulti	
XDmaMulti_SgData	
XDsAdc	
XDsAdc_Config	
XDsDac	
XDsDac_Config	
XEmac	
XEmac_Config	
XEmac_Stats	
XEmacLite	
XEmacLite_Config	
XEmc	
XEmc_Config	
XENV_TIME_STAMP	
XFlash_Config	
XFlashGeometry	
XFlashPartID	
XFlashProgCap	

XFlashProperties	
XFlashTag	
XFlashTiming	
XGemac	
XGemac_Config	
XGemac_HardStats	
XGemac_SoftStats	
XGpio	
XGpio_Config	
XHdlc	
XHdlc_Channel	
XHdlc_ChannelID	
XHdlc_Config	
XHdlc_RegisterOffsets	
XHdlc_Stats	
XHwIcap	
XHwIcap_Config	
Xlic	
Xlic_Config	
XlicStats	
XIntc	
XIntc_Config	
XOpb2Plb	
XOpb2Plb_Config	
XOpbArb	
XOpbArb_Config	
XPacketFifoV200a	
XPci	
XPciArb	
XPciArb_Config	
XPciError	
XPlb2Opb	

XPlb2Opb_Config	
XPlbArb	
XPlbArb_Config	
XSpi	
XSpi_Config	
XSpi_Stats	
XSysAce	
XSysAce_CFPParameters	
XSysAce_Config	
XTmrCtr	
XTmrCtr_Config	
XTmrCtrStats	
XUartLite	
XUartLite_Buffer	
XUartLite_Config	
XUartLite_Stats	
XUartNs550	
XUartNs550_Config	
XUartNs550Format	
XUartNs550Stats	
XWdtTb	
XWdtTb_Config	

Xilinx Device Drivers File List

Here is a list of all documented files with brief descriptions:

atmc/v1_00_c/src/xatmc.c	
atmc/v1_00_c/src/xatmc.h	
atmc/v1_00_c/src/xatmc_cfg.c	
atmc/v1_00_c/src/xatmc_g.c	
atmc/v1_00_c/src/xatmc_i.h	
atmc/v1_00_c/src/xatmc_l.c	
atmc/v1_00_c/src/xatmc_l.h	
common/v1_00_a/src/xbasic_types.c	
common/v1_00_a/src/xbasic_types.h	
common/v1_00_a/src/xenv.h	
common/v1_00_a/src/xenv_linux.h	
common/v1_00_a/src/xenv_none.h	
common/v1_00_a/src/xenv_vxworks.h	
common/v1_00_a/src/xparameters.h	
common/v1_00_a/src/xstatus.h	
common/v1_00_a/src/xutil.h	
common/v1_00_a/src/xutil_memtest.c	
cpu/v1_00_a/src/xio.c	
cpu/v1_00_a/src/xio.h	
cpu_ppc405/v1_00_a/src/xio.c	
cpu_ppc405/v1_00_a/src/xio.h	
cpu_ppc405/v1_00_a/src/xio_dcr.c	
cpu_ppc405/v1_00_a/src/xio_dcr.h	
ddr/v1_00_b/src/xddr.c	
ddr/v1_00_b/src/xddr.h	

ddr/v1_00_b/src/xddr_g.c	
ddr/v1_00_b/src/xddr_intr.c	
ddr/v1_00_b/src/xddr_l.h	
ddr/v1_00_b/src/xddr_selftest.c	
ddr/v1_00_b/src/xddr_stats.c	
dma/v1_00_a/src/xdma_multi.c	
dma/v1_00_a/src/xdma_multi.h	
dma/v1_00_a/src/xdma_multi_i.h	
dma/v1_00_a/src/xdma_multi_sg.c	
dmacentral/v1_00_a/src/xdmacentral.c	
dmacentral/v1_00_a/src/xdmacentral.h	
dmacentral/v1_00_a/src/xdmacentral_g.c	
dmacentral/v1_00_a/src/xdmacentral_intr.c	
dmacentral/v1_00_a/src/xdmacentral_l.h	
dmacentral/v1_00_a/src/xdmacentral_selftest.c	
dsadc/v1_00_a/src/xdsadc.c	
dsadc/v1_00_a/src/xdsadc.h	
dsadc/v1_00_a/src/xdsadc_g.c	
dsadc/v1_00_a/src/xdsadc_intr.c	
dsadc/v1_00_a/src/xdsadc_l.h	
dsadc/v1_00_a/src/xdsadc_selftest.c	
dsdac/v1_00_a/src/xdsdac.c	
dsdac/v1_00_a/src/xdsdac.h	
dsdac/v1_00_a/src/xdsdac_g.c	
dsdac/v1_00_a/src/xdsdac_intr.c	
dsdac/v1_00_a/src/xdsdac_l.h	
dsdac/v1_00_a/src/xdsdac_selftest.c	
emac/v1_00_e/src/xemac.c	
emac/v1_00_e/src/xemac.h	
emac/v1_00_e/src/xemac_g.c	
emac/v1_00_e/src/xemac_i.h	
emac/v1_00_e/src/xemac_intr.c	

emac/v1_00_e/src/xemac_intr_dma.c	
emac/v1_00_e/src/xemac_intr_fifo.c	
emac/v1_00_e/src/xemac_l.c	
emac/v1_00_e/src/xemac_l.h	
emac/v1_00_e/src/xemac_multicast.c	
emac/v1_00_e/src/xemac_options.c	
emac/v1_00_e/src/xemac_phy.c	
emac/v1_00_e/src/xemac_polled.c	
emac/v1_00_e/src/xemac_selftest.c	
emac/v1_00_e/src/xemac_stats.c	
emaclite/v1_01_a/src/xemaclite.c	
emaclite/v1_01_a/src/xemaclite.h	
emaclite/v1_01_a/src/xemaclite_g.c	
emaclite/v1_01_a/src/xemaclite_i.h	
emaclite/v1_01_a/src/xemaclite_intr.c	
emaclite/v1_01_a/src/xemaclite_l.c	
emaclite/v1_01_a/src/xemaclite_l.h	
emaclite/v1_01_a/src/xemaclite_selftest.c	
emc/v1_00_a/src/xemc.c	
emc/v1_00_a/src/xemc.h	
emc/v1_00_a/src/xemc_g.c	
emc/v1_00_a/src/xemc_i.h	
emc/v1_00_a/src/xemc_l.h	
emc/v1_00_a/src/xemc_selftest.c	
flash/v1_00_a/src/xflash.c	
flash/v1_00_a/src/xflash.h	
flash/v1_00_a/src/xflash_cfi.c	
flash/v1_00_a/src/xflash_cfi.h	
flash/v1_00_a/src/xflash_g.c	
flash/v1_00_a/src/xflash_geometry.c	
flash/v1_00_a/src/xflash_geometry.h	
flash/v1_00_a/src/xflash_intel.c	

flash/v1_00_a/src/xflash_intel.h	
flash/v1_00_a/src/xflash_intel_l.c	
flash/v1_00_a/src/xflash_intel_l.h	
flash/v1_00_a/src/xflash_properties.h	
gemac/v1_00_f/src/xgemac.c	
gemac/v1_00_f/src/xgemac.h	
gemac/v1_00_f/src/xgemac_control.c	
gemac/v1_00_f/src/xgemac_g.c	
gemac/v1_00_f/src/xgemac_i.h	
gemac/v1_00_f/src/xgemac_intr.c	
gemac/v1_00_f/src/xgemac_intr_dma.c	
gemac/v1_00_f/src/xgemac_intr_fifo.c	
gemac/v1_00_f/src/xgemac_l.h	
gemac/v1_00_f/src/xgemac_multicast.c	
gemac/v1_00_f/src/xgemac_options.c	
gemac/v1_00_f/src/xgemac_polled.c	
gemac/v1_00_f/src/xgemac_selftest.c	
gemac/v1_00_f/src/xgemac_stats.c	
gpio/v2_00_a/src/xgpio.c	
gpio/v2_00_a/src/xgpio.h	
gpio/v2_00_a/src/xgpio_extra.c	
gpio/v2_00_a/src/xgpio_g.c	
gpio/v2_00_a/src/xgpio_i.h	
gpio/v2_00_a/src/xgpio_intr.c	
gpio/v2_00_a/src/xgpio_l.h	
gpio/v2_00_a/src/xgpio_selftest.c	
hdlc/v2_01_a/src/xhdlc.c	
hdlc/v2_01_a/src/xhdlc.h	
hdlc/v2_01_a/src/xhdlc_dmasg.c	
hdlc/v2_01_a/src/xhdlc_g.c	
hdlc/v2_01_a/src/xhdlc_i.c	
hdlc/v2_01_a/src/xhdlc_i.h	

hdlc/v2_01_a/src/xhdlc_intr_fifo.c	
hdlc/v2_01_a/src/xhdlc_l.c	
hdlc/v2_01_a/src/xhdlc_l.h	
hdlc/v2_01_a/src/xhdlc_options.c	
hdlc/v2_01_a/src/xhdlc_selftest.c	
hdlc/v2_01_a/src/xhdlc_stats.c	
hwicap/v1_00_a/src/xhwicap.h	
hwicap/v1_00_a/src/xhwicap_clb_ff.h	
hwicap/v1_00_a/src/xhwicap_clb_lut.h	
hwicap/v1_00_a/src/xhwicap_clb_srin.v.h	
hwicap/v1_00_a/src/xhwicap_device_read_frame.c	
hwicap/v1_00_a/src/xhwicap_device_write_frame.c	
hwicap/v1_00_a/src/xhwicap_g.c	
hwicap/v1_00_a/src/xhwicap_i.h	
hwicap/v1_00_a/src/xhwicap_l.h	
hwicap/v1_00_a/src/xhwicap_set_configuration.c	
hwicap/v1_00_a/src/xhwicap_srp.c	
iic/v1_01_d/src/xiic.c	
iic/v1_01_d/src/xiic.h	
iic/v1_01_d/src/xiic_g.c	
iic/v1_01_d/src/xiic_i.h	
iic/v1_01_d/src/xiic_intr.c	
iic/v1_01_d/src/xiic_l.c	
iic/v1_01_d/src/xiic_l.h	
iic/v1_01_d/src/xiic_master.c	
iic/v1_01_d/src/xiic_multi_master.c	
iic/v1_01_d/src/xiic_options.c	
iic/v1_01_d/src/xiic_selftest.c	
iic/v1_01_d/src/xiic_slave.c	
iic/v1_01_d/src/xiic_stats.c	
intc/v1_00_c/src/xintc.c	
intc/v1_00_c/src/xintc.h	

intc/v1_00_c/src/xintc_g.c	
intc/v1_00_c/src/xintc_i.h	
intc/v1_00_c/src/xintc_intr.c	
intc/v1_00_c/src/xintc_l.c	
intc/v1_00_c/src/xintc_l.h	
intc/v1_00_c/src/xintc_options.c	
intc/v1_00_c/src/xintc_selftest.c	
opb2plb/v1_00_a/src/xopb2plb.c	
opb2plb/v1_00_a/src/xopb2plb.h	
opb2plb/v1_00_a/src/xopb2plb_g.c	
opb2plb/v1_00_a/src/xopb2plb_i.h	
opb2plb/v1_00_a/src/xopb2plb_l.h	
opb2plb/v1_00_a/src/xopb2plb_selftest.c	
opbarb/v1_02_a/src/xopbarb.c	
opbarb/v1_02_a/src/xopbarb.h	
opbarb/v1_02_a/src/xopbarb_g.c	
opbarb/v1_02_a/src/xopbarb_l.h	
packet_fifo/v2_00_a/src/xpacket_fifo_1_v2_00_a.c	
packet_fifo/v2_00_a/src/xpacket_fifo_1_v2_00_a.h	
packet_fifo/v2_00_a/src/xpacket_fifo_v2_00_a.c	
packet_fifo/v2_00_a/src/xpacket_fifo_v2_00_a.h	
pci/v1_00_b/src/xpci.h	
pci/v1_00_b/src/xpci_config.c	
pci/v1_00_b/src/xpci_g.c	
pci/v1_00_b/src/xpci_intr.c	
pci/v1_00_b/src/xpci_l.h	
pci/v1_00_b/src/xpci_selftest.c	
pci/v1_00_b/src/xpci_v3.c	
pciarb/v1_00_a/src/xpciarb.c	
pciarb/v1_00_a/src/xpciarb.h	
pciarb/v1_00_a/src/xpciarb_g.c	
pciarb/v1_00_a/src/xpciarb_l.h	

pciarb/v1_00_a/src/xpciarb_selftest.c	
plb2opb/v1_00_a/src/xplb2opb.c	
plb2opb/v1_00_a/src/xplb2opb.h	
plb2opb/v1_00_a/src/xplb2opb_g.c	
plb2opb/v1_00_a/src/xplb2opb_i.h	
plb2opb/v1_00_a/src/xplb2opb_l.h	
plb2opb/v1_00_a/src/xplb2opb_selftest.c	
plbarb/v1_01_a/src/xplbarb.c	
plbarb/v1_01_a/src/xplbarb.h	
plbarb/v1_01_a/src/xplbarb_g.c	
plbarb/v1_01_a/src/xplbarb_i.h	
plbarb/v1_01_a/src/xplbarb_l.h	
plbarb/v1_01_a/src/xplbarb_selftest.c	
rapidio/v1_01_a/src/xrapidio_l.c	
rapidio/v1_01_a/src/xrapidio_l.h	
spi/v1_00_b/src/xspi.c	
spi/v1_00_b/src/xspi.h	
spi/v1_00_b/src/xspi_g.c	
spi/v1_00_b/src/xspi_i.h	
spi/v1_00_b/src/xspi_l.h	
spi/v1_00_b/src/xspi_options.c	
spi/v1_00_b/src/xspi_selftest.c	
spi/v1_00_b/src/xspi_stats.c	
sysace/v1_00_a/src/xsysace.c	
sysace/v1_00_a/src/xsysace.h	
sysace/v1_00_a/src/xsysace_compactflash.c	
sysace/v1_00_a/src/xsysace_g.c	
sysace/v1_00_a/src/xsysace_intr.c	
sysace/v1_00_a/src/xsysace_jtagcfg.c	
sysace/v1_00_a/src/xsysace_l.c	
sysace/v1_00_a/src/xsysace_l.h	
sysace/v1_00_a/src/xsysace_selftest.c	

tmrctr/v1_00_b/src/xtmrctr.c	
tmrctr/v1_00_b/src/xtmrctr.h	
tmrctr/v1_00_b/src/xtmrctr_g.c	
tmrctr/v1_00_b/src/xtmrctr_i.h	
tmrctr/v1_00_b/src/xtmrctr_intr.c	
tmrctr/v1_00_b/src/xtmrctr_l.c	
tmrctr/v1_00_b/src/xtmrctr_l.h	
tmrctr/v1_00_b/src/xtmrctr_options.c	
tmrctr/v1_00_b/src/xtmrctr_selftest.c	
tmrctr/v1_00_b/src/xtmrctr_stats.c	
uartlite/v1_00_b/src/xuartlite.c	
uartlite/v1_00_b/src/xuartlite.h	
uartlite/v1_00_b/src/xuartlite_g.c	
uartlite/v1_00_b/src/xuartlite_i.h	
uartlite/v1_00_b/src/xuartlite_intr.c	
uartlite/v1_00_b/src/xuartlite_l.c	
uartlite/v1_00_b/src/xuartlite_l.h	
uartlite/v1_00_b/src/xuartlite_selftest.c	
uartlite/v1_00_b/src/xuartlite_stats.c	
uartns550/v1_00_b/src/xuartns550.c	
uartns550/v1_00_b/src/xuartns550.h	
uartns550/v1_00_b/src/xuartns550_format.c	
uartns550/v1_00_b/src/xuartns550_g.c	
uartns550/v1_00_b/src/xuartns550_i.h	
uartns550/v1_00_b/src/xuartns550_intr.c	
uartns550/v1_00_b/src/xuartns550_l.c	
uartns550/v1_00_b/src/xuartns550_l.h	
uartns550/v1_00_b/src/xuartns550_options.c	
uartns550/v1_00_b/src/xuartns550_selftest.c	
uartns550/v1_00_b/src/xuartns550_stats.c	
wdttb/v1_00_b/src/xwdttb.c	
wdttb/v1_00_b/src/xwdttb.h	

wdttb/v1_00_b/src/xwdttb_g.c	
wdttb/v1_00_b/src/xwdttb_i.h	
wdttb/v1_00_b/src/xwdttb_l.h	
wdttb/v1_00_b/src/xwdttb_selftest.c	

Generated on 24 Jun 2004 for Xilinx Device Drivers

Xilinx Device Drivers Data Fields

[a](#) | [b](#) | [c](#) | [d](#) | [e](#) | [f](#) | [g](#) | [h](#) | [i](#) | [l](#) | [m](#) | [n](#) | [o](#) | [p](#) | [r](#) | [s](#) | [t](#) | [u](#) | [v](#) | [w](#) | [x](#)

Here is a list of all documented struct and union fields with links to the structures/unions they belong to:

- a -

- AbsoluteBlock : [XFlashGeometry](#)
- AbsoluteOffset : [XFlashGeometry](#)
- AckBeforeService : [XIntc_Config](#)
- ActiveDescriptorCount : [XDmaMulti_SgData](#)
- AddrTablePtr : [XDmaMulti](#)
- ArbitrationLost : [XIicStats](#)
- AtmcInterrupts : [XAtmc_Stats](#)

- b -

- BaseAddr : [XWdtTb_Config](#), [XFlash_Config](#)
- BaseAddress : [XUartNs550_Config](#), [XTmrCtr_Config](#), [XSysAce_Config](#), [XSpi_Config](#), [XPlbArb_Config](#), [XPlb2Opb_Config](#), [XPCIArb](#), [XPCIArb_Config](#), [XOpbArb_Config](#), [XOpb2Plb_Config](#), [XIntc_Config](#), [XIic_Config](#), [XHwIcap_Config](#), [XHdlc](#), [XHdlc_Config](#), [XGemac_Config](#), [XFlashGeometry](#), [XEmacLite_Config](#), [XEmac_Config](#), [XDSDac](#), [XDSDac_Config](#), [XDSDac](#), [XDSDac_Config](#), [XDmaCentral](#), [XDmaCentral_Config](#), [XDmaMulti](#), [XDdr](#), [XAtmc_Config](#)
- BaudRate : [XUartNs550Format](#), [XUartLite_Config](#)
- BufferSize : [XSysAce_CFPParameters](#)
- BufferType : [XSysAce_CFPParameters](#)
- BusBusy : [XIicStats](#)
- BytesTransferred : [XSpi_Stats](#)

- c -

- Capabilities : **XSysAce_CFPParameters**
- ChannelCount : **XDmaMulti**
- ChannelDataAccessPtrs : **XHdlc**
- ChannelDataPtr : **XHdlc**
- ChannelID : **XHdlc_Channel**
- CharactersReceived : **XUartNs550Stats, XUartLite_Stats**
- CharactersTransmitted : **XUartNs550Stats, XUartLite_Stats**
- CommandSet : **XFlashPartID**
- CommitPtr : **XDmaMulti_SgData**
- ConfigPtr : **XHdlc**
- CurNumCylinders : **XSysAce_CFPParameters**
- CurNumHeads : **XSysAce_CFPParameters**
- CurSectorsPerCard : **XSysAce_CFPParameters**
- CurSectorsPerTrack : **XSysAce_CFPParameters**

- d -

- DataBits : **XUartNs550Format, XUartLite_Config**
- DblWord : **XSysAce_CFPParameters**
- DeviceID : **XFlashPartID**
- DeviceId : **XWdtTb_Config, XUartNs550_Config, XUartLite_Config, XTmrCtr_Config, XSysAce_Config, XSpi_Config, XPlbArb_Config, XPlb2Opb_Config, XPCIArb_Config, XOpbArb_Config, XOpb2Plb_Config, XIntc_Config, Xlic_Config, XHwIcap_Config, XHdlc_Config, XGemac_Config, XFlash_Config, XEmc_Config, XEmacLite_Config, XEmac_Config, XDsDac_Config, XDsAdc_Config, XDmaCentral_Config, XAtmc_Config**
- DeviceSize : **XFlashGeometry**
- DmaChannelStructure : **XHdlc**
- DmaErrors : **XHdlc_Stats, XGemac_SoftStats, XEmac_Stats, XAtmc_Stats**
- DmaMode : **XSysAce_CFPParameters**
- DmaRegBaseAddr : **XPCI**
- DmaType : **XPCI**

- e -

- EmacInterrupts : **XGemac_SoftStats, XEmac_Stats**
- EraseBlock_Ms : **XFlashTiming**
- EraseChip_Ms : **XFlashTiming**
- EraseQueueSize : **XFlashProgCap**

- f -

- FifoErrors : [XHdlc_Stats](#), [XGemac_SoftStats](#), [XEmac_Stats](#), [XAtmc_Stats](#)
- FwVersion : [XSysAce_CFPParameters](#)

- g -

- GetPtr : [XDmaMulti_SgData](#)
- GpOutWidth : [XIic_Config](#)

- h -

- HandlerTable : [XIntc_Config](#)
- Has10BitAddr : [XIic_Config](#)
- HasCounters : [XGemac_Config](#), [XEmac_Config](#)
- HasFifos : [XSpi_Config](#)
- HasGmii : [XGemac_Config](#)
- HasMii : [XEmac_Config](#)
- HdlcInterrupts : [XHdlc_Stats](#)

- i -

- IicInterrupts : [XIicStats](#)
- InputClockHz : [XUartNs550_Config](#)
- Interrupts : [XTmrCtrStats](#)
- IntrFifoAddress : [XDmaMulti](#)
- IpIfDmaConfig : [XHdlc_Config](#), [XGemac_Config](#), [XEmac_Config](#), [XAtmc_Config](#)
- IsError : [XPCIError](#)
- IsReady : [XPCIArb](#), [XPCI](#), [XHdlc](#), [XDSDac](#), [XDSDac](#), [XDmaCentral](#), [XDmaMulti](#), [XDdr](#)
- IsStarted : [XHdlc](#)

- l -

- LastPtr : [XDmaMulti_SgData](#)
- LbaSectors : [XSysAce_CFPParameters](#)
- LocalBusReadAddr : [XPCIError](#)
- LocalBusReason : [XPCIError](#)
- LocalBusWriteAddr : [XPCIError](#)

- m -

- ManufacturerID : **XFlashPartID**
- MaxSectors : **XSysAce_CFPParameters**
- MemoryLayout : **XFlashGeometry**
- ModeFaults : **XSpi_Stats**
- ModelNo : **XSysAce_CFPParameters**
- ModemInterrupts : **XUartNs550Stats**
- MultipleSectors : **XSysAce_CFPParameters**

- n -

- NumBanks : **XEmc_Config**
- Number : **XFlashGeometry**
- NumBlocks : **XFlashGeometry**
- NumBytesPerSector : **XSysAce_CFPParameters**
- NumBytesPerTrack : **XSysAce_CFPParameters**
- NumCylinders : **XSysAce_CFPParameters**
- NumEccBytes : **XSysAce_CFPParameters**
- NumEraseRegions : **XFlashGeometry**
- NumHeads : **XSysAce_CFPParameters**
- NumInterrupts : **XSpi_Stats**
- NumMasters : **XPlbArb_Config, XPlb2Opb_Config, XPCIarb, XPCIarb_Config, XOpbArb_Config**
- NumParts : **XFlash_Config**
- NumSectorsPerCard : **XSysAce_CFPParameters**
- NumSectorsPerTrack : **XSysAce_CFPParameters**
- NumSlaveBits : **XSpi_Config**

- o -

- Options : **XIntc_Config**

- p -

- Parity : **XUartNs550Format**
- ParityOdd : **XUartLite_Config**
- PartID : **XFlashProperties**
- PartMode : **XFlash_Config**

- PartWidth : **XFlash_Config**
- PciReadAddr : **XPciError**
- PciReason : **XPciError**
- PciSerrReadAddr : **XPciError**
- PciSerrReason : **XPciError**
- PciSerrWriteAddr : **XPciError**
- PciWriteAddr : **XPciError**
- PhysChanSubCount : **XHdlc**
- PioMode : **XSysAce_CFPParameters**
- PowerDesc : **XSysAce_CFPParameters**
- ProgCap : **XFlashProperties**
- PutPtr : **XDmaMulti_SgData**

- r -

- ReceiveBreakDetected : **XUartNs550Stats**
- ReceiveFramingErrors : **XUartNs550Stats, XUartLite_Stats**
- ReceiveInterrupts : **XUartNs550Stats, XUartLite_Stats**
- ReceiveOverrunErrors : **XUartNs550Stats, XUartLite_Stats**
- ReceiveParityErrors : **XUartNs550Stats, XUartLite_Stats**
- Recv1024_MaxByte : **XGemac_HardStats**
- Recv128_255Byte : **XGemac_HardStats**
- Recv256_511Byte : **XGemac_HardStats**
- Recv512_1023Byte : **XGemac_HardStats**
- Recv64Byte : **XGemac_HardStats**
- Recv65_127Byte : **XGemac_HardStats**
- RecvAlignmentErrors : **XHdlc_Stats, XEmac_Stats**
- RecvBadOpcode : **XGemac_HardStats**
- RecvBroadcast : **XGemac_HardStats**
- RecvBytes : **XIicStats, XHdlc_Stats, XGemac_HardStats, XEmac_Stats**
- RecvCells : **XAtmc_Stats**
- RecvChannelNum : **XHdlc_Channel**
- RecvCollisionErrors : **XEmac_Stats**
- RecvControl : **XGemac_HardStats**
- RecvCounterRolls : **XHdlc_Stats**
- RecvFcs : **XGemac_HardStats**
- RecvFcsErrors : **XHdlc_Stats, XGemac_SoftStats, XEmac_Stats**
- RecvFragment : **XGemac_HardStats**
- RecvFrames : **XHdlc_Stats, XGemac_HardStats, XEmac_Stats**

- RecvFramesAborted : **XHdlc_Stats**
- RecvHecErrors : **XAtmc_Stats**
- RecvInterrupts : **XIicStats, XHdlc_Stats, XGemac_SoftStats, XEmac_Stats, XAtmc_Stats**
- RecvLengthFieldErrors : **XGemac_SoftStats, XEmac_Stats**
- RecvLengthRange : **XGemac_HardStats**
- RecvLong : **XGemac_HardStats**
- RecvLongCells : **XAtmc_Stats**
- RecvLongErrors : **XGemac_SoftStats, XEmac_Stats**
- RecvMissedFrameErrors : **XEmac_Stats**
- RecvMulticast : **XGemac_HardStats**
- RecvOverrunErrors : **XHdlc_Stats, XGemac_SoftStats, XEmac_Stats**
- RecvOverruns : **XSpi_Stats**
- RecvParityErrors : **XAtmc_Stats**
- RecvPause : **XGemac_HardStats**
- RecvShort : **XGemac_HardStats**
- RecvShortCells : **XAtmc_Stats**
- RecvShortErrors : **XEmac_Stats**
- RecvSlotLengthErrors : **XGemac_SoftStats**
- RecvUnderrunErrors : **XGemac_SoftStats, XEmac_Stats**
- RecvUnexpectedHeaders : **XAtmc_Stats**
- RecvVlan : **XGemac_HardStats**
- RegBaseAddr : **XUartLite_Config, XPCI, XEmc_Config**
- RepeatedStarts : **XIicStats**
- RxPingPong : **XEmacLite_Config**

- S -

- SecurityStatus : **XSysAce_CFPParameters**
- SendBytes : **XIicStats**
- SendChannelNum : **XHdlc_Channel**
- SendInterrupts : **XIicStats**
- SerialNo : **XSysAce_CFPParameters**
- SgDataTablePtr : **XDmaMulti**
- Signature : **XSysAce_CFPParameters**
- Size : **XFlashGeometry**
- SlaveModeFaults : **XSpi_Stats**
- SlaveOnly : **XSpi_Config**
- StatusInterrupts : **XUartNs550Stats**
- StopBits : **XUartNs550Format**

- SupportReadRegs : [XDmaCentral](#), [XDmaCentral_Config](#)

- t -

- TimeMax : [XFlashProperties](#)
- TimeTypical : [XFlashProperties](#)
- TotalDescriptorCount : [XDmaMulti_SgData](#)
- TotalInterrupts : [XGemac_SoftStats](#)
- TotalIntrs : [XEmac_Stats](#)
- TotalNumberOfChannels : [XHdlc](#)
- TranslationValid : [XSysAce_CFPParameters](#)
- TransmitInterrupts : [XUartNs550Stats](#), [XUartLite_Stats](#)
- TxErrors : [XIicStats](#)
- TxPingPong : [XEmacLite_Config](#)

- u -

- UseParity : [XUartLite_Config](#)
- UserData : [XHdlc_Channel](#)

- v -

- VendorUnique : [XSysAce_CFPParameters](#)
- VendorUniqueBytes : [XSysAce_CFPParameters](#)

- w -

- WriteBuffer_Us : [XFlashTiming](#)
- WriteBufferAlignmentMask : [XFlashProgCap](#)
- WriteBufferSize : [XFlashProgCap](#)
- WriteSingle_Us : [XFlashTiming](#)

- x -

- Xmit1024_MaxByte : [XGemac_HardStats](#)
- Xmit128_255Byte : [XGemac_HardStats](#)
- Xmit1stCollision : [XGemac_HardStats](#)
- Xmit256_511Byte : [XGemac_HardStats](#)
- Xmit512_1023Byte : [XGemac_HardStats](#)

- Xmit64Byte : **XGemac_HardStats**
- Xmit65_127Byte : **XGemac_HardStats**
- XmitBroadcast : **XGemac_HardStats**
- XmitBytes : **XHdlc_Stats, XGemac_HardStats, XEmac_Stats**
- XmitCarrierSense : **XGemac_HardStats**
- XmitCells : **XAtmc_Stats**
- XmitControl : **XGemac_HardStats**
- XmitDeferred : **XGemac_HardStats**
- XmitExcessCollision : **XGemac_HardStats**
- XmitExcessDeferral : **XEmac_Stats**
- XmitExcessDeferralErrors : **XGemac_SoftStats**
- XmitExcessDeferred : **XGemac_HardStats**
- XmitFrames : **XHdlc_Stats, XGemac_HardStats, XEmac_Stats**
- XmitInterrupts : **XHdlc_Stats, XGemac_SoftStats, XEmac_Stats, XAtmc_Stats**
- XmitLateCollision : **XGemac_HardStats**
- XmitLateCollErrors : **XGemac_SoftStats**
- XmitLateCollisionErrors : **XEmac_Stats**
- XmitLong : **XGemac_HardStats**
- XmitMulticast : **XGemac_HardStats**
- XmitMultiCollision : **XGemac_HardStats**
- XmitOverrunErrors : **XGemac_SoftStats, XEmac_Stats**
- XmitPause : **XGemac_HardStats**
- XmitPFifoUnderrunErrors : **XGemac_SoftStats**
- XmitUnderrun : **XGemac_HardStats**
- XmitUnderrunErrors : **XHdlc_Stats, XGemac_SoftStats, XEmac_Stats**
- XmitUnderruns : **XSpi_Stats**
- XmitVlan : **XGemac_HardStats**

Xilinx Device Drivers Globals

[l](#) | [s](#) | [t](#) | [x](#)

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- l -

- LOOPBACK_WORD_COUNT : [xhdlc_selftest.c](#)

- s -

- StubHandler() : [xemaclite_i.h](#), [xemaclite.c](#)

- t -

- TEST_VALUE : [xdsdac_selftest.c](#), [xdsadc_selftest.c](#)

- x -

- XADC_DATA_OFFSET : [xdsadc_1.h](#)
- XADC_FSTM_MASK : [xdsadc_1.h](#)
- XADC_FSTM_OFFSET : [xdsadc_1.h](#)
- XADC_GIER_GIE_MASK : [xdsadc_1.h](#)
- XADC_GIER_OFFSET : [xdsadc_1.h](#)
- XADC_IPIER_OFFSET : [xdsadc_1.h](#)
- XADC_IPISR_OFFSET : [xdsadc_1.h](#)
- XADC_IPIXR_READY_MASK : [xdsadc_1.h](#)
- XAssert() : [xbasic_types.h](#), [xbasic_types.c](#)
- XASSERT_NONVOID : [xbasic_types.h](#)
- XASSERT_NONVOID_ALWAYS : [xbasic_types.h](#)
- XASSERT_VOID : [xbasic_types.h](#)
- XASSERT_VOID_ALWAYS : [xbasic_types.h](#)

- XAssertCallback : [xbasic_types.h](#)
- XAssertSetCallback() : [xbasic_types.h](#), [xbasic_types.c](#)
- XAssertStatus : [xbasic_types.h](#), [xbasic_types.c](#)
- XAT_CELL_STATUS_BAD_HEC : [xatmc.h](#)
- XAT_CELL_STATUS_BAD_PARITY : [xatmc.h](#)
- XAT_CELL_STATUS_LONG : [xatmc.h](#)
- XAT_CELL_STATUS_NO_ERROR : [xatmc.h](#)
- XAT_CELL_STATUS_SHORT : [xatmc.h](#)
- XAT_CELL_STATUS_VXI_MISMATCH : [xatmc.h](#)
- XAT_DISCARD_HEC_OPTION : [xatmc.h](#)
- XAT_DISCARD_LONG_OPTION : [xatmc.h](#)
- XAT_DISCARD_PARITY_OPTION : [xatmc.h](#)
- XAT_DISCARD_SHORT_OPTION : [xatmc.h](#)
- XAT_DISCARD_VXI_OPTION : [xatmc.h](#)
- XAT_LOOPBACK_OPTION : [xatmc.h](#)
- XAT_NO_SEND_PARITY_OPTION : [xatmc.h](#)
- XAT_PAYLOAD_ONLY_OPTION : [xatmc.h](#)
- XAT_POLLED_OPTION : [xatmc.h](#)
- XAtmc_ClearStats() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_ConfigTable : [xatmc_i.h](#), [xatmc_g.c](#)
- XAtmc_ErrorHandler : [xatmc.h](#)
- XAtmc_GetHeader() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_GetOptions() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_GetPhyAddress() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_GetPktThreshold() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_GetPktWaitBound() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_GetStats() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_GetUserDefined() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_Initialize() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_InterruptHandler() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_LookupConfig() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_mDisable : [xatmc_l.h](#)
- XAtmc_mEnable : [xatmc_l.h](#)
- XAtmc_mIsRxEmpty : [xatmc_l.h](#)
- XAtmc_mIsSgDma : [xatmc_i.h](#)
- XAtmc_mIsTxDone : [xatmc_l.h](#)
- XAtmc_mReadReg : [xatmc_l.h](#)
- XAtmc_mWriteReg : [xatmc_l.h](#)
- XAtmc_PollRecv() : [xatmc.h](#), [xatmc.c](#)

- XAtmc_PollSend() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_RecvCell() : [xatmc_1.h](#), [xatmc_1.c](#)
- XAtmc_Reset() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SelfTest() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SendCell() : [xatmc_1.h](#), [xatmc_1.c](#)
- XAtmc_SetErrorHandler() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetHeader() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetOptions() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetPhyAddress() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetPktThreshold() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetPktWaitBound() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetSgRecvHandler() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetSgRecvSpace() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetSgSendHandler() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetSgSendSpace() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetUserDefined() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SgGetRecvCell() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_SgGetSendCell() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_SgHandler : [xatmc.h](#)
- XAtmc_SgRecv() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_SgSend() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_Start() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_Stop() : [xatmc_cfg.c](#), [xatmc.h](#)
- Xboolean : [xbasic_types.h](#)
- XDAC_CR_EN_MASK : [xdsdac_1.h](#)
- XDAC_CR_FIFO_RESET_MASK : [xdsdac_1.h](#)
- XDAC_CR_OFFSET : [xdsdac_1.h](#)
- XDAC_FIFO_DATA_MASK : [xdsdac_1.h](#)
- XDAC_FIFO_OFFSET : [xdsdac_1.h](#)
- XDAC_GIER_GIE_MASK : [xdsdac_1.h](#)
- XDAC_GIER_OFFSET : [xdsdac_1.h](#)
- XDAC_IPIER_OFFSET : [xdsdac_1.h](#)
- XDAC_IPISR_OFFSET : [xdsdac_1.h](#)
- XDAC_IPISRR_OFFSET : [xdsdac_1.h](#)
- XDAC_IPISRR_RESET_MASK : [xdsdac_1.h](#)
- XDAC_IPIXR_EMPTY_MASK : [xdsdac_1.h](#)
- XDAC_IPIXR_PIRQ_MASK : [xdsdac_1.h](#)
- XDAC_OCCY_MASK : [xdsdac_1.h](#)
- XDAC_OCCY_OFFSET : [xdsdac_1.h](#)

- XDAC_PIRQ_MASK : [xdsdac_1.h](#)
- XDAC_PIRQ_OFFSET : [xdsdac_1.h](#)
- XDdr_ClearStats() : [xddr_stats.c](#), [xddr.h](#)
- XDdr_ClearStatus() : [xddr.h](#), [xddr.c](#)
- XDdr_ConfigTable : [xddr_g.c](#)
- XDDR_DGIE_GIE_MASK : [xddr_1.h](#)
- XDDR_DGIE_OFFSET : [xddr_1.h](#)
- XDdr_DisableEcc() : [xddr.h](#), [xddr.c](#)
- XDDR_ECCCR_FORCE_DE_MASK : [xddr_1.h](#)
- XDDR_ECCCR_FORCE_PE_MASK : [xddr_1.h](#)
- XDDR_ECCCR_FORCE_SE_MASK : [xddr_1.h](#)
- XDDR_ECCCR_OFFSET : [xddr_1.h](#)
- XDDR_ECCCR_RE_MASK : [xddr_1.h](#)
- XDDR_ECCCR_WE_MASK : [xddr_1.h](#)
- XDDR_ECCDEC_OFFSET : [xddr_1.h](#)
- XDDR_ECCPEC_OFFSET : [xddr_1.h](#)
- XDDR_ECCPETR_OFFSET : [xddr_1.h](#)
- XDDR_ECCSEC_OFFSET : [xddr_1.h](#)
- XDDR_ECCSETR_OFFSET : [xddr_1.h](#)
- XDDR_ECCSR_DE_MASK : [xddr_1.h](#)
- XDDR_ECCSR_OFFSET : [xddr_1.h](#)
- XDDR_ECCSR_PE_MASK : [xddr_1.h](#)
- XDDR_ECCSR_SE_MASK : [xddr_1.h](#)
- XDDR_ECCSR_SE_SYND_MASK : [xddr_1.h](#)
- XDdr_EnableEcc() : [xddr.h](#), [xddr.c](#)
- XDdr_GetControl() : [xddr.h](#), [xddr.c](#)
- XDdr_GetStats() : [xddr_stats.c](#), [xddr.h](#)
- XDdr_GetStatus() : [xddr.h](#), [xddr.c](#)
- XDdr_Initialize() : [xddr.h](#), [xddr.c](#)
- XDdr_InterruptClear() : [xddr_intr.c](#), [xddr.h](#)
- XDdr_InterruptDisable() : [xddr_intr.c](#), [xddr.h](#)
- XDdr_InterruptEnable() : [xddr_intr.c](#), [xddr.h](#)
- XDdr_InterruptGetEnabled() : [xddr_intr.c](#), [xddr.h](#)
- XDdr_InterruptGetStatus() : [xddr_intr.c](#), [xddr.h](#)
- XDdr_InterruptGlobalDisable() : [xddr_intr.c](#), [xddr.h](#)
- XDdr_InterruptGlobalEnable() : [xddr_intr.c](#), [xddr.h](#)
- XDDR_IPIER_OFFSET : [xddr_1.h](#)
- XDDR_IPIF_OFFSET : [xddr_1.h](#)
- XDDR_IPISR_OFFSET : [xddr_1.h](#)

- XDDR_IPIXR_DE_IX_MASK : [xddr_1.h](#)
- XDDR_IPIXR_PE_IX_MASK : [xddr_1.h](#)
- XDDR_IPIXR_SE_IX_MASK : [xddr_1.h](#)
- XDdr_LookupConfig() : [xddr.h](#), [xddr.c](#)
- XDdr_mReadReg : [xddr_1.h](#)
- XDdr_mWriteReg : [xddr_1.h](#)
- XDdr_SelfTest() : [xddr_selftest.c](#), [xddr.h](#)
- XDdr_SetControl() : [xddr.h](#), [xddr.c](#)
- XDM_CONTROL_LAST_BD_MASK : [xdma_multi_i.h](#)
- XDM_DMACR_SOURCE_INCR_MASK : [xdma_multi.h](#)
- XDM_DMASR_BUSY_MASK : [xdma_multi.h](#)
- XDM_GCSR_GLOBAL_ENABLE_MASK : [xdma_multi.h](#)
- XDM_GEFIFO_CHAN_SHIFT : [xdma_multi_i.h](#)
- XDM_GEFIFO_SGEN_MASK : [xdma_multi.h](#)
- XDM_GMIR_REG_OFFSET : [xdma_multi_i.h](#)
- XDM_STATUS_LAST_BD_MASK : [xdma_multi_i.h](#)
- XDM_SWCR_EESGEN_MASK : [xdma_multi.h](#)
- XDM_SWCR_PWB_SHIFT : [xdma_multi_i.h](#)
- XDM_SYS_REG_ANCY_SHIFT : [xdma_multi_i.h](#)
- XDM_SYS_REG_OFFSET : [xdma_multi_i.h](#)
- XDmaCentral_GetControl() : [xdmacentral.h](#), [xdmacentral.c](#)
- XDmaCentral_GetDestAddress() : [xdmacentral.h](#), [xdmacentral.c](#)
- XDmaCentral_GetSrcAddress() : [xdmacentral.h](#), [xdmacentral.c](#)
- XDmaCentral_GetStatus() : [xdmacentral.h](#), [xdmacentral.c](#)
- XDmaCentral_Initialize() : [xdmacentral.h](#), [xdmacentral.c](#)
- XDmaCentral_InterruptClear() : [xdmacentral_intr.c](#), [xdmacentral.h](#)
- XDmaCentral_InterruptEnableGet() : [xdmacentral_intr.c](#), [xdmacentral.h](#)
- XDmaCentral_InterruptEnableSet() : [xdmacentral_intr.c](#), [xdmacentral.h](#)
- XDmaCentral_InterruptStatusGet() : [xdmacentral_intr.c](#), [xdmacentral.h](#)
- XDmaCentral_LookupConfig() : [xdmacentral.h](#), [xdmacentral.c](#)
- XDmaCentral_mReadReg : [xdmacentral_1.h](#)
- XDmaCentral_mWriteReg : [xdmacentral_1.h](#)
- XDmaCentral_Reset() : [xdmacentral.h](#), [xdmacentral.c](#)
- XDmaCentral_SelfTest() : [xdmacentral_selftest.c](#), [xdmacentral.h](#)
- XDmaCentral_SetControl() : [xdmacentral.h](#), [xdmacentral.c](#)
- XDmaCentral_Transfer() : [xdmacentral.h](#), [xdmacentral.c](#)
- XDmaMulti_CommitPuts() : [xdma_multi_sg.c](#), [xdma_multi.h](#)
- XDmaMulti_CreateSgList() : [xdma_multi_sg.c](#), [xdma_multi.h](#)
- XDmaMulti_GetDescriptor() : [xdma_multi_sg.c](#), [xdma_multi.h](#)

- XDmaMulti_Initialize() : [xdma_multi.h](#), [xdma_multi.c](#)
- XDmaMulti_IsSgListEmpty() : [xdma_multi_sg.c](#), [xdma_multi.h](#)
- XDmaMulti_mGetControl : [xdma_multi.h](#)
- XDmaMulti_mGetEventChannel : [xdma_multi.h](#)
- XDmaMulti_mGetEventEnable : [xdma_multi.h](#)
- XDmaMulti_mGetEventParameter : [xdma_multi.h](#)
- XDmaMulti_mGetEventStatus : [xdma_multi.h](#)
- XDmaMulti_mGetGlobalControl : [xdma_multi.h](#)
- XDmaMulti_mGetPktThreshold : [xdma_multi.h](#)
- XDmaMulti_mGetPktWaitBound : [xdma_multi.h](#)
- XDmaMulti_mGetStatus : [xdma_multi.h](#)
- XDmaMulti_mSetControl : [xdma_multi.h](#)
- XDmaMulti_mSetEventEnable : [xdma_multi.h](#)
- XDmaMulti_mSetGlobalControl : [xdma_multi.h](#)
- XDmaMulti_mSetPktThreshold : [xdma_multi.h](#)
- XDmaMulti_mSetPktWaitBound : [xdma_multi.h](#)
- XDmaMulti_mSizeNeeded : [xdma_multi.h](#)
- XDmaMulti_PutDescriptor() : [xdma_multi_sg.c](#), [xdma_multi.h](#)
- XDmaMulti_Reset() : [xdma_multi.h](#), [xdma_multi.c](#)
- XDmaMulti_SelfTest() : [xdma_multi.h](#), [xdma_multi.c](#)
- XDmaMulti_SgStart() : [xdma_multi_sg.c](#), [xdma_multi.h](#)
- XDmaMulti_SgStop() : [xdma_multi_sg.c](#), [xdma_multi.h](#)
- XDmaMulti_Transfer() : [xdma_multi.h](#), [xdma_multi.c](#)
- XDMC_DA_OFFSET : [xdmacentral_1.h](#)
- XDMC_DMACR_DATASIZE_1_MASK : [xdmacentral_1.h](#)
- XDMC_DMACR_DATASIZE_2_MASK : [xdmacentral_1.h](#)
- XDMC_DMACR_DATASIZE_4_MASK : [xdmacentral_1.h](#)
- XDMC_DMACR_DATASIZE_MASK : [xdmacentral_1.h](#)
- XDMC_DMACR_DEST_INCR_MASK : [xdmacentral_1.h](#)
- XDMC_DMACR_OFFSET : [xdmacentral_1.h](#)
- XDMC_DMACR_SOURCE_INCR_MASK : [xdmacentral_1.h](#)
- XDMC_DMASR_BUS_ERROR_MASK : [xdmacentral_1.h](#)
- XDMC_DMASR_BUS_TIMEOUT_MASK : [xdmacentral_1.h](#)
- XDMC_DMASR_BUSY_MASK : [xdmacentral_1.h](#)
- XDMC_DMASR_OFFSET : [xdmacentral_1.h](#)
- XDMC_IER_OFFSET : [xdmacentral_1.h](#)
- XDMC_ISR_OFFSET : [xdmacentral_1.h](#)
- XDMC_IXR_DMA_DONE_MASK : [xdmacentral_1.h](#)
- XDMC_IXR_DMA_ERROR_MASK : [xdmacentral_1.h](#)

- XDMC_LENGTH_OFFSET : [xdmacentral_1.h](#)
- XDMC_MIR_OFFSET : [xdmacentral_1.h](#)
- XDMC_RST_MASK : [xdmacentral_1.h](#)
- XDMC_RST_OFFSET : [xdmacentral_1.h](#)
- XDMC_SA_OFFSET : [xdmacentral_1.h](#)
- XDMC_SELFTEST_BUFSIZE : [xdmacentral_selftest.c](#)
- XDsAdc_ClearInterruptStatus() : [xdsadc_intr.c](#), [xdsadc.h](#)
- XDsAdc_ConfigTable : [xdsadc_g.c](#), [xdsadc.c](#)
- XDsAdc_DisableInterrupts() : [xdsadc_intr.c](#), [xdsadc.h](#)
- XDsAdc_EnableInterrupts() : [xdsadc_intr.c](#), [xdsadc.h](#)
- XDsAdc_GetInterruptStatus() : [xdsadc_intr.c](#), [xdsadc.h](#)
- XDsAdc_Initialize() : [xdsadc.h](#), [xdsadc.c](#)
- XDsAdc_LookupConfig() : [xdsadc.h](#), [xdsadc.c](#)
- XDsAdc_mGetFstmReg : [xdsadc.h](#)
- XDsAdc_mIsEmpty : [xdsadc.h](#)
- XDsAdc_mReadReg : [xdsadc_1.h](#)
- XDsAdc_mSetFstmReg : [xdsadc.h](#)
- XDsAdc_mWriteReg : [xdsadc_1.h](#)
- XDsAdc_ReadInput() : [xdsadc.h](#), [xdsadc.c](#)
- XDsDac_ClearInterruptStatus() : [xdsdac_intr.c](#), [xdsdac.h](#)
- XDsDac_ConfigTable : [xdsdac_g.c](#), [xdsdac.c](#)
- XDsDac_DisableInterrupts() : [xdsdac_intr.c](#), [xdsdac.h](#)
- XDsDac_EnableInterrupts() : [xdsdac_intr.c](#), [xdsdac.h](#)
- XDsDac_GetInterruptStatus() : [xdsdac_intr.c](#), [xdsdac.h](#)
- XDsDac_Initialize() : [xdsdac.h](#), [xdsdac.c](#)
- XDsDac_LookupConfig() : [xdsdac.h](#), [xdsdac.c](#)
- XDsDac_mFifoIsFull : [xdsdac.h](#)
- XDsDac_mGetVacancy : [xdsdac.h](#)
- XDsDac_mIsEmpty : [xdsdac.h](#)
- XDsDac_mReadReg : [xdsdac_1.h](#)
- XDsDac_mWriteReg : [xdsdac_1.h](#)
- XDsDac_Reset() : [xdsdac.h](#), [xdsdac.c](#)
- XDsDac_SetInterruptDepth() : [xdsdac_intr.c](#), [xdsdac.h](#)
- XDsDac_Start() : [xdsdac.h](#), [xdsdac.c](#)
- XDsDac_Stop() : [xdsdac.h](#), [xdsdac.c](#)
- XDsDac_WriteOutput() : [xdsdac.h](#), [xdsdac.c](#)
- XEL_BUFFER_OFFSET : [xemaclite_1.h](#)
- XEL_FCS_SIZE : [xemaclite_1.h](#)
- XEL_GIER_GIE_MASK : [xemaclite_1.h](#)

- XEL_GIER_OFFSET : [xemaclite_1.h](#)
- XEL_HEADER_OFFSET : [xemaclite_1.h](#)
- XEL_HEADER_SHIFT : [xemaclite_1.h](#)
- XEL_HEADER_SIZE : [xemaclite_1.h](#)
- XEL_MAC_ADDR_SIZE : [xemaclite_1.h](#)
- XEL_MAC_HI_OFFSET : [xemaclite_1.h](#)
- XEL_MAC_LO_OFFSET : [xemaclite_1.h](#)
- XEL_MAX_FRAME_SIZE : [xemaclite_1.h](#)
- XEL_MTU_SIZE : [xemaclite_1.h](#)
- XEL_RPLR_LENGTH_MASK_HI : [xemaclite_1.h](#)
- XEL_RPLR_LENGTH_MASK_LO : [xemaclite_1.h](#)
- XEL_RPLR_OFFSET : [xemaclite_1.h](#)
- XEL_RSR_OFFSET : [xemaclite_1.h](#)
- XEL_RSR_RECV_DONE_MASK : [xemaclite_1.h](#)
- XEL_RSR_RECV_IE_MASK : [xemaclite_1.h](#)
- XEL_RXBUFF_OFFSET : [xemaclite_1.h](#)
- XEL_TPLR_LENGTH_MASK_HI : [xemaclite_1.h](#)
- XEL_TPLR_LENGTH_MASK_LO : [xemaclite_1.h](#)
- XEL_TPLR_OFFSET : [xemaclite_1.h](#)
- XEL_TSR_OFFSET : [xemaclite_1.h](#)
- XEL_TSR_PROG_MAC_ADDR : [xemaclite_1.h](#)
- XEL_TSR_PROGRAM_MASK : [xemaclite_1.h](#)
- XEL_TSR_XMIT_ACTIVE_MASK : [xemaclite_1.h](#)
- XEL_TSR_XMIT_BUSY_MASK : [xemaclite_1.h](#)
- XEL_TSR_XMIT_IE_MASK : [xemaclite_1.h](#)
- XEL_TXBUFF_OFFSET : [xemaclite_1.h](#)
- XEM_BROADCAST_OPTION : [xemac.h](#)
- XEM_FDUPLEX_OPTION : [xemac.h](#)
- XEM_FLOW_CONTROL_OPTION : [xemac.h](#)
- XEM_INSERT_ADDR_OPTION : [xemac.h](#)
- XEM_INSERT_FCS_OPTION : [xemac.h](#)
- XEM_INSERT_PAD_OPTION : [xemac.h](#)
- XEM_LOOPBACK_OPTION : [xemac.h](#)
- XEM_MULTICAST_OPTION : [xemac.h](#)
- XEM_NO_SGEND_INT_OPTION : [xemac.h](#)
- XEM_OVWRT_ADDR_OPTION : [xemac.h](#)
- XEM_POLLED_OPTION : [xemac.h](#)
- XEM_PROMISC_OPTION : [xemac.h](#)
- XEM_STRIP_PAD_FCS_OPTION : [xemac.h](#)

- XEM_UNICAST_OPTION : [xemac.h](#)
- XEmac_ClearStats() : [xemac_stats.c](#), [xemac.h](#)
- XEmac_ConfigTable : [xemac_i.h](#), [xemac_g.c](#)
- XEmac_ErrorHandler : [xemac.h](#)
- XEmac_FifoHandler : [xemac.h](#)
- XEmac_FifoRecv() : [xemac_intr_fifo.c](#), [xemac.h](#)
- XEmac_FifoSend() : [xemac_intr_fifo.c](#), [xemac.h](#)
- XEmac_GetInterframeGap() : [xemac_options.c](#), [xemac.h](#)
- XEmac_GetMacAddress() : [xemac.h](#), [xemac.c](#)
- XEmac_GetOptions() : [xemac_options.c](#), [xemac.h](#)
- XEmac_GetPktThreshold() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_GetPktWaitBound() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_GetSgRecvFreeDesc() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_GetSgSendFreeDesc() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_GetStats() : [xemac_stats.c](#), [xemac.h](#)
- XEmac_Initialize() : [xemac.h](#), [xemac.c](#)
- XEmac_IntrHandlerDma() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_IntrHandlerFifo() : [xemac_intr_fifo.c](#), [xemac.h](#)
- XEmac_LookupConfig() : [xemac.h](#), [xemac.c](#)
- XEmac_mDisable : [xemac_l.h](#)
- XEmac_mEnable : [xemac_l.h](#)
- XEmac_mIsDma : [xemac.h](#)
- XEmac_mIsRxEmpty : [xemac_l.h](#)
- XEmac_mIsSgDma : [xemac.h](#)
- XEmac_mIsSimpleDma : [xemac.h](#)
- XEmac_mIsTxDone : [xemac_l.h](#)
- XEmac_mPhyReset : [xemac_l.h](#)
- XEmac_mReadReg : [xemac_l.h](#)
- XEmac_mSetControlReg : [xemac_l.h](#)
- XEmac_mSetMacAddress : [xemac_l.h](#)
- XEmac_MulticastAdd() : [xemac_multicast.c](#), [xemac.h](#)
- XEmac_MulticastClear() : [xemac_multicast.c](#), [xemac.h](#)
- XEmac_mWriteReg : [xemac_l.h](#)
- XEmac_PhyRead() : [xemac_phy.c](#), [xemac.h](#)
- XEmac_PhyWrite() : [xemac_phy.c](#), [xemac.h](#)
- XEmac_PollRecv() : [xemac_polled.c](#), [xemac.h](#)
- XEmac_PollSend() : [xemac_polled.c](#), [xemac.h](#)
- XEmac_RecvFrame() : [xemac_l.h](#), [xemac_l.c](#)
- XEmac_Reset() : [xemac.h](#), [xemac.c](#)

- XEmac_SelfTest() : [xemac_selftest.c](#), [xemac.h](#)
- XEmac_SendFrame() : [xemac_l.h](#), [xemac_l.c](#)
- XEmac_SetErrorHandler() : [xemac_intr.c](#), [xemac.h](#)
- XEmac_SetFifoRecvHandler() : [xemac_intr_fifo.c](#), [xemac.h](#)
- XEmac_SetFifoSendHandler() : [xemac_intr_fifo.c](#), [xemac.h](#)
- XEmac_SetInterframeGap() : [xemac_options.c](#), [xemac.h](#)
- XEmac_SetMacAddress() : [xemac.h](#), [xemac.c](#)
- XEmac_SetOptions() : [xemac_options.c](#), [xemac.h](#)
- XEmac_SetPktThreshold() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SetPktWaitBound() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SetSgRecvHandler() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SetSgRecvSpace() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SetSgSendHandler() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SetSgSendSpace() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SgHandler : [xemac.h](#)
- XEmac_SgRecv() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SgSend() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_Start() : [xemac.h](#), [xemac.c](#)
- XEmac_Stop() : [xemac.h](#), [xemac.c](#)
- XEmacLite_AlignedRead() : [xemaclite_l.c](#), [xemaclite_i.h](#)
- XEmacLite_AlignedWrite() : [xemaclite_l.c](#), [xemaclite_i.h](#)
- XEmacLite_ConfigTable : [xemaclite_i.h](#), [xemaclite_g.c](#)
- XEmacLite_DisableInterrupts() : [xemaclite_intr.c](#), [xemaclite.h](#)
- XEmacLite_EnableInterrupts() : [xemaclite_intr.c](#), [xemaclite.h](#)
- XEmacLite_FlushReceive() : [xemaclite.h](#), [xemaclite.c](#)
- XEmacLite_Initialize() : [xemaclite.h](#), [xemaclite.c](#)
- XEmacLite_InterruptHandler() : [xemaclite_intr.c](#), [xemaclite.h](#)
- XEmacLite_LookupConfig() : [xemaclite.h](#), [xemaclite.c](#)
- XEmacLite_mGetTxActive : [xemaclite_i.h](#)
- XEmacLite_mIsRxEmpty : [xemaclite_l.h](#)
- XEmacLite_mIsTxDone : [xemaclite_l.h](#)
- XEmacLite_mNextReceiveAddr : [xemaclite.h](#)
- XEmacLite_mNextTransmitAddr : [xemaclite.h](#)
- XEmacLite_mSetTxActive : [xemaclite_i.h](#)
- XEmacLite_Recv() : [xemaclite.h](#), [xemaclite.c](#)
- XEmacLite_RecvFrame() : [xemaclite_l.h](#), [xemaclite_l.c](#)
- XEmacLite_SelfTest() : [xemaclite_selftest.c](#), [xemaclite.h](#)
- XEmacLite_Send() : [xemaclite.h](#), [xemaclite.c](#)
- XEmacLite_SendFrame() : [xemaclite_l.h](#), [xemaclite_l.c](#)

- XEmacLite_SetMacAddress() : [xemaclite.h](#), [xemaclite.c](#)
- XEmacLite_SetRecvHandler() : [xemaclite_intr.c](#), [xemaclite.h](#)
- XEmacLite_SetSendHandler() : [xemaclite_intr.c](#), [xemaclite.h](#)
- XEmacLite_TxBufferAvailable() : [xemaclite.h](#), [xemaclite.c](#)
- XEmc_GetAccessSpeed() : [xemc.h](#), [xemc.c](#)
- XEmc_GetPageMode() : [xemc.h](#), [xemc.c](#)
- XEmc_Initialize() : [xemc.h](#), [xemc.c](#)
- XEmc_LookupConfig() : [xemc.h](#), [xemc.c](#)
- XEmc_mDisableFastAccess : [xemc_1.h](#)
- XEmc_mDisablePageMode : [xemc_1.h](#)
- XEmc_mEnableFastAccess : [xemc_1.h](#)
- XEmc_mEnablePageMode : [xemc_1.h](#)
- XEmc_mGetControlReg : [xemc_1.h](#)
- XEmc_mGetOffset : [xemc_1.h](#)
- XEmc_mSetControlReg : [xemc_1.h](#)
- XEmc_SelfTest() : [xemc_selftest.c](#), [xemc.h](#)
- XEmc_SetAccessSpeed() : [xemc.h](#), [xemc.c](#)
- XEmc_SetPageMode() : [xemc.h](#), [xemc.c](#)
- XENV_MEM_COPY : [xenv_vxworks.h](#), [xenv_none.h](#), [xenv_linux.h](#)
- XENV_MEM_FILL : [xenv_vxworks.h](#), [xenv_none.h](#), [xenv_linux.h](#)
- XENV_TIME_STAMP : [xenv_none.h](#), [xenv_linux.h](#)
- XENV_TIME_STAMP_DELTA_MS : [xenv_vxworks.h](#), [xenv_none.h](#), [xenv_linux.h](#)
- XENV_TIME_STAMP_DELTA_US : [xenv_vxworks.h](#), [xenv_none.h](#), [xenv_linux.h](#)
- XENV_TIME_STAMP_GET : [xenv_vxworks.h](#), [xenv_none.h](#), [xenv_linux.h](#)
- XENV_USLEEP : [xenv_vxworks.h](#), [xenv_none.h](#), [xenv_linux.h](#)
- XFALSE : [xbasic_types.h](#)
- XFL_CFI_ADVANCE_PTR16 : [xflash_cfi.h](#)
- XFL_CFI_ADVANCE_PTR8 : [xflash_cfi.h](#)
- XFL_CFI_POSITION_PTR : [xflash_cfi.h](#)
- XFL_CFI_READ16 : [xflash_cfi.h](#)
- XFL_CFI_READ8 : [xflash_cfi.h](#)
- XFL_GEOMETRY_BLOCK_DIFF : [xflash_geometry.h](#)
- XFL_GEOMETRY_BLOCKS_LEFT : [xflash_geometry.h](#)
- XFL_GEOMETRY_INCREMENT : [xflash_geometry.h](#)
- XFL_GEOMETRY_IS_ABSOLUTE_VALID : [xflash_geometry.h](#)
- XFL_GEOMETRY_IS_BLOCK_VALID : [xflash_geometry.h](#)
- XFL_MANUFACTURER_ID_INTEL : [xflash.h](#)
- XFL_MAX_ERASE_REGIONS : [xflash_geometry.h](#)
- XFL_NON_BLOCKING_ERASE_OPTION : [xflash.h](#)

- XFL_NON_BLOCKING_WRITE_OPTION : [xflash.h](#)
- XFlash : [xflash.h](#)
- XFlash_ConfigTable : [xflash_g.c](#)
- XFlash_DeviceControl() : [xflash.h](#), [xflash.c](#)
- XFlash_Erase() : [xflash.h](#), [xflash.c](#)
- XFlash_EraseBlock() : [xflash.h](#), [xflash.c](#)
- XFlash_EraseBlockResume() : [xflash.h](#), [xflash.c](#)
- XFlash_EraseBlockSuspend() : [xflash.h](#), [xflash.c](#)
- XFlash_EraseChip() : [xflash.h](#), [xflash.c](#)
- XFlash_EraseResume() : [xflash.h](#), [xflash.c](#)
- XFlash_EraseSuspend() : [xflash.h](#), [xflash.c](#)
- XFlash_GetBlockStatus() : [xflash.h](#), [xflash.c](#)
- XFlash_GetGeometry() : [xflash.h](#), [xflash.c](#)
- XFlash_GetOptions() : [xflash.h](#), [xflash.c](#)
- XFlash_GetProperties() : [xflash.h](#), [xflash.c](#)
- XFlash_GetStatus() : [xflash.h](#), [xflash.c](#)
- XFlash_Initialize() : [xflash.h](#), [xflash.c](#)
- XFlash_IsReady() : [xflash.c](#)
- XFlash_Lock() : [xflash.h](#), [xflash.c](#)
- XFlash_LockBlock() : [xflash.h](#), [xflash.c](#)
- XFlash_LookupConfig() : [xflash.h](#), [xflash.c](#)
- XFlash_Read() : [xflash.h](#), [xflash.c](#)
- XFlash_ReadBlock() : [xflash.h](#), [xflash.c](#)
- XFlash_Reset() : [xflash.h](#), [xflash.c](#)
- XFlash_SelfTest() : [xflash.h](#), [xflash.c](#)
- XFlash_SetOptions() : [xflash.h](#), [xflash.c](#)
- XFlash_Unlock() : [xflash.h](#), [xflash.c](#)
- XFlash_UnlockBlock() : [xflash.h](#), [xflash.c](#)
- XFlash_Write() : [xflash.h](#), [xflash.c](#)
- XFlash_WriteBlock() : [xflash.h](#), [xflash.c](#)
- XFlash_WriteBlockResume() : [xflash.h](#), [xflash.c](#)
- XFlash_WriteBlockSuspend() : [xflash.h](#), [xflash.c](#)
- XFlash_WriteResume() : [xflash.h](#), [xflash.c](#)
- XFlash_WriteSuspend() : [xflash.h](#), [xflash.c](#)
- XFlashCFI_ReadCommon() : [xflash_cfi.h](#), [xflash_cfi.c](#)
- XFlashGeometry_ConvertLayout() : [xflash_geometry.h](#), [xflash_geometry.c](#)
- XFlashGeometry_ToAbsolute() : [xflash_geometry.h](#), [xflash_geometry.c](#)
- XFlashGeometry_ToBlock() : [xflash_geometry.h](#), [xflash_geometry.c](#)
- XFlashIntel_DeviceControl() : [xflash_intel.h](#), [xflash_intel.c](#)

- XFlashIntel_Erase() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_EraseAddr() : [xflash_intel_l.h](#), [xflash_intel_l.c](#)
- XFlashIntel_EraseBlock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_EraseBlockResume() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_EraseBlockSuspend() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_EraseChip() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_EraseResume() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_EraseSuspend() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_GetBlockStatus() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_GetGeometry() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_GetOptions() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_GetProperties() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_GetStatus() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_Initialize() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_Lock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_LockAddr() : [xflash_intel_l.h](#), [xflash_intel_l.c](#)
- XFlashIntel_LockBlock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_mSendCmd : [xflash_intel_l.h](#)
- XFlashIntel_Read() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_ReadAddr() : [xflash_intel_l.h](#), [xflash_intel_l.c](#)
- XFlashIntel_ReadBlock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_Reset() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_SelfTest() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_SetOptions() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_Unlock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_UnlockAddr() : [xflash_intel_l.h](#), [xflash_intel_l.c](#)
- XFlashIntel_UnlockBlock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_WaitReady() : [xflash_intel_l.h](#), [xflash_intel_l.c](#)
- XFlashIntel_Write() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_WriteAddr() : [xflash_intel_l.h](#), [xflash_intel_l.c](#)
- XFlashIntel_WriteBlock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_WriteBlockResume() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_WriteBlockSuspend() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_WriteResume() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_WriteSuspend() : [xflash_intel.h](#), [xflash_intel.c](#)
- Xfloat32 : [xbasic_types.h](#)
- Xfloat64 : [xbasic_types.h](#)
- XGE_AUTO_NEGOTIATE_OPTION : [xgemac.h](#)
- XGE_BROADCAST_OPTION : [xgemac.h](#)

- XGE_CEAH_OFFSET : [xgemac_1.h](#)
- XGE_CEAL_OFFSET : [xgemac_1.h](#)
- XGE_DMA_OFFSET : [xgemac_1.h](#)
- XGE_DMA_RECV_OFFSET : [xgemac_1.h](#)
- XGE_DMA_SEND_OFFSET : [xgemac_1.h](#)
- XGE_ECR_OFFSET : [xgemac_1.h](#)
- XGE_EMIR_OFFSET : [xgemac_1.h](#)
- XGE_FDUPLEX_OPTION : [xgemac.h](#)
- XGE_FLOW_CONTROL_OPTION : [xgemac.h](#)
- XGE_IFGP_OFFSET : [xgemac_1.h](#)
- XGE_INSERT_ADDR_OPTION : [xgemac.h](#)
- XGE_INSERT_FCS_OPTION : [xgemac.h](#)
- XGE_INSERT_PAD_OPTION : [xgemac.h](#)
- XGE_ISR_OFFSET : [xgemac_1.h](#)
- XGE_JUMBO_OPTION : [xgemac.h](#)
- XGE_LOOPBACK_OPTION : [xgemac.h](#)
- XGE_MGTCR_OFFSET : [xgemac_1.h](#)
- XGE_MGTDR_OFFSET : [xgemac_1.h](#)
- XGE_MULTICAST_OPTION : [xgemac.h](#)
- XGE_NO_SGEND_INT_OPTION : [xgemac.h](#)
- XGE_OVWRT_ADDR_OPTION : [xgemac.h](#)
- XGE_PFIFO_OFFSET : [xgemac_1.h](#)
- XGE_PFIFO_RXDATA_OFFSET : [xgemac_1.h](#)
- XGE_PFIFO_RXREG_OFFSET : [xgemac_1.h](#)
- XGE_PFIFO_TXDATA_OFFSET : [xgemac_1.h](#)
- XGE_PFIFO_TXREG_OFFSET : [xgemac_1.h](#)
- XGE_POLLED_OPTION : [xgemac.h](#)
- XGE_PROMISC_OPTION : [xgemac.h](#)
- XGE_RPLR_OFFSET : [xgemac_1.h](#)
- XGE_RSR_OFFSET : [xgemac_1.h](#)
- XGE_SAH_OFFSET : [xgemac_1.h](#)
- XGE_SAL_OFFSET : [xgemac_1.h](#)
- XGE_STAT_1023RXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_1023TXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_127RXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_127TXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_255RXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_255TXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_511RXOK_OFFSET : [xgemac_1.h](#)

- XGE_STAT_511TXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_64RXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_64TXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_BFRXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_BFTXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_CARRIERERR_OFFSET : [xgemac_1.h](#)
- XGE_STAT_CFRXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_CFTXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_CFUNSUP_OFFSET : [xgemac_1.h](#)
- XGE_STAT_DEFERRED_OFFSET : [xgemac_1.h](#)
- XGE_STAT_EXCESSDEF_OFFSET : [xgemac_1.h](#)
- XGE_STAT_FCSERR_OFFSET : [xgemac_1.h](#)
- XGE_STAT_FRAGRX_OFFSET : [xgemac_1.h](#)
- XGE_STAT_LATECOLL_OFFSET : [xgemac_1.h](#)
- XGE_STAT_LTERROR_OFFSET : [xgemac_1.h](#)
- XGE_STAT_MAXRXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_MAXTXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_MCOLL_OFFSET : [xgemac_1.h](#)
- XGE_STAT_MCRXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_MFTXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_OFRXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_OFTXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_PFRXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_PFTXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_REG_OFFSET : [xgemac_1.h](#)
- XGE_STAT_RXBYTES_OFFSET : [xgemac_1.h](#)
- XGE_STAT_RXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_SCOLL_OFFSET : [xgemac_1.h](#)
- XGE_STAT_TXABORTED_OFFSET : [xgemac_1.h](#)
- XGE_STAT_TXBYTES_OFFSET : [xgemac_1.h](#)
- XGE_STAT_TXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_TXURUNERR_OFFSET : [xgemac_1.h](#)
- XGE_STAT_UFRX_OFFSET : [xgemac_1.h](#)
- XGE_STAT_VLANRXOK_OFFSET : [xgemac_1.h](#)
- XGE_STAT_VLANTXOK_OFFSET : [xgemac_1.h](#)
- XGE_STRIP_PAD_FCS_OPTION : [xgemac.h](#)
- XGE_TPLR_OFFSET : [xgemac_1.h](#)
- XGE_TPPR_OFFSET : [xgemac_1.h](#)
- XGE_TSR_OFFSET : [xgemac_1.h](#)

- XGE_UNICAST_OPTION : [xgemac.h](#)
- XGE_VLAN_OPTION : [xgemac.h](#)
- XGEmac_ClearSoftStats() : [xgemac_stats.c](#), [xgemac.h](#)
- XGEmac_ConfigTable : [xgemac_i.h](#), [xgemac_g.c](#)
- XGEmac_ErrorHandler : [xgemac.h](#)
- XGEmac_FifoHandler : [xgemac.h](#)
- XGEmac_FifoRecv() : [xgemac_intr_fifo.c](#), [xgemac.h](#)
- XGEmac_FifoSend() : [xgemac_intr_fifo.c](#), [xgemac.h](#)
- XGEmac_GetHardStats() : [xgemac_stats.c](#), [xgemac.h](#)
- XGEmac_GetInterframeGap() : [xgemac_control.c](#), [xgemac.h](#)
- XGEmac_GetMacAddress() : [xgemac.h](#), [xgemac.c](#)
- XGEmac_GetOptions() : [xgemac_options.c](#), [xgemac.h](#)
- XGEmac_GetPktThreshold() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGEmac_GetPktWaitBound() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGEmac_GetSgRecvFreeDesc() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGEmac_GetSgSendFreeDesc() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGEmac_GetSoftStats() : [xgemac_stats.c](#), [xgemac.h](#)
- XGEmac_Initialize() : [xgemac.h](#), [xgemac.c](#)
- XGEmac_IntrHandlerDma() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGEmac_IntrHandlerFifo() : [xgemac_intr_fifo.c](#), [xgemac.h](#)
- XGEmac_LookupConfig() : [xgemac.h](#), [xgemac.c](#)
- XGEmac_mDisable : [xgemac_l.h](#)
- XGEmac_mEnable : [xgemac_l.h](#)
- XGEmac_MgtRead() : [xgemac_control.c](#), [xgemac.h](#)
- XGEmac_MgtWrite() : [xgemac_control.c](#), [xgemac.h](#)
- XGEmac_mIsDma : [xgemac.h](#)
- XGEmac_mIsRxEmpty : [xgemac_l.h](#)
- XGEmac_mIsSgDma : [xgemac.h](#)
- XGEmac_mIsSimpleDma : [xgemac.h](#)
- XGEmac_mIsTxDone : [xgemac_l.h](#)
- XGEmac_mPhyReset : [xgemac_l.h](#)
- XGEmac_mReadReg : [xgemac_l.h](#)
- XGEmac_mSetControlReg : [xgemac_l.h](#)
- XGEmac_mSetMacAddress : [xgemac_l.h](#)
- XGEmac_MulticastAdd() : [xgemac_multicast.c](#), [xgemac.h](#)
- XGEmac_MulticastClear() : [xgemac_multicast.c](#), [xgemac.h](#)
- XGEmac_mWriteReg : [xgemac_l.h](#)
- XGEmac_PollRecv() : [xgemac_polled.c](#), [xgemac.h](#)
- XGEmac_PollSend() : [xgemac_polled.c](#), [xgemac.h](#)

- XGemac_Reset() : [xgemac.h](#), [xgemac.c](#)
- XGemac_SelfTest() : [xgemac_selftest.c](#), [xgemac.h](#)
- XGemac_SendPause() : [xgemac_control.c](#), [xgemac.h](#)
- XGemac_SetErrorHandler() : [xgemac_intr.c](#), [xgemac.h](#)
- XGemac_SetFifoRecvHandler() : [xgemac_intr_fifo.c](#), [xgemac.h](#)
- XGemac_SetFifoSendHandler() : [xgemac_intr_fifo.c](#), [xgemac.h](#)
- XGemac_SetInterframeGap() : [xgemac_control.c](#), [xgemac.h](#)
- XGemac_SetMacAddress() : [xgemac.h](#), [xgemac.c](#)
- XGemac_SetOptions() : [xgemac_options.c](#), [xgemac.h](#)
- XGemac_SetPktThreshold() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGemac_SetPktWaitBound() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGemac_SetSgRecvHandler() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGemac_SetSgRecvSpace() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGemac_SetSgSendHandler() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGemac_SetSgSendSpace() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGemac_SgHandler : [xgemac.h](#)
- XGemac_SgRecv() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGemac_SgSend() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGemac_Start() : [xgemac.h](#), [xgemac.c](#)
- XGemac_Stop() : [xgemac.h](#), [xgemac.c](#)
- XGpio_ConfigTable : [xgpio_i.h](#), [xgpio_g.c](#)
- XGPIO_DATA2_OFFSET : [xgpio_l.h](#)
- XGPIO_DATA_OFFSET : [xgpio_l.h](#)
- XGpio_DiscreteClear() : [xgpio_extra.c](#), [xgpio.h](#)
- XGpio_DiscreteRead() : [xgpio.h](#), [xgpio.c](#)
- XGpio_DiscreteSet() : [xgpio_extra.c](#), [xgpio.h](#)
- XGpio_DiscreteWrite() : [xgpio.h](#), [xgpio.c](#)
- XGPIO_GIER_OFFSET : [xgpio_l.h](#)
- XGPIO_IER_OFFSET : [xgpio_l.h](#)
- XGpio_Initialize() : [xgpio.h](#), [xgpio.c](#)
- XGpio_InterruptClear() : [xgpio_intr.c](#), [xgpio.h](#)
- XGpio_InterruptDisable() : [xgpio_intr.c](#), [xgpio.h](#)
- XGpio_InterruptEnable() : [xgpio_intr.c](#), [xgpio.h](#)
- XGpio_InterruptGetEnabled() : [xgpio_intr.c](#), [xgpio.h](#)
- XGpio_InterruptGetStatus() : [xgpio_intr.c](#), [xgpio.h](#)
- XGpio_InterruptGlobalDisable() : [xgpio_intr.c](#), [xgpio.h](#)
- XGpio_InterruptGlobalEnable() : [xgpio_intr.c](#), [xgpio.h](#)
- XGPIO_IR_CH1_MASK : [xgpio_l.h](#)
- XGPIO_IR_CH2_MASK : [xgpio_l.h](#)

- XGPIO_IR_MASK : [xgpio_1.h](#)
- XGPIO_ISR_OFFSET : [xgpio_1.h](#)
- XGpio_LookupConfig() : [xgpio.h](#), [xgpio.c](#)
- XGpio_mGetDataReg : [xgpio_1.h](#)
- XGpio_mReadReg : [xgpio_1.h](#)
- XGpio_mSetDataReg : [xgpio_1.h](#)
- XGpio_mWriteReg : [xgpio_1.h](#)
- XGpio_SelfTest() : [xgpio_selftest.c](#), [xgpio.h](#)
- XGpio_SetDataDirection() : [xgpio.h](#), [xgpio.c](#)
- XGPIO_TRI2_OFFSET : [xgpio_1.h](#)
- XGPIO_TRI_OFFSET : [xgpio_1.h](#)
- XHD_DFT_BUFFERS : [xhdlc.h](#)
- XHD_DFT_RECV_DESC : [xhdlc.h](#)
- XHD_DFT_SEND_DESC : [xhdlc.h](#)
- XHD_MIN_BUFFERS : [xhdlc.h](#)
- XHD_MIN_RECV_DESC : [xhdlc.h](#)
- XHD_MIN_SEND_DESC : [xhdlc.h](#)
- XHD_OPTION_CRC_32 : [xhdlc.h](#)
- XHD_OPTION_CRC_DISABLE : [xhdlc.h](#)
- XHD_OPTION_RX_16_ADDR : [xhdlc.h](#)
- XHD_OPTION_RX_BROADCAST : [xhdlc.h](#)
- XHD_OPTION_RX_FILTER_ADDR : [xhdlc.h](#)
- XHD_OPTION_RX_REMOVE_ADDR : [xhdlc.h](#)
- XHD_SGDMA_DFT_THRESHOLD : [xhdlc.h](#)
- XHD_SGDMA_DFT_WAITBOUND : [xhdlc.h](#)
- XHD_SGDMA_MAX_THRESHOLD : [xhdlc.h](#)
- XHD_SGDMA_MAX_WAITBOUND : [xhdlc.h](#)
- XHdlc_ClearStats() : [xhdlc_stats.c](#), [xhdlc.h](#)
- XHdlc_ConfigTable : [xhdlc_i.h](#), [xhdlc_g.c](#)
- XHdlc_DeviceDisable() : [xhdlc_1.h](#), [xhdlc_1.c](#)
- XHdlc_DeviceEnable() : [xhdlc_1.h](#), [xhdlc_1.c](#)
- XHdlc_DisableLoopback() : [xhdlc_options.c](#), [xhdlc.h](#)
- XHdlc_DisableSubChannel() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_EnableLoopback() : [xhdlc_options.c](#), [xhdlc.h](#)
- XHdlc_EnableSubChannel() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_ErrorHandler : [xhdlc.h](#)
- XHdlc_EventFlush() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_FifoHandler : [xhdlc.h](#)
- XHdlc_GetAddress() : [xhdlc_options.c](#), [xhdlc.h](#)

- XHdlc_GetChannelUserData() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_GetOptions() : [xhdlc_options.c](#), [xhdlc.h](#)
- XHdlc_GetStats() : [xhdlc_stats.c](#), [xhdlc.h](#)
- XHdlc_Initialize() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_InitializeRegisterAccess() : [xhdlc_1.h](#), [xhdlc_1.c](#)
- XHdlc_InterruptHandlerFifo() : [xhdlc_intr_fifo.c](#), [xhdlc.h](#)
- XHdlc_InterruptHandlerSg() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_IsPhysicalSubEnabled() : [xhdlc_i.h](#), [xhdlc_i.c](#)
- XHdlc_LogicalToPhysicalSub() : [xhdlc_i.h](#), [xhdlc_i.c](#)
- XHdlc_LookupConfig() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_mChannelIDValid : [xhdlc.h](#)
- XHdlc_mCreateChannelID : [xhdlc_1.h](#)
- XHdlc_mLogicalToRxSgdmaChannel : [xhdlc_i.h](#)
- XHdlc_mLogicalToTxSgdmaChannel : [xhdlc_i.h](#)
- XHdlc_mNumberOfPhysicalChannels : [xhdlc_1.h](#)
- XHdlc_mNumberOfSubChannels : [xhdlc_1.h](#)
- XHdlc_mReadFifoSubReg : [xhdlc_1.h](#)
- XHdlc_mReadPhysChanReg : [xhdlc_1.h](#)
- XHdlc_mReadReg : [xhdlc_1.h](#)
- XHdlc_mReadSubChanReg : [xhdlc_1.h](#)
- XHdlc_mRxFifoChannelID : [xhdlc_i.h](#)
- XHdlc_mSizeNeeded : [xhdlc.h](#)
- XHdlc_mStatusFifoChannelID : [xhdlc_1.h](#)
- XHdlc_mTxFifoChannelID : [xhdlc_i.h](#)
- XHdlc_mWriteFifoSubReg : [xhdlc_1.h](#)
- XHdlc_mWritePhysChanReg : [xhdlc_1.h](#)
- XHdlc_mWriteReg : [xhdlc_1.h](#)
- XHdlc_mWriteSubChanReg : [xhdlc_1.h](#)
- XHdlc_PhysDisable() : [xhdlc_1.h](#), [xhdlc_1.c](#)
- XHdlc_PhysEnable() : [xhdlc_1.h](#), [xhdlc_1.c](#)
- XHdlc_PhysicalSubToLogical() : [xhdlc_i.h](#), [xhdlc_i.c](#)
- XHdlc_PolledSubEnable() : [xhdlc_1.h](#), [xhdlc_1.c](#)
- XHdlc_PolledSubTxEnable() : [xhdlc_1.h](#), [xhdlc_1.c](#)
- XHdlc_PollRecv() : [xhdlc_1.h](#), [xhdlc_1.c](#)
- XHdlc_PollSend() : [xhdlc_1.h](#), [xhdlc_1.c](#)
- XHdlc_Recv() : [xhdlc_intr_fifo.c](#), [xhdlc.h](#)
- XHdlc_Reset() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_SelfTest() : [xhdlc_selftest.c](#), [xhdlc.h](#)
- XHdlc_Send() : [xhdlc_intr_fifo.c](#), [xhdlc.h](#)

- XHdlc_SetAddress() : [xhdlc_options.c](#), [xhdlc.h](#)
- XHdlc_SetChannelUserData() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_SetErrorHandler() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_SetFifoRecvHandler() : [xhdlc_intr_fifo.c](#), [xhdlc.h](#)
- XHdlc_SetFifoSendHandler() : [xhdlc_intr_fifo.c](#), [xhdlc.h](#)
- XHdlc_SetOptions() : [xhdlc_options.c](#), [xhdlc.h](#)
- XHdlc_SetSgRecvHandler() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SetSgRecvSpace() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SetSgSendHandler() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SetSgSendSpace() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SgGetRecvFrame() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SgGetSendFrame() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SgHandler : [xhdlc.h](#)
- XHdlc_SgRecv() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SgSend() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_Start() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_Stop() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_SubDisable() : [xhdlc_l.h](#), [xhdlc_l.c](#)
- XHdlc_SubEnable() : [xhdlc_l.h](#), [xhdlc_l.c](#)
- XHdlc_SubTxDisable() : [xhdlc_l.h](#), [xhdlc_l.c](#)
- XHdlc_SubTxEnable() : [xhdlc_l.h](#), [xhdlc_l.c](#)
- XHI_CLB_FF : [xhwicap_clb_ff.h](#)
- XHI_CLB_LUT : [xhwicap_clb_lut.h](#)
- XHI_CLB_LUT_F : [xhwicap_clb_lut.h](#)
- XHI_CLB_LUT_G : [xhwicap_clb_lut.h](#)
- XHI_CLB_SRINV : [xhwicap_clb_srinvs.h](#)
- XHI_CLB_XQ : [xhwicap_clb_ff.h](#)
- XHI_CLB_YQ : [xhwicap_clb_ff.h](#)
- XHwIcap_CommandCapture() : [xhwicap_srp.c](#), [xhwicap.h](#)
- XHwIcap_CommandDesync() : [xhwicap_srp.c](#), [xhwicap.h](#)
- XHwIcap_ConfigTable : [xhwicap_i.h](#), [xhwicap_g.c](#)
- XHwIcap_DeviceRead() : [xhwicap_srp.c](#), [xhwicap.h](#)
- XHwIcap_DeviceReadFrame() : [xhwicap_device_read_frame.c](#), [xhwicap.h](#)
- XHwIcap_DeviceWrite() : [xhwicap_srp.c](#), [xhwicap.h](#)
- XHwIcap_DeviceWriteFrame() : [xhwicap_device_write_frame.c](#), [xhwicap.h](#)
- XHwIcap_GetClbBits() : [xhwicap.h](#)
- XHwIcap_GetConfigReg() : [xhwicap_srp.c](#), [xhwicap.h](#)
- XHwIcap_Initialize() : [xhwicap_srp.c](#), [xhwicap.h](#)
- XHwIcap_mGetBram : [xhwicap_l.h](#)

- XHwIcap_mGetDoneReg : [xhwicap_1.h](#)
- XHwIcap_mGetOffsetReg : [xhwicap_1.h](#)
- XHwIcap_mGetSizeReg : [xhwicap_1.h](#)
- XHwIcap_mGetStatusReg : [xhwicap_1.h](#)
- XHwIcap_mSetBram : [xhwicap_1.h](#)
- XHwIcap_mSetOffsetReg : [xhwicap_1.h](#)
- XHwIcap_mSetRncReg : [xhwicap_1.h](#)
- XHwIcap_mSetSizeReg : [xhwicap_1.h](#)
- XHwIcap_mSliceX2Col : [xhwicap.h](#)
- XHwIcap_mSliceXY2Slice : [xhwicap.h](#)
- XHwIcap_mSliceY2Row : [xhwicap.h](#)
- XHwIcap_SetClbBits() : [xhwicap.h](#)
- XHwIcap_SetConfiguration() : [xhwicap_set_configuration.c](#), [xhwicap.h](#)
- XHwIcap_SetupFar : [xhwicap_i.h](#)
- XHwIcap_StorageBufferRead() : [xhwicap_srp.c](#), [xhwicap.h](#)
- XHwIcap_StorageBufferWrite() : [xhwicap_srp.c](#), [xhwicap.h](#)
- XHwIcap_Type1Read : [xhwicap_i.h](#)
- XHwIcap_Type1Write : [xhwicap_i.h](#)
- XII_ADDR_TO_RESPOND_TYPE : [xiic.h](#)
- XII_ADDR_TO_SEND_TYPE : [xiic.h](#)
- XII_ARB_LOST_EVENT : [xiic.h](#)
- XII_BUS_NOT_BUSY_EVENT : [xiic.h](#)
- XII_GENERAL_CALL_EVENT : [xiic.h](#)
- XII_GENERAL_CALL_OPTION : [xiic.h](#)
- XII_MASTER_READ_EVENT : [xiic.h](#)
- XII_MASTER_WRITE_EVENT : [xiic.h](#)
- XII_REPEATED_START_OPTION : [xiic.h](#)
- XII_SEND_10_BIT_OPTION : [xiic.h](#)
- XII_SLAVE_NO_ACK_EVENT : [xiic.h](#)
- Xlic_ClearStats() : [xiic_stats.c](#), [xiic.h](#)
- Xlic_ConfigTable : [xiic_i.h](#), [xiic_g.c](#)
- Xlic_GetAddress() : [xiic.h](#), [xiic.c](#)
- Xlic_GetGpOutput() : [xiic.h](#), [xiic.c](#)
- Xlic_GetOptions() : [xiic_options.c](#), [xiic.h](#)
- Xlic_GetStats() : [xiic_stats.c](#), [xiic.h](#)
- Xlic_Handler : [xiic.h](#)
- Xlic_Initialize() : [xiic.h](#), [xiic.c](#)
- Xlic_InterruptHandler() : [xiic_intr.c](#), [xiic.h](#)
- Xlic_IsSlave() : [xiic.h](#), [xiic.c](#)

- `XIic_LookupConfig()` : [xiic.h](#), [xiic.c](#)
- `XIic_MasterRecv()` : [xiic_master.c](#), [xiic.h](#)
- `XIic_MasterSend()` : [xiic_master.c](#), [xiic.h](#)
- `XIic_MultiMasterInclude()` : [xiic_multi_master.c](#), [xiic.h](#)
- `XIic_Recv()` : [xiic_l.h](#), [xiic_l.c](#)
- `XIic_Reset()` : [xiic.h](#), [xiic.c](#)
- `XIic_SelfTest()` : [xiic_selftest.c](#), [xiic.h](#)
- `XIic_Send()` : [xiic_l.h](#), [xiic_l.c](#)
- `XIic_SetAddress()` : [xiic.h](#), [xiic.c](#)
- `XIic_SetGpOutput()` : [xiic.h](#), [xiic.c](#)
- `XIic_SetOptions()` : [xiic_options.c](#), [xiic.h](#)
- `XIic_SetRecvHandler()` : [xiic.h](#), [xiic.c](#)
- `XIic_SetSendHandler()` : [xiic.h](#), [xiic.c](#)
- `XIic_SetStatusHandler()` : [xiic.h](#), [xiic.c](#)
- `XIic_Start()` : [xiic.h](#), [xiic.c](#)
- `XIic_StatusHandler` : [xiic.h](#)
- `XIic_Stop()` : [xiic.h](#), [xiic.c](#)
- `XIN_REAL_MODE` : [xintc.h](#)
- `XIN_SIMULATION_MODE` : [xintc.h](#)
- `XIN_SVC_ALL_ISRS_OPTION` : [xintc.h](#)
- `XIN_SVC_SGL_ISR_OPTION` : [xintc.h](#)
- `Xint16` : [xbasic_types.h](#)
- `Xint32` : [xbasic_types.h](#)
- `Xint8` : [xbasic_types.h](#)
- `XIntc_Acknowledge()` : [xintc.h](#), [xintc.c](#)
- `XIntc_ConfigTable` : [xintc_i.h](#), [xintc_g.c](#)
- `XIntc_Connect()` : [xintc.h](#), [xintc.c](#)
- `XIntc_DeviceInterruptHandler()` : [xintc_l.h](#), [xintc_l.c](#)
- `XIntc_Disable()` : [xintc.h](#), [xintc.c](#)
- `XIntc_Disconnect()` : [xintc.h](#), [xintc.c](#)
- `XIntc_Enable()` : [xintc.h](#), [xintc.c](#)
- `XIntc_GetOptions()` : [xintc_options.c](#), [xintc.h](#)
- `XIntc_Initialize()` : [xintc.h](#), [xintc.c](#)
- `XIntc_InterruptHandler()` : [xintc_intr.c](#), [xintc.h](#)
- `XIntc_LookupConfig()` : [xintc.h](#), [xintc.c](#)
- `XIntc_mAckIntr` : [xintc_l.h](#)
- `XIntc_mDisableIntr` : [xintc_l.h](#)
- `XIntc_mEnableIntr` : [xintc_l.h](#)
- `XIntc_mGetIntrStatus` : [xintc_l.h](#)

- XIntc_mMasterDisable : [xintc_1.h](#)
- XIntc_mMasterEnable : [xintc_1.h](#)
- XIntc_RegisterHandler() : [xintc_1.h](#), [xintc_1.c](#)
- XIntc_SelfTest() : [xintc_selftest.c](#), [xintc.h](#)
- XIntc_SetIntrSvcOption() : [xintc_1.h](#), [xintc_1.c](#)
- XIntc_SetOptions() : [xintc_options.c](#), [xintc.h](#)
- XIntc_SimulateIntr() : [xintc_selftest.c](#), [xintc.h](#)
- XIntc_Start() : [xintc.h](#), [xintc.c](#)
- XIntc_Stop() : [xintc.h](#), [xintc.c](#)
- XInterruptHandler : [xbasic_types.h](#)
- XIo_Address : [xio.h](#)
- XIo_DcrIn() : [xio_dcr.h](#), [xio_dcr.c](#)
- XIo_DcrOut() : [xio_dcr.h](#), [xio_dcr.c](#)
- XIo_EndianSwap16() : [xio.h](#), [xio.c](#)
- XIo_EndianSwap16OLD() : [xio.c](#)
- XIo_EndianSwap32() : [xio.h](#), [xio.c](#)
- XIo_EndianSwap32OLD() : [xio.c](#)
- XIo_In16 : [xio.h](#), [xio.c](#), [xio.h](#)
- XIo_In32 : [xio.h](#), [xio.c](#), [xio.h](#)
- XIo_In8 : [xio.h](#), [xio.c](#), [xio.h](#)
- XIo_InSwap16() : [xio.h](#), [xio.c](#), [xio.h](#), [xio.c](#)
- XIo_InSwap32() : [xio.h](#), [xio.c](#), [xio.h](#), [xio.c](#)
- XIo_Out16 : [xio.h](#), [xio.c](#), [xio.h](#)
- XIo_Out32 : [xio.h](#), [xio.c](#), [xio.h](#)
- XIo_Out8 : [xio.h](#), [xio.c](#), [xio.h](#)
- XIo_OutSwap16() : [xio.h](#), [xio.c](#), [xio.h](#), [xio.c](#)
- XIo_OutSwap32() : [xio.h](#), [xio.c](#), [xio.h](#), [xio.c](#)
- XNULL : [xbasic_types.h](#)
- XNullHandler() : [xbasic_types.h](#), [xbasic_types.c](#)
- XO2P_NO_ERROR : [xopb2plb.h](#)
- XO2P_READ_ERROR : [xopb2plb.h](#)
- XO2P_WRITE_ERROR : [xopb2plb.h](#)
- XOA_DYNAMIC_PRIORITY_OPTION : [xopbarb.h](#)
- XOA_PARK_BY_ID_OPTION : [xopbarb.h](#)
- XOA_PARK_ENABLE_OPTION : [xopbarb.h](#)
- XOpb2Plb_ClearErrors() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_ConfigTable : [xopb2plb_i.h](#), [xopb2plb_g.c](#)
- XOpb2Plb_DisableInterrupt() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_DisableLock() : [xopb2plb.h](#), [xopb2plb.c](#)

- XOpb2Plb_EnableInterrupt() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_EnableLock() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_GetErrorAddress() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_GetErrorStatus() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_Initialize() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_IsError() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_LookupConfig() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_mGetBearReg : [xopb2plb_1.h](#)
- XOpb2Plb_mGetBesrReg : [xopb2plb_1.h](#)
- XOpb2Plb_mGetControlReg : [xopb2plb_1.h](#)
- XOpb2Plb_mSetControlReg : [xopb2plb_1.h](#)
- XOpb2Plb_Reset() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_SelfTest() : [xopb2plb_selftest.c](#), [xopb2plb.h](#)
- XOpbArb_ConfigTable : [xopbarb_g.c](#)
- XOpbArb_GetOptions() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_GetParkId() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_GetPriorityLevel() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_Initialize() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_LookupConfig() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_mClearParkMasterNot : [xopbarb_1.h](#)
- XOpbArb_mClearPriorityRegsValid : [xopbarb_1.h](#)
- XOpbArb_mDisableDynamic : [xopbarb_1.h](#)
- XOpbArb_mDisableParking : [xopbarb_1.h](#)
- XOpbArb_mEnableDynamic : [xopbarb_1.h](#)
- XOpbArb_mEnableParking : [xopbarb_1.h](#)
- XOpbArb_mGetControlReg : [xopbarb_1.h](#)
- XOpbArb_mGetPriorityReg : [xopbarb_1.h](#)
- XOpbArb_mSetControlReg : [xopbarb_1.h](#)
- XOpbArb_mSetParkedMasterId : [xopbarb_1.h](#)
- XOpbArb_mSetParkMasterNot : [xopbarb_1.h](#)
- XOpbArb_mSetPriorityReg : [xopbarb_1.h](#)
- XOpbArb_mSetPriorityRegsValid : [xopbarb_1.h](#)
- XOpbArb_ResumePriorityLevels() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_SelfTest() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_SetOptions() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_SetParkId() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_SetPriorityLevel() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_SuspendPriorityLevels() : [xopbarb.h](#), [xopbarb.c](#)
- XP2O_DRIVING_BEAR_MASK : [xplb2opb.h](#)

- XP2O_ERROR_READ_MASK : [xplb2opb.h](#)
- XP2O_ERROR_TYPE_MASK : [xplb2opb.h](#)
- XP2O_LOCK_ERR_MASK : [xplb2opb.h](#)
- XPacketFifoV200a_Initialize() : [xpacket_fifo_v2_00_a.h](#), [xpacket_fifo_v2_00_a.c](#)
- XPacketFifoV200a_Read() : [xpacket_fifo_v2_00_a.h](#), [xpacket_fifo_v2_00_a.c](#)
- XPacketFifoV200a_SelfTest() : [xpacket_fifo_v2_00_a.h](#), [xpacket_fifo_v2_00_a.c](#)
- XPacketFifoV200a_Write() : [xpacket_fifo_v2_00_a.h](#), [xpacket_fifo_v2_00_a.c](#)
- XPacketFifoV200a_WriteDre() : [xpacket_fifo_v2_00_a.h](#), [xpacket_fifo_v2_00_a.c](#)
- XPCA_CR_FIXED_ENABLE_MASK : [xpciarb_1.h](#)
- XPCA_CR_OFFSET : [xpciarb_1.h](#)
- XPCA_CR_PARK_ENABLE_MASK : [xpciarb_1.h](#)
- XPCA_MIR_OFFSET : [xpciarb_1.h](#)
- XPCA_PARK_OFFSET : [xpciarb_1.h](#)
- XPCA_RESET_MASK : [xpciarb_1.h](#)
- XPCA_RESET_OFFSET : [xpciarb_1.h](#)
- XPCI_AckRead() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_AckSend() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_BUSNO_BUS_MASK : [xpci_1.h](#)
- XPCI_BUSNO_OFFSET : [xpci_1.h](#)
- XPCI_BUSNO_SUBBUS_MASK : [xpci_1.h](#)
- XPCI_CAR_OFFSET : [xpci_1.h](#)
- XPCI_CDR_OFFSET : [xpci_1.h](#)
- XPCI_ConfigIn() : [xpci.h](#)
- XPCI_ConfigIn16() : [xpci_config.c](#), [xpci.h](#)
- XPCI_ConfigIn32() : [xpci_config.c](#), [xpci.h](#)
- XPCI_ConfigIn8() : [xpci_config.c](#), [xpci.h](#)
- XPCI_ConfigOut() : [xpci.h](#)
- XPCI_ConfigOut16() : [xpci_config.c](#), [xpci.h](#)
- XPCI_ConfigOut32() : [xpci_config.c](#), [xpci.h](#)
- XPCI_ConfigOut8() : [xpci_config.c](#), [xpci.h](#)
- XPCI_ConfigPack() : [xpci.h](#)
- XPCI_DMA_TYPE_NONE : [xpci_1.h](#)
- XPCI_DMA_TYPE_SG : [xpci_1.h](#)
- XPCI_DMA_TYPE_SIMPLE : [xpci_1.h](#)
- XPCI_ErrorClear() : [xpci.h](#)
- XPCI_ErrorGet() : [xpci.h](#)
- XPCI_GetBusNumber() : [xpci.h](#)
- XPCI_GetDmaImplementation() : [xpci.h](#)
- XPCI_HDR_BAR0 : [xpci_1.h](#)

- XPCI_HDR_BAR1 : [xpci_1.h](#)
- XPCI_HDR_BAR2 : [xpci_1.h](#)
- XPCI_HDR_BAR3 : [xpci_1.h](#)
- XPCI_HDR_BAR4 : [xpci_1.h](#)
- XPCI_HDR_BAR5 : [xpci_1.h](#)
- XPCI_HDR_BAR_ADDR_MASK : [xpci_1.h](#)
- XPCI_HDR_BAR_PREFETCH_NO : [xpci_1.h](#)
- XPCI_HDR_BAR_PREFETCH_YES : [xpci_1.h](#)
- XPCI_HDR_BAR_SPACE_IO : [xpci_1.h](#)
- XPCI_HDR_BAR_SPACE_MEMORY : [xpci_1.h](#)
- XPCI_HDR_BAR_TYPE_ABOVE_4GB : [xpci_1.h](#)
- XPCI_HDR_BAR_TYPE_BELOW_1MB : [xpci_1.h](#)
- XPCI_HDR_BAR_TYPE_BELOW_4GB : [xpci_1.h](#)
- XPCI_HDR_BAR_TYPE_MASK : [xpci_1.h](#)
- XPCI_HDR_BAR_TYPE_RESERVED : [xpci_1.h](#)
- XPCI_HDR_BIST : [xpci_1.h](#)
- XPCI_HDR_CACHE_LINE_SZ : [xpci_1.h](#)
- XPCI_HDR_CAP_PTR : [xpci_1.h](#)
- XPCI_HDR_CARDBUS_PTR : [xpci_1.h](#)
- XPCI_HDR_CLASSCODE : [xpci_1.h](#)
- XPCI_HDR_COMMAND : [xpci_1.h](#)
- XPCI_HDR_DEVICE : [xpci_1.h](#)
- XPCI_HDR_INT_LINE : [xpci_1.h](#)
- XPCI_HDR_INT_PIN : [xpci_1.h](#)
- XPCI_HDR_LATENCY : [xpci_1.h](#)
- XPCI_HDR_MAX_LAT : [xpci_1.h](#)
- XPCI_HDR_MIN_GNT : [xpci_1.h](#)
- XPCI_HDR_NUM_BAR : [xpci_1.h](#)
- XPCI_HDR_REVID : [xpci_1.h](#)
- XPCI_HDR_ROM_BASE : [xpci_1.h](#)
- XPCI_HDR_STATUS : [xpci_1.h](#)
- XPCI_HDR_SUB_DEVICE : [xpci_1.h](#)
- XPCI_HDR_SUB_VENDOR : [xpci_1.h](#)
- XPCI_HDR_TYPE : [xpci_1.h](#)
- XPCI_HDR_VENDOR : [xpci_1.h](#)
- XPCI_IAR_OFFSET : [xpci_1.h](#)
- XPCI_INHIBIT_LOCAL_BUS_R : [xpci_1.h](#)
- XPCI_INHIBIT_LOCAL_BUS_W : [xpci_1.h](#)
- XPCI_INHIBIT_MASK : [xpci_1.h](#)

- XPCI_INHIBIT_OFFSET : [xpci_1.h](#)
- XPCI_INHIBIT_PCI_R : [xpci_1.h](#)
- XPCI_INHIBIT_PCI_W : [xpci_1.h](#)
- XPCI_InhibitAfterError() : [xpci.h](#)
- XPCI_Initialize() : [xpci.h](#)
- XPCI_InterruptClear() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptDisable() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptEnable() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptGetEnabled() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptGetHighestPending() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptGetPending() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptGetStatus() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptGlobalDisable() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptGlobalEnable() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptPciClear() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptPciDisable() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptPciEnable() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptPciGetEnabled() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_InterruptPciGetStatus() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_IR_LM_BR_W : [xpci_1.h](#)
- XPCI_IR_LM_BRANGE_W : [xpci_1.h](#)
- XPCI_IR_LM_BRD_W : [xpci_1.h](#)
- XPCI_IR_LM_BRT_W : [xpci_1.h](#)
- XPCI_IR_LM_MA_W : [xpci_1.h](#)
- XPCI_IR_LM_PERR_R : [xpci_1.h](#)
- XPCI_IR_LM_PERR_W : [xpci_1.h](#)
- XPCI_IR_LM_SERR_R : [xpci_1.h](#)
- XPCI_IR_LM_SERR_W : [xpci_1.h](#)
- XPCI_IR_LM_TA_R : [xpci_1.h](#)
- XPCI_IR_LM_TA_W : [xpci_1.h](#)
- XPCI_IR_MASK : [xpci_1.h](#)
- XPCI_IR_PI_SERR_R : [xpci_1.h](#)
- XPCI_IR_PI_SERR_W : [xpci_1.h](#)
- XPCI_LMA_R_OFFSET : [xpci_1.h](#)
- XPCI_LMA_W_OFFSET : [xpci_1.h](#)
- XPCI_LMADDR_BR_W : [xpci_1.h](#)
- XPCI_LMADDR_BRANGE_W : [xpci_1.h](#)
- XPCI_LMADDR_BRD_W : [xpci_1.h](#)
- XPCI_LMADDR_BRT_W : [xpci_1.h](#)

- XPCI_LMADDR_MA_W : [xpci_1.h](#)
- XPCI_LMADDR_MASK : [xpci_1.h](#)
- XPCI_LMADDR_OFFSET : [xpci_1.h](#)
- XPCI_LMADDR_PERR_R : [xpci_1.h](#)
- XPCI_LMADDR_PERR_W : [xpci_1.h](#)
- XPCI_LMADDR_SERR_R : [xpci_1.h](#)
- XPCI_LMADDR_SERR_W : [xpci_1.h](#)
- XPCI_LMADDR_TA_R : [xpci_1.h](#)
- XPCI_LMADDR_TA_W : [xpci_1.h](#)
- XPci_LookupConfig() : [xpci.h](#)
- XPci_mAckRead : [xpci_1.h](#)
- XPci_mAckSend : [xpci_1.h](#)
- XPci_mConfigIn : [xpci_1.h](#)
- XPci_mConfigOut : [xpci_1.h](#)
- XPci_mIntrClear : [xpci_1.h](#)
- XPci_mIntrDisable : [xpci_1.h](#)
- XPci_mIntrEnable : [xpci_1.h](#)
- XPci_mIntrGlobalDisable : [xpci_1.h](#)
- XPci_mIntrGlobalEnable : [xpci_1.h](#)
- XPci_mIntrPciClear : [xpci_1.h](#)
- XPci_mIntrPciDisable : [xpci_1.h](#)
- XPci_mIntrPciEnable : [xpci_1.h](#)
- XPci_mIntrPciReadIER : [xpci_1.h](#)
- XPci_mIntrPciReadISR : [xpci_1.h](#)
- XPci_mIntrPciWriteISR : [xpci_1.h](#)
- XPci_mIntrReadID : [xpci_1.h](#)
- XPci_mIntrReadIER : [xpci_1.h](#)
- XPci_mIntrReadIPR : [xpci_1.h](#)
- XPci_mIntrReadISR : [xpci_1.h](#)
- XPci_mIntrWriteISR : [xpci_1.h](#)
- XPci_mLocal2Pci : [xpci_1.h](#)
- XPci_mPci2Local : [xpci_1.h](#)
- XPci_mReadReg : [xpci_1.h](#)
- XPci_mReset : [xpci_1.h](#)
- XPci_mSpecialCycle : [xpci_1.h](#)
- XPci_mWriteReg : [xpci_1.h](#)
- XPCI_PIA_R_OFFSET : [xpci_1.h](#)
- XPCI_PIA_W_OFFSET : [xpci_1.h](#)
- XPCI_PIADDR_ERRACK_R : [xpci_1.h](#)

- XPCI_PIADDR_ERRACK_W : [xpci_1.h](#)
- XPCI_PIADDR_MASK : [xpci_1.h](#)
- XPCI_PIADDR_OFFSET : [xpci_1.h](#)
- XPCI_PIADDR_RANGE_W : [xpci_1.h](#)
- XPCI_PIADDR_RETRY_W : [xpci_1.h](#)
- XPCI_PIADDR_TIMEOUT_W : [xpci_1.h](#)
- XPCI_PREOVRD_OFFSET : [xpci_1.h](#)
- XPCI_Reset() : [xpci.h](#)
- XPCI_SC_DATA_OFFSET : [xpci_1.h](#)
- XPCI_SelfTest() : [xpci_selftest.c](#), [xpci.h](#)
- XPCI_SERR_R_OFFSET : [xpci_1.h](#)
- XPCI_SERR_W_OFFSET : [xpci_1.h](#)
- XPCI_SetBusNumber() : [xpci.h](#)
- XPCI_SpecialCycle() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_STATCMD_66MHZ_CAP : [xpci_1.h](#)
- XPCI_STATCMD_BACK_EN : [xpci_1.h](#)
- XPCI_STATCMD_BUSM_EN : [xpci_1.h](#)
- XPCI_STATCMD_DEVSEL_FAST : [xpci_1.h](#)
- XPCI_STATCMD_DEVSEL_MED : [xpci_1.h](#)
- XPCI_STATCMD_DEVSEL_MSK : [xpci_1.h](#)
- XPCI_STATCMD_ERR_MASK : [xpci_1.h](#)
- XPCI_STATCMD_INT_DISABLE : [xpci_1.h](#)
- XPCI_STATCMD_INT_STATUS : [xpci_1.h](#)
- XPCI_STATCMD_IO_EN : [xpci_1.h](#)
- XPCI_STATCMD_MEM_EN : [xpci_1.h](#)
- XPCI_STATCMD_MEMWR_INV_EN : [xpci_1.h](#)
- XPCI_STATCMD_MPERR : [xpci_1.h](#)
- XPCI_STATCMD_MSTABRT_RCV : [xpci_1.h](#)
- XPCI_STATCMD_OFFSET : [xpci_1.h](#)
- XPCI_STATCMD_PARITY : [xpci_1.h](#)
- XPCI_STATCMD_PERR_DET : [xpci_1.h](#)
- XPCI_STATCMD_SERR_EN : [xpci_1.h](#)
- XPCI_STATCMD_SERR_SIG : [xpci_1.h](#)
- XPCI_STATCMD_SPECIALCYC : [xpci_1.h](#)
- XPCI_STATCMD_STEPPING : [xpci_1.h](#)
- XPCI_STATCMD_TGTABRT_RCV : [xpci_1.h](#)
- XPCI_STATCMD_TGTABRT_SIG : [xpci_1.h](#)
- XPCI_STATCMD_VGA_SNOOP_EN : [xpci_1.h](#)
- XPCI_STATV3_DATA_XFER : [xpci_1.h](#)

- XPCI_STATV3_DISC_WDATA : [xpci_1.h](#)
- XPCI_STATV3_DISC_WODATA : [xpci_1.h](#)
- XPCI_STATV3_MASK : [xpci_1.h](#)
- XPCI_STATV3_MASTER_ABRT : [xpci_1.h](#)
- XPCI_STATV3_NORM_TERM : [xpci_1.h](#)
- XPCI_STATV3_OFFSET : [xpci_1.h](#)
- XPCI_STATV3_PCI_RETRY_R : [xpci_1.h](#)
- XPCI_STATV3_PCI_RETRY_W : [xpci_1.h](#)
- XPCI_STATV3_TGT_ABRT : [xpci_1.h](#)
- XPCI_STATV3_TGT_TERM : [xpci_1.h](#)
- XPCI_STATV3_TRANS_END : [xpci_1.h](#)
- XPCI_STATV3_WRITE_BUSY : [xpci_1.h](#)
- XPci_V3StatusCommandGet() : [xpci_v3.c](#), [xpci.h](#)
- XPci_V3TransactionStatusClear() : [xpci_v3.c](#), [xpci.h](#)
- XPci_V3TransactionStatusGet() : [xpci_v3.c](#), [xpci.h](#)
- XPciArb_ConfigTable : [xpciarb_g.c](#), [xpciarb.c](#)
- XPciArb_DisableFixed() : [xpciarb.h](#), [xpciarb.c](#)
- XPciArb_DisableParkReg() : [xpciarb.h](#), [xpciarb.c](#)
- XPciArb_EnableFixed() : [xpciarb.h](#), [xpciarb.c](#)
- XPciArb_EnableParkReg() : [xpciarb.h](#), [xpciarb.c](#)
- XPciArb_Initialize() : [xpciarb.h](#), [xpciarb.c](#)
- XPciArb_LookupConfig() : [xpciarb.h](#), [xpciarb.c](#)
- XPciArb_mReadReg : [xpciarb_1.h](#)
- XPciArb_mWriteReg : [xpciarb_1.h](#)
- XPciArb_Reset() : [xpciarb.h](#), [xpciarb.c](#)
- XPciArb_SelfTest() : [xpciarb_selftest.c](#), [xpciarb.h](#)
- XPF_V200A_GET_COUNT : [xpacket_fifo_v2_00_a.h](#)
- XPF_V200A_IS_ALMOST_EMPTY : [xpacket_fifo_v2_00_a.h](#)
- XPF_V200A_IS_ALMOST_FULL : [xpacket_fifo_v2_00_a.h](#)
- XPF_V200A_IS_DEADLOCKED : [xpacket_fifo_v2_00_a.h](#)
- XPF_V200A_IS_EMPTY : [xpacket_fifo_v2_00_a.h](#)
- XPF_V200A_IS_FULL : [xpacket_fifo_v2_00_a.h](#)
- XPF_V200A_RESET : [xpacket_fifo_v2_00_a.h](#)
- XPlb2Opb_ClearErrors() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_ConfigTable : [xplb2opb_i.h](#), [xplb2opb_g.c](#)
- XPlb2Opb_DisableInterrupt() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_EnableInterrupt() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_GetErrorAddress() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_GetErrorByteEnables() : [xplb2opb.h](#), [xplb2opb.c](#)

- XPlb2Opb_GetErrorStatus() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_GetMasterDrivingError() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_GetNumMasters() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_Initialize() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_IsError() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_LookupConfig() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_mGetByteEnableReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetControlReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetErrorAddressReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetErrorDetectReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetErrorTypeReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetLockBitReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetMasterDrivingReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetReadWriteReg : [xplb2opb_1.h](#)
- XPlb2Opb_mSetControlReg : [xplb2opb_1.h](#)
- XPlb2Opb_mSetErrorDetectReg : [xplb2opb_1.h](#)
- XPlb2Opb_Reset() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_SelfTest() : [xplb2opb_selftest.c](#), [xplb2opb.h](#)
- XPlbArb_ClearErrors() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_ConfigTable : [xplbarb_i.h](#), [xplbarb_g.c](#)
- XPlbArb_DisableInterrupt() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_EnableInterrupt() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_GetErrorAddress() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_GetErrorStatus() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_GetNumMasters() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_Initialize() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_IsError() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_LookupConfig() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_mDisableInterrupt : [xplbarb_1.h](#)
- XPlbArb_mEnableInterrupt : [xplbarb_1.h](#)
- XPlbArb_mGetControlReg : [xplbarb_1.h](#)
- XPlbArb_mGetParrAddrReg : [xplbarb_1.h](#)
- XPlbArb_mGetParrByteEnReg : [xplbarb_1.h](#)
- XPlbArb_mGetParrLockReg : [xplbarb_1.h](#)
- XPlbArb_mGetParrMDriveReg : [xplbarb_1.h](#)
- XPlbArb_mGetParrMerrReg : [xplbarb_1.h](#)
- XPlbArb_mGetParrRnwReg : [xplbarb_1.h](#)
- XPlbArb_mReset : [xplbarb_1.h](#)
- XPlbArb_mSetParrMerrReg : [xplbarb_1.h](#)

- XPlbArb_Reset() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_SelfTest() : [xplbarb_selftest.c](#), [xplbarb.h](#)
- XRRapidIo_mGetLinkStatus : [xrapidio_1.h](#)
- XRRapidIo_mReadReg : [xrapidio_1.h](#)
- XRRapidIo_mReset : [xrapidio_1.h](#)
- XRRapidIo_mWriteReg : [xrapidio_1.h](#)
- XRRapidIo_RecvPkt() : [xrapidio_1.h](#), [xrapidio_1.c](#)
- XRRapidIo_SendPkt() : [xrapidio_1.h](#), [xrapidio_1.c](#)
- XSA_BMR_16BIT_MASK : [xsysace_1.h](#)
- XSA_BMR_OFFSET : [xsysace_1.h](#)
- XSA_CF_SECTOR_SIZE : [xsysace_1.h](#)
- XSA_CLR_LBA_MASK : [xsysace_1.h](#)
- XSA_CLR_OFFSET : [xsysace_1.h](#)
- XSA_CR_CFGADDR_MASK : [xsysace_1.h](#)
- XSA_CR_CFGADDR_SHIFT : [xsysace_1.h](#)
- XSA_CR_CFGDONEIRQ_MASK : [xsysace_1.h](#)
- XSA_CR_CFGMODE_MASK : [xsysace_1.h](#)
- XSA_CR_CFGPROG_MASK : [xsysace_1.h](#)
- XSA_CR_CFGRESET_MASK : [xsysace_1.h](#)
- XSA_CR_CFGSEL_MASK : [xsysace_1.h](#)
- XSA_CR_CFGSTART_MASK : [xsysace_1.h](#)
- XSA_CR_DATARDYIRQ_MASK : [xsysace_1.h](#)
- XSA_CR_ERRORIRQ_MASK : [xsysace_1.h](#)
- XSA_CR_FORCECFGADDR_MASK : [xsysace_1.h](#)
- XSA_CR_FORCECFGMODE_MASK : [xsysace_1.h](#)
- XSA_CR_FORCELOCK_MASK : [xsysace_1.h](#)
- XSA_CR_LOCKREQ_MASK : [xsysace_1.h](#)
- XSA_CR_OFFSET : [xsysace_1.h](#)
- XSA_CR_RESETIRQ_MASK : [xsysace_1.h](#)
- XSA_DATA_BUFFER_SIZE : [xsysace_1.h](#)
- XSA_DBR_OFFSET : [xsysace_1.h](#)
- XSA_ER_ABORT : [xsysace_1.h](#)
- XSA_ER_BAD_BLOCK : [xsysace_1.h](#)
- XSA_ER_CARD_READ : [xsysace_1.h](#)
- XSA_ER_CARD_READY : [xsysace_1.h](#)
- XSA_ER_CARD_RESET : [xsysace_1.h](#)
- XSA_ER_CARD_WRITE : [xsysace_1.h](#)
- XSA_ER_CFG_ADDR : [xsysace_1.h](#)
- XSA_ER_CFG_FAIL : [xsysace_1.h](#)

- XSA_ER_CFG_INIT : [xsysace_1.h](#)
- XSA_ER_CFG_INSTR : [xsysace_1.h](#)
- XSA_ER_CFG_READ : [xsysace_1.h](#)
- XSA_ER_GENERAL : [xsysace_1.h](#)
- XSA_ER_OFFSET : [xsysace_1.h](#)
- XSA_ER_RESERVED : [xsysace_1.h](#)
- XSA_ER_SECTOR_ID : [xsysace_1.h](#)
- XSA_ER_SECTOR_READY : [xsysace_1.h](#)
- XSA_ER_UNCORRECTABLE : [xsysace_1.h](#)
- XSA_EVENT_CFG_DONE : [xsysace.h](#)
- XSA_EVENT_DATA_DONE : [xsysace.h](#)
- XSA_EVENT_ERROR : [xsysace.h](#)
- XSA_FAT_12_BOOT_REC : [xsysace_1.h](#)
- XSA_FAT_12_CALC : [xsysace_1.h](#)
- XSA_FAT_12_PART_REC : [xsysace_1.h](#)
- XSA_FAT_16_BOOT_REC : [xsysace_1.h](#)
- XSA_FAT_16_CALC : [xsysace_1.h](#)
- XSA_FAT_16_PART_REC : [xsysace_1.h](#)
- XSA_FAT_VALID_BOOT_REC : [xsysace_1.h](#)
- XSA_FAT_VALID_PART_REC : [xsysace_1.h](#)
- XSA_FSR_OFFSET : [xsysace_1.h](#)
- XSA_MLR_LBA_MASK : [xsysace_1.h](#)
- XSA_MLR_OFFSET : [xsysace_1.h](#)
- XSA_SCCR_ABORT_MASK : [xsysace_1.h](#)
- XSA_SCCR_CMD_MASK : [xsysace_1.h](#)
- XSA_SCCR_COUNT_MASK : [xsysace_1.h](#)
- XSA_SCCR_IDENTIFY_MASK : [xsysace_1.h](#)
- XSA_SCCR_OFFSET : [xsysace_1.h](#)
- XSA_SCCR_READDATA_MASK : [xsysace_1.h](#)
- XSA_SCCR_RESET_MASK : [xsysace_1.h](#)
- XSA_SCCR_WRITEDATA_MASK : [xsysace_1.h](#)
- XSA_SR_CFBSY_MASK : [xsysace_1.h](#)
- XSA_SR_CFCERROR_MASK : [xsysace_1.h](#)
- XSA_SR_CFCORR_MASK : [xsysace_1.h](#)
- XSA_SR_CFDETECT_MASK : [xsysace_1.h](#)
- XSA_SR_CFDRQ_MASK : [xsysace_1.h](#)
- XSA_SR_CFDSC_MASK : [xsysace_1.h](#)
- XSA_SR_CFDWF_MASK : [xsysace_1.h](#)
- XSA_SR_CFERR_MASK : [xsysace_1.h](#)

- XSA_SR_CFGADDR_MASK : [xsysace_1.h](#)
- XSA_SR_CFGDONE_MASK : [xsysace_1.h](#)
- XSA_SR_CFGERROR_MASK : [xsysace_1.h](#)
- XSA_SR_CFGLOCK_MASK : [xsysace_1.h](#)
- XSA_SR_CFGMODE_MASK : [xsysace_1.h](#)
- XSA_SR_CFRDY_MASK : [xsysace_1.h](#)
- XSA_SR_DATABUFMODE_MASK : [xsysace_1.h](#)
- XSA_SR_DATABUFRDY_MASK : [xsysace_1.h](#)
- XSA_SR_MPULOCK_MASK : [xsysace_1.h](#)
- XSA_SR_OFFSET : [xsysace_1.h](#)
- XSA_SR_RDYFORCMD_MASK : [xsysace_1.h](#)
- XSA_VR_OFFSET : [xsysace_1.h](#)
- XSP_CLK_ACTIVE_LOW_OPTION : [xspi.h](#)
- XSP_CLK_PHASE_1_OPTION : [xspi.h](#)
- XSP_LOOPBACK_OPTION : [xspi.h](#)
- XSP_MANUAL_SSELECT_OPTION : [xspi.h](#)
- XSP_MASTER_OPTION : [xspi.h](#)
- XSpi_Abort() : [xspi_i.h](#), [xspi.c](#)
- XSpi_ClearStats() : [xspi_stats.c](#), [xspi.h](#)
- XSpi_ConfigTable : [xspi_i.h](#), [xspi_g.c](#)
- XSpi_GetOptions() : [xspi_options.c](#), [xspi.h](#)
- XSpi_GetSlaveSelect() : [xspi.h](#), [xspi.c](#)
- XSpi_GetStats() : [xspi_stats.c](#), [xspi.h](#)
- XSpi_Initialize() : [xspi.h](#), [xspi.c](#)
- XSpi_IRQHandler() : [xspi.h](#), [xspi.c](#)
- XSpi_LookupConfig() : [xspi.h](#), [xspi.c](#)
- XSpi_mDisable : [xspi_1.h](#)
- XSpi_mEnable : [xspi_1.h](#)
- XSpi_mGetControlReg : [xspi_1.h](#)
- XSpi_mGetSlaveSelectReg : [xspi_1.h](#)
- XSpi_mGetStatusReg : [xspi_1.h](#)
- XSpi_mRecvByte : [xspi_1.h](#)
- XSpi_mSendByte : [xspi_1.h](#)
- XSpi_mSetControlReg : [xspi_1.h](#)
- XSpi_mSetSlaveSelectReg : [xspi_1.h](#)
- XSpi_Reset() : [xspi.h](#), [xspi.c](#)
- XSpi_SelfTest() : [xspi_selftest.c](#), [xspi.h](#)
- XSpi_SetOptions() : [xspi_options.c](#), [xspi.h](#)
- XSpi_SetSlaveSelect() : [xspi.h](#), [xspi.c](#)

- XSpi_SetStatusHandler() : **xsapi.h, xsapi.c**
- XSpi_Start() : **xsapi.h, xsapi.c**
- XSpi_StatusHandler : **xsapi.h**
- XSpi_Stop() : **xsapi.h, xsapi.c**
- XSpi_Transfer() : **xsapi.h, xsapi.c**
- XStatus : **xstatus.h**
- XSysAce_AbortCF() : **xsysace_compactflash.c, xsysace.h**
- XSysAce_ConfigTable : **xsysace_g.c**
- XSysAce_DisableInterrupt() : **xsysace_intr.c, xsysace.h**
- XSysAce_EnableInterrupt() : **xsysace_intr.c, xsysace.h**
- XSysAce_EventHandler : **xsysace.h**
- XSysAce_GetCfgSector() : **xsysace_jtagcfg.c, xsysace.h**
- XSysAce_GetErrors() : **xsysace.h, xsysace.c**
- XSysAce_GetFatStatus() : **xsysace_compactflash.c, xsysace.h**
- XSysAce_GetVersion() : **xsysace_selftest.c, xsysace.h**
- XSysAce_IdentifyCF() : **xsysace_compactflash.c, xsysace.h**
- XSysAce_Initialize() : **xsysace.h, xsysace.c**
- XSysAce_InterruptHandler() : **xsysace_intr.c, xsysace.h**
- XSysAce_IsCfgDone() : **xsysace_jtagcfg.c, xsysace.h**
- XSysAce_IsCFReady() : **xsysace_compactflash.c, xsysace.h**
- XSysAce_Lock() : **xsysace.h, xsysace.c**
- XSysAce_LookupConfig() : **xsysace.h, xsysace.c**
- XSysAce_mAndControlReg : **xsysace_l.h**
- XSysAce_mDisableIntr : **xsysace_l.h**
- XSysAce_mEnableIntr : **xsysace_l.h**
- XSysAce_mGetControlReg : **xsysace_l.h**
- XSysAce_mGetErrorReg : **xsysace_l.h**
- XSysAce_mGetStatusReg : **xsysace_l.h**
- XSysAce_mIsCfgDone : **xsysace_l.h**
- XSysAce_mIsIntrEnabled : **xsysace_l.h**
- XSysAce_mIsMpuLocked : **xsysace_l.h**
- XSysAce_mIsReadyForCmd : **xsysace_l.h**
- XSysAce_mOrControlReg : **xsysace_l.h**
- XSysAce_mSetCfgAddr : **xsysace_l.h**
- XSysAce_mSetControlReg : **xsysace_l.h**
- XSysAce_mWaitForLock : **xsysace_l.h**
- XSysAce_ProgramChain() : **xsysace_jtagcfg.c, xsysace.h**
- XSysAce_ReadDataBuffer() : **xsysace_l.h, xsysace_l.c**
- XSysAce_ReadSector() : **xsysace_l.h, xsysace_l.c**

- XSysAce_RegRead16() : [xsysace_1.h](#), [xsysace_1.c](#)
- XSysAce_RegRead32() : [xsysace_1.h](#), [xsysace_1.c](#)
- XSysAce_RegWrite16() : [xsysace_1.h](#), [xsysace_1.c](#)
- XSysAce_RegWrite32() : [xsysace_1.h](#), [xsysace_1.c](#)
- XSysAce_ResetCF() : [xsysace_compactflash.c](#), [xsysace.h](#)
- XSysAce_ResetCfg() : [xsysace_jtagcfg.c](#), [xsysace.h](#)
- XSysAce_SectorRead() : [xsysace_compactflash.c](#), [xsysace.h](#)
- XSysAce_SectorWrite() : [xsysace_compactflash.c](#), [xsysace.h](#)
- XSysAce_SelfTest() : [xsysace_selftest.c](#), [xsysace.h](#)
- XSysAce_SetCfgAddr() : [xsysace_jtagcfg.c](#), [xsysace.h](#)
- XSysAce_SetEventHandler() : [xsysace_intr.c](#), [xsysace.h](#)
- XSysAce_SetStartMode() : [xsysace_jtagcfg.c](#), [xsysace.h](#)
- XSysAce_Unlock() : [xsysace.h](#), [xsysace.c](#)
- XSysAce_WriteDataBuffer() : [xsysace_1.h](#), [xsysace_1.c](#)
- XSysAce_WriteSector() : [xsysace_1.h](#), [xsysace_1.c](#)
- XTC_AUTO_RELOAD_OPTION : [xtmrctr.h](#)
- XTC_CAPTURE_MODE_OPTION : [xtmrctr.h](#)
- XTC_CSR_AUTO_RELOAD_MASK : [xtmrctr_1.h](#)
- XTC_CSR_CAPTURE_MODE_MASK : [xtmrctr_1.h](#)
- XTC_CSR_DOWN_COUNT_MASK : [xtmrctr_1.h](#)
- XTC_CSR_ENABLE_ALL_MASK : [xtmrctr_1.h](#)
- XTC_CSR_ENABLE_INT_MASK : [xtmrctr_1.h](#)
- XTC_CSR_ENABLE_PWM_MASK : [xtmrctr_1.h](#)
- XTC_CSR_ENABLE_TMR_MASK : [xtmrctr_1.h](#)
- XTC_CSR_EXT_CAPTURE_MASK : [xtmrctr_1.h](#)
- XTC_CSR_EXT_GENERATE_MASK : [xtmrctr_1.h](#)
- XTC_CSR_INT_OCCURED_MASK : [xtmrctr_1.h](#)
- XTC_CSR_LOAD_MASK : [xtmrctr_1.h](#)
- XTC_DEVICE_TIMER_COUNT : [xtmrctr_1.h](#)
- XTC_DOWN_COUNT_OPTION : [xtmrctr.h](#)
- XTC_ENABLE_ALL_OPTION : [xtmrctr.h](#)
- XTC_EXT_COMPARE_OPTION : [xtmrctr.h](#)
- XTC_INT_MODE_OPTION : [xtmrctr.h](#)
- XTC_TCR_OFFSET : [xtmrctr_1.h](#)
- XTC_TCSR_OFFSET : [xtmrctr_1.h](#)
- XTC_TLR_OFFSET : [xtmrctr_1.h](#)
- XTimerCtr_mReadReg : [xtmrctr_1.h](#)
- XTmrCtr_ClearStats() : [xtmrctr_stats.c](#), [xtmrctr.h](#)
- XTmrCtr_ConfigTable : [xtmrctr_i.h](#), [xtmrctr_g.c](#)

- XTmrCtr_GetCaptureValue() : [xmrctr.h](#), [xmrctr.c](#)
- XTmrCtr_GetOptions() : [xmrctr_options.c](#), [xmrctr.h](#)
- XTmrCtr_GetStats() : [xmrctr_stats.c](#), [xmrctr.h](#)
- XTmrCtr_GetValue() : [xmrctr.h](#), [xmrctr.c](#)
- XTmrCtr_Handler : [xmrctr.h](#)
- XTmrCtr_Initialize() : [xmrctr.h](#), [xmrctr.c](#)
- XTmrCtr_InterruptHandler() : [xmrctr_intr.c](#), [xmrctr.h](#)
- XTmrCtr_IsExpired() : [xmrctr.h](#), [xmrctr.c](#)
- XTmrCtr_mDisable : [xmrctr_l.h](#)
- XTmrCtr_mDisableIntr : [xmrctr_l.h](#)
- XTmrCtr_mEnable : [xmrctr_l.h](#)
- XTmrCtr_mEnableIntr : [xmrctr_l.h](#)
- XTmrCtr_mGetControlStatusReg : [xmrctr_l.h](#)
- XTmrCtr_mGetLoadReg : [xmrctr_l.h](#)
- XTmrCtr_mGetTimerCounterReg : [xmrctr_l.h](#)
- XTmrCtr_mHasEventOccurred : [xmrctr_l.h](#)
- XTmrCtr_mLoadTimerCounterReg : [xmrctr_l.h](#)
- XTmrCtr_mSetControlStatusReg : [xmrctr_l.h](#)
- XTmrCtr_mSetLoadReg : [xmrctr_l.h](#)
- XTmrCtr_mWriteReg : [xmrctr_l.h](#)
- XTmrCtr_Reset() : [xmrctr.h](#), [xmrctr.c](#)
- XTmrCtr_SelfTest() : [xmrctr_selftest.c](#), [xmrctr.h](#)
- XTmrCtr_SetHandler() : [xmrctr_intr.c](#), [xmrctr.h](#)
- XTmrCtr_SetOptions() : [xmrctr_options.c](#), [xmrctr.h](#)
- XTmrCtr_SetResetValue() : [xmrctr.h](#), [xmrctr.c](#)
- XTmrCtr_Start() : [xmrctr.h](#), [xmrctr.c](#)
- XTmrCtr_Stop() : [xmrctr.h](#), [xmrctr.c](#)
- XTRUE : [xbasic_types.h](#)
- XUartLite_ClearStats() : [xuartlite_stats.c](#), [xuartlite.h](#)
- XUartLite_ConfigTable : [xuartlite_i.h](#), [xuartlite_g.c](#)
- XUartLite_DisableInterrupt() : [xuartlite_intr.c](#), [xuartlite.h](#)
- XUartLite_EnableInterrupt() : [xuartlite_intr.c](#), [xuartlite.h](#)
- XUartLite_GetStats() : [xuartlite_stats.c](#), [xuartlite.h](#)
- XUartLite_Handler : [xuartlite.h](#)
- XUartLite_Initialize() : [xuartlite.h](#), [xuartlite.c](#)
- XUartLite_InterruptHandler() : [xuartlite_intr.c](#), [xuartlite.h](#)
- XUartLite_IsSending() : [xuartlite.h](#), [xuartlite.c](#)
- XUartLite_mDisableIntr : [xuartlite_l.h](#)
- XUartLite_mEnableIntr : [xuartlite_l.h](#)

- XUartLite_mGetStatusReg : [xuartlite_1.h](#)
- XUartLite_mIsIntrEnabled : [xuartlite_1.h](#)
- XUartLite_mIsReceiveEmpty : [xuartlite_1.h](#)
- XUartLite_mIsTransmitFull : [xuartlite_1.h](#)
- XUartLite_mSetControlReg : [xuartlite_1.h](#)
- XUartLite_ReceiveBuffer() : [xuartlite_i.h](#), [xuartlite.c](#)
- XUartLite_Recv() : [xuartlite.h](#), [xuartlite.c](#)
- XUartLite_RecvByte() : [xuartlite_1.h](#), [xuartlite_1.c](#)
- XUartLite_ResetFifos() : [xuartlite.h](#), [xuartlite.c](#)
- XUartLite_SelfTest() : [xuartlite_selftest.c](#), [xuartlite.h](#)
- XUartLite_Send() : [xuartlite.h](#), [xuartlite.c](#)
- XUartLite_SendBuffer() : [xuartlite_i.h](#), [xuartlite.c](#)
- XUartLite_SendByte() : [xuartlite_1.h](#), [xuartlite_1.c](#)
- XUartLite_SetRecvHandler() : [xuartlite_intr.c](#), [xuartlite.h](#)
- XUartLite_SetSendHandler() : [xuartlite_intr.c](#), [xuartlite.h](#)
- XUartNs550_ClearStats() : [xuartns550_stats.c](#), [xuartns550.h](#)
- XUartNs550_ConfigTable : [xuartns550_i.h](#), [xuartns550_g.c](#)
- XUartNs550_GetDataFormat() : [xuartns550_format.c](#), [xuartns550.h](#)
- XUartNs550_GetFifoThreshold() : [xuartns550_options.c](#), [xuartns550.h](#)
- XUartNs550_GetLastErrors() : [xuartns550_options.c](#), [xuartns550.h](#)
- XUartNs550_GetModemStatus() : [xuartns550_options.c](#), [xuartns550.h](#)
- XUartNs550_GetOptions() : [xuartns550_options.c](#), [xuartns550.h](#)
- XUartNs550_GetStats() : [xuartns550_stats.c](#), [xuartns550.h](#)
- XUartNs550_Handler : [xuartns550.h](#)
- XUartNs550_Initialize() : [xuartns550.h](#), [xuartns550.c](#)
- XUartNs550_InterruptHandler() : [xuartns550_intr.c](#), [xuartns550.h](#)
- XUartNs550_IsSending() : [xuartns550_options.c](#), [xuartns550.h](#)
- XUartNs550_LookupConfig() : [xuartns550.h](#), [xuartns550.c](#)
- XUartNs550_mDisableIntr : [xuartns550_1.h](#)
- XUartNs550_mEnableIntr : [xuartns550_1.h](#)
- XUartNs550_mGetLineControlReg : [xuartns550_1.h](#)
- XUartNs550_mGetLineStatusReg : [xuartns550_1.h](#)
- XUartNs550_mIsReceiveData : [xuartns550_1.h](#)
- XUartNs550_mIsTransmitEmpty : [xuartns550_1.h](#)
- XUartNs550_mReadReg : [xuartns550_1.h](#)
- XUartNs550_mSetLineControlReg : [xuartns550_1.h](#)
- XUartNs550_mWriteReg : [xuartns550_1.h](#)
- XUartNs550_ReceiveBuffer() : [xuartns550_i.h](#), [xuartns550.c](#)
- XUartNs550_Recv() : [xuartns550.h](#), [xuartns550.c](#)

- XUartNs550_RecvByte() : [xuartns550_1.h](#), [xuartns550_1.c](#)
- XUartNs550_SelfTest() : [xuartns550_selftest.c](#), [xuartns550.h](#)
- XUartNs550_Send() : [xuartns550.h](#), [xuartns550.c](#)
- XUartNs550_SendBuffer() : [xuartns550_i.h](#), [xuartns550.c](#)
- XUartNs550_SendByte() : [xuartns550_1.h](#), [xuartns550_1.c](#)
- XUartNs550_SetDataFormat() : [xuartns550_format.c](#), [xuartns550.h](#)
- XUartNs550_SetFifoThreshold() : [xuartns550_options.c](#), [xuartns550.h](#)
- XUartNs550_SetHandler() : [xuartns550_intr.c](#), [xuartns550.h](#)
- XUartNs550_SetOptions() : [xuartns550_options.c](#), [xuartns550.h](#)
- Xuint16 : [xbasic_types.h](#)
- Xuint32 : [xbasic_types.h](#)
- XUINT64_LSW : [xbasic_types.h](#)
- XUINT64_MSW : [xbasic_types.h](#)
- Xuint8 : [xbasic_types.h](#)
- XUN_ERROR_BREAK_MASK : [xuartns550.h](#)
- XUN_ERROR_FRAMING_MASK : [xuartns550.h](#)
- XUN_ERROR_NONE : [xuartns550.h](#)
- XUN_ERROR_OVERRUN_MASK : [xuartns550.h](#)
- XUN_ERROR_PARITY_MASK : [xuartns550.h](#)
- XUN_EVENT_MODEM : [xuartns550.h](#)
- XUN_EVENT_RECV_DATA : [xuartns550.h](#)
- XUN_EVENT_RECV_ERROR : [xuartns550.h](#)
- XUN_EVENT_RECV_TIMEOUT : [xuartns550.h](#)
- XUN_EVENT_SENT_DATA : [xuartns550.h](#)
- XUN_FORMAT_1_STOP_BIT : [xuartns550.h](#)
- XUN_FORMAT_2_STOP_BIT : [xuartns550.h](#)
- XUN_FORMAT_5_BITS : [xuartns550.h](#)
- XUN_FORMAT_6_BITS : [xuartns550.h](#)
- XUN_FORMAT_7_BITS : [xuartns550.h](#)
- XUN_FORMAT_8_BITS : [xuartns550.h](#)
- XUN_FORMAT_EVEN_PARITY : [xuartns550.h](#)
- XUN_FORMAT_NO_PARITY : [xuartns550.h](#)
- XUN_FORMAT_ODD_PARITY : [xuartns550.h](#)
- XUN_MODEM_CTS_DELTA_MASK : [xuartns550.h](#)
- XUN_MODEM_CTS_MASK : [xuartns550.h](#)
- XUN_MODEM_DCD_DELTA_MASK : [xuartns550.h](#)
- XUN_MODEM_DCD_MASK : [xuartns550.h](#)
- XUN_MODEM_DSR_DELTA_MASK : [xuartns550.h](#)
- XUN_MODEM_DSR_MASK : [xuartns550.h](#)

- XUN_MODEM_RING_STOP_MASK : [xuartns550.h](#)
 - XUN_MODEM_RINGING_MASK : [xuartns550.h](#)
 - XUN_OPTION_ASSERT_DTR : [xuartns550.h](#)
 - XUN_OPTION_ASSERT_OUT1 : [xuartns550.h](#)
 - XUN_OPTION_ASSERT_OUT2 : [xuartns550.h](#)
 - XUN_OPTION_ASSERT_RTS : [xuartns550.h](#)
 - XUN_OPTION_DATA_INTR : [xuartns550.h](#)
 - XUN_OPTION_FIFOS_ENABLE : [xuartns550.h](#)
 - XUN_OPTION_LOOPBACK : [xuartns550.h](#)
 - XUN_OPTION_MODEM_INTR : [xuartns550.h](#)
 - XUN_OPTION_RESET_RX_FIFO : [xuartns550.h](#)
 - XUN_OPTION_RESET_TX_FIFO : [xuartns550.h](#)
 - XUN_OPTION_SET_BREAK : [xuartns550.h](#)
 - XUT_ALLMEMTESTS : [xutil.h](#)
 - XUT_FIXEDPATTERN : [xutil.h](#)
 - XUT_INCREMENT : [xutil.h](#)
 - XUT_INVERSEADDR : [xutil.h](#)
 - XUT_MAXTEST : [xutil.h](#)
 - XUT_WALKONES : [xutil.h](#)
 - XUT_WALKZEROS : [xutil.h](#)
 - XUtil_MemoryTest16() : [xutil_memtest.c](#), [xutil.h](#)
 - XUtil_MemoryTest32() : [xutil_memtest.c](#), [xutil.h](#)
 - XUtil_MemoryTest8() : [xutil_memtest.c](#), [xutil.h](#)
 - XWaitInAssert : [xbasic_types.c](#)
 - XWdtTb_ConfigTable : [xwdttb_i.h](#), [xwdttb_g.c](#)
 - XWdtTb_GetTbValue() : [xwdttb.h](#), [xwdttb.c](#)
 - XWdtTb_Initialize() : [xwdttb.h](#), [xwdttb.c](#)
 - XWdtTb_IsWdtExpired() : [xwdttb.h](#), [xwdttb.c](#)
 - XWdtTb_mDisableWdt : [xwdttb_l.h](#)
 - XWdtTb_mEnableWdt : [xwdttb_l.h](#)
 - XWdtTb_mGetTimebaseReg : [xwdttb_l.h](#)
 - XWdtTb_mHasExpired : [xwdttb_l.h](#)
 - XWdtTb_mHasReset : [xwdttb_l.h](#)
 - XWdtTb_mRestartWdt : [xwdttb_l.h](#)
 - XWdtTb_RestartWdt() : [xwdttb.h](#), [xwdttb.c](#)
 - XWdtTb_SelfTest() : [xwdttb_selftest.c](#), [xwdttb.h](#)
 - XWdtTb_Start() : [xwdttb.h](#), [xwdttb.c](#)
 - XWdtTb_Stop() : [xwdttb.h](#), [xwdttb.c](#)
-

hdlc/v2_01_a/src/xhdlc_selftest.c File Reference

Detailed Description

Self-test and diagnostic functions of the **XHdlc** driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
2.00a	ecm	09/16/03	First release
2.01a	ecm	02/16/04	Add multichannel scatter-gather DMA support

```
#include "xbasic_types.h"  
#include "xhdlc_i.h"  
#include "xhdlc_l.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"  
#include "xchannel_fifo_v1_00_a.h"
```

Defines

```
#define LOOPBACK_WORD_COUNT
```

Functions

XStatus XHdlc_SelfTest (**XHdlc** *InstancePtr)

Define Documentation

#define

LOOPBACK_WORD_COUNT

Performs a loopback test on the HDLC device by sending and receiving a frame using loopback mode on physical channel 0, sub channel 0. This channel pair is required for any and all instances of the HDLC hardware.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

XST_SUCCESS	Loopback was successful
XST_LOOPBACK_ERROR	Loopback was unsuccessful

Note:

None.

Function Documentation

XStatus (**XHdlc ***)
XHdlc_SelfTest *InstancePtr*

Performs a self-test on the HDLC device. The test includes:

- Run self-test on the FIFOs, and IPIF components
- Reset the HDLC device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode and uses physical channel 0, sub channel 0

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run, and starting it when ready to send and receive frames.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

XST_SUCCESS	Self-test was successful
XST_REGISTER_ERROR	HDLC failed register reset test

XST_LOOPBACK_ERROR

Internal loopback failed

Note:

_Initialize must be run prior to running this test.

Generated on 24 Jun 2004 for Xilinx Device Drivers

XHdlc Struct Reference

```
#include <xhdlc.h>
```

Detailed Description

The XHdlc driver instance data. The user is required to allocate a variable of this type for every HDLC device in the system. A pointer to a variable of this type is then passed to the driver API functions.

Data Fields

Xuint32 BaseAddress

Xuint32 IsStarted

Xuint32 IsReady

XHdlc_Config * ConfigPtr

Xuint32 TotalNumberOfChannels

XDmaMulti DmaChannelStructure

Xuint8 PhysChanSubCount [XHD_MAX_NUM_PHY_CHANNELS]

Xuint32 * ChannelDataAccessPtrs

XHdlc_Channel * ChannelDataPtr

Field Documentation

Xuint32 XHdlc::

BaseAddress

Base address for device
(IPIF)

Xuint32* XHdlc::

ChannelDataAccessPtrs

This array is used to keep track of the pointers to the allocated sub channel specific data blocks in the memory pointed to by ChannelDataPtr

XHdlc_Channel* XHdlc::

ChannelDataPtr

This pointer is used as the base address of the block of sub channel specific data blocks composed of **XHdlc_Channel** structures

XHdlc_Config* XHdlc::

ConfigPtr

A pointer to the device configuration

XDmaMulti XHdlc::

DmaChannelStructure

This structure is for the multichannel DMA instance data

Xuint32 XHdlc::

IsReady

Device is initialized and ready

Xuint32 XHdlc::

IsStarted

Device is currently started

Xuint8 XHdlc::PhysChanSubCount

[XHD_MAX_NUM_PHY_CHANNELS]

This array is used to keep track of the number of configured sub channels for each physical channel

Xuint32 XHdlc::

TotalNumberOfChannels

total number of sub
channels

The documentation for this struct was generated from the following file:

- [hdlc/v2_01_a/src/xhdlc.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

common/v1_00_a/src/xbasic_types.h File Reference

Detailed Description

This file contains basic types for Xilinx software IP. These types do not follow the standard naming convention with respect to using the component name in front of each name because they are considered to be primitives.

Note:

This file contains items which are architecture dependent.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	12/14/01	First release
	rmm	05/09/03	Added "xassert always" macros to rid ourselves of diab compiler warnings
1.00a	rpm	11/07/03	Added XNullHandler function as a stub interrupt handler

Data Structures

struct **Xuint64**

Primitive types

These primitive types are created for transportability. They are dependent upon the target architecture.

```
typedef unsigned char Xuint8
    typedef char Xint8
typedef unsigned short Xuint16
    typedef short Xint16
typedef unsigned long Xuint32
    typedef long Xint32
    typedef float Xfloat32
    typedef double Xfloat64
typedef unsigned long Xboolean
```

Defines

```
#define XTRUE
#define XFALSE
#define XNULL
#define XUINT64_MSW(x)
#define XUINT64_LSW(x)
#define XASSERT_VOID(expression)
#define XASSERT_NONVOID(expression)
#define XASSERT_VOID_ALWAYS()
#define XASSERT_NONVOID_ALWAYS()
```

Typedefs

```
typedef void(* XInterruptHandler )(void *InstancePtr)
typedef void(* XAssertCallback )(char *FilenamePtr, int LineNumber)
```

Functions

```
void XAssert (char *, int)
void XAssertSetCallback (XAssertCallback Routine)
void XNullHandler (void *NullParameter)
```

Variables

```
unsigned int XAssertStatus
```

Define Documentation

#define XASSERT_NONVOID(expression)

This assert macro is to be used for functions that do return a value. This in conjunction with the XWaitInAssert boolean can be used to accomodate tests so that asserts which fail allow execution to continue.

Parameters:

expression is the expression to evaluate. If it evaluates to false, the assert occurs.

Returns:

Returns 0 unless the XWaitInAssert variable is true, in which case no return is made and an infinite loop is entered.

Note:

None.

#define XASSERT_NONVOID_ALWAYS()

Always assert. This assert macro is to be used for functions that do return a value. Use for instances where an assert should always occur.

Returns:

Returns void unless the XWaitInAssert variable is true, in which case no return is made and an infinite loop is entered.

Note:

None.

#define XASSERT_VOID(expression)

This assert macro is to be used for functions that do not return anything (void). This in conjunction with the XWaitInAssert boolean can be used to accomodate tests so that asserts which fail allow execution to continue.

Parameters:

expression is the expression to evaluate. If it evaluates to false, the assert occurs.

Returns:

Returns void unless the XWaitInAssert variable is true, in which case no return is made and an infinite loop is entered.

Note:

None.

#define XASSERT_VOID_ALWAYS()

Always assert. This assert macro is to be used for functions that do not return anything (void). Use for instances where an assert should always occur.

Returns:

Returns void unless the XWaitInAssert variable is true, in which case no return is made and an infinite loop is entered.

Note:

None.

**#define
XFALSE**

Xboolean
false

**#define
XNULL**

Null

**#define
XTRUE**

Xboolean
true

```
#define XUINT64_LSW( x )
```

Return the least significant half of the 64 bit data type.

Parameters:

x is the 64 bit word.

Returns:

The lower 32 bits of the 64 bit word.

Note:

None.

```
#define XUINT64_MSW( x )
```

Return the most significant half of the 64 bit data type.

Parameters:

x is the 64 bit word.

Returns:

The upper 32 bits of the 64 bit word.

Note:

None.

Typedef Documentation

```
typedef void(* XAssertCallback)(char* FilenamePtr, int  
LineNumber)
```

This data type defines a callback to be invoked when an assert occurs. The callback is invoked only when asserts are enabled

**typedef unsigned long
Xboolean**

boolean (XTRUE or
XFALSE)

**typedef float
Xfloat32**

32-bit floating
point

**typedef double
Xfloat64**

64-bit double precision floating
point

**typedef short
Xint16**

signed 16-
bit

**typedef long
Xint32**

signed 32-
bit

**typedef char
Xint8**

signed 8-
bit

**typedef void(* XInterruptHandler)(void
*InstancePtr)**

This data type defines an interrupt handler for a device. The argument points to the instance of the component

**typedef unsigned short
Xuint16**

unsigned 16-
bit

**typedef unsigned long
Xuint32**

unsigned 32-
bit

**typedef unsigned char
Xuint8**

unsigned 8-
bit

Function Documentation

```
void      ( char *  
XAssert  File,  
         int  Line  
         )
```

Implements assert. Currently, it calls a user-defined callback function if one has been set. Then, it potentially enters an infinite loop depending on the value of the XWaitInAssert variable.

Parameters:

File is the name of the filename of the
source

Line is the linenumber within File

Returns:

None.

Note:

None.

```
void      ( XAssertCallback  
XAssertSetCallback Routine  
         )
```

Sets up a callback function to be invoked when an assert occurs. If there was already a callback installed, then it is replaced.

Parameters:

Routine is the callback to be invoked when an assert is taken

Returns:

None.

Note:

This function has no effect if NDEBUG is set

```
void          ( void *          )  
XNullHandler      NullParameter
```

Null handler function. This follows the XInterruptHandler signature for interrupt handlers. It can be used to assign a null handler (a stub) to an interrupt controller vector table.

Parameters:

NullParameter is an arbitrary void pointer and not used.

Returns:

None.

Note:

None.

Variable Documentation

```
unsigned int          ( )  
XAssertStatus
```

This variable allows testing to be done easier with asserts. An assert sets this variable such that a driver can evaluate this variable to determine if an assert occurred.

XHdlc_Config Struct Reference

```
#include <xhdlc.h>
```

Detailed Description

This typedef contains configuration information for a device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xuint8 IpIfDmaConfig](#)

Field Documentation

[Xuint32 XHdlc_Config::](#)

BaseAddress

Device base
address

[Xuint16 XHdlc_Config::](#)

DeviceId

Unique ID of
device

[Xuint8 XHdlc_Config::](#)

IpIfDmaConfig

IPIF/DMA hardware configuration

The documentation for this struct was generated from the following file:

- `hdlc/v2_01_a/src/xhdlc.h`
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

hdlc/v2_01_a/src/xhdlc.h File Reference

Detailed Description

The Xilinx HDLC driver component which supports the Xilinx HDLC device.

Driver Description

The device driver enables higher layer software (e.g., an application) to communicate to HDLC devices. The driver handles transmission and reception of HDLC frames, as well as configuration of the devices. A single device driver can support multiple multi-channel HDLC devices.

The driver is designed for a zero-copy buffer scheme. That is, the driver will not copy buffers. This avoids potential throughput bottlenecks within the driver.

Since the driver is a simple pass-through mechanism between an application and the HDLC devices, no assembly or disassembly of HDLC frames is done at the driver-level. This assumes that the application passes a correctly formatted HDLC frame to the driver for transmission, and that the driver does not validate the contents of an incoming frame.

The driver supports FIFO interrupt mode without DMA and interrupt mode with scatter-gather DMA. The default mode of operation is FIFO interrupt mode.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are stored in the [xhdlc_g.c](#) file. A table is defined where each entry contains configuration information for an HDLC device. This information includes such things as the base address of the memory-mapped device, the number of channels, and depth of FIFO's.

Device Driver Instance Configuration

The driver configuration requires both allocating an instance of the driver and a block of memory for the management of the individual channels. The size of this block of data is dependent on the number of channels configured in the hardware. The pointer to this block of memory is provided as an argument to the `_Initialize` function. The `XHdlc_mSizeNeeded`(NumChannels) macro can be used to determine the number of bytes needed for this block of memory. This block of memory is 32-bit aligned to prevent alignment errors in various processors.

After `_Initialize` is called, `_Start` is called to enable all of the Physical Channels that are in the hardware. After initializing the Interrupt handling such as `SetHandler` and `SetUserData` if required, and enabling the interrupt, `_EnableSubChannel` must be called for each channel that is expected to send and/or receive data. When processing is complete, disable the sub channels that were enabled and then call `_Stop` to disable the Physical channels and disable the device.

HDLC Frame Description

An HDLC frame contains a number of fields as illustrated below. The size of several fields, the Address and FCS fields, are variable depending on the configuration of the device as set through the options.

<Opening Flag><Address><Control><Data><FCS><Closing Flag>

Asserts

Asserts are used within all Xilinx drivers to enforce constraints on argument values. Asserts can be turned off on a system-wide basis by defining, at compile time, the `NDEBUG` identifier. By default, asserts are turned on and it is recommended that application developers leave asserts on during development.

Note:

This driver requires HDLC hardware version 2.00a and higher. It is not compatible with earlier versions of the HDLC hardware. Use version 1.00a software driver for hardware version 1.00a.

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
2.00a	ecm	09/16/03	First release
2.01a	ecm	02/16/04	Add multichannel scatter-gather DMA support

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xchannel_fifo_v1_00_a.h"
#include "xdma_multi.h"
#include "xhdlc_1.h"
#include "xparameters.h"
```

Data Structures

```
struct XHdlc
struct XHdlc_Channel
struct XHdlc_Config
struct XHdlc_Stats
```

Configuration options

Device configuration options (see the [XHdlc_SetOptions\(\)](#) and [XHdlc_GetOptions\(\)](#) for information on how to use these options). Note that the option to remove addresses from the receive buffers is only valid when address filtering is on.

```
#define XHD_OPTION_CRC_32
#define XHD_OPTION_CRC_DISABLE
#define XHD_OPTION_RX_FILTER_ADDR
#define XHD_OPTION_RX_REMOVE_ADDR
#define XHD_OPTION_RX_BROADCAST
#define XHD_OPTION_RX_16_ADDR
```

Typedefs for callbacks

Callback functions.

```
typedef void(* XHdlc_FifoHandler )(void *CallBackRef, unsigned ByteCount, XHdlc_ChannelID ChannelID, Xuint32 Status, Xuint32 UserData)
typedef void(* XHdlc_SgHandler )(void *CallBackRef, unsigned FrameCount, XHdlc_ChannelID ChannelID, Xuint32 UserData)
typedef void(* XHdlc_ErrorHandler )(void *CallBackRef, XStatus ErrorCode, XHdlc_ChannelID ChannelID, Xuint32 UserData)
```

Defines

```
#define XHD_SGDMA_DFT_THRESHOLD
#define XHD_SGDMA_MAX_THRESHOLD
#define XHD_SGDMA_DFT_WAITBOUND
#define XHD_SGDMA_MAX_WAITBOUND
#define XHD_MIN_BUFFERS
#define XHD_DFT_BUFFERS
#define XHD_MIN_RECV_DESC
#define XHD_DFT_RECV_DESC
#define XHD_MIN_SEND_DESC
#define XHD_DFT_SEND_DESC
```

```
#define XHdlc_mSizeNeeded(NumChannels)
#define XHdlc_mChannelIDValid(InstancePtr, ChannelID)
```

Functions

```
XStatus XHdlc_Initialize (XHdlc *InstancePtr, Xuint32 *MemoryPtr, Xuint16 DeviceId)
XStatus XHdlc_Start (XHdlc *InstancePtr)
XStatus XHdlc_Stop (XHdlc *InstancePtr)
void XHdlc_EnableSubChannel (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)
void XHdlc_DisableSubChannel (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)
void XHdlc_Reset (XHdlc *InstancePtr)
void XHdlc_EventFlush (XHdlc *InstancePtr)
XHdlc_Config * XHdlc_LookupConfig (Xuint16 DeviceId)
void XHdlc_SetErrorHandler (XHdlc *InstancePtr, void *CallBackRef, XHdlc_ErrorHandler FuncPtr)
void XHdlc_SetChannelUserData (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, Xuint32 UserData)
Xuint32 XHdlc_GetChannelUserData (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)
XStatus XHdlc_SelfTest (XHdlc *InstancePtr)
XStatus XHdlc_SetOptions (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, Xuint16 Options)
Xuint16 XHdlc_GetOptions (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)
void XHdlc_SetAddress (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, Xuint16 Address)
Xuint16 XHdlc_GetAddress (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)
void XHdlc_EnableLoopback (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)
void XHdlc_DisableLoopback (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)
void XHdlc_GetStats (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, XHdlc_Stats *StatsPtr)
void XHdlc_ClearStats (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)
Xuint32 XHdlc_Send (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, Xuint32 *FramePtr, Xuint32 ByteCount, Xboolean EndOfPacket)
Xuint32 XHdlc_Recv (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, Xuint32 *FramePtr, Xuint32 ByteCount)
void XHdlc_InterruptHandlerFifo (void *InstancePtr)
void XHdlc_SetFifoRecvHandler (XHdlc *InstancePtr, void *CallBackRef, XHdlc_FifoHandler FuncPtr)
void XHdlc_SetFifoSendHandler (XHdlc *InstancePtr, void *CallBackRef, XHdlc_FifoHandler FuncPtr)
XStatus XHdlc_SgSend (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, XBufDescriptor *BdPtr)
XStatus XHdlc_SgRecv (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, XBufDescriptor *BdPtr)
XStatus XHdlc_SgGetSendFrame (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, XBufDescriptor **PtrToBdPtr, unsigned
*BdCountPtr)
XStatus XHdlc_SgGetRecvFrame (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, XBufDescriptor **PtrToBdPtr, int *BdCountPtr)
void XHdlc_InterruptHandlerSg (void *InstancePtr)
XStatus XHdlc_SetSgRecvSpace (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, Xuint32 *MemoryPtr, unsigned ByteCount)
XStatus XHdlc_SetSgSendSpace (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, Xuint32 *MemoryPtr, unsigned ByteCount)
void XHdlc_SetSgRecvHandler (XHdlc *InstancePtr, void *CallBackRef, XHdlc_SgHandler FuncPtr)
void XHdlc_SetSgSendHandler (XHdlc *InstancePtr, void *CallBackRef, XHdlc_SgHandler FuncPtr)
```

Define Documentation

```
#define
XHD_DFT_BUFFERS

default number of receive
buffers
```

```
#define
XHD_DFT_RECV_DESC

default # of recv
descriptors
```

```
#define
XHD_DFT_SEND_DESC
```

default # of send
descriptors

#define
XHD_MIN_BUFFERS

minimum number of receive
buffers

#define
XHD_MIN_RECV_DESC

minimum # of recv
descriptors

#define
XHD_MIN_SEND_DESC

minimum # of send
descriptors

#define
XHD_OPTION_CRC_32

send/receive 32 bit CRCs default is 16 bit
CRCs

#define
XHD_OPTION_CRC_DISABLE

disable sending
CRCs

#define
XHD_OPTION_RX_16_ADDR

receive 16 bit addresses, 8 bit is the
default

#define
XHD_OPTION_RX_BROADCAST

receive broadcast
addresses

#define
XHD_OPTION_RX_FILTER_ADDR

receive address
filtering

#define
XHD_OPTION_RX_REMOVE_ADDR

don't buffer receive
addresses

#define
XHD_SGDMA_DFT_THRESHOLD

Default pkt
threshold

#define
XHD_SGDMA_DFT_WAITBOUND

Default pkt wait bound
(msec)

```
#define  
XHD_SGDMA_MAX_THRESHOLD
```

Maximum pkt
theshold

```
#define  
XHD_SGDMA_MAX_WAITBOUND
```

Maximum pkt wait bound
(msec)

```
#define XHdlc_mChannelIDValid( InstancePtr,  
ChannelID )
```

Determine if the ChannelID is valid.

Parameters:

InstancePtr points to instance to be used for
operation
ChannelID is pair to be tested

Returns:

boolean XTRUE if the channelID is in the PhysChanSubCount array.

Note:

Hardware is not queried for this check. XHdlc_Initialize must be run prior to running this function.

```
#define XHdlc_mSizeNeeded( NumChannels )
```

Determine the size needed for the channel structure pointer.

Parameters:

NumChannels is the total number of SubChannels configured in the hardware

Returns:

The size of the structure in bytes. Includes array used for pointers at the beginning of the data.

Note:

(sizeof(XHdlc_Channel) + sizeof(Xuint32)) is the size required for one channel. The first number is the size of the structure needed for each channel and the second is the size of the pointer used to access the structure allocated in the first sizeof() function. If DMA is not used, define XDM_BYTES_PER_CHANNEL as zero.

Typedef Documentation

```
typedef void(* XHdlc_ErrorHandler)(void *CallbackRef, XStatus ErrorCode, XHdlc_ChannelID ChannelID, Xuint32  
UserData)
```

Callback when errors occur in interrupt or sgdma mode

Parameters:

- CallBackRef* is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.
- ErrorCode* indicates the error that occurred. This is XST_RESET_ERROR to indicate an error that requires reset
- ChannelID* is channel the data is associated with.
- UserData* is available for use by user for channel specific data.

```
typedef void(* XHdlc_FifoHandler)(void *CallBackRef, unsigned ByteCount, XHdlc_ChannelID ChannelID, Xuint32 Status, Xuint32 UserData)
```

Callback when data is sent or received with interrupt driven mode.

Parameters:

- CallBackRef* is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.
- ByteCount* is the number of bytes sent or received. On receive the ByteCount is the exact amount of data to be read. It is possible that there is more data in the FIFO than the event actually indicates due to another packet coming in after the current one to be read.
- ChannelID* is channel the data is associated with.
- Status* is is the status from the read of the FIFO's. The values passed through this variable are XHD_STAT_FIFO_*_MASK from ([xhdlc_1.h](#)) and include End of Packet and error indications.
- UserData* is available for use by user for channel specific data.

```
typedef void(* XHdlc_SgHandler)(void *CallBackRef, unsigned FrameCount, XHdlc_ChannelID ChannelID, Xuint32 UserData)
```

Callback when data is sent or received with interrupt driven mode.

Parameters:

- CallBackRef* is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.
- ByteCount* is the number of bytes sent or received. On receive the ByteCount is the exact amount of data to be read. It is possible that there is more data in the FIFO than the event actually indicates due to another packet coming in after the current one to be read.
- ChannelID* is channel the data is associated with.
- Status* is is the status from the read of the FIFO's. The values passed through this variable are XHD_STAT_FIFO_*_MASK from ([xhdlc_1.h](#)) and include End of Packet and error indications.
- UserData* is available for use by user for channel specific data.

Function Documentation

```
void XHdlc_ClearStats ( XHdlc * InstancePtr, XHdlc_ChannelID ChannelID )
```

Clear the statistics for the specified HDLC driver instance and physical channel/ sub channel pair.

Parameters:

- InstancePtr* is a pointer to the [XHdlc](#) instance to be worked on.
- ChannelID* is the intended channel

Returns:

None.

Note:

None.

```
void  
XHdlc_DisableLoopback ( XHdlc *  
                        InstancePtr,  
                        XHdlc_ChannelID ChannelID  
                        )
```

Disable the loopback for the Physical channel provided. The sub channel in the ChannelID is ignored.

This function clears the loopback functionality for the physical channel. This takes all of the associated sub channels out of loopback also.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel, the sub channel is ignored

Returns:

None.

Note:

None.

```
void  
XHdlc_DisableSubChannel ( XHdlc *  
                          InstancePtr,  
                          XHdlc_ChannelID ChannelID  
                          )
```

This function disables the sub channel associated with the provided ChannelID. This function leaves the physical channel and the device enabled.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel.

Returns:

None.

Note:

None.

```
void  
XHdlc_EnableLoopback ( XHdlc *  
                       InstancePtr,  
                       XHdlc_ChannelID ChannelID  
                       )
```

Enable the loopback for the Physical channel provided. The sub channel in the ChannelID is ignored.

This function set the loopback functionality for the physical channel. This puts all of the associated sub channels into loopback also.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel, the sub channel is ignored

Returns:

None.

Note:

None

```
void
XHdlc_EnableSubChannel ( XHdlc *
                        InstancePtr,
                        XHdlc_ChannelID ChannelID
                        )
```

Enable the HDLC sub Channel. This function enables the sub channel associated with the provided ChannelID. This allows the channel to be used for data transfer.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.
ChannelID is the intended channel.

Returns:

None

Note:

```
void
XHdlc_EventFlush ( XHdlc *
                  InstancePtr
                  )
```

Empty the HDLC Event and Data FIFO's. This empties the Status, Receive Event, and Transmit Event FIFO's. Then the Receive Channel FIFO's are emptied to prevent initial overruns and garbled data. This data is usually left over after either a power up reset or a user directed reset occurs. EventFlush must only be called after the driver has been initialized.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint16
XHdlc_GetAddress ( XHdlc *
                  InstancePtr,
                  XHdlc_ChannelID ChannelID
                  )
```

Get the receive address for this Sub Channel. The address is a 8 or 16 bit value within an HDLC frame. The Address applies only to the specified physical channel/ sub channel pair provided in the arguments.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.
ChannelID is the intended channel

Returns:

The receive address of the HDLC device.

Note:

None.

```
Xuint32  
XHdlc_GetChannelUserData ( XHdlc *  
                           InstancePtr,  
                           XHdlc_ChannelID ChannelID  
                           )
```

Gets the user data for the channel. This uses the channel specific data area provided by the application at initialization. The data itself can be anything that fits into an Xuint32 memory location.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the channel the data was stored in.

Returns:

- o UserData is available for application use.

Note:

XHdlc_Initialize must be run prior to running this function

```
Xuint16  
XHdlc_GetOptions ( XHdlc *  
                   InstancePtr,  
                   XHdlc_ChannelID ChannelID  
                   )
```

Get HDLC Sub Channel Options. A value is returned which is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel

Returns:

The value of the HDLC options. The value is a bit-mask representing all options that are currently enabled. See [xhdlc.h](#) for a description of the available options.

Note:

None.

```
void  
XHdlc_GetStats ( XHdlc *  
                 InstancePtr,  
                 XHdlc_ChannelID ChannelID,  
                 XHdlc_Stats * StatsPtr  
                 )
```

Get a copy of the statistics structure, which contains the current statistics for this instance and physical channel/ sub channel pair. The statistics are only cleared at initialization or on demand using the **XHdlc_ClearStats()** function.

The FifoErrors counts indicate that the device has been or needs to be reset. Reset of the device is the responsibility of the caller.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None.

Note:

None.

```
XStatus          ( XHdlc *  
XHdlc_Initialize      InstancePtr,  
                    Xuint32 *  
                    MemoryPtr,  
                    Xuint16 DeviceId  
                    )
```

Initialize a specific **XHdlc** instance/driver. The initialization entails:

- Initialize fields of the **XHdlc** instance structure and access arrays.
- Clear the HDLC statistics for each channel configured in this device.
- Configure the FIFO components and DMA channels if available.
- Reset the HDLC device.

The driver defaults to interrupt mode operation.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

MemoryPtr is a pointer to the block of memory for the channel specific data such as the statistics (see notes). 32-bit aligned.

DeviceId is the unique id of the device controlled by this **XHdlc** instance. Passing in a device id associates the generic **XHdlc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- 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.
- XST_NO_FEATURE if the device configuration information indicates a feature that is not supported by this driver (no IPIF or DMA).

Note:

MemoryPtr points to a block of memory for the management of the individual channels. The size of this block of data is dependent on the number of channels available in the hardware. The **XHdlc_mSizeNeeded**(NumChannels) macro can be used to determine the number of bytes needed for this block of memory. This memory is divided up into two regions, an array of pointers which allow direct access into the larger block of memory which is the channel specific data itself. This block of memory is 32-bit aligned to prevent alignment errors in various processors.

```
void              ( void *  
XHdlc_InterruptHandlerFifo      InstancePtr
```

Interrupt handler for the HDLC driver. It performs the following processing:

- Get the interrupt status from the IpIf to determine the source of the interrupt.
- Call the appropriate handler based on the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the HDLC device instance for the interrupt.

Returns:

None.

Note:

This handler reads all three FIFO's, Status, TX Event, and RX Event, if there are events in them and processes the event (s).

```
void ( void * )
XHdlc_InterruptHandlerSg InstancePtr
```

Interrupt handler for the HDLC driver. It performs the following processing:

- Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: the HDLC device, the send packet FIFO, the receive packet FIFO, the send DMA channel, or the receive DMA channel. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the HDLC device.
- Call the appropriate handler based on the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the HDLC device instance for the interrupt.

Returns:

None.

Note:

None.

```
XHdlc_Config* ( Xuint16 )
XHdlc_LookupConfig DeviceId
```

Lookup the device configuration based on the unique device ID. The table *XHdlc_ConfigTable* contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

```
Xuint32 ( XHdlc *
XHdlc_Recv InstancePtr,
XHdlc_ChannelID ChannelID,
Xuint32 * FramePtr,
Xuint32 ByteCount
)
```

Gets the received frame.

This function is called to initiate the reception of the frame, usually from within the callback. and, in the case of frames longer than the FIFO size, called repeatedly from within the callback to continue receiving the frame.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended receive channel.

FramePtr is the address of a buffer for received data, 32-bit aligned.

ByteCount is the size of the available buffer.

Returns:

The number of bytes received.

Note:

None.

```
void (XHdlc * )  
XHdlc_Reset InstancePtr
```

Reset the HDLC instance. This is a graceful reset in that the device is stopped first then it resets the FIFOs. Reset must only be called after the driver has been initialized.

The configuration after this reset is as follows:

- Disable all of the valid sub channels.
- Disable all of the valid physical channels.
- Disable the device itself.
- Device interrupts are disabled.

The upper layer software is responsible for re-configuring (if necessary) and restarting the HDLC device after the reset. Note also that driver statistics are not cleared on reset. It is up to the upper layer software to clear the statistics if needed.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus (XHdlc * )  
XHdlc_SelfTest InstancePtr
```

Performs a self-test on the HDLC device. The test includes:

- Run self-test on the FIFOs, and IPIF components
- Reset the HDLC device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode and uses physical channel 0, sub channel 0

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run, and starting it when ready to send and receive frames.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

XST_SUCCESS	Self-test was successful
XST_REGISTER_ERROR	HDLC failed register reset test
XST_LOOPBACK_ERROR	Internal loopback failed

Note:

_Initialize must be run prior to running this test.

```
Xuint32 (XHdlc * InstancePtr,  
XHdlc_ChannelID ChannelID,  
Xuint32 * FramePtr,  
Xuint32 ByteCount,  
Xboolean EndOfPacket  
)
```

Sends a HDLC frame using interrupt mode.

It is assumed that the upper layer software supplies a correctly formatted HDLC frame based upon the configuration of the HDLC device. The HDLC device must be started before calling this function.

This function is called to initiate a transmission and, in the case of frames longer than the FIFO size, called from within the callback to continue the transmission.

Parameters:

- InstancePtr* is a pointer to the **XHdlc** instance to be worked on.
- ChannelID* is the intended transmit channel.
- FramePtr* is the address of a buffer to transmit, 32-bit aligned.
- ByteCount* is the number of bytes to be sent.
- EndOfPacket* indicates that this send is the end of packet.

Returns:

The number of bytes actually sent.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

The EndOfPacket flag allows the calling function to control whether the EndOfPacket is indicated to the hardware when the last of the data is written to the FIFO. This allows for multiple calls to the `_Send` function without having the EOP inserted at the wrong time. This flag is not the final decision on whether the EOP is indicated. In the situation where the data to be sent is larger than the the space available in the FIFO, the function will not indicate an EOP to the hardware and return to the caller with the actual number of bytes sent which will be less than the number of bytes requested to be sent. At this point, the calling function can call the `_Send` function again, after updating the buffer pointer to reflect the amount of data sent, with the `EndOfPacket = XTRUE` and the function will again try to put all of the requested data into the FIFO. Looping on the `_Send` until the entire buffer is sent effectively turns the non-blocking call to `_Send` into a blocking call until the buffer is completely sent.

Warning: `EndOfPacket = XFALSE` and not getting back to the `_Send` before the FIFO is empty causes underruns which can force the entire packet to be resent from the beginning.

```
void  
XHdlc_SetAddress ( XHdlc *  
                  InstancePtr,  
                  XHdlc_ChannelID  
                  Xuint16  
                  ChannelID,  
                  Address  
                  )
```

Set the receive address for this Sub Channel. The address is a 8 or 16 bit value within an HDLC frame. The Address applies only to the specified physical channel/ sub channel pair provided in the arguments.

Parameters:

- InstancePtr* is a pointer to the **XHdlc** instance to be worked on.
- ChannelID* is the intended channel.
- Address* is the address to be set.

Returns:

None.

Note:

None.

Sets the callback function for handling received frames in FIFO mode. The upper layer software should call this function during initialization. The callback is called when a frame is received. The callback function should communicate the data to a thread such that the processing is not performed in an interrupt context.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallBackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void  
XHdlc_SetFifoSendHandler ( XHdlc * InstancePtr,  
void * CallBackRef,  
XHdlc_FifoHandler FuncPtr  
)
```

Sets the callback function for handling confirmation of transmitted frame in FIFO mode. The upper layer software should call this function during initialization. The callback is called when a frames is sent. The callback function should communicate the data to a thread such that processing is not performed in an interrupt context.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Note that a callback may occur with a Status of 0 and a ByteCount of 0. This represents a half-empty interrupt. The callback is still invoked here so the user has a chance to refill the FIFO before an underrun if not the end of packet.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallBackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus  
XHdlc_SetOptions ( XHdlc * InstancePtr,  
XHdlc_ChannelID ChannelID,  
Xuint16 Options  
)
```

Set HDLC Sub Channel options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option (i.e., you can OR the options together). A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel

Options is a bit-mask representing the HDLC options to turn on or off. See **xhdlc.h** for a description of the available options.

Returns:

- o XST_SUCCESS if the options were set successfully
- o XST_IS_STOPPED if the channel has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

```
void  
XHdlc_SetSgRecvHandler ( XHdlc * InstancePtr,  
                          void * CallbackRef,  
                          XHdlc_SgHandler FuncPtr  
                        )
```

Sets the callback function for handling received cells in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of cells, determined by the DMA scatter-gather packet threshold, are received. The number of received cells is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each received cell from the list and should attach a new buffer to each descriptor. It is important that the specified number of cells passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus  
XHdlc_SetSgRecvSpace ( XHdlc * InstancePtr,  
                       XHdlc_ChannelID ChannelID,  
                       Xuint32 * MemoryPtr,  
                       unsigned ByteCount  
                     )
```

Gives the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the HDLC driver. The memory space must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the specific channel associated with this data.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- o XST_SUCCESS if the space was initialized successfully
- o XST_NOT_SGDMA if the HDLC device is not configured for scatter-gather DMA
- o XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the XHdlc_Initialize function because the DMA channel components must be initialized before the memory space is set.

```
void XHdlc_SetSgSendHandler ( XHdlc * InstancePtr, void * CallbackRef, XHdlc_SgHandler FuncPtr )
```

Sets the callback function for handling confirmation of transmitted cells in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of cells, determined by the DMA scatter-gather packet threshold, are sent. The number of sent cells is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each sent cell from the list and should also free the buffers attached to the descriptors if necessary. It is important that the specified number of cells passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XHdlc_SetSgSendSpace ( XHdlc * InstancePtr, XHdlc_ChannelID ChannelID, Xuint32 * MemoryPtr, unsigned ByteCount )
```

Gives the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the HDLC driver. The memory space must be word-aligned.

Parameters:

- InstancePtr* is a pointer to the **XHdlc** instance to be worked on.
- ChannelID* is the specific channel associated with this data.
- MemoryPtr* is a pointer to the word-aligned memory.
- ByteCount* is the length, in bytes, of the memory space.

Returns:

- o XST_SUCCESS if the space was initialized successfully
- o XST_NOT_SGDMA if the HDLC device is not configured for scatter-gather DMA
- o XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the XHdlc_Initialize function because the DMA channel components must be initialized before the memory space is set.

```
XStatus ( XHdlc * InstancePtr,
XHdlc_SgGetRecvFrame XHdlc_ChannelID ChannelID,
XBufDescriptor ** PtrToBdPtr,
int * BdCountPtr
)
```

Gets the first buffer descriptor of the oldest frame which was received by the scatter-gather DMA channel of the HDLC device. This function is provided to be called from a callback function such that the buffer descriptors for received frames can be processed. The function should be called by the application repetitively for the number of frames indicated as an argument in the callback function.

Parameters:

- InstancePtr* is a pointer to the **XHdlc** instance to be worked on.
- ChannelID* is the specific channel associated with this data.
- PtrToBdPtr* is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the frame. This input argument is also an output.
- BdCountPtr* is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the frame. This input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the frame pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- o XST_SUCCESS if a descriptor was successfully returned to the driver.
- o XST_NOT_SGDMA if the device is not in scatter-gather DMA mode.
- o XST_DMA_SG_NO_LIST if the scatter gather list has not been created.
- o XST_DMA_SG_LIST_EMPTY if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```
XStatus ( XHdlc * InstancePtr,
XHdlc_SgGetSendFrame XHdlc_ChannelID ChannelID,
XBufDescriptor ** PtrToBdPtr,
unsigned * BdCountPtr
)
```

Gets the first buffer descriptor of the oldest frame which was sent by the scatter-gather DMA channel of the HDLC device. This function is provided to be called from a callback function such that the buffer descriptors for sent frames can be processed. The function should be called by the application repetitively for the number of frames indicated as an argument in the callback function.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the specific channel associated with this data.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the frame. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the frame. this input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the frame pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- o XST_SUCCESS if a descriptor was successfully returned to the driver.
- o XST_NOT_SGDMA if the device is not in scatter-gather DMA mode.
- o XST_DMA_SG_NO_LIST if the scatter gather list has not been created.
- o XST_DMA_SG_LIST_EMPTY if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```
XStatus ( XHdlc *  
XHdlc_SgRecv InstancePtr,  
XHdlc_ChannelID ChannelID,  
XBufDescriptor * BdPtr  
)
```

Adds this descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be word-aligned. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of frames to replace filled buffers with empty buffers. The contents of the specified buffer descriptor are copied into the scatter-gather transmit list. This function can be called when the device is started or stopped.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the specific channel associated with this data.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- o XST_SUCCESS if a descriptor was successfully returned to the driver
- o XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- o XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- o XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- o XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

Note:

None.

```
XStatus ( XHdlc *  
XHdlc_SgSend InstancePtr,  
XHdlc_ChannelID ChannelID,  
XBufDescriptor * BdPtr  
)
```

Sends a HDLC frame using scatter-gather DMA. The caller attaches the frame to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire frame may or may not be contained within one descriptor. The contents of the buffer descriptor are copied into the scatter-gather transmit list. The caller is responsible for providing mutual exclusion to guarantee that a frame is contiguous in the transmit list. The buffer attached to the descriptor must be word-aligned.

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the frame, the inserts are committed, which means the descriptors for this frame are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted HDLC frame based upon the configuration of the HDLC device. The HDLC device must be started before calling this function.

Parameters:

- InstancePtr* is a pointer to the **XHdlc** instance to be worked on.
- ChannelID* is the specific channel associated with this data.
- BdPtr* is the address of a descriptor to be inserted into the transmit ring.

Returns:

- o XST_SUCCESS if the buffer was successfully sent
- o XST_IS_STOPPED if the HDLC device has not been started yet
- o XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- o XST_DMA_SG_LIST_FULL if the descriptor list for the DMA channel is full
- o XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- o XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

XStatus (**XHdlc** *)
XHdlc_Start *InstancePtr*

Start the HDLC Channel and driver by enabling the hardware device, enabling the DMA functionality if configured for it, Enables all of the available physical channels, and enables the interrupts from the device.

This function must be called before other functions to send or receive data. It supports interrupt driven modes with FIFOs and scatter gather DMA. The user should have connected the interrupt handler of the driver to an interrupt source such as an interrupt controller or the processor interrupt prior to this function being called.

This function starts all of the configured Physical Channels and enables the interrupt outputs from the device and finally enables the device itself. The application must then enable any/all sub channels using **XHdlc_EnableSubChannel()** before data will be transmitted and/or received across that channel.

Parameters:

- InstancePtr* is a pointer to the **XHdlc** instance to be worked on.

Returns:

- o XST_SUCCESS if the device was started successfully.
- o XST_DEVICE_IS_STARTED if the device is already started.
- o XST_NO_CALLBACK if a callback function has not yet been registered using the SetxxxHandler function. This is required if in interrupt mode.
- o XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XStatus (**XHdlc** *)

XHdlc_Stop *InstancePtr*

Stop the HDLC device as follows:

- If the device is configured with DMA, stop the DMA channels (wait for acknowledgment of stop).
- Disables the available sub channels.
- Disables the available physical channels.
- Disables the device.
- Disables the interrupts from the device(the higher layer software is responsible for disabling interrupts at the interrupt controller).

If the device is configured for scatter-gather DMA, the DMA engine stops at the next buffer descriptor in its list. The remaining descriptors in the list are not removed, so anything in the list will be transmitted or received when the device is restarted. The side effect of doing this is that the last buffer descriptor processed by the DMA engine before stopping may not be the last descriptor in the HDLC frame. So when the device is restarted, a partial frame (i.e., a bad frame) may be transmitted/received. This is only a concern if a frame can span multiple buffer descriptors, which is dependent on the size of the network buffers.

To start using the device again, `_Start` must be called and any sub channels that are to be used must be enabled prior to use. Calling `_Reset` is recommended since there is the possibility that there will be garbage in the FIFO's.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully.
- XST_IS_STOPPED if the device is already stopped.

Note:

None.

hdlc/v2_01_a/src/xhdlc_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of HDLC devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
2.00a	ecm	09/16/03	First release
2.01a	ecm	02/16/04	Add multichannel scatter-gather DMA support

```
#include "xhdlc.h"
```

Variables

[XHdlc_Config](#) [XHdlc_ConfigTable](#) [XPAR_XHDLC_NUM_INSTANCES]

Variable Documentation

[XHdlc_Config](#) [XHdlc_ConfigTable](#)
[XPAR_XHDLC_NUM_INSTANCES]

This table contains configuration information for each HDLC device in the system.

common/v1_00_a/src/xstatus.h File Reference

Detailed Description

This file contains Xilinx software status codes. Status codes have their own data type called XStatus. These codes are used throughout the Xilinx device drivers.

```
#include "xbasic\_types.h"
```

Typedefs

```
typedef Xuint32 XStatus
```

Typedef Documentation

```
typedef Xuint32
```

```
XStatus
```

The status
typedef.

dma/v1_00_a/src/xdma_multi.h File Reference

Detailed Description

Description

This file contains the multichannel DMA implementation. This component supports a channelized DMA design in which each device can have its own dedicated multichannel DMA, as opposed to a centralized DMA design. A device which uses DMA typically contains at least two channels of DMA, one for sending data and the other for receiving data.

This component is designed to be used as a basic building block for designing a device driver. It provides register accesses such that all DMA processing can be maintained easier, but the device driver designer must still understand all the details of the multichannel DMA.

The multichannel DMA allows a CPU to minimize the CPU interaction required to move data between a memory and a device. The CPU requests the DMA hardware to perform a DMA operation and typically continues performing other processing until the DMA operation completes. DMA could be considered a primitive form of multiprocessing such that caching and address translation can be an issue.

Scatter Gather Operations

The multichannel DMA may support scatter gather operations. A scatter gather operation automates the DMA operation such that multiple buffers can be sent or received with minimal software interaction with the hardware. Buffer descriptors, contained in the XBufDescriptor component, are used by the scatter gather operations of the multichannel DMA to describe the buffers to be processed.

Scatter Gather List Operations

A scatter gather list may be supported by each channel of DMA. The scatter gather list allows buffer descriptors to be put into the list by a device driver which requires scatter gather. The hardware processes the buffer descriptors which are contained in the list and modifies the buffer descriptors to reflect the status of the DMA operations. The device driver is notified by interrupt that specific DMA

events occur including scatter gather events. The device driver removes the completed buffer descriptors from the scatter gather list to evaluate the status of each DMA operation.

The scatter gather list is created and buffer descriptors are inserted into the list. Buffer descriptors are never removed from the list after it's creation such that a put operation copies from a temporary buffer descriptor to a buffer descriptor in the list. Get operations don't copy from the list to a temporary, but return a pointer to the buffer descriptor in the list. A buffer descriptor in the list may be locked to prevent it from being overwritten by a put operation. This allows the device driver to get a descriptor from a scatter gather list and prevent it from being overwritten until the buffer associated with the buffer descriptor has been processed.

Typical Scatter Gather Processing

The following steps illustrate the typical processing to use the scatter gather features of a multichannel DMA.

1. Create a scatter gather list for the each channel of DMA which puts empty buffer descriptors into the list.
2. Create buffer descriptors which describe the buffers to be filled with receive data or the buffers which contain data to be sent.
3. Put buffer descriptors into the multichannel DMA scatter list such that scatter gather operations are requested.
4. Commit the buffer descriptors in the list such that they are ready to be used by the multichannel DMA hardware.
5. Start the scatter gather operations of the multichannel DMA.
6. Process any interrupts which occur as a result of the scatter gather operations or poll the each channel of DMA to determine the status.

Interrupts

Each channel of the DMA has the ability to generate an interrupt. This component does not perform processing for the interrupt as this processing is typically tightly coupled with the device which is using the multichannel DMA. It is the responsibility of the caller of DMA functions to manage the interrupt including connecting to the interrupt and enabling/disabling the interrupt.

Critical Sections

It is the responsibility of the device driver designer to use critical sections as necessary when calling functions of the multichannel DMA. This component does not protect critical sections and it does access registers using read-modify-write operations. Calls to DMA functions from a main thread and from an interrupt context could produce unpredictable behavior such that the caller must provide the appropriate protection.

Address Translation

All addresses of data structures which are passed to DMA functions must be physical (real) addresses as opposed to logical (virtual) addresses.

Caching

The memory which is passed to the function which creates the scatter gather list must not be cached such that buffer descriptors are non-cached. This is necessary because the buffer descriptors are kept in a ring buffer and not directly accessible to the caller of DMA functions.

The caller of DMA functions is responsible for ensuring that any data buffers which are passed to the multichannel DMA are cache-line aligned if necessary.

The caller of DMA functions is responsible for ensuring that any data buffers which are passed to the multichannel DMA have been flushed from the cache.

The caller of DMA functions is responsible for ensuring that the cache is invalidated prior to using any data buffers which are the result of a DMA operation.

<Memory Alignment

The addresses of data buffers which are passed to DMA functions must be 32 bit word aligned if the peripheral is on the OPB bus and must be 64-bit word aligned if the peripheral is on the PLB bus since the DMA hardware performs bus width word transfers. The alignment of the provided buffers is not checked by the driver.

Mutual Exclusion

The functions of the multichannel DMA are not thread safe such that the caller of all DMA functions is responsible for ensuring mutual exclusion for each channel of DMA. Mutual exclusion across multiple instances of the multichannel DMAs is not necessary.

Asserts

Asserts are used within all Xilinx drivers to enforce constraints on argument values. Asserts can be turned off on a system-wide basis by defining, at compile time, the NDEBUG identifier. By default, asserts are turned on and it is recommended that application developers leave asserts on during development.

Note:

Many of the provided functions which are register accessors don't provide a lot of error detection. The caller is expected to understand the impact of a function call based upon the current state of the multichannel DMA. This is done to minimize the overhead in this component.

In the following diagram, arrows are used to illustrate addresses or pointers such that this diagram is aimed for illustration purposes rather than a literal implementation.

The user memory block is passed in and used by the device driver. The user memory is partitioned into two parts with the 1st part being a table of addresses (indexed by the channel number) which contains the address of each channels registers. The 2nd part of the memory is a table of structures (indexed by the channel number) which contains the scatter gather data for each channel.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	09/16/03	First release

```
#include "xdma_multi_i.h"  
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xbuf_descriptor.h"
```

Data Structures

```
struct XDmaMulti  
struct XDmaMulti_SgData
```

Macro functions

```
#define XDmaMulti_mSizeNeeded(NumChannels)  
#define XDmaMulti_mGetGlobalControl(InstancePtr)  
#define XDmaMulti_mSetGlobalControl(InstancePtr, Control)  
#define XDmaMulti_mGetControl(InstancePtr, Channel)  
#define XDmaMulti_mSetControl(InstancePtr, Channel, Control)  
#define XDmaMulti_mSetEventEnable(InstancePtr, Channel, Enable)  
#define XDmaMulti_mGetEventEnable(InstancePtr, Channel)  
#define XDmaMulti_mGetEventStatus(InstancePtr)  
#define XDmaMulti_mGetEventChannel(Status)  
#define XDmaMulti_mGetEventParameter(Status)  
#define XDmaMulti_mGetStatus(InstancePtr, Channel)  
#define XDmaMulti_mSetPktThreshold(InstancePtr, Channel, Threshold)
```

```
#define XDmaMulti_mGetPktThreshold(InstancePtr, Channel)
#define XDmaMulti_mSetPktWaitBound(InstancePtr, Channel, WaitBound)
#define XDmaMulti_mGetPktWaitBound(InstancePtr, Channel)
```

Defines

```
#define XDM_DMACR_SOURCE_INCR_MASK
#define XDM_DMASR_BUSY_MASK
#define XDM_SWCR_EESGEND_MASK
#define XDM_GCSR_GLOBAL_ENABLE_MASK
#define XDM_GEFIFO_SGEND_MASK
```

Functions

```
XStatus XDmaMulti_Initialize (XDmaMulti *InstancePtr, Xuint32 BaseAddress, Xuint32
    *UserMemoryPtr, Xuint16 ChannelCount)
XStatus XDmaMulti_SelfTest (XDmaMulti *InstancePtr)
void XDmaMulti_Reset (XDmaMulti *InstancePtr, unsigned Channel)
void XDmaMulti_Transfer (XDmaMulti *InstancePtr, unsigned Channel, Xuint32 *SourcePtr,
    Xuint32 *DestinationPtr, Xuint32 ByteCount)
XStatus XDmaMulti_SgStart (XDmaMulti *InstancePtr, unsigned Channel)
XStatus XDmaMulti_SgStop (XDmaMulti *InstancePtr, unsigned Channel, XBufDescriptor
    **BufDescriptorPtr)
XStatus XDmaMulti_CreateSgList (XDmaMulti *InstancePtr, unsigned Channel, Xuint32
    *BdMemoryPtr, Xuint32 ByteCount)
Xboolean XDmaMulti_IsSgListEmpty (XDmaMulti *InstancePtr, unsigned Channel)
XStatus XDmaMulti_PutDescriptor (XDmaMulti *InstancePtr, unsigned Channel,
    XBufDescriptor *BufDescriptorPtr)
XStatus XDmaMulti_CommitPuts (XDmaMulti *InstancePtr, unsigned Channel)
XStatus XDmaMulti_GetDescriptor (XDmaMulti *InstancePtr, unsigned Channel,
    XBufDescriptor **BufDescriptorPtr)
```

Define Documentation

```
#define
XDM_DMACR_SOURCE_INCR_MASK
```

The following constants provide access to the bit fields of the DMA control register (DMACR).


```
#define XDmaMulti_mGetEventChannel( Status )
```

Determine the Channel Number of the provided Status/Event.

Parameters:

Status is the Status/Event read from Event FIFO.

Returns:

The 32-bit value of Channel that caused the Status/Event.

Note:

Xuint32 [XDmaMulti_mGetEventChannel](#)(Xuint32 Status);

```
#define XDmaMulti_mGetEventEnable( InstancePtr,  
                                Channel )
```

Read the Event Enable Register of the given channel.

Parameters:

InstancePtr is the instance to be used.

Channel is the channel of interest, zero based.

Returns:

The 32-bit value of the register.

The interrupt enable of the DMA channel. One or more of the following values may be contained the register. Each of the values are unique bit masks. See [xdma_multi.h](#) for a description of possible values. The return values are prefixed with XDM_SWCR_*.*

Note:

Xuint32 [XDmaMulti_mGetEventEnable](#)(XDmaMulti *InstancePtr, unsigned Channel);

```
#define XDmaMulti_mGetEventParameter( Status )
```

Determine the Event Parameter of the provided Status/Event.

Parameters:

Status is the Status/Event read from Event FIFO.

Returns:

The 32-bit value of associated Event Parameter.

Note:

Xuint32 [XDmaMulti_mGetEventChannel](#)(Xuint32 Status);

```
#define XDmaMulti_mGetEventStatus( InstancePtr )
```

Read the Event FIFO Status Register of the Device.

Parameters:

InstancePtr is the instance to be used

Returns:

The 32-bit value of the register

The Event FIFO Status Register contents. This register is a FIFO and it can only be read once per event. Once it is read the event is discarded and the next event appears. The user must ensure that this FIFO is read only once per event to maintain synchronization with hardware.

One or more of the following values may be contained the register. Each of the values are unique bit masks. See [xdma_multi.h](#) for a description of possible values. The return values are prefixed with XDM_GEFIFO_*.*

Note:

Xuint32 [XDmaMulti_mGetEventStatus](#)(XDmaMulti *InstancePtr);

```
#define XDmaMulti_mGetGlobalControl( InstancePtr )
```

Read the Global Control Register.

Parameters:

InstancePtr is the instance to be used.

Returns:

The 32-bit value of the register.

The control register contents of the DMA Hardware. One or more of the following values may be contained the register. Each of the values are unique bit masks. See [xdma_multi.h](#) for a description of possible values. The return values are prefixed with XDM_GCSR_*.*

Note:

Xuint32 [XDmaMulti_mGetGlobalControl](#)(XDmaMulti *InstancePtr);

```
#define XDmaMulti_mGetPktThreshold( InstancePtr,  
                                   Channel      )
```

Read the Packet Count Threshold of the given channel.

Parameters:

InstancePtr is the instance to be used.

Channel is the channel of interest, zero based.

Returns:

The 10-bit value of the Packet Count Threshold. Range is 0-1023, 0 is disabled.

This function reads the value of the packet count threshold register of the DMA channel. It reflects the number of packets that must be sent or received before generating an interrupt.

Note:

Xuint32 [XDmaMulti_mGetPktThreshold](#)(XDmaMulti *InstancePtr, unsigned Channel);

```
#define XDmaMulti_mGetPktWaitBound( InstancePtr,  
                                   Channel      )
```

Read the Packet Wait Bound of the given channel.

Parameters:

InstancePtr is the instance to be used.

Channel is the channel of interest, zero based.

Returns:

The 10-bit value of the Packet Wait Bound. Range is 0-1023, 0 is disabled.

The packet wait bound register contents for the DMA channel.

Note:

Xuint32 **XDmaMulti_mGetPktWaitBound**(XDmaMulti *InstancePtr, unsigned Channel);

```
#define XDmaMulti_mGetStatus( InstancePtr,  
                             Channel      )
```

Read the Status Register of the given channel.

Parameters:

InstancePtr is the instance to be used.

Channel is the channel of interest, zero based.

Returns:

The 32-bit value of the register.

Note:

Xuint32 **XDmaMulti_mGetStatus**(XDmaMulti *InstancePtr, unsigned Channel);

```
#define XDmaMulti_mSetControl( InstancePtr,  
                              Channel,  
                              Control      )
```

Set the Control Register of the given channel with the provided value.

Parameters:

InstancePtr is the instance to be used.

Channel is the channel of interest, zero based.

Control is the 32-bit value to write to the register.

Control contains the value to be written to the control register of the DMA channel. One or more of the following values may be contained the register. Each of the values are unique bit masks such that they may be ORed together to enable multiple bits or inverted and ANDed to disable multiple bits. Use the XDM_DMOCR_* constants defined in [xdma_multi.h](#) to create the bit-mask to be written to the register.

Returns:

None.

Note:

```
void XDmaMulti_mSetControl(XDmaMulti *InstancePtr, unsigned Channel, Xuint32 Control);
```

```
#define XDmaMulti_mSetEventEnable( InstancePtr,  
                                Channel,  
                                Enable    )
```

Set the Event Enable of the given channel with the provided value.

Parameters:

InstancePtr is the instance to be used.

Channel is the channel of interest, zero based.

Enable is the 32-bit value to write to the register.

Enable contains the event enable register contents to be written in the DMA channel. One or more of the following values may be contained the register. Each of the values are unique bit masks such that they may be ORed together to enable multiple bits or inverted and ANDed to disable multiple bits. Use the XDM_SWCR_* constants defined in [xdma_multi.h](#) to create the bit-mask to be written to the register.

Returns:

None.

Note:

```
void XDmaMulti_mSetEventEnable(XDmaMulti *InstancePtr, unsigned Channel, Xuint32 Enable);
```

```
#define XDmaMulti_mSetGlobalControl( InstancePtr,  
                                   Control    )
```

Write the Global Control Register.

Parameters:

InstancePtr is the instance to be used.

Control is the 32-bit value to write to the register.

Returns:

None

Write the contents to the DMA Hardware. Use the `XDM_GCSR_*` constants defined in [xdma_multi.h](#) to create the bit-mask to be written to the register.

Note:

```
void XDmaMulti_mSetGlobalControl(XDmaMulti *InstancePtr, Xuint32 Control);
```

```
#define XDmaMulti_mSetPktThreshold( InstancePtr,  
                                   Channel,  
                                   Threshold  )
```

Set the Packet Count Threshold Value for the given channel with the provided value.

This function sets the value of the packet count threshold register of the DMA channel. It reflects the number of packets that must be sent or received before generating an interrupt. This value helps implement a concept called "interrupt coalescing", which is used to reduce the number of interrupts from devices with high data rates.

Parameters:

InstancePtr is the instance to be used.

Channel is the channel of interest, zero based.

Threshold is the 10-bit value to write to the register. Range is 0-1023, 0 is disabled.

Returns:

None

Note:

```
void XDmaMulti_mSetPktThreshold(XDmaMulti *InstancePtr, unsigned Channel,  
Xuint32 Threshold);
```

```
#define XDmaMulti_mSetPktWaitBound( InstancePtr,  
                                   Channel,  
                                   WaitBound )
```

Set the Packet Wait Bound Value for the given channel with the provided value.

WaitBound is the value, in milliseconds, to be stored in the wait bound register of the DMA channel and is a value in the range 0 - 1023. A value of 0 disables the packet wait bound timer.

Parameters:

InstancePtr is the instance to be used.

Channel is the channel of interest, zero based.

WaitBound is the 10-bit value to write to the Packet Wait Bound. Range is 0-1023, 0 is disabled.

Returns:

None.

Note:

```
void XDmaMulti_mSetPktWaitBound(XDmaMulti *InstancePtr, unsigned Channel,  
Xuint32 WaitBound);
```

```
#define XDmaMulti_mSizeNeeded( NumChannels )
```

Determine the size needed for the DMA channel structure.

Parameters:

NumChannels is the total number of DMA Channels configured in the hardware.

Returns:

The size of the structure in bytes. Includes array used for addresses at the beginning of the data.

Note:

Xuint32 [XDmaMulti_mSizeNeeded](#)(Xuint32 NumChannels);

Function Documentation

```
XStatus ( XDmaMulti *  
XDmaMulti_CommitPuts InstancePtr,  
                unsigned Channel  
                )
```

This function commits the buffer descriptors which have been put into the scatter list for a DMA channel since the last commit operation was performed. This enables the calling functions to put several buffer descriptors into the list (e.g., a packet's worth) before allowing the scatter gather operations to start. This prevents the DMA channel hardware from starting to use the buffer descriptors in the list before they are ready to be used (multiple buffer descriptors for a single packet).

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.

Channel is the particular channel of interest.

Returns:

- XST_SUCCESS if the buffer descriptors of the list were successfully committed.
- XST_DMA_SG_NOTHING_TO_COMMIT indicates that the buffer descriptors were not committed because there was nothing to commit in the list. All the buffer descriptors which are in the list are committed.

Note:

None.

```

XStatus
XDmaMulti_CreateSgList
    ( XDmaMulti *
      InstancePtr,
      unsigned
      Channel,
      Xuint32 *
      BdMemoryPtr,
      Xuint32
      ByteCount
    )

```

This function creates a scatter gather list for a channel of DMA. A scatter gather list consists of a list of buffer descriptors that are available to be used for scatter gather operations. Buffer descriptors are put into the list to request a scatter gather operation to be performed.

A number of buffer descriptors are created from the specified memory and put into a buffer descriptor list as empty buffer descriptors. This function must be called before non-empty buffer descriptors may be put into the DMA channel to request scatter gather operations.

Parameters:

- InstancePtr* contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.
- Channel* is the particular channel of interest.
- BdMemoryPtr* contains a pointer to the memory which is to be used for buffer descriptors and must not be cached.
- ByteCount* contains the number of bytes for the specified memory to be used for buffer descriptors.

Returns:

- XST_SUCCESS if the scatter gather list was successfully created.
- XST_DMA_SG_LIST_EXISTS indicates that the scatter gather list was not created because the list has already been created.

Note:

None.

```

XStatus
XDmaMulti_GetDescriptor
    ( XDmaMulti *
      InstancePtr,
      unsigned
      XBufDescriptor ** Channel,
      BufDescriptorPtr
    )

```

This function gets a buffer descriptor from the scatter gather list of the DMA channel. The buffer descriptor is retrieved from the scatter gather list and the scatter gather list is updated to not include the retrieved buffer descriptor. This is typically done after a scatter gather operation completes indicating that a data buffer has been successfully sent or data has been received into the data buffer. The purpose of this function is to allow the device using the scatter gather operation to get the results of the operation.

Parameters:

- InstancePtr* contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.
- Channel* is the particular channel of interest.
- BufDescriptorPtr* is a pointer to a pointer to the buffer descriptor which was retrieved from the list. The buffer descriptor is not really removed from the list, but it is changed to a state such that the hardware will not use it again until it is put into the scatter gather list of a DMA channel.

Returns:

- XST_SUCCESS if a buffer descriptor was retrieved
- XST_DMA_SG_NO_LIST indicates the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY indicates no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.
- BufDescriptorPtr is updated to point to the buffer descriptor which was retrieved from the list if the status indicates success.

Note:

None.

```

XStatus
XDmaMulti_Initialize
    ( XDmaMulti *
      InstancePtr,
      Xuint32
      BaseAddress,
      Xuint32 *
      UserMemoryPtr,
      Xuint16
      ChannelCount
    )

```

This function initializes the multichannel DMA. This function must be called prior to using the multichannel DMA. Initialization of the instance includes setting up the registers base address, register access tables, and resetting the channels such that they are in a known state. Interrupts for the channels are disabled when the channels are reset.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on.

BaseAddress contains the base address of the registers for the multichannel DMA.

UserMemoryPtr is a pointer to user allocated memory for the register access tables and the buffer descriptor tables. The **XDmaMulti_mSizeNeeded()** macro returns the amount of memory needed for the number of channels configured.

ChannelCount is the total number of transmit and receive channels associated with the device

Returns:

- o XST_SUCCESS indicating initialization was successful.

Note:

None.

```

Xboolean
XDmaMulti_IsSgListEmpty
    ( XDmaMulti *
      InstancePtr,
      unsigned
      Channel
    )

```

This function determines if the scatter gather list of a DMA channel is empty with regard to buffer descriptors which are pointing to buffers to be used for scatter gather operations.

Channel is the particular channel of interest.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.

Channel is the particular channel of interest.

Returns:

- o XTRUE if the scatter gather list is empty, otherwise a value of XFALSE.

Note:

None.

```
XStatus ( XDmaMulti * InstancePtr,  
XDmaMulti_PutDescriptor unsigned Channel,  
XBufDescriptor * BufferDescriptorPtr  
)
```

DESCRIPTION:

This function puts a buffer descriptor into the DMA channel scatter gather list. A DMA channel maintains a list of buffer descriptors which are to be processed. This function puts the specified buffer descriptor at the next location in the list. Note that since the list is already intact, the information in the parameter is copied into the list (rather than modify list pointers on the fly).

After buffer descriptors are put into the list, they must also be committed by calling another function. This allows multiple buffer descriptors which span a single packet to be put into the list while preventing the hardware from starting the first buffer descriptor of the packet.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.

Channel is the particular channel of interest.

BufferDescriptorPtr is a pointer to the buffer descriptor to be put into the next available location of the scatter gather list.

Returns:

- XST_SUCCESS if the buffer descriptor was successfully put into the scatter gather list.
- XST_DMA_SG_NO_LIST indicates the scatter gather list has not been created.
- XST_DMA_SG_LIST_FULL indicates the buffer descriptor was not put into the list because the list was full.
- XST_DMA_SG_BD_LOCKED indicates the buffer descriptor was not put into the list because the buffer descriptor in the list which is to be overwritten was locked. A locked buffer descriptor indicates the higher layered software is still using the buffer descriptor.

Note:

It is necessary to create a scatter gather list for a DMA channel before putting buffer descriptors into it.

```
void          ( XDmaMulti *  
XDmaMulti_Reset          InstancePtr,  
          unsigned          Channel  
          )
```

This function resets a particular channel of the multichannel DMA. This is a destructive operation such that it should not be done while a channel is being used. If the channel of the DMA is transferring data into other blocks, such as a FIFO, it may be necessary to reset other blocks. This function does not modify the contents of the scatter gather list for the specified channel such that the user is responsible for getting buffer descriptors from the list if necessary.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on.

Channel is the particular channel that is to be reset.

Returns:

None.

Note:

The registers are set to zero because they are in BRAM in this device. There is no initial state for BRAM and therefore it must be initialized to actually accomplish the reset state for the device..

XStatus (**XDmaMulti ***)
XDmaMulti_SelfTest *InstancePtr*

DESCRIPTION:

This function performs a self test on the multichannel DMA. This self test is destructive in that channel 0 of the DMA is reset and DMACR defaults are verified.

Parameters:

InstancePtr is a pointer to the multichannel DMA to be operated on.

Returns:

- XST_SUCCESS is returned if the self test is successful, or one of the following errors.
- XST_FAILURE Indicates the test failed

Note:

This test does not perform a DMA transfer to test the channel because the DMA hardware will not currently allow a non-local memory transfer to non-local memory (memory copy), but only allows a non-local memory to or from the device memory (typically a FIFO).

XStatus (**XDmaMulti ***
XDmaMulti_SgStart *InstancePtr,*
unsigned *Channel*
)

This function starts a scatter gather operation for a scatter gather DMA channel. The first buffer descriptor in the buffer descriptor list will be started with the scatter gather operation. A scatter gather list should have previously been created for the channel of DMA and buffer descriptors put into the scatter gather list such that there are scatter operations ready to be performed.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.

Channel is the particular channel of interest.

Returns:

- XST_SUCCESS if scatter gather was started successfully
- XST_DMA_SG_NO_LIST indicates the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY indicates scatter gather was not started because the scatter gather list of the DMA channel does not contain any buffer descriptors that are ready to be processed by the hardware.
- XST_DMA_SG_IS_STARTED indicates scatter gather was not started because the scatter gather was not stopped, but was already started.
- XST_DMA_SG_BD_NOT_COMMITTED indicates the buffer descriptor of scatter gather list which was to be started is not committed to the list. This status is more likely if this function is being called from an ISR and non-ISR processing is putting descriptors into the list.
- XST_DMA_SG_NO_DATA indicates that the buffer descriptor of the scatter gather list which was to be started had already been used by the hardware for a DMA transfer that has been completed.

Note:

It is the responsibility of the caller to get all the buffer descriptors after performing a stop operation and before performing a start operation. If buffer descriptors are not retrieved between stop and start operations, buffer descriptors may be processed by the hardware more than once.

```

XStatus          ( XDmaMulti *
XDmaMulti_SgStop      InstancePtr,
                        unsigned      Channel,
                        XBufDescriptor ** BufDescriptorPtr
                        )

```

This function stops a scatter gather operation for a scatter gather DMA channel. This function starts the process of stopping a scatter gather operation that is in progress and waits for the stop to be completed. Since it waits for the operation to be stopped before returning, this function could take an amount of time relative to the size of the DMA scatter gather operation which is in progress. The scatter gather list of the DMA channel is not modified by this function such that starting the scatter gather channel after stopping it will cause it to resume. This operation is considered to be a graceful stop in that the scatter gather operation completes the current buffer descriptor before stopping.

If the interrupt is enabled, an interrupt will be generated when the operation is stopped and the caller is responsible for handling the interrupt.

Parameters:

- InstancePtr* contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.
- Channel* is the particular channel of interest.
- BufDescriptorPtr* is also a return value which contains a pointer to the buffer descriptor which the scatter gather operation completed when it was stopped.

Returns:

- XST_SUCCESS if scatter gather was stopped successfully
- XST_DMA_SG_IS_STOPPED indicates scatter gather was not stopped because the scatter gather is not started, but was already stopped.
- BufDescriptorPtr contains a pointer to the buffer descriptor which was completed when the operation was stopped.

Note:

This function implements a loop which polls the hardware for the Stop indication. If the hardware is hung or malfunctioning, this function will loop for an infinite amount of time and this function may never return.

```

void
XDmaMulti_Transfer
    ( XDmaMulti *
      InstancePtr,
      unsigned
      Xuint32 *
      Channel,
      Xuint32 *
      SourcePtr,
      Xuint32 *
      DestinationPtr,
      Xuint32
      ByteCount
    )

```

This function starts the specified channel of DMA transferring data from a memory source to a memory destination. This function only starts the operation and returns before the operation may be complete. If the interrupt is enabled, an interrupt will be generated when the operation is complete, otherwise it is necessary to poll the channel status to determine when it's complete. It is the responsibility of the caller to determine when the operation is complete by handling the generated interrupt or polling the status. It is also the responsibility of the caller to ensure that the DMA channel specified is not busy with another transfer before calling this function.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on.
Channel is the particular channel of interest.
SourcePtr contains a pointer to the source memory where the data is to be transferred from and must be 32 bit aligned.
DestinationPtr contains a pointer to the destination memory where the data is to be transferred and must be 32 bit aligned.
ByteCount contains the number of bytes to transfer during the DMA operation.

Returns:

None.

Note:

The DMA h/w will not currently allow a non-local memory transfer to non-local memory (memory copy), but only allows a non-local memory to or from the device memory (typically a FIFO).

It is the responsibility of the caller to ensure that the cache is flushed and invalidated both before and after the DMA operation completes if the memory pointed to is cached. The caller must also ensure that the pointers contain a physical address rather than a virtual address if address translation is being used.

dma/v1_00_a/src/xdma_multi_i.h File Reference

Detailed Description

Description

This file contains data which is shared internal data for the multichannel DMA component. It is also shared with the buffer descriptor component which is very tightly coupled with the multichannel DMA component.

Note:

The last buffer descriptor constants must be located here to prevent a circular dependency between the multichannel DMA component and the buffer descriptor component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	09/16/03	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

Defines

```
#define XDM_CONTROL_LAST_BD_MASK  
#define XDM_STATUS_LAST_BD_MASK  
#define XDM_GMIR_REG_OFFSET  
#define XDM_SYS_REG_OFFSET  
#define XDM_SWCR_PWB_SHIFT  
#define XDM_GEFIFO_CHAN_SHIFT
```

```
#define XDM_SYS_REG_ANCY_SHIFT
```

Define Documentation

```
#define  
XDM_CONTROL_LAST_BD_MASK
```

The following constant provides access to the bit fields of the DMA control register (DMACR) which must be shared between the multichannel DMA component and the buffer descriptor component.

```
#define  
XDM_GEFIFO_CHAN_SHIFT
```

The following constant is the shift value for the Event Channel Field.

```
#define  
XDM_GMIR_REG_OFFSET
```

The following constants provide access to each of the Global Registers for the multichannel DMA.

```
#define  
XDM_STATUS_LAST_BD_MASK
```

The following constant provides access to the bit fields of the DMA status register (DMASR) which must be shared between the multichannel DMA component and the buffer descriptor component.

```
#define  
XDM_SWCR_PWB_SHIFT
```

The following constant is the shift value for the Packet Wait Bound.

```
#define  
XDM_SYS_REG_ANCY_SHIFT
```

The following constant is the shift value for the Event Channel Field.

#define

XDM_SYS_REG_OFFSET

The following constants provide access to each of the Channel Specific multichannel DMA Registers.

Generated on 24 Jun 2004 for Xilinx Device Drivers

XDmaMulti Struct Reference

```
#include <xdma_multi.h>
```

Detailed Description

The following structure contains data which is on a per instance basis for the XDmaMulti component.

Data Fields

Xuint32 BaseAddress

Xuint32 IntrFifoAddress

Xuint32 IsReady

Xuint16 ChannelCount

Xuint32 * AddrTablePtr

XDmaMulti_SgData * SgDataTablePtr

Field Documentation

Xuint32* XDmaMulti::

AddrTablePtr

Beginning of Register address
table

Xuint32 XDmaMulti::

BaseAddress

Base address of
channels

Xuint16 XDmaMulti::

ChannelCount

Number of DMA channels on device

Xuint32 XDmaMulti::

IntrFifoAddress

Interrupt FIFO address

Xuint32 XDmaMulti::

IsReady

Device is initialized and ready

XDmaMulti_SgData* XDmaMulti::

SgDataTablePtr

Beginning of SGDMA data structures

The documentation for this struct was generated from the following file:

- [dma/v1_00_a/src/xdma_multi.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

XDmaMulti_SgData Struct Reference

```
#include <xdma_multi.h>
```

Detailed Description

The following structure contains data which is used to maintain the buffer descriptor list.

Data Fields

XBufDescriptor * [PutPtr](#)

XBufDescriptor * [GetPtr](#)

XBufDescriptor * [CommitPtr](#)

XBufDescriptor * [LastPtr](#)

[Xuint16 TotalDescriptorCount](#)

[Xuint16 ActiveDescriptorCount](#)

Field Documentation

[Xuint16 XDmaMulti_SgData::](#)

[ActiveDescriptorCount](#)

of descriptors pointing to
buffers

[XBufDescriptor* XDmaMulti_SgData::](#)

[CommitPtr](#)

Keep track of where to commit in
list

**XBufDescriptor* XDmaMulti_SgData::
GetPtr**

Keep track of where to get from
list

**XBufDescriptor* XDmaMulti_SgData::
LastPtr**

Keep track of the last put in the
list

**XBufDescriptor* XDmaMulti_SgData::
PutPtr**

Keep track of where to put into
list

**Xuint16 XDmaMulti_SgData::
TotalDescriptorCount**

Total # of descriptors in the
list

The documentation for this struct was generated from the following file:

- [dma/v1_00_a/src/xdma_multi.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

hdlc/v2_01_a/src/xhdlc_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xhdlc.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
2.00a	ecm	04/17/03	First release
2.01a	ecm	02/16/04	Add multichannel scatter-gather DMA support

```
#include "xbasic_types.h"  
#include "xio.h"  
#include "xstatus.h"
```

Data Structures

```
struct XHdlc\_ChannelID  
struct XHdlc\_RegisterOffsets
```

Defines

```
#define XHdlc\_mStatusFifoChannelID(Status, ChannelID)  
#define XHdlc\_mReadReg(BaseAddress, RegOffset)  
#define XHdlc\_mWriteReg(BaseAddress, RegOffset, Data)  
#define XHdlc\_mReadPhysChanReg(BaseAddress, ChannelID, Register)  
#define XHdlc\_mWritePhysChanReg(BaseAddress, ChannelID, Register, Data)
```

```
#define XHdlc_mReadSubChanReg(BaseAddress, ChannelID, Register)
#define XHdlc_mWriteSubChanReg(BaseAddress, ChannelID, Register, Data)
#define XHdlc_mReadFifoSubReg(BaseAddress, ChannelID, Register)
#define XHdlc_mWriteFifoSubReg(BaseAddress, ChannelID, Register, Data)
#define XHdlc_mNumberOfPhysicalChannels(BaseAddress)
#define XHdlc_mNumberOfSubChannels(BaseAddress, ChannelID)
#define XHdlc_mCreateChannelID(Phys, Sub, ChannelID)
```

Functions

```
void XHdlc_InitializeRegisterAccess (Xuint32 BaseAddress)
Xuint32 XHdlc_PollSend (Xuint32 BaseAddress, XHdlc_ChannelID ChannelID, Xuint32
    *FramePtr, Xuint32 ByteCount, Xuint32 *ChannelStatusPtr)
Xuint32 XHdlc_PollRecv (Xuint32 BaseAddress, XHdlc_ChannelID *ChannelIDPtr, Xuint32
    *FramePtr, Xuint32 ByteCount, Xuint32 *ChannelStatusPtr, Xuint32 *RxErrorStatus)
void XHdlc_DeviceEnable (Xuint32 BaseAddress)
void XHdlc_DeviceDisable (Xuint32 BaseAddress)
void XHdlc_PhysEnable (Xuint32 BaseAddress, XHdlc_ChannelID ChannelID)
void XHdlc_PhysDisable (Xuint32 BaseAddress, XHdlc_ChannelID ChannelID)
void XHdlc_PolledSubEnable (Xuint32 BaseAddress, XHdlc_ChannelID ChannelID)
void XHdlc_SubEnable (Xuint32 BaseAddress, XHdlc_ChannelID ChannelID)
void XHdlc_SubDisable (Xuint32 BaseAddress, XHdlc_ChannelID ChannelID)
void XHdlc_PolledSubTxEnable (Xuint32 BaseAddress, XHdlc_ChannelID ChannelID)
void XHdlc_SubTxEnable (Xuint32 BaseAddress, XHdlc_ChannelID ChannelID)
void XHdlc_SubTxDisable (Xuint32 BaseAddress, XHdlc_ChannelID ChannelID)
```

Define Documentation

```
#define XHdlc_mCreateChannelID( Phys,
                               Sub,
                               ChannelID )
```

Create ChannelID from provided number pair.

Parameters:

Phys is the physical channel.

Sub is the associated sub channel.

Returns:

ChannelID is the structure containing the provided Pyhs and Sub.

Note:

None.

```
void XHdlc_mCreateChannelID(int Phys, int Sub, XHdlc_Channel  
ChannelID);
```

```
#define XHdlc_mNumberOfPhysicalChannels( BaseAddress )
```

Returns the number of physical channels configured. Actually reads the hardware register to determine how the hardware is configured.

Parameters:

BaseAddress is the base address of the device.

ChannelID is specific channel.

Returns:

Number of physical channels configured.

Note:

None.

```
Xuint32 XHdlc_mNumberOfPhysicalChannels(Xuint32 BaseAddress);
```

```
#define XHdlc_mNumberOfSubChannels( BaseAddress,  
ChannelID )
```

Returns the number of sub channels configured. Actually reads the hardware register to determine how the hardware is configured. This sub channel data in the ChannelID is ignored.

Parameters:

BaseAddress is the base address of the device.

ChannelID is specific channel.

Returns:

Number of physical channels configured.

Note:

None.

Xuint32 XHdlc_mNumberOfSubChannels(Xuint32 BaseAddress, XHdlc_ChannelChannelID);

```
#define XHdlc_mReadFifoSubReg( BaseAddress,  
                               ChannelID,  
                               Register    )
```

Read the HDLC FIFO Sub Channel Control Register. A 32 bit read of the register is performed.

Parameters:

BaseAddress is the base address of the device.

ChannelID is the Channel desired.

Register is the register to be read.

Returns:

The 32 bit value of the register

Note:

None.

Xuint32 XHdlc_mReadFifoSubReg(Xuint32 BaseAddress, **XHdlc_Channel** ChannelID, Xuint32 Register);

```
#define XHdlc_mReadPhysChanReg( BaseAddress,  
                                ChannelID,  
                                Register    )
```

Read the HDLC Physical Channel Control Register. A 32 bit read of the register is performed.

Parameters:

BaseAddress is the base address of the device.

ChannelID is the Channel desired.

Register is the register to be read.

Returns:

The 32 bit value of the register.

Note:

None.

Xuint32 XHdlc_mReadPhysChanReg(Xuint32 BaseAddress, **XHdlc_Channel** ChannelID, Xuint32 Register);

```
#define XHdlc_mReadReg( BaseAddress,  
                      RegOffset  )
```

Read the specified HDLC register. A 32 bit read of the register is performed.

Parameters:

BaseAddress is the base address of the device.

RegOffset is the register offset to be read.

Returns:

The 32 bit value of the register.

Note:

None.

Xuint32 **XHdlc_mReadReg**(Xuint32 BaseAddress, int RegOffset);

```
#define XHdlc_mReadSubChanReg( BaseAddress,  
                              ChannelID,  
                              Register  )
```

Read the HDLC SubChannel register. A 32 bit read of the register is performed.

Parameters:

BaseAddress is the base address of the device.

ChannelID is the Channel desired.

Register is the register to be read.

Returns:

The 32 bit value of the register.

Note:

None.

Xuint32 XHdlc_mReadSubChanReg(Xuint32 BaseAddress, **XHdlc_Channel** ChannelID, Xuint32 Register);

```
#define XHdlc_mStatusFifoChannelID( Status,  
                                   ChannelID )
```

Extract the ChannelID from the Status FIFO data.

Parameters:

Status is the data from the Status FIFO.

ChannelID is ChannelID contained in the Data.

Returns:

None.

Note:

XHdlc_Initialize must be run prior to running this function

void **XHdlc_mStatusFifoChannelID**(Xuint32 Status, XHdlc_ChannelID ChannelID)

```
#define XHdlc_mWriteFifoSubReg( BaseAddress,  
                               ChannelID,  
                               Register,  
                               Data )
```

Write the specified data to the HDLC FIFO Sub Channel Control Register.

Parameters:

BaseAddress is the base address of the device.

ChannelID is the Channel desired.

Register is the register to be written to.

Data is the 32-bit value to write to the register.

Returns:

None.

Note:

None.

```
void XHdlc_mWriteFifoSubReg(Xuint32 BaseAddress, XHdlc_Channel ChannelID, Xuint32 Register, Xuint32 Data);
```

```
#define XHdlc_mWritePhysChanReg( BaseAddress,  
                                ChannelID,  
                                Register,  
                                Data          )
```

Write the specified data to the HDLC Physical Channel Control Register.

Parameters:

BaseAddress is the base address of the device.

ChannelID is the Channel desired.

Register is the register to be written to.

Data is the 32-bit value to write to the register.

Returns:

None.

Note:

None.

```
void XHdlc_mWritePhysChanReg(Xuint32 BaseAddress, XHdlc_Channel ChannelID, Xuint32 Register, Xuint32 Data);
```

```
#define XHdlc_mWriteReg( BaseAddress,  
                        RegOffset,  
                        Data      )
```

Write the specified data to the specified register.

Parameters:

BaseAddress is the base address of the device.

RegOffset is the register offset to be written.

Data is the 32-bit value to write to the register.

Returns:

None.

Note:

None.

```
void XHdlc_mWriteReg(Xuint32 BaseAddress, int RegOffset, Xuint32  
Data);
```

```
#define XHdlc_mWriteSubChanReg( BaseAddress,  
                               ChannelID,  
                               Register,  
                               Data      )
```

Write the specified data to the HDLC Subchannel Register.

Parameters:

BaseAddress is the base address of the device.

ChannelID is the Channel desired.

Register is the register to be written to.

Data is the 32-bit value to write to the register.

Returns:

None.

Note:

None.

```
void XHdlc_mWriteSubChanReg(Xuint32 BaseAddress, XHdlc_Channel ChannelID, Xuint32  
Register, Xuint32 Data);
```

Function Documentation

```
void ( Xuint32 )  
XHdlc_DeviceDisable BaseAddress
```

Disable the HDLC Device. This prevents transmit and receive for the entire device. If there are channel(s) in use when the device is disabled, there will likely be garbage in the FIFO of the receive that needs to be cleaned out prior to using the channel(s) again.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
void ( Xuint32 )  
XHdlc_DeviceEnable BaseAddress
```

Enable the HDLC Device. This allows the channels to transmit and receive data if available and the physical channel(s) and sub channel(s) are enabled.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
void ( Xuint32 )  
XHdlc_InitializeRegisterAccess BaseAddress
```

Level 0 driver initialization. This function initializes the register access arrays allowing faster access to the hardware registers. This function also initializes the sub channel registers in the device itself since they are actually FPGA Block RAM (BRAM) which is not initialized in the FPGA configuration process.

Parameters:

param BaseAddress is the base address of the device

Returns:

none

Note:

None.

```
void XHdlc_PhysDisable ( Xuint32 BaseAddress, XHdlc_ChannelID ChannelID )
```

Disable the physical channel, both TX and RX. The sub channel of the channelID is ignored. If there are channel(s) in use when the device is disabled, there will likely be garbage in the FIFO of the receive that needs to be cleaned out prior to using the channel(s) again.

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```
void XHdlc_PhysEnable ( Xuint32 BaseAddress, XHdlc_ChannelID ChannelID )
```

Enable the physical channel, both TX and RX. The sub channel of the channelID is ignored.

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```
void XHdlc_PolledSubEnable ( Xuint32 BaseAddress, XHdlc_ChannelID ChannelID )
```

Enable the sub channel, both TX and RX for Polled Operation.

This configuration is intended for the polling operation in that the TX/RX complete event is written to the event FIFOs to allow the send/ receive function to just poll on that event FIFOs and not have to sort through the status FIFO events waiting for the complete. If the XHD_SCCR_*XCR_EVT0_MASK bit is not set, the TX/RX complete events are not sent to the FIFO's and the routines will not know when to write/read to/from the FIFO's

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```

void          ( Xuint32
XHdlc_PolledSubTxEnable      BaseAddress,
                               ChannelID
                               )

```

Enable the transmitter only on the sub channel for Polled Operation.

This configuration is intended for the polling operation in that the TX complete event is written to the event FIFOs to allow the send function to just poll on that event FIFOs and not have to sort through the status FIFO events waiting for the complete. If the XHD_SCCR_*XCR_EVT0_MASK bit is not set, the TX complete events are not sent to the FIFO's and the routines will not know when to write to the FIFO's

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```

Xuint32      ( Xuint32
XHdlc_PollRecv      BaseAddress,
                    ChannelIDPtr,
                    FramePtr,
                    ByteCount,
                    ChannelStatusPtr,
                    RxErrorStatus
                    )

```

Receive a frame. Wait for a frame to arrive.

Parameters:

BaseAddress is the base address of the device
ChannelID is the channel data was received on.
FramePtr is a pointer to a 32 bit word-aligned buffer where the frame
ByteCount contains the number of bytes available in the buffer.
ChannelStatusPtr is a pointer to a channel status that will be valid after this function returns. The definitions of the bit positions are in [xhdlc_1.h](#) in the XHD_STAT_FIFO_*_MASK group of defines
RxErrorStatus is a pointer to a receive event that will be valid after this function returns. The definitions of the bit positions are in [xhdlc_1.h](#) in the XHD_RXEF_*_MASK group of defines

Returns:

number of Bytes read from the FIFO

Note:

This function assumes that the channel has been enabled at the sub channel and physical channel and that the entire device has been enabled. The function will hang if all of the enables are not correct.

This function assumes packet size is such that the entire received packet/ frame fits within the FIFO including CRC.

```
Xuint32 ( Xuint32 BaseAddress,  
XHdlc_PollSend XHdlc_ChannelID ChannelID,  
Xuint32 * FramePtr,  
Xuint32 ByteCount,  
Xuint32 * ChannelStatusPtr  
)
```

Send a HDLC frame. This size is the total frame size, including header. This function blocks waiting for the frame to be transmitted.

Parameters:

BaseAddress is the base address of the device
ChannelID is the channel data is to be transmitted on.
FramePtr is a pointer to 32 bit word-aligned frame
ByteCount is the number of bytes in the frame to be sent
ChannelStatusPtr is a pointer to a channel status that will be valid after this function returns. The definitions of the bit positions are in [xhdlc_1.h](#) in the XHD_TXEF_*_MASK group of defines

Returns:

number of Bytes that still need to be sent, i.e. Byte Count requested - Actual Bytes sent

Note:

This function assumes that the channel has been enabled at the sub channel and physical channel and that the entire device has been enabled. The function will hang if all of the enables are not correct. This function also assumes that the FIFO will hold the entire packet to be sent.

```
void XHdlc_SubDisable ( Xuint32 BaseAddress,  
                      XHdlc_ChannelID ChannelID  
                      )
```

Disable the sub channel, both TX and RX.

Parameters:

BaseAddress is the base address of the device
ChannelID is specific channel

Returns:

None.

Note:

None.

```
void XHdlc_SubEnable ( Xuint32 BaseAddress,  
                    XHdlc_ChannelID ChannelID  
                    )
```

Enable the sub channel, both TX and RX.

This function sets the TX and RX enable bits in the sub channel control register. It can be used for interrupt driven with FIFOs and SGDMA modes of operation.

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```
void XHdlc_SubTxDisable ( Xuint32 BaseAddress,  
                        XHdlc_ChannelID ChannelID  
                        )
```

Disable the transmitter only on sub channel.

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```
void  
XHdlc_SubTxEnable ( Xuint32  
                   BaseAddress,  
                   XHdlc_ChannelID ChannelID  
                   )
```

Enable the transmitter only on the sub channel.

This function sets the TX enable bit in the sub channel control register. It can be used for interrupt driven with FIFOs and SGDMA modes of operation.

Parameters:

BaseAddress is the base address of the
device

ChannelID is specific channel

Returns:

None.

Note:

None.

cpu_ppc405/v1_00_a/src/xio.h File Reference

Detailed Description

This file contains the interface for the XIo component, which encapsulates the Input/Output functions for the PowerPC architecture.

Note:

This file contains architecture-dependent items (memory mapped or non memory mapped I/O).

```
#include "xbasic_types.h"
```

Typedefs

```
typedef Xuint32 XIo_Address
```

Functions

```
Xuint8 XIo_In8 (XIo\_Address InAddress)
```

```
Xuint16 XIo_In16 (XIo\_Address InAddress)
```

```
Xuint32 XIo_In32 (XIo\_Address InAddress)
```

```
void XIo_Out8 (XIo\_Address OutAddress, Xuint8 Value)
```

```
void XIo_Out16 (XIo\_Address OutAddress, Xuint16 Value)
```

```
void XIo_Out32 (XIo\_Address OutAddress, Xuint32 Value)
```

```
Xuint16 XIo_InSwap16 (XIo\_Address InAddress)
```

```
Xuint32 XIo_InSwap32 (XIo\_Address InAddress)
```

```
void XIo_OutSwap16 (XIo\_Address OutAddress, Xuint16 Value)
```

```
void XIo_OutSwap32 (XIo\_Address OutAddress, Xuint32 Value)
```

Typedef Documentation

typedef [Xuint32](#)

XIo_Address

Typedef for an I/O address. Typically correlates to the width of the address bus.

Function Documentation

[Xuint16](#) (**[XIo_Address](#)**)

XIo_In16 *InAddress*

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

Note:

None.

[Xuint32](#) (**[XIo_Address](#)**)

XIo_In32 *InAddress*

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

Note:

None.

Xuint8 (**XIo_Address**)

XIo_In8 *InAddress*

Performs an input operation for an 8-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

Note:

None.

Xuint16 (**XIo_Address**)

XIo_InSwap16 *InAddress*

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the byte-swapped value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The byte-swapped value read from the specified input address.

Note:

None.

Xuint32 (**XIo_Address**)

XIo_InSwap32 *InAddress*

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the byte-swapped value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The byte-swapped value read from the specified input address.

Note:

None.

```
void      ( XIo_Address  
XIo_Out16      OutAddress,  
           Xuint16      Value  
           )
```

Performs an output operation for a 16-bit memory location by writing the specified value to the the specified address.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void      ( XIo_Address  
XIo_Out32      OutAddress,  
           Xuint32      Value  
           )
```

Performs an output operation for a 32-bit memory location by writing the specified value to the the specified address.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void      ( XIo_Address  
XIo_Out8      OutAddress,  
             Xuint8      Value  
            )
```

Performs an output operation for an 8-bit memory location by writing the specified value to the the specified address.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void      ( XIo_Address  
XIo_OutSwap16      OutAddress,  
                  Xuint16      Value  
                  )
```

Performs an output operation for a 16-bit memory location by writing the specified value to the the specified address. The value is byte-swapped before being written.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void      ( XIo_Address
XIo_OutSwap32      OutAddress,
                Xuint32      Value
                )
```

Performs an output operation for a 32-bit memory location by writing the specified value to the the specified address. The value is byte-swapped before being written.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

XHdlc_ChannelID Struct Reference

```
#include <xhdlc_1.h>
```

Detailed Description

The [XHdlc](#) ChannelID structure.

The documentation for this struct was generated from the following file:

- [hdlc/v2_01_a/src/xhdlc_1.h](#)
-

XHdlc_RegisterOffsets Struct Reference

```
#include <xhdlc_1.h>
```

Detailed Description

The [XHdlc](#) register access structure. These are the individual channel control register offsets from the base address. Initialized in the `_InitializeRegisterAccess` routine for faster access.

The documentation for this struct was generated from the following file:

- `hdlc/v2_01_a/src/xhdlc_1.h`
-

XHdlc_Channel Struct Reference

```
#include <xhdlc.h>
```

Detailed Description

The [XHdlc](#) sub channel data.

Data Fields

[Xuint32 UserData](#)

[XHdlc_ChannelID ChannelID](#)

[Xuint32 RecvChannelNum](#)

[Xuint32 SendChannelNum](#)

Field Documentation

[XHdlc_ChannelID](#) **XHdlc_Channel::**

ChannelID

channel ID associated with this statistics /
block

[Xuint32](#) **XHdlc_Channel::**

RecvChannelNum

DMA receive channel
number

Xuint32 XHdlc_Channel::

SendChannelNum

DMA send channel
Number

Xuint32 XHdlc_Channel::

UserData

used for user specific
data

The documentation for this struct was generated from the following file:

- [hdlc/v2_01_a/src/xhdlc.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

common/v1_00_a/src/xparameters.h File Reference

Detailed Description

This file contains system parameters for the Xilinx device driver environment. It is a representation of the system in that it contains the number of each device in the system as well as the parameters and memory map for each device. The user can view this file to obtain a summary of the devices in their system and the device parameters.

This file may be automatically generated by a design tool such as System Generator.

XHdlc_Stats Struct Reference

```
#include <xhdlc.h>
```

Detailed Description

HDLC statistics (see [XHdlc_GetStats\(\)](#) and [XHdlc_ClearStats\(\)](#))

Data Fields

[Xuint16 XmitFrames](#)

[Xuint16 XmitBytes](#)

[Xuint16 XmitUnderrunErrors](#)

[Xuint16 RecvFrames](#)

[Xuint16 RecvBytes](#)

[Xuint16 RecvFcsErrors](#)

[Xuint16 RecvAlignmentErrors](#)

[Xuint16 RecvOverrunErrors](#)

[Xuint16 RecvFramesAborted](#)

[Xuint16 RecvCounterRolls](#)

[Xuint16 FifoErrors](#)

[Xuint16 DmaErrors](#)

[Xuint16 RecvInterrupts](#)

[Xuint16 XmitInterrupts](#)

[Xuint16 HdlcInterrupts](#)

Field Documentation

Xuint16 XHdlc_Stats::

DmaErrors

Number of DMA errors

Xuint16 XHdlc_Stats::

FifoErrors

Number of FIFO errors since init

Xuint16 XHdlc_Stats::

HdlcInterrupts

Number of HDLC (device) interrupts

Xuint16 XHdlc_Stats::

RecvAlignmentErrors

Number of frames received with alignment errors

Xuint16 XHdlc_Stats::

RecvBytes

Number of bytes received

Xuint16 XHdlc_Stats::

RecvCounterRolls

Number of times the receive counters have rolled over

Xuint16 XHdlc_Stats::

RecvFcsErrors

Number of frames discarded due to FCS errors

Xuint16 XHdlc_Stats::

RecvFrames

Number of frames
received

Xuint16 XHdlc_Stats::

RecvFramesAborted

Number of Receive frames
aborted

Xuint16 XHdlc_Stats::

RecvInterrupts

Number of Receive
interrupts

Xuint16 XHdlc_Stats::

RecvOverrunErrors

Number of Receive overrun
errors

Xuint16 XHdlc_Stats::

XmitBytes

Number of bytes
transmitted

Xuint16 XHdlc_Stats::

XmitFrames

Number of frames
transmitted

Xuint16 XHdlc_Stats::

XmitInterrupts

Number of Transmit
interrupts

Xuint16 XHdlc_Stats::

XmitUnderrunErrors

Number of Transmit underrun errors

The documentation for this struct was generated from the following file:

- [hdlc/v2_01_a/src/xhdlc.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

hdlc/v2_01_a/src/xhdlc_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between **XHdlc** components. The identifiers in this file are not intended for use external to the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
2.00a	ecm	09/16/03	First release
2.01a	ecm	02/16/04	Add multichannel scatter-gather DMA support

```
#include "xhdlc_1.h"  
#include "xhdlc.h"
```

Defines

```
#define XHdlc_mRxFifoChannelID(Event, ChannelID)  
#define XHdlc_mTxFifoChannelID(Event, ChannelID)  
#define XHdlc_mLogicalToTxSgdmaChannel(Channel)  
#define XHdlc_mLogicalToRxSgdmaChannel(Channel)
```

Functions

```
void XHdlc_LogicalToPhysicalSub (XHdlc *InstancePtr, Xuint32 LogicalChannel, XHdlc_ChannelID *ChannelID)  
void XHdlc_PhysicalSubToLogical (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, Xuint32 *LogicalChannel)  
Xboolean XHdlc_IsPhysicalSubEnabled (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)
```

Variables

```
XHdlc_Config XHdlc_ConfigTable []
```

Define Documentation

#define XHdlc_mLogicalToRxSgdmaChannel(Channel)

Convert logical channel to RX SGDMA channel equivalent.

Parameters:

Channel is the logical Channel to convert to RX SGDMA channel

Returns:

SGDMA equivalent channel.

Note:

The custom is that the RX channel is the odd channel number of the pair

#define XHdlc_mLogicalToTxSgdmaChannel(Channel)

Convert logical channel to TX SGDMA channel equivalent.

Parameters:

Channel is the logical Channel to convert to TX SGDMA channel

Returns:

SGDMA equivalent channel.

Note:

The custom is that the TX channel is the even channel number of the pair

#define XHdlc_mRxFifoChannelID(Event, ChannelID)

Extract the ChannelID from the RX Event FIFO data.

Parameters:

Status is the data from the RX Event
FIFO

ChannelID is ChannelID contained in the Data

Returns:

None.

Note:

XHdlc_Initialize must be run prior to running this function

```
#define XHdlc_mTxFifoChannelID( Event,  
                               ChannelID )
```

Extract the ChannelID from the TX Event FIFO data.

Parameters:

Status is the data from the TX Event
FIFO
ChannelID is ChannelID contained in the Data

Returns:

None.

Note:

XHdlc_Initialize must be run prior to running this
function

Function Documentation

```
Xboolean ( XHdlc *  
XHdlc_IsPhysicalSubEnabled InstancePtr,  
XHdlc_ChannelID ChannelID  
)
```

Determine if the Physical channel/ Sub channel pair is enabled at the hardware level. If the hardware itself is disabled, by default from a functional perspective, the channel pair is also disabled regardless of the status of the bits in the respective registers.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.
ChannelID contains the PhysicalChannel/Subchannel pair to be checked.

Returns:

- o XTRUE if the physical channel, sub channel, and device are enabled.
- o XFALSE otherwise.

Note:

XHdlc_Initialize must be run prior to running this function.

```

void
XHdlc_LogicalToPhysicalSub ( XHdlc * InstancePtr,
                             Xuint32 LogicalChannel,
                             XHdlc_ChannelID * ChannelIDPtr
                             )

```

Convert the logical ID to a ChannelID which is a physical channel number/ subchannel number pair. This functions converts the provided logical number to the physical/sub channel pair by stepping through the PhysChanSubCount array in the instance which contains the available channels associated with a given physical channel.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.
LogicalChannel to be converted from.
ChannelIDPtr pointer for the resulting data pair to be written to.

Returns:

the ChannelID pair corresponding to the Logical ID provided in ChannelID.

Note:

XHdlc_Initialize must be run prior to running this function.

```

void
XHdlc_PhysicalSubToLogical ( XHdlc * InstancePtr,
                             XHdlc_ChannelID ChannelID,
                             Xuint32 * LogicalChannelPtr
                             )

```

Convert the ChannelID which is a physical channel number/ subchannel number pair into an incrementing Logical Channel Number.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.
ChannelID data pair to be converted from.
LogicalChannelPtr pointer for the resulting channel to be written to.

Returns:

the Logical ID corresponding to the ChannelID pair provided.

Note:

XHdlc_Initialize must be run prior to running this function.

Variable Documentation

[XHdlc_Config](#) XHdlc_ConfigTable ()

[]

This table contains configuration information for each HDLC device in the system.

Generated on 24 Jun 2004 for Xilinx Device Drivers

emacLite/v1_01_a/src/xemacLite_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between the files of the driver. It is intended for internal use only.

NOTES:

None.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.01a	ecm	05/21/04	First release

```
#include "xemacLite.h"
```

Defines

```
#define XEmacLite\_mGetTxActive(BaseAddress)  
#define XEmacLite\_mSetTxActive(BaseAddress, Mask)
```

Functions

```
void XEmacLite\_AlignedWrite (void *SrcPtr, Xuint32 *DestPtr, unsigned  
ByteCount)
```


Set the TX active location to update status. This is used to set the bit indicating which TX buffer is currently active. There isn't any way in the hardware to implement this but the register is fully populated so the driver can set the bit in the send routine and the ISR can clear the bit when the handler is complete. This mimics the correct operation of the hardware if it was possible to do this in hardware.

Parameters:

BaseAddress is the base address of the device
Mask is the data to be written

Returns:

None

Note:

void [XEmacLite_mSetTxActive](#)(Xuint32 BaseAddress, Xuint32 Mask)

Function Documentation

```
void StubHandler ( void * CallbackRef )
```

A stub for the asynchronous callback. The stub is here in case the upper layers forget to set the handler.

Parameters:

CallbackRef is a pointer to the upper layer callback reference

Returns:

None.

Note:

None.

```
void XEmacLite_AlignedRead ( Xuint32 * SrcPtr,  
void * DestPtr,  
unsigned ByteCount  
)
```

This function reads from a 32-bit aligned source address range and aligns the writes to the provided destination pointer alignment.

Parameters:

SrcPtr is a pointer to incoming data of 32-bit alignment.
DestPtr is a pointer to outgoing data of any alignment.
ByteCount is the number of bytes to read.

Returns:

None.

Note:

None.

```
void XEmacLite_AlignedWrite ( void * SrcPtr,  
Xuint32 * DestPtr,  
unsigned ByteCount  
)
```

This function aligns the incoming data and writes it out to a 32-bit aligned destination address range.

Parameters:

SrcPtr is a pointer to incoming data of any alignment.
DestPtr is a pointer to outgoing data of 32-bit alignment.
ByteCount is the number of bytes to write.

Returns:

None.

Note:

None.

Variable Documentation

XEmacLite_Config XEmacLite_ConfigTable ()

[]

This table contains configuration information for each EMACLite device in the system.

Generated on 24 Jun 2004 for Xilinx Device Drivers

emac_lite/v1_01_a/src/xemac_lite.h File Reference

Detailed Description

The Xilinx Ethernet Lite driver component. This component supports the Xilinx Lite Ethernet 10/100 MAC (EMAC Lite).

The Xilinx Ethernet Lite 10/100 MAC supports the following features:

- Media Independent Interface (MII) for connection to external 10/100 Mbps PHY transceivers.
- Independent internal transmit and receive buffers
- CSMA/CD compliant operations for half-duplex modes
- Unicast and broadcast
- Automatic FCS insertion
- Automatic pad insertion on transmit
- Configurable ping/pong buffer scheme for either/both transmit and receive buffer areas.
- Interrupt driven mode available.

The Xilinx Ethernet Lite 10/100 MAC does not support the following features:

- multi-frame buffering only 1 transmit frame is allowed into each transmit buffer only 1 receive frame is allowed into each receive buffer. the hardware blocks reception until buffer is emptied
- Pause frame (flow control) detection in full-duplex mode
- Programmable interframe gap
- Multicast and promiscuous address filtering
- Internal loopback
- Automatic source address insertion or overwrite

Driver Description

The device driver enables higher layer software (e.g., an application) to communicate to the EMAC Lite. The driver handles transmission and reception of Ethernet frames, as well as configuration of the controller. It does not handle protocol stack functionality such as Link Layer Control (LLC) or the Address Resolution Protocol (ARP). The protocol stack that makes use of the driver handles this

functionality. This implies that the driver is simply a pass-through mechanism between a protocol stack and the EMAC Lite.

Since the driver is a simple pass-through mechanism between a protocol stack and the EMAC Lite, no assembly or disassembly of Ethernet frames is done at the driver-level. This assumes that the protocol stack passes a correctly formatted Ethernet frame to the driver for transmission, and that the driver does not validate the contents of an incoming frame. A single device driver can support multiple EmacLite devices.

The driver supports interrupt driven mode and the default mode of operation is polled mode. If interrupts are desired, XEmacLite_InterruptEnable() must be called.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are stored in the [xemaclite_g.c](#) file. A table is defined where each entry contains configuration information for an EmacLite device. This information includes such things as the base address of the memory-mapped device and the number of buffers.

Interrupt Processing

After _Initialize is called, _InterruptEnable can be called to enable the interrupt driven functionality. If polled operation is desired, just call _Send and check the return code. If XST_FAILURE is returned, call _Send with the same data until XST_SUCCESS is returned. The same idea applies to _Recv. Call _Recv until the returned length is non-zero at which point the received data is in the buffer provided in the function call.

The Transmit and Receive interrupts are enabled within the _InterruptEnable function and disabled in the _InterruptDisable function. The _Send and _Recv functions acknowledge the EMACLite generated interrupts associated with each function. It is the application's responsibility to acknowledge any associated Interrupt Controller interrupts if it is used in the system.

Memory Buffer Alignment

The alignment of the input/output buffers for the _Send and _Recv routine is not required to be 32 bits. If the buffer is not aligned on a 32-bit boundary there will be a performance impact while the driver aligns the data for transmission or upon reception.

For optimum performance, the user should provide a 32-bit aligned buffer to the _Send and _Recv routines.

Asserts

Asserts are used within all Xilinx drivers to enforce constraints on argument values. Asserts can be turned off on a system-wide basis by defining, at compile time, the NDEBUG identifier. By default, asserts are turned on and it is recommended that application developers leave asserts on during development.

Note:

This driver requires EmacLite hardware version 1.01a and higher. It is not compatible with earlier versions of the EmacLite hardware. Use version 1.00a software driver for hardware version 1.00a/b.

The RX hardware is enabled from powerup and there is no disable. It is possible that frames have been received prior to the initialization of the driver. If this situation is possible, call XEmacLite_mFlushReceive() to empty the receive buffers after initialization.

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.01a	ecm	01/30/04	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xemaclite_1.h"
```

Data Structures

```
struct XEmacLite  
struct XEmacLite_Config
```

Defines

```
#define XEmacLite_mNextTransmitAddr(InstancePtr)  
#define XEmacLite_mNextReceiveAddr(InstancePtr)
```

Functions

XStatus XEmaLite_Initialize (XEmaLite *InstancePtr, **Xuint16** DeviceId)

void **XEmaLite_SetMacAddress** (XEmaLite *InstancePtr, **Xuint8** *AddressPtr)

Xboolean XEmaLite_TxBufferAvailable (XEmaLite *InstancePtr)

void **XEmaLite_FlushReceive** (XEmaLite *InstancePtr)

XEmaLite_Config * **XEmaLite_LookupConfig** (**Xuint16** DeviceId)

XStatus XEmaLite_Send (XEmaLite *InstancePtr, **Xuint8** *FramePtr, unsigned ByteCount)

Xuint16 XEmaLite_Recv (XEmaLite *InstancePtr, **Xuint8** *FramePtr)

XStatus XEmaLite_EnableInterrupts (XEmaLite *InstancePtr)

void **XEmaLite_DisableInterrupts** (XEmaLite *InstancePtr)

void **XEmaLite_InterruptHandler** (void *InstancePtr)

void **XEmaLite_SetRecvHandler** (XEmaLite *InstancePtr, void *CallbackRef, XEmaLite_Handler FuncPtr)

void **XEmaLite_SetSendHandler** (XEmaLite *InstancePtr, void *CallbackRef, XEmaLite_Handler FuncPtr)

XStatus XEmaLite_SelfTest (XEmaLite *InstancePtr)

Define Documentation

```
#define XEmaLite_mNextReceiveAddr( InstancePtr )
```

Return the next expected Receive Buffer's address .

Parameters:

InstancePtr is the pointer to the instance of the driver to be worked on

Note:

This macro returns the address of the next receive buffer to read data from. This is the expected receive buffer address if the driver is in sync.

```
#define XEmaLite_mNextTransmitAddr( InstancePtr )
```

Return the next expected Transmit Buffer's address .

Parameters:

InstancePtr is the pointer to the instance of the driver to be worked on

Note:

This macro returns the address of the next transmit buffer to put data into. This is used to determine the destination of the next transmit data frame.

Function Documentation

```
void ( XEmacLite * )  
XEmacLite_DisableInterrupts InstancePtr
```

Disables the interrupts from the device(the higher layer software is responsible for disabling interrupts at the interrupt controller).

To start using the device again, `_EnableInterrupts` must be called.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( XEmacLite * )  
XEmacLite_EnableInterrupts InstancePtr
```

Enable the EmaLite Interrupts.

This function must be called before other functions to send or receive data in interrupt driven mode. The user should have connected the interrupt handler of the driver to an interrupt source such as an interrupt controller or the processor interrupt prior to this function being called.

Parameters:

InstancePtr is a pointer to the **XEmaLite** instance to be worked on.

Returns:

- XST_SUCCESS if the device interrupts were enabled successfully.
- XST_NO_CALLBACK if the callbacks were not set.

Note:

None

```
void ( XEmaLite * )  
XEmaLite_FlushReceive InstancePtr
```

Flush the Receive buffers. All data will be lost.

Parameters:

InstancePtr is the pointer to the instance of the driver to be worked on

Returns:

None.

Note:

```
XStatus ( XEmaLite *  
XEmaLite_Initialize InstancePtr,  
Xuint16 DeviceId  
)
```

Initialize a specific **XEmacLite** instance/driver. The initialization entails:

- Initialize fields of the **XEmacLite** instance structure.

The driver defaults to polled mode operation.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XEmacLite** instance. Passing in a device id associates the generic **XEmacLite** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful.
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None

```
void XEmacLite_InterruptHandler ( void * InstancePtr )
```

Interrupt handler for the EMACLite driver. It performs the following processing:

- Get the interrupt status from the registers to determine the source of the interrupt.
- Call the appropriate handler based on the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the EMACLite device instance for the interrupt.

Returns:

None.

Note:

XEmaLite_Config* (**Xuint16**)
XEmaLite_LookupConfig *DeviceId*

Lookup the device configuration based on the unique device ID. The table XEmaLite_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

Xuint16 (**XEmaLite ***
XEmaLite_Recv *InstancePtr,*
Xuint8 * *FramePtr*
)

Receive a frame. Intended to be called from the interrupt context or with a wrapper which waits for the receive frame to be available.

Parameters:

InstancePtr is a pointer to the **XEmaLite** instance to be worked on.

FramePtr is a pointer to a buffer where the frame will be stored. The buffer must be at least XEL_MAX_FRAME_SIZE bytes. For optimal performance, a 32-bit aligned buffer should be used but it is not required, the function will align the data if necessary.

Returns:

The type/length field of the frame received. When the type/length field contains the type, XEL_MAX_FRAME_SIZE bytes will be copied out of the buffer and it is up to the higher layers to sort out the frame. Function returns 0 if there is no data waiting in the receive buffer or the pong buffer if configured.

Note:

This function call is not blocking in nature, i.e. it will not wait until a frame arrives.

```
XStatus ( XEmacLite * )  
XEmacLite_SelfTest InstancePtr
```

Performs a SelfTest on the EmacLite device as follows:

- Writes to the mandatory TX buffer and reads back to verify.
- If configured, writes to the secondary TX buffer and reads back to verify.
- Writes to the mandatory RX buffer and reads back to verify.
- If configured, writes to the secondary RX buffer and reads back to verify.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

Returns:

- XST_SUCCESS if the device Passed the Self Test.
- XST_FAILURE if any of the data read backs fail.

Note:

None.

```
XStatus ( XEmacLite *  
XEmacLite_Send InstancePtr,  
 Xuint8 * FramePtr,  
 unsigned ByteCount  
 )
```

Send an Ethernet frame. The ByteCount is the total frame size, including header.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

FramePtr is a pointer to frame. For optimal performance, a 32-bit aligned buffer should be used but it is not required, the function will align the data if necessary.

ByteCount is the size, in bytes, of the frame

Returns:

- XST_SUCCESS if data was transmitted.
- XST_FAILURE if buffer(s) was (were) full and no valid data was transmitted.

Note:

This function call is not blocking in nature, i.e. it will not wait until the frame is transmitted.

```

void
XEmacLite_SetMacAddress ( XEmacLite *
                          InstancePtr,
                          Xuint8 *
                          AddressPtr
                          )

```

Set the MAC address for this device. The address is a 48-bit value.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

AddressPtr is a pointer to a 6-byte MAC address. the format of the MAC address is major octet to minor octet

Returns:

None.

Note:

TX must be idle and RX should be idle for deterministic results.

Function will not return if hardware is absent or not functioning properly.

```

void
XEmacLite_SetRecvHandler ( XEmacLite *
                           InstancePtr,
                           void *
                           CallbackRef,
                           XEmacLite_Handler
                           FuncPtr
                           )

```

Sets the callback function for handling received frames in interrupt mode. The upper layer software should call this function during initialization. The callback is called when a frame is received. The callback function should communicate the data to a thread such that the processing is not performed in an interrupt context.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void XEmacLite_SetSendHandler ( XEmacLite * InstancePtr,  
                               void * CallbackRef,  
                               XEmacLite_Handler FuncPtr  
                               )
```

Sets the callback function for handling transmitted frames in interrupt mode. The upper layer software should call this function during initialization. The callback is called when a frame is transmitted. The callback function should communicate the data to a thread such that the processing is not performed in an interrupt context.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
Xboolean XEmacLite_TxBufferAvailable ( XEmacLite * InstancePtr )
```

Determine if there is a transmit buffer available.

Parameters:

InstancePtr is the pointer to the instance of the driver to be worked on

Returns:

XTRUE if there is a TX buffer available for data to be written into, XFALSE otherwise.

Note:

Generated on 24 Jun 2004 for Xilinx Device Drivers

emacLite/v1_01_a/src/xemacLite_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of EMACLite devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.01a	ecm	02/16/04	First release

```
#include "xparameters.h"  
#include "xemacLite.h"
```

Variables

[XEmacLite_Config](#) [XEmacLite_ConfigTable](#) [XPAR_XEMACLITE_NUM_INSTANCES]

Variable Documentation

[XEmacLite_Config](#) [XEmacLite_ConfigTable](#)
[XPAR_XEMACLITE_NUM_INSTANCES]

This table contains configuration information for each EMACLite device in the system.

XEmacLite_Config Struct Reference

```
#include <xemaclite.h>
```

Detailed Description

This typedef contains configuration information for a device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xuint8 TxPingPong](#)

[Xuint8 RxPingPong](#)

Field Documentation

[Xuint32 XEmacLite_Config::](#)

BaseAddress

Device base
address

[Xuint16 XEmacLite_Config::](#)

DeviceId

Unique ID of
device

Xuint8 XEmaclite_Config::

RxPingPong

1 if RX Pong buffer configured,0
otherwise

Xuint8 XEmaclite_Config::

TxPingPong

1 if TX Pong buffer configured,0
otherwise

The documentation for this struct was generated from the following file:

- [emacLite/v1_01_a/src/xemaclite.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

emac_lite/v1_01_a/src/xemac_lite_i.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions and macros that can be used to access the device.

The Xilinx Ethernet Lite driver component. This component supports the Xilinx Lite Ethernet 10/100 MAC (EMAC Lite).

Refer to [xemac_lite.h](#) for more details.

Note:

The functions and macros in this file assume that the proper device address is provided in the argument. If the ping buffer is the source or destination, the argument should be DeviceAddress + XEL_(T/R)XBUFF_OFFSET. If the pong buffer is the source or destination, the argument should be DeviceAddress + XEL_(T/R)XBUFF_OFFSET + XEL_BUFFER_OFFSET. The driver does not take the different buffers into consideration. For more details on the ping/pong buffer configuration please refer to the OPB Ethernet Lite Media Access Controller hardware specification.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	06/01/02	First release
1.01a	ecm	03/31/04	Additional functionality and the <code>_AlignedRead</code> and <code>_AlignedWrite</code> functions. Moved the bulk of description to xemac_lite.h

```
#include "xbasic_types.h"
#include "xio.h"
```

Defines

```
#define XEL_TXBUFF_OFFSET
#define XEL_GIER_OFFSET
#define XEL_TSR_OFFSET
#define XEL_TPLR_OFFSET
#define XEL_RXBUFF_OFFSET
#define XEL_RSR_OFFSET
#define XEL_RPLR_OFFSET
#define XEL_MAC_HI_OFFSET
#define XEL_MAC_LO_OFFSET
#define XEL_BUFFER_OFFSET
#define XEL_GIER_GIE_MASK
#define XEL_TSR_XMIT_BUSY_MASK
#define XEL_TSR_PROGRAM_MASK
#define XEL_TSR_XMIT_IE_MASK
#define XEL_TSR_XMIT_ACTIVE_MASK
#define XEL_TSR_PROG_MAC_ADDR
#define XEL_RSR_RECV_DONE_MASK
#define XEL_RSR_RECV_IE_MASK
#define XEL_TPLR_LENGTH_MASK_HI
#define XEL_TPLR_LENGTH_MASK_LO
#define XEL_RPLR_LENGTH_MASK_HI
#define XEL_RPLR_LENGTH_MASK_LO
#define XEL_HEADER_SIZE
#define XEL_MTU_SIZE
#define XEL_FCS_SIZE
#define XEL_HEADER_OFFSET
#define XEL_HEADER_SHIFT
#define XEL_MAX_FRAME_SIZE
#define XEL_MAC_ADDR_SIZE
#define XEmacLite_mIsTxDone(BaseAddress)
#define XEmacLite_mIsRxEmpty(BaseAddress)
```

Functions

void **XEmaLite_SendFrame** (**Xuint32** BaseAddress, **Xuint8** *FramePtr, unsigned ByteCount)
Xuint16 XEmaLite_RecvFrame (**Xuint32** BaseAddress, **Xuint8** *FramePtr)

Define Documentation

#define

XEL_BUFFER_OFFSET

Next buffer's offset same for both TX and RX

#define

XEL_FCS_SIZE

Size of CRC

#define

XEL_GIER_GIE_MASK

Global Enable

#define

XEL_GIER_OFFSET

Offset for the GIE bit

#define

XEL_HEADER_OFFSET

Offset to length field

#define

XEL_HEADER_SHIFT

Right shift value to align
length

```
#define  
XEL_HEADER_SIZE
```

Size of header in
bytes

```
#define  
XEL_MAC_ADDR_SIZE
```

length of MAC
address

```
#define  
XEL_MAC_HI_OFFSET
```

MAC address hi
offset

```
#define  
XEL_MAC_LO_OFFSET
```

MAC address lo
offset

```
#define  
XEL_MAX_FRAME_SIZE
```

Maximum length of rx frame used if length/type field contains the type (>
1500)

```
#define  
XEL_MTU_SIZE
```

Max size of data in
frame

```
#define  
XEL_RPLR_LENGTH_MASK_HI
```

Receive packet length upper
byte

```
#define  
XEL_RPLR_LENGTH_MASK_LO
```

Receive packet length lower
byte

```
#define  
XEL_RPLR_OFFSET
```

Rx packet
length

```
#define  
XEL_RSR_OFFSET
```

Rx
status

```
#define  
XEL_RSR_RECV_DONE_MASK
```

Recv
complete

```
#define  
XEL_RSR_RECV_IE_MASK
```

Recv interrupt enable
bit

```
#define  
XEL_RXBUFF_OFFSET
```

Receive
Buffer

```
#define  
XEL_TPLR_LENGTH_MASK_HI
```

Transmit packet length upper
byte

#define

XEL_TPLR_LENGTH_MASK_LO

Transmit packet length lower
byte

#define

XEL_TPLR_OFFSET

Tx packet
length

#define

XEL_TSR_OFFSET

Tx
status

#define

XEL_TSR_PROG_MAC_ADDR

define for programming the MAC address into the EMAC
Lite

#define

XEL_TSR_PROGRAM_MASK

Program the MAC
address

#define

XEL_TSR_XMIT_ACTIVE_MASK

Buffer is active, SW bit
only

#define

XEL_TSR_XMIT_BUSY_MASK

Xmit
complete

#define

XEL_TSR_XMIT_IE_MASK

Xmit interrupt enable
bit

```
#define  
XEL_TXBUFF_OFFSET
```

Transmit
Buffer

```
#define XEmacLite_mIsRxEmpty( BaseAddress )
```

Check to see if the receive is empty.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is empty, or XFALSE if it is not.

Note:

Xboolean [XEmacLite_mIsRxEmpty](#)(Xuint32
BaseAddress)

```
#define XEmacLite_mIsTxDone( BaseAddress )
```

Check to see if the transmission is complete.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is done, or XFALSE if it is not.

Note:

Xboolean [XEmacLite_mIsTxDone](#)(Xuint32
BaseAddress)

```
Xuint16 ( Xuint32  
XEmacLite_RecvFrame BaseAddress,  
 Xuint8 * FramePtr  
 )
```

Receive a frame. Wait for a frame to arrive.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to a buffer where the frame will be stored.

Returns:

The type/length field of the frame received. When the type/length field contains the type , XEL_MAX_FRAME_SIZE bytes will be copied out of the buffer and it is up to the higher layers to sort out the frame.

Note:

This function call is blocking in nature, i.e. it will wait until a frame arrives.

If the ping buffer is the source of the data, the argument should be DeviceAddress + XEL_RXBUFF_OFFSET. If the pong buffer is the source of the data, the argument should be DeviceAddress + XEL_RXBUFF_OFFSET + XEL_BUFFER_OFFSET. The function does not take the different buffers into consideration.

```
void ( Xuint32  
XEmacLite_SendFrame BaseAddress,  
 Xuint8 * FramePtr,  
 unsigned ByteCount  
 )
```

Send an Ethernet frame. The size is the total frame size, including header. This function blocks waiting for the frame to be transmitted.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to frame

ByteCount is the size, in bytes, of the frame

Returns:

None.

Note:

This function call is blocking in nature, i.e. it will wait until the frame is transmitted. This function can hang and not exit if the hardware is not configured properly.

If the ping buffer is the destination of the data, the argument should be DeviceAddress + XEL_TXBUFF_OFFSET. If the pong buffer is the destination of the data, the argument should be DeviceAddress + XEL_TXBUFF_OFFSET + XEL_BUFFER_OFFSET. The function does not take the different buffers into consideration.

Generated on 24 Jun 2004 for Xilinx Device Drivers

XEmacLite Struct Reference

```
#include <xemaclite.h>
```

Detailed Description

The XEmacLite driver instance data. The user is required to allocate a variable of this type for every EmacLite device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [emacLite/v1_01_a/src/xemaclite.h](#)
-

emacLite/v1_01_a/src/xemacLite.c File Reference

Detailed Description

Functions in this file are the minimum required functions for the EMAC Lite driver. See [xemacLite.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.01a	ecm	01/31/04	First release

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xio.h"  
#include "xenv.h"  
#include "xemacLite.h"  
#include "xemacLite_1.h"  
#include "xemacLite_i.h"
```

Functions

XStatus [XEmacLite_Initialize](#) ([XEmacLite](#) *InstancePtr, **Xuint16** DeviceId)

XStatus [XEmacLite_Send](#) ([XEmacLite](#) *InstancePtr, **Xuint8** *FramePtr, unsigned
ByteCount)

Xuint16 [XEmacLite_Recv](#) ([XEmacLite](#) *InstancePtr, **Xuint8** *FramePtr)

void [XEmacLite_SetMacAddress](#) ([XEmacLite](#) *InstancePtr, **Xuint8** *AddressPtr)

[XEmacLite_Config](#) * [XEmacLite_LookupConfig](#) (**Xuint16** DeviceId)

void **StubHandler** (void *CallbackRef)
Xboolean XEmacLite_TxBufferAvailable (**XEmacLite** *InstancePtr)
void **XEmacLite_FlushReceive** (**XEmacLite** *InstancePtr)

Function Documentation

void (**void ***)
StubHandler *CallbackRef*

This is a stub for the send and recv callbacks. The stub is here in case the upper layers forget to set the handlers.

Parameters:

CallbackRef is a pointer to the upper layer callback reference

Returns:

None.

Note:

None.

void (**XEmacLite ***)
XEmacLite_FlushReceive *InstancePtr*

Flush the Receive buffers. All data will be lost.

Parameters:

InstancePtr is the pointer to the instance of the driver to be worked on

Returns:

None.

Note:

XStatus (**XEmacLite ***
XEmacLite_Initialize *InstancePtr,*
Xuint16 *DeviceId*
)

Initialize a specific **XEmacLite** instance/driver. The initialization entails:

- Initialize fields of the **XEmacLite** instance structure.

The driver defaults to polled mode operation.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XEmacLite** instance. Passing in a device id associates the generic **XEmacLite** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful.
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None

XEmacLite_Config*	(Xuint16)
XEmacLite_LookupConfig	<i>DeviceId</i>

Lookup the device configuration based on the unique device ID. The table XEmacLite_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

```

Xuint16                ( XEmaLite *
XEmaLite_Recv          InstancePtr,
                          Xuint8 *   FramePtr
                          )

```

Receive a frame. Intended to be called from the interrupt context or with a wrapper which waits for the receive frame to be available.

Parameters:

InstancePtr is a pointer to the **XEmaLite** instance to be worked on.

FramePtr is a pointer to a buffer where the frame will be stored. The buffer must be at least XEL_MAX_FRAME_SIZE bytes. For optimal performance, a 32-bit aligned buffer should be used but it is not required, the function will align the data if necessary.

Returns:

The type/length field of the frame received. When the type/length field contains the type, XEL_MAX_FRAME_SIZE bytes will be copied out of the buffer and it is up to the higher layers to sort out the frame. Function returns 0 if there is no data waiting in the receive buffer or the pong buffer if configured.

Note:

This function call is not blocking in nature, i.e. it will not wait until a frame arrives.

```

XStatus                ( XEmaLite *
XEmaLite_Send          InstancePtr,
                          Xuint8 *   FramePtr,
                          unsigned   ByteCount
                          )

```

Send an Ethernet frame. The ByteCount is the total frame size, including header.

Parameters:

InstancePtr is a pointer to the **XEmaLite** instance to be worked on.

FramePtr is a pointer to frame. For optimal performance, a 32-bit aligned buffer should be used but it is not required, the function will align the data if necessary.

ByteCount is the size, in bytes, of the frame

Returns:

- XST_SUCCESS if data was transmitted.
- XST_FAILURE if buffer(s) was (were) full and no valid data was transmitted.

Note:

This function call is not blocking in nature, i.e. it will not wait until the frame is transmitted.

```
void  
XEmacLite_SetMacAddress ( XEmacLite *  
                          InstancePtr,  
                          Xuint8 *   AddressPtr  
                          )
```

Set the MAC address for this device. The address is a 48-bit value.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

AddressPtr is a pointer to a 6-byte MAC address. the format of the MAC address is major octet to minor octet

Returns:

None.

Note:

TX must be idle and RX should be idle for deterministic results.

Function will not return if hardware is absent or not functioning properly.

```
Xboolean  
XEmacLite_TxBufferAvailable ( XEmacLite *  
                              InstancePtr
```

Determine if there is a transmit buffer available.

Parameters:

InstancePtr is the pointer to the instance of the driver to be worked on

Returns:

XTRUE if there is a TX buffer available for data to be written into, XFALSE otherwise.

Note:

common/v1_00_a/src/xenv.h File Reference

Detailed Description

Defines common services that are typically found in a host operating environment. This include file simply includes an OS specific file based on the compile-time constant BUILD_ENV_*, where * is the name of the target environment.

All services are defined as macros.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	ch	10/24/02	Added XENV_LINUX
1.00a	rmm	04/17/02	First release

```
#include "xenv\_none.h"
```

common/v1_00_a/src/xenv_none.h File Reference

Detailed Description

Defines common services specified by [xenv.h](#). Some of these services are defined as not performing any action. The implementation of these services are left to the user.

Note:

This file is not intended to be included directly by driver code. Instead, the generic [xenv.h](#) file is intended to be included by driver code.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	03/21/02	First release

Defines

```
#define XENV\_MEM\_COPY(DestPtr, SrcPtr, Bytes)
#define XENV\_MEM\_FILL(DestPtr, Data, Bytes)
#define XENV\_TIME\_STAMP\_GET(StampPtr)
#define XENV\_TIME\_STAMP\_DELTA\_US(Stamp1Ptr, Stamp2Ptr)
#define XENV\_TIME\_STAMP\_DELTA\_MS(Stamp1Ptr, Stamp2Ptr)
#define XENV\_USLEEP(delay)
```

Typedefs

typedef int [XENV_TIME_STAMP](#)

Define Documentation

```
#define XENV_MEM_COPY( DestPtr,  
                      SrcPtr,  
                      Bytes  )
```

Copies a non-overlapping block of memory.

Parameters:

DestPtr is the destination address to copy data to.

SrcPtr is the source address to copy data from.

Bytes is the number of bytes to copy.

Returns:

None.

Note:

Signature: void [XENV_MEM_COPY](#)(void *DestPtr, void *SrcPtr, unsigned Bytes)

```
#define XENV_MEM_FILL( DestPtr,  
                     Data,  
                     Bytes  )
```

Fills an area of memory with constant data.

Parameters:

DestPtr is the destination address to set.

Data contains the value to set.

Bytes is the number of bytes to set.

Returns:

None.

Note:

Signature: void [XENV_MEM_FILL](#)(void *DestPtr, char Data, unsigned

Bytes)

```
#define XENV_TIME_STAMP_DELTA_MS( Stamp1Ptr,  
                                Stamp2Ptr )
```

This macro is not yet implemented and always returns 0.

Parameters:

Stamp1Ptr is the first sampled time stamp.

Stamp2Ptr is the second sampled time stamp.

Returns:

0

Note:

This macro must be implemented by the user

```
#define XENV_TIME_STAMP_DELTA_US( Stamp1Ptr,  
                                Stamp2Ptr )
```

This macro is not yet implemented and always returns 0.

Parameters:

Stamp1Ptr is the first sampled time stamp.

Stamp2Ptr is the second sampled time stamp.

Returns:

0

Note:

This macro must be implemented by the user

```
#define XENV_TIME_STAMP_GET( StampPtr )
```

Time is derived from the 64 bit PPC timebase register

Parameters:

StampPtr is the storage for the retrieved time stamp.

Returns:

None.

Note:

Signature: void **XENV_TIME_STAMP_GET**(XTIME_STAMP *StampPtr)

Note:

This macro must be implemented by the user

```
#define XENV_USLEEP( delay )
```

```
XENV_USLEEP(unsigned delay)
```

Delay the specified number of microseconds. Not implemented without OS support.

Parameters:

delay is the number of microseconds to delay.

Returns:

None

Typedef Documentation

```
typedef int
```

```
XENV_TIME_STAMP
```

A structure that contains a time stamp used by other time stamp macros defined below. This structure is processor dependent.

dsdac/v1_00_a/src/xdsdac_selftest.c File Reference

Detailed Description

The function in this file is needed to run the selftest on the hardware. See [xdsdac.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	03/30/04	First release

```
#include "xstatus.h"  
#include "xio.h"  
#include "xdsdac.h"
```

Defines

```
#define TEST_VALUE
```

Define Documentation

```
#define  
TEST_VALUE
```

Performs a SelfTest on the DsDac device as follows: Writes a known value to the Occupancy Register then reads it back to confirm that the hardware is there. The hardware is reset to powerup state at completion of test.

Parameters:

InstancePtr is a pointer to the **XDsDac** instance to be worked on.

Returns:

- XST_SUCCESS if the device Passed the Self Test.
- XST_FAILURE if the register read back fails.

Note:

Resets the DAC. Configuration will have to be run again.

dsdac/v1_00_a/src/xdsdac.h File Reference

Detailed Description

The Xilinx Delta-Sigma Digital-to-Analog-Converter (DAC) driver. This driver supports the Xilinx Delta-Sigma DAC (D-S DAC) hardware.

Driver Description

The device driver enables higher layer software (e.g., an application) to communicate to the D-S DAC .

The driver supports interrupt driven mode and the default mode of operation is polled mode.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are stored in the [xdsdac_g.c](#) file. A table is defined where each entry contains configuration information for a DsDac device. This information includes such things as the base address of the memory-mapped device.

Interrupt Processing

After [XDSDac_Initialize\(\)](#) and [XDSDac_Start\(\)](#) are called, [XDSDac_EnableInterrupts\(\)](#) is called to enable interrupts after initializing the Interrupt handling such as [XIntc_SetHandler\(\)](#) through the [XIntc](#) driver and enabling the interrupt. There is no callback handler provided by this driver. An example is provided in the examples directory on how to construct a user defined handler.

When processing is complete, call [XDSDac_DisableInterrupts\(\)](#) to disable the interrupts.

The Threshold and Empty interrupts are enabled within the [XDSDac_EnableInterrupts\(\)](#) function and disabled in the [XDSDac_DisableInterrupts\(\)](#) function. The [XDSDac_WriteOutput\(\)](#) function does not acknowledge the D-S DAC generated interrupts. The application must call [XDSDac_GetInterruptStatus\(\)](#) to determine which interrupts are pending and call

[XDsDac_ClearInterruptStatus\(\)](#) to acknowledge the appropriate interrupt sources.

It is the application's responsibility to acknowledge any associated Interrupt Controller interrupts if it is used in the system.

Buffer Management

There is no buffer management provided in the driver. It is up to the application to provide the buffer space and data necessary to keep the DAC from underrunning on the data to be converted. If there isn't any new data when the DAC needs it, the last sample is repeated until the device is reset or new data is provided.

Asserts

Asserts are used within all Xilinx drivers to enforce constraints on argument values. Asserts can be turned off on a system-wide basis by defining, at compile time, the NDEBUG identifier. By default, asserts are turned on and it is recommended that application developers leave asserts on during development.

Note:

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	03/30/04	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xdsdac_1.h"
```

Data Structures

```
struct XDsDac  
struct XDsDac\_Config
```

Defines

```
#define XDsDac_mIsEmpty(InstancePtr)
#define XDsDac_mFifoIsFull(InstancePtr)
#define XDsDac_mGetVacancy(InstancePtr)
```

Functions

```
XStatus XDsDac_Initialize (XDsDac *InstancePtr, Xuint16 DeviceId)
```

```
void XDsDac_Start (XDsDac *InstancePtr)
```

```
void XDsDac_Stop (XDsDac *InstancePtr)
```

```
void XDsDac_Reset (XDsDac *InstancePtr)
```

```
XDsDac_Config * XDsDac_LookupConfig (Xuint16 DeviceId)
```

```
Xuint32 XDsDac_WriteOutput (XDsDac *InstancePtr, Xuint16 *DataPtr, unsigned  
SampleCount)
```

```
void XDsDac_EnableInterrupts (XDsDac *InstancePtr)
```

```
void XDsDac_DisableInterrupts (XDsDac *InstancePtr)
```

```
Xuint32 XDsDac_GetInterruptStatus (XDsDac *InstancePtr)
```

```
void XDsDac_ClearInterruptStatus (XDsDac *InstancePtr, Xuint32 Mask)
```

```
void XDsDac_SetInterruptDepth (XDsDac *InstancePtr, Xuint32 Depth)
```

Define Documentation

```
#define XDsDac_mFifoIsFull( InstancePtr )
```

Is the FIFO full. Can be used in polling also to keep the FIFO full.

Parameters:

InstancePtr is the instance of the device to operate on

Returns:

XFALSE if at least 1 sample can be written, or XTRUE if it not.

Note:

C-style signature Xboolean **XDsDac_mFifoIsFull**(XDsDac
*InstancePtr)

#define XDsDac_mGetVacancy(InstancePtr)

Get the space available in the FIFO in sample count.

Parameters:

InstancePtr is the instance of the device to operate on

Returns:

The current vacancy of the DS DAC FIFO.

Note:

C-style signature Xuint32 **XDsDac_mGetVacancy**(XDsDac *InstancePtr)

#define XDsDac_mIsEmpty(InstancePtr)

Check to see if the FIFO is empty. This macro is typically used when the polling method is employed.

Parameters:

InstancePtr is the instance of the device to operate on

Returns:

XTRUE if it is Empty, or XFALSE if it is not.

Note:

C-style signature Xboolean **XDsDac_mIsEmpty**(XDsDac *InstancePtr)

Function Documentation

```
void XDsDac_ClearInterruptStatus ( XDsDac * InstancePtr, Xuint32 Mask )
```

Clears the interrupt status in the register as directed by the bits set in the provided Mask

Parameters:

InstancePtr contains a pointer to the D-S DAC device instance for the interrupt.

Mask contains the interrupt sources to be cleared. Bits are defined in [xdsdac_l.h](#).

Returns:

None.

Note:

```
void ( XDsDac * )  
XDsDac_DisableInterrupts InstancePtr
```

Disables the interrupts from the device(the higher layer software is responsible for disabling interrupts at the interrupt controller).

To start using the device again in interrupt driven mode, [XDsDac_EnableInterrupts\(\)](#) must be called.

Parameters:

InstancePtr is a pointer to the [XDsDac](#) instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XDsDac * )  
XDsDac_EnableInterrupts InstancePtr
```

Enable the DsDac Interrupts.

This function must be called before other functions to transmit data in interrupt driven mode. The user should have connected the interrupt handler within the application to an interrupt source such as an interrupt controller or the processor interrupt prior to this function being called.

Parameters:

InstancePtr is a pointer to the **XDsDac** instance to be worked on.

Returns:

None.

Note:

None

```
Xuint32 ( XDsDac * )  
XDsDac_GetInterruptStatus InstancePtr
```

Get the interrupt status from the registers to determine the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the D-S DAC device instance for the interrupt.

Returns:

The currently asserted interrupts. Bits are defined in **xdsdac_1.h**.

None.

Note:

```
XStatus ( XDsDac *  
XDsDac_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize a specific **XDsDac** instance/driver. The initialization entails:

- Initialize fields of the **XDsDac** instance structure.
- Resets the hardware to ensure it is ready.

The driver defaults to polled mode operation.

Parameters:

InstancePtr is a pointer to the **XDsDac** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XDsDac** instance. Passing in a device id associates the generic **XDsDac** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful.
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None

XDsDac_Config*	(Xuint16)
XDsDac_LookupConfig	<i>DeviceId</i>

Lookup the device configuration based on the unique device ID. The table XDsDac_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

```
void ( XDsDac * )
XDsDac_Reset InstancePtr
```

Resets the Delta-Sigma DAC Hardware.

The Device must be configured again and **XDsDac_Start()** must be called prior to using the device again.

Parameters:

InstancePtr is a pointer to the **XDsDac** instance to be worked on.

Returns:

None

Note:

None

```
void ( XDsDac *
XDsDac_SetInterruptDepth InstancePtr,
Xuint32 Depth
)
```

Set the level of the FIFO which triggers the interrupt.

Parameters:

InstancePtr is the instance of the device to operate on

Depth is the intended interrupt trigger depth (0 through XDAC_MAX_DATA_LENGTH-1 samples)

Returns:

none.

Note:

```
void ( XDsDac *
XDsDac_Start InstancePtr
```

Starts the Delta-Sigma DAC Hardware.

The Device must be initialized and configured prior to **XDsDac_Start()** being called. Enables the DAC hardware to start producing analog outputs.

Parameters:

InstancePtr is a pointer to the **XDsDac** instance to be worked on.

Returns:

None

Note:

None

```
void ( XDsDac * )  
XDsDac_Stop InstancePtr
```

Stops the Delta-Sigma DAC Hardware. Prevents the DAC from producing analog outputs.

Parameters:

InstancePtr is a pointer to the **XDsDac** instance to be worked on.

Returns:

None

Note:

None

```
Xuint32 ( XDsDac * InstancePtr,  
XDsDac_WriteOutput DataPtr,  
Xuint16 * SampleCount  
unsigned SampleCount  
)
```

Transmit Delta-Sigma DAC data. The `SampleCount` is the total data size to be converted. If the amount of data to be sent will not fit in the FIFO, the return value indicates the amount of data sent to the DAC. This allows for this function to be used in a polling situation also.

Parameters:

InstancePtr is a pointer to the `XDsDac` instance to be worked on.

DataPtr is a pointer to data

SampleCount is the size, in samples, of the data

Returns:

The number of samples sent to the DAC

Note:

This function call is not blocking in nature, i.e. it will not wait until the data is sent to the hardware. Also, the interrupts are not disabled while the data is written to the DAC. If this is desired, the application must disable the interrupts prior to calling this function and then re-enable the interrupts when this function returns.

dsdac/v1_00_a/src/xdsdac_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of D-S DAC devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	03/31/04	First release

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

```
#include "xdsdac.h"
```

Variables

[XDsdac_Config](#) [XDsdac_ConfigTable](#) [XPAR_XDSDAC_NUM_INSTANCES]

Variable Documentation

[XDsdac_Config](#) [XDsdac_ConfigTable](#)
[XPAR_XDSDAC_NUM_INSTANCES]

This table contains configuration information for each D-S DAC device in the system.

XDsDac_Config Struct Reference

```
#include <xdsdac.h>
```

Detailed Description

This typedef contains configuration information for a device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

Field Documentation

[Xuint32 XDsDac_Config::](#)

BaseAddress

Device base
address

[Xuint16 XDsDac_Config::](#)

DeviceId

Unique ID of
device

The documentation for this struct was generated from the following file:

- dsdac/v1_00_a/src/[xdsdac.h](#)

dsdac/v1_00_a/src/xdsdac.c File Reference

Detailed Description

Functions in this file are the minimum required functions for the D-S DAC driver. See [xdsdac.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	03/30/04	First release

```
#include "xstatus.h"  
#include "xio.h"  
#include "xenv.h"  
#include "xparameters.h"  
#include "xdsdac.h"
```

Functions

XStatus **XDsDac_Initialize** (**XDsDac** *InstancePtr, **Xuint16** DeviceId)

void **XDsDac_Start** (**XDsDac** *InstancePtr)

void **XDsDac_Stop** (**XDsDac** *InstancePtr)

Xuint32 **XDsDac_WriteOutput** (**XDsDac** *InstancePtr, **Xuint16** *DataPtr, unsigned
SampleCount)

void **XDsDac_Reset** (**XDsDac** *InstancePtr)

XDsDac_Config * **XDsDac_LookupConfig** (**Xuint16** DeviceId)

Variables

[XDsdac_Config](#) [XDsdac_ConfigTable](#) []

Function Documentation

```
XStatus ( XDsdac *  
XDsdac_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize a specific **XDsdac** instance/driver. The initialization entails:

- Initialize fields of the **XDsdac** instance structure.
- Resets the hardware to ensure it is ready.

The driver defaults to polled mode operation.

Parameters:

InstancePtr is a pointer to the **XDsdac** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XDsdac** instance. Passing in a device id associates the generic **XDsdac** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful.
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None

```
XDsdac_Config* ( Xuint16 )  
XDsdac_LookupConfig DeviceId
```

Lookup the device configuration based on the unique device ID. The table XDsDac_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

```
void ( XDsDac * )  
XDsDac_Reset InstancePtr
```

Resets the Delta-Sigma DAC Hardware.

The Device must be configured again and [XDsDac_Start\(\)](#) must be called prior to using the device again.

Parameters:

InstancePtr is a pointer to the [XDsDac](#) instance to be worked on.

Returns:

None

Note:

None

```
void ( XDsDac * )  
XDsDac_Start InstancePtr
```

Starts the Delta-Sigma DAC Hardware.

The Device must be initialized and configured prior to **XDsDac_Start()** being called. Enables the DAC hardware to start producing analog outputs.

Parameters:

InstancePtr is a pointer to the **XDsDac** instance to be worked on.

Returns:

None

Note:

None

```
void ( XDsDac * )  
XDsDac_Stop InstancePtr
```

Stops the Delta-Sigma DAC Hardware. Prevents the DAC from producing analog outputs.

Parameters:

InstancePtr is a pointer to the **XDsDac** instance to be worked on.

Returns:

None

Note:

None

```
Xuint32 ( XDsDac * InstancePtr,  
XDsDac_WriteOutput DataPtr,  
Xuint16 * SampleCount  
unsigned SampleCount  
)
```

Transmit Delta-Sigma DAC data. The `SampleCount` is the total data size to be converted. If the amount of data to be sent will not fit in the FIFO, the return value indicates the amount of data sent to the DAC. This allows for this function to be used in a polling situation also.

Parameters:

InstancePtr is a pointer to the **XDsDac** instance to be worked on.

DataPtr is a pointer to data

SampleCount is the size, in samples, of the data

Returns:

The number of samples sent to the DAC

Note:

This function call is not blocking in nature, i.e. it will not wait until the data is sent to the hardware. Also, the interrupts are not disabled while the data is written to the DAC. If this is desired, the application must disable the interrupts prior to calling this function and then re-enable the interrupts when this function returns.

Variable Documentation

XDsDac_Config XDsDac_ConfigTable ()

[]

This table contains configuration information for each D-S DAC device in the system.

XDsDac Struct Reference

```
#include <xdsdac.h>
```

Detailed Description

The XDsDac driver instance data. The user is required to allocate a variable of this type for every DsDac device in the system. A pointer to a variable of this type is then passed to the driver API functions.

Data Fields

[Xuint32 BaseAddress](#)

[Xuint32 IsReady](#)

Field Documentation

[Xuint32 XDsDac::](#)

BaseAddress

Base address for device
(IPIF)

[Xuint32 XDsDac::](#)

IsReady

Device is initialized and
ready

The documentation for this struct was generated from the following file:

- [dsdac/v1_00_a/src/xdsdac.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

XIntc Struct Reference

```
#include <xintc.h>
```

Detailed Description

The XIntc driver instance data. The user is required to allocate a variable of this type for every intc device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [intc/v1_00_c/src/xintc.h](#)
-

intc/v1_00_c/src/xintc.h File Reference

Detailed Description

The Xilinx interrupt controller driver component. This component supports the Xilinx interrupt controller.

The interrupt controller driver uses the idea of priority for the various handlers. Priority is an integer within the range of 0 and 31 inclusive with 0 being the highest priority interrupt source.

The Xilinx interrupt controller supports the following features:

- specific individual interrupt enabling/disabling
- specific individual interrupt acknowledging
- attaching specific callback function to handle interrupt source
- master enable/disable
- single callback per interrupt or all pending interrupts handled for each interrupt of the processor

The acknowledgement of the interrupt within the interrupt controller is selectable, either prior to the device's handler being called or after the handler is called. This is necessary to support interrupt signal inputs which are either edge or level signals. Edge driven interrupt signals require that the interrupt is acknowledged prior to the interrupt being serviced in order to prevent the loss of interrupts which are occurring extremely close together. A level driven interrupt input signal requires the interrupt to be acknowledged after servicing the interrupt to ensure that the interrupt only generates a single interrupt condition.

Details about connecting the interrupt handler of the driver are contained in the source file specific to interrupt processing, [xintc_intr.c](#).

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

Interrupt Vector Tables

The interrupt vector table for each interrupt controller device is declared statically in [xintc_g.c](#) within the configuration data for each instance. The device ID of the interrupt controller device is used by the driver as a direct index into the configuration data table - to retrieve the vector table for an instance of the interrupt controller. The user should populate the vector table with handlers and callbacks at run-time using the [XIntc_Connect\(\)](#) and [XIntc_Disconnect\(\)](#) functions.

Each vector table entry corresponds to a device that can generate an interrupt. Each entry contains an interrupt handler function and an argument to be passed to the handler when an interrupt occurs. The tools default this argument to the base address of the interrupting device. Note that Level 1 device driver interrupt handlers do not take a base address as an argument, but instead take a pointer to the driver instance. This means that although the table is created statically, the user must still use [XIntc_Connect\(\)](#) when the interrupt handler takes an argument other than the base address. This is only to say that the existence of the static vector tables should not mislead the user into thinking they no longer need to register/connect interrupt handlers with this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00a	rpm	01/09/02	Removed the AckLocation argument from XIntc_Connect() . This information is now internal in xintc_g.c .
1.00b	jhl	02/13/02	Repartitioned the driver for smaller files
1.00b	jhl	04/24/02	Made LookupConfig function global and relocated config data type
1.00c	rpm	10/17/03	New release. Support the static vector table created in the xintc_g.c configuration table. Moved vector table and options out of instance structure and into the configuration table.

```
#include "xbasic_types.h"  
#include "xparameters.h"  
#include "xstatus.h"  
#include "xintc_1.h"
```

Data Structures

```
struct XIntc  
struct XIntc\_Config
```

Configuration options

These options are used in [XIntc_SetOptions\(\)](#) to configure the device.

```
#define XIN\_SVC\_SGL\_ISR\_OPTION  
#define XIN\_SVC\_ALL\_ISRS\_OPTION
```

Start modes

One of these values is passed to **XIntc_Start()** to start the device.

```
#define XIN_SIMULATION_MODE  
#define XIN_REAL_MODE
```

Functions

XStatus XIntc_Initialize (**XIntc** *InstancePtr, **Xuint16** DeviceId)

XStatus XIntc_Start (**XIntc** *InstancePtr, **Xuint8** Mode)

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

XStatus XIntc_Connect (**XIntc** *InstancePtr, **Xuint8** Id, **XInterruptHandler** Handler, void *CallBackRef)

void **XIntc_Disconnect** (**XIntc** *InstancePtr, **Xuint8** Id)

void **XIntc_Enable** (**XIntc** *InstancePtr, **Xuint8** Id)

void **XIntc_Disable** (**XIntc** *InstancePtr, **Xuint8** Id)

void **XIntc_Acknowledge** (**XIntc** *InstancePtr, **Xuint8** Id)

XIntc_Config * **XIntc_LookupConfig** (**Xuint16** DeviceId)

void **XIntc_InterruptHandler** (**XIntc** *InstancePtr)

XStatus XIntc_SetOptions (**XIntc** *InstancePtr, **Xuint32** Options)

Xuint32 XIntc_GetOptions (**XIntc** *InstancePtr)

XStatus XIntc_SelfTest (**XIntc** *InstancePtr)

XStatus XIntc_SimulateIntr (**XIntc** *InstancePtr, **Xuint8** Id)

Define Documentation

```
#define  
XIN_REAL_MODE
```

Real mode, no simulation allowed, hardware interrupts recognized

```
#define  
XIN_SIMULATION_MODE
```

Simulation only mode, no hardware interrupts recognized

```
#define  
XIN_SVC_ALL_ISRS_OPTION
```

XIN_SVC_SGL_ISR_OPTION	Service the highest priority pending interrupt and then return.
XIN_SVC_ALL_ISRS_OPTION	Service all of the pending interrupts and then return.

```
#define
XIN_SVC_SGL_ISR_OPTION
    XIN_SVC_SGL_ISR_OPTION    Service the highest priority pending interrupt
                              and then return.
    XIN_SVC_ALL_ISRS_OPTION   Service all of the pending interrupts and then
                              return.
```

Function Documentation

```
void
XIntc_Acknowledge ( XIntc *
                   InstancePtr,
                   Xuint8 Id
                 )
```

Acknowledges the interrupt source provided as the argument *Id*. When the interrupt is acknowledged, it causes the interrupt controller to clear its interrupt condition.

Parameters:

- InstancePtr* is a pointer to the **XIntc** instance to be worked on.
- 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.

Returns:

None.

Note:

None.

```
XStatus
XIntc_Connect ( XIntc *
               InstancePtr,
               Xuint8 Id,
               XInterruptHandler Handler,
               void * CallbackRef
             )
```

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.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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 to the handler for that interrupt.

CallbackRef is the callback reference, usually the instance pointer of the connecting driver.

Returns:

- o XST_SUCCESS if the handler was connected correctly.

Note:

WARNING: The handler provided as an argument will overwrite any handler that was previously connected.

```
void ( XIntc *  
XIntc_Disable InstancePtr,  
Xuint8 Id  
)
```

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.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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.

Returns:

None.

Note:

None.

```
void ( XIntc *  
XIntc_Disconnect InstancePtr,  
Xuint8 Id  
)
```

Updates the interrupt table with the Null Handler and XNULL arguments at the location pointed at by the Id. This effectively disconnects that interrupt source from any handler. The interrupt is disabled also.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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.

Returns:

None.

Note:

None.

```
void  
XIntc_Enable ( XIntc *  
              InstancePtr,  
              Xuint8 Id  
            )
```

Enables the interrupt source provided as the argument Id. Any pending interrupt condition for the specified Id will occur after this function is called.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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.

Returns:

None.

Note:

None.

```
Xuint32  
XIntc_GetOptions ( XIntc *  
                  InstancePtr
```

Return the currently set options.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

The currently set options. The options are described in [xintc.h](#).

Note:

None.

```
XStatus ( XIntc *  
XIntc_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

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

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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.

Returns:

- 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.

```
void ( XIntc *  
XIntc_InterruptHandler InstancePtr  
 )
```

The interrupt handler for the driver. This function is provided mostly for backward compatibility. The user should use **XIntc_DeviceInterruptHandler()**, defined in **xintc_l.c** when possible and pass the device ID of the interrupt controller device as its argument.

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.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

None.

Note:

None.

XIntc_Config* (**Xuint16**)
XIntc_LookupConfig *DeviceId*

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique identifier for a device.

Returns:

A pointer to the **XIntc** configuration structure for the specified device, or XNULL if the device was not found.

Note:

None.

XStatus (**XIntc ***)
XIntc_SelfTest *InstancePtr*

Run a self-test on the driver/device. This is a destructive test.

This involves forcing interrupts into the controller and verifying that they are recognized and can be acknowledged. This test will not succeed if the interrupt controller has been started in real mode such that interrupts cannot be forced.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

- XST_SUCCESS if self-test is successful.
- XST_INTC_FAIL_SELFTEST if the Interrupt controller fails the self-test. It will fail the self test if the device has previously been started in real mode.

Note:

None.

```
XStatus ( XIntc *  
XIntc_SetOptions InstancePtr,  
 Xuint32 Options  
 )
```

Set the options for the interrupt controller driver.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Options to be set. The available options are described in [xintc.h](#).

Returns:

- XST_SUCCESS if the options were set successfully
- XST_INVALID_PARAM if the specified option was not valid

Note:

None.

```
XStatus ( XIntc *  
XIntc_SimulateIntr InstancePtr,  
 Xuint8 Id  
 )
```

Allows software to simulate an interrupt in the interrupt controller. This function will only be successful when the interrupt controller has been started in simulation mode. Once it has been started in real mode, interrupts cannot be simulated. A simulated interrupt allows the interrupt controller to be tested without any device to drive an interrupt input signal into it.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id is the interrupt ID for which to simulate an interrupt.

Returns:

XST_SUCCESS if successful, or XST_FAILURE if the interrupt could not be simulated because the interrupt controller is or has previously been in real mode.

Note:

None.

```
XStatus ( XIntc *  
XIntc_Start InstancePtr,  
 Xuint8 Mode  
 )
```

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.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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

Returns:

- 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.

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

Stops the interrupt controller by disabling the output from the controller so that no interrupts will be caused by the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

None.

Note:

None.

intc/v1_00_c/src/xintc_intr.c File Reference

Detailed Description

This file contains the interrupt processing for the **XIntc** component which is the driver for the Xilinx Interrupt Controller. The interrupt processing is partitioned separately such that users are not required to use the provided interrupt processing. This file requires other files of the driver to be linked in also.

Two different interrupt handlers are provided for this driver such that the user must select the appropriate handler for the application. The first interrupt handler, `XIntc_VoidInterruptHandler`, is provided for systems which use only a single interrupt controller or for systems that cannot otherwise provide an argument to the **XIntc** interrupt handler (e.g., the RTOS interrupt vector handler may not provide such a facility). The constant `XPAR_INTC_SINGLE_DEVICE_ID` must be defined for this handler to be included in the driver. The second interrupt handler, `XIntc_InterruptHandler`, uses an input argument which is an instance pointer to an interrupt controller driver such that multiple interrupt controllers can be supported. This handler requires the calling function to pass it the appropriate argument, so another level of indirection may be required.

Note that both of these handlers are now only provided for backward compatibility. The handler defined in **`xintc_1.c`** is the recommended handler.

The interrupt processing may be used by connecting one of the interrupt handlers to the interrupt system. These handlers do not save and restore the processor context but only handle the processing of the Interrupt Controller. The two handlers are provided as working examples. The user is encouraged to supply their own interrupt handler when performance tuning is deemed necessary.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00b	jhl	02/13/02	First release
-------	-----	----------	---------------

1.00c	rpm	10/17/03	New release. Support the static vector table created
-------	-----	----------	--

handlers in the `xintc_g.c` configuration table. Collapse to use the `XIntc_DeviceInterruptHandler()` in `xintc_l.c`.

1.00c rpm 04/09/04 Added conditional compilation around the old handler `XIntc_VoidInterruptHandler()`. This handler will only be include/compiled if `XPAR_INTC_SINGLE_DEVICE_ID` is defined.

```
#include "xbasic_types.h"
#include "xparameters.h"
#include "xintc.h"
```

Functions

void `XIntc_InterruptHandler` (`XIntc` *InstancePtr)

Function Documentation

void `XIntc_InterruptHandler` (`XIntc` * *InstancePtr*)

The interrupt handler for the driver. This function is provided mostly for backward compatibility. The user should use `XIntc_DeviceInterruptHandler()`, defined in `xintc_l.c` when possible and pass the device ID of the interrupt controller device as its argument.

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.

Parameters:

InstancePtr is a pointer to the `XIntc` instance to be worked on.

Returns:

None.

Note:

None.

Generated on 24 Jun 2004 for Xilinx Device Drivers

intc/v1_00_c/src/xintc_i.c File Reference

Detailed Description

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.

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 <code>XPAR_INTC_SINGLE_DEVICE_ID</code> is defined.

```
#include "xparameters.h"  
#include "xbasic\_types.h"  
#include "xintc.h"  
#include "xintc\_i.h"
```

Functions

void **XIntc_DeviceInterruptHandler** (void *DeviceId)
void **XIntc_SetIntrSvcOption** (**Xuint32** BaseAddress, int Option)
void **XIntc_RegisterHandler** (**Xuint32** BaseAddress, int InterruptId, **XInterruptHandler** Handler,
void *CallbackRef)

Function Documentation

void (**void ***)
XIntc_DeviceInterruptHandler *DeviceId*

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 Level 1 **XIntc_SetOptions()** function or the Level 0 **XIntc_SetIntrSvcOption()** 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.

Parameters:

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.

Returns:

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          ( Xuint32
XIntc_RegisterHandler
                BaseAddress,
                int      InterruptId,
                XInterruptHandler Handler,
                void *    CallbackRef
                )

```

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 `XNULL` as the callback reference.

Parameters:

BaseAddress is the base address of the interrupt controller whose vector table will be modified.

InterruptId is the interrupt ID to be associated with the input handler.

Handler is the function pointer that will be added to the vector table for the given interrupt ID. It adheres to the `XInterruptHandler` signature found in [xbasic_types.h](#).

CallbackRef is the argument that will be passed to the new handler function when it is called. This is user-specific.

Returns:

None.

Note:

Note that this function has no effect if the input base address is invalid.

```

void          ( Xuint32
XIntc_SetIntrSvcOption
                BaseAddress,
                int      Option
                )

```

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 a Level 1 driver is to service only a single interrupt, whereas the default behavior when using a Level 0 driver is to service all outstanding interrupts when an interrupt occurs.

Parameters:

BaseAddress is the unique identifier for a device.

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.

Returns:

None.

Note:

Note that this function has no effect if the input base address is invalid.

intc/v1_00_c/src/xintc_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of interrupt controller devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	01/09/02	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files
1.00b	jhl	04/24/02	Compressed the ack table 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.

```
#include "xintc.h"  
#include "xparameters.h"
```

Variables

[XIntc_Config](#) [XIntc_ConfigTable](#) [XPAR_XINTC_NUM_INSTANCES]

Variable Documentation

[XIntc_Config](#) [XIntc_ConfigTable](#)
[XPAR_XINTC_NUM_INSTANCES]

This table contains configuration information for each intc device in the system. The **XIntc** driver must know when to acknowledge the interrupt. The entry which specifies this as a bit mask where each bit corresponds to a specific interrupt. A bit set indicates to ack it before servicing it. Generally, acknowledge before service is used when the interrupt signal is edge-sensitive, and after when the signal is level-sensitive.

Refer to the **XIntc_Config** data structure in **xintc.h** for details on how this table should be initialized.

Generated on 24 Jun 2004 for Xilinx Device Drivers

XIntc_Config Struct Reference

```
#include <xintc.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

Xuint16 DeviceId
Xuint32 BaseAddress
Xuint32 AckBeforeService
Xuint32 Options

XIntc_VectorTableEntry **HandlerTable** [XPAR_INTC_MAX_NUM_INTR_INPUTS]

Field Documentation

Xuint32 XIntc_Config::

AckBeforeService

Ack location per
interrupt

Xuint32 XIntc_Config::

BaseAddress

Register base
address

Xuint16 XIntc_Config::

DeviceId

Unique ID of
device

XIntc_VectorTableEntry XIntc_Config::HandlerTable [XPAR_INTC_MAX_NUM_INTR_INPUTS]

Static vector table of interrupt
handlers

Xuint32 XIntc_Config::

Options

Device
options

The documentation for this struct was generated from the following file:

- [intc/v1_00_c/src/xintc.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

intc/v1_00_c/src/xintc_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. High-level driver functions are defined in [xintc.h](#).

Note that users of this Level 0 driver interface can register an interrupt handler dynamically (at run-time) using the [XIntc_RegisterHandler\(\)](#) function. User of the Level 1 xintc driver interface should still use [XIntc_Connect\(\)](#), as always. Also see the discussion of the interrupt vector tables in [xintc.h](#).

There are currently two interrupt handlers specified in this interface.

- [XIntc_LowLevelInterruptHandler\(\)](#) is a handler without any arguments that is used in cases where there is a single interrupt controller device in the system and the handler cannot be passed an argument. This function is provided mostly for backward compatibility.
- [XIntc_DeviceInterruptHandler\(\)](#) is a handler that takes a device ID as an argument, indicating which interrupt controller device in the system is causing the interrupt - thereby supporting multiple interrupt controllers.

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.

```
#include "xbasic_types.h"
```

```
#include "xparameters.h"
#include "xio.h"
```

Data Structures

```
struct XIntc_VectorTableEntry
```

Defines

```
#define XIntc_mMasterEnable(BaseAddress)
#define XIntc_mMasterDisable(BaseAddress)
#define XIntc_mEnableIntr(BaseAddress, EnableMask)
#define XIntc_mDisableIntr(BaseAddress, DisableMask)
#define XIntc_mAckIntr(BaseAddress, AckMask)
#define XIntc_mGetIntrStatus(BaseAddress)
```

Functions

```
void XIntc_DeviceInterruptHandler (void *DeviceId)
void XIntc_SetIntrSvcOption (Xuint32 BaseAddress, int Option)
void XIntc_RegisterHandler (Xuint32 BaseAddress, int InterruptId, XInterruptHandler
    Handler, void *CallbackRef)
```

Define Documentation

```
#define XIntc_mAckIntr( BaseAddress,
                       AckMask      )
```

Acknowledge specific interrupt(s) in the interrupt controller.

Parameters:

BaseAddress is the base address of the device

AckMask is the 32-bit value to write to the acknowledge register. Each bit of the mask corresponds to an interrupt input signal that is connected to the interrupt controller (INT0 = LSB). Only the bits which are set in the mask will acknowledge interrupts.

Returns:

None.

Note:

Signature: void [XIntc_mAckIntr](#)(Xuint32 BaseAddress, Xuint32 AckMask)

```
#define XIntc_mDisableIntr( BaseAddress,  
                           DisableMask )
```

Disable specific interrupt(s) in the interrupt controller.

Parameters:

BaseAddress is the base address of the device

DisableMask is the 32-bit value to write to the enable register. Each bit of the mask corresponds to an interrupt input signal that is connected to the interrupt controller (INT0 = LSB). Only the bits which are set in the mask will disable interrupts.

Returns:

None.

Note:

Signature: void [XIntc_mDisableIntr](#)(Xuint32 BaseAddress, Xuint32 DisableMask)

```
#define XIntc_mEnableIntr( BaseAddress,  
                           EnableMask )
```

Enable specific interrupt(s) in the interrupt controller.

Parameters:

BaseAddress is the base address of the device

EnableMask is the 32-bit value to write to the enable register. Each bit of the mask corresponds to an interrupt input signal that is connected to the interrupt controller (INT0 = LSB). Only the bits which are set in the mask will enable interrupts.

Returns:

None.

Note:

Signature: void **XIntc_mEnableIntr**(Xuint32 BaseAddress, Xuint32 EnableMask)

```
#define XIntc_mGetIntrStatus( BaseAddress )
```

Get the interrupt status from the interrupt controller which indicates which interrupts are active and enabled.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit contents of the interrupt status register. Each bit corresponds to an interrupt input signal that is connected to the interrupt controller (INT0 = LSB). Bits which are set indicate an active interrupt which is also enabled.

Note:

Signature: Xuint32 **XIntc_mGetIntrStatus**(Xuint32 BaseAddress)

```
#define XIntc_mMasterDisable( BaseAddress )
```

Disable all interrupts in the Master Enable register of the interrupt controller.

Parameters:

BaseAddress is the base address of the device.

Returns:

None.

Note:

Signature: void **XIntc_mMasterDisable**(Xuint32 BaseAddress)

```
#define XIntc_mMasterEnable( BaseAddress )
```

Enable all interrupts in the Master Enable register of the interrupt controller. The interrupt controller defaults to all interrupts disabled from reset such that this macro must be used to enable interrupts.

Parameters:

BaseAddress is the base address of the device.

Returns:

None.

Note:

Signature: void **XIntc_mMasterEnable**(Xuint32 BaseAddress)

Function Documentation

```
void  
XIntc_DeviceInterruptHandler ( void *  
                               DeviceId )
```

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 Level 1 [XIntc_SetOptions\(\)](#) function or the Level 0 [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.

Parameters:

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.

Returns:

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_RegisterHandler ( Xuint32 BaseAddress,  
                           int InterruptId,  
                           XInterruptHandler Handler,  
                           void * CallbackRef  
                           )
```

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 `XNULL` as the callback reference.

Parameters:

BaseAddress is the base address of the interrupt controller whose vector table will be modified.

InterruptId is the interrupt ID to be associated with the input handler.

Handler is the function pointer that will be added to the vector table for the given interrupt ID. It adheres to the `XInterruptHandler` signature found in [xbasic_types.h](#).

CallbackRef is the argument that will be passed to the new handler function when it is called. This is user-specific.

Returns:

None.

Note:

Note that this function has no effect if the input base address is invalid.

```
void XIntc_SetIntrSvcOption ( Xuint32 BaseAddress,
                             int Option
                             )
```

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 a Level 1 driver is to service only a single interrupt, whereas the default behavior when using a Level 0 driver is to service all outstanding interrupts when an interrupt occurs.

Parameters:

BaseAddress is the unique identifier for a device.

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.

Returns:

None.

Note:

Note that this function has no effect if the input base address is invalid.

intc/v1_00_c/src/xintc.c File Reference

Detailed Description

Contains required functions for the [XIntc](#) driver for the Xilinx Interrupt Controller. See [xintc.h](#) for a detailed description of the driver.

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.

```
#include "xbasic_types.h"  
#include "xintc.h"  
#include "xintc_l.h"  
#include "xintc_i.h"
```

Functions

XStatus **XIntc_Initialize** (**XIntc** *InstancePtr, **Xuint16** DeviceId)

XStatus **XIntc_Start** (**XIntc** *InstancePtr, **Xuint8** Mode)

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

XStatus **XIntc_Connect** (**XIntc** *InstancePtr, **Xuint8** Id, **XInterruptHandler** Handler, void *CallBackRef)

void **XIntc_Disconnect** (**XIntc** *InstancePtr, **Xuint8** Id)

void **XIntc_Enable** (**XIntc** *InstancePtr, **Xuint8** Id)

void **XIntc_Disable** (**XIntc** *InstancePtr, **Xuint8** Id)

void **XIntc_Acknowledge** (**XIntc** *InstancePtr, **Xuint8** Id)

XIntc_Config * **XIntc_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
void  
XIntc_Acknowledge ( XIntc *  
                   InstancePtr,  
                   Xuint8 Id  
                   )
```

Acknowledges the interrupt source provided as the argument *Id*. When the interrupt is acknowledged, it causes the interrupt controller to clear its interrupt condition.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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.

Returns:

None.

Note:

None.

```

XStatus          ( XIntc *
XIntc_Connect   InstancePtr,
                   Xuint8   Id,
                   XInterruptHandler Handler,
                   void *    CallbackRef
                   )

```

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.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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 to the handler for that interrupt.

CallbackRef is the callback reference, usually the instance pointer of the connecting driver.

Returns:

- XST_SUCCESS if the handler was connected correctly.

Note:

WARNING: The handler provided as an argument will overwrite any handler that was previously connected.

```

void             ( XIntc *
XIntc_Disable   InstancePtr,
                   Xuint8 Id
                   )

```

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.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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.

Returns:

None.

Note:

None.

```
void  
XIntc_Disconnect ( XIntc *  
                  InstancePtr,  
                  Xuint8 Id  
                  )
```

Updates the interrupt table with the Null Handler and XNULL arguments at the location pointed at by the *Id*. This effectively disconnects that interrupt source from any handler. The interrupt is disabled also.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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.

Returns:

None.

Note:

None.

```
void          ( XIntc *  
XIntc_Enable      InstancePtr,  
                Xuint8 Id  
                )
```

Enables the interrupt source provided as the argument *Id*. Any pending interrupt condition for the specified *Id* will occur after this function is called.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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.

Returns:

None.

Note:

None.

```
XStatus          ( XIntc *  
XIntc_Initialize      InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

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

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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.

Returns:

- 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.

```
XIntc_Config* ( Xuint16 )  
XIntc_LookupConfig DeviceId
```

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique identifier for a device.

Returns:

A pointer to the **XIntc** configuration structure for the specified device, or XNULL if the device was not found.

Note:

None.

```
XStatus ( XIntc *  
XIntc_Start InstancePtr,  
Xuint8 Mode  
)
```

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.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

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

Returns:

- 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.

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

Stops the interrupt controller by disabling the output from the controller so that no interrupts will be caused by the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

None.

Note:

None.

intc/v1_00_c/src/xintc_i.h File Reference

Detailed Description

This file contains data which is shared between files and internal to the **XIntc** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00b	jhl	02/06/02	First release
-------	-----	----------	---------------

1.00b	jhl	04/24/02	Moved register definitions to xintc_1.h
-------	-----	----------	---

1.00c	rpm	10/17/03	New release. Removed extern of global, single instance pointer.
-------	-----	----------	---

```
#include "xbasic_types.h"
```

```
#include "xintc.h"
```

Variables

[XIntc_Config](#) [XIntc_ConfigTable](#) []

Variable Documentation

[XIntc_Config](#) [XIntc_ConfigTable](#) ()

[]

This table contains configuration information for each intc device in the system. The **XIntc** driver must know when to acknowledge the interrupt. The entry which specifies this as a bit mask where each bit corresponds to a specific interrupt. A bit set indicates to ack it before servicing it. Generally, acknowledge before service is used when the interrupt signal is edge-sensitive, and after when the signal is level-sensitive.

Refer to the **XIntc_Config** data structure in **xintc.h** for details on how this table should be initialized.

Generated on 24 Jun 2004 for Xilinx Device Drivers

dsdac/v1_00_a/src/xdsdac_I.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions and macros that can be used to access the device.

The Xilinx DS DAC driver component. This component supports the Xilinx Delta-Sigma Digital-to-Analog Converter (DAC).

Refer to [xdsdac.h](#) for more details.

Note:

None

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	03/30/04	First release

```
#include "xbasic_types.h"
```

```
#include "xio.h"
```

Defines

```
#define XDAC_GIER_OFFSET
```

```
#define XDAC_IPISR_OFFSET
```

```
#define XDAC_IPIER_OFFSET
```

```
#define XDAC_IPISRR_OFFSET
```

```
#define XDAC_CR_OFFSET
```

```
#define XDAC_FIFO_OFFSET
#define XDAC_OCCY_OFFSET
#define XDAC_PIRQ_OFFSET
#define XDAC_GIER_GIE_MASK
#define XDAC_IPIXR_PIRQ_MASK
#define XDAC_IPIXR_EMPTY_MASK
#define XDAC_IPISRR_RESET_MASK
#define XDAC_CR_FIFO_RESET_MASK
#define XDAC_CR_EN_MASK
#define XDAC_FIFO_DATA_MASK
#define XDAC_OCCY_MASK
#define XDAC_PIRQ_MASK
#define XDac_mWriteReg(BaseAddress, RegOffset, Data)
#define XDac_mReadReg(BaseAddress, RegOffset)
```

Define Documentation

```
#define  
XDAC_CR_EN_MASK
```

Device enable
bit

```
#define  
XDAC_CR_FIFO_RESET_MASK
```

FIFO reset
bit

```
#define  
XDAC_CR_OFFSET
```

DAC control
register

```
#define  
XDAC_FIFO_DATA_MASK
```

valid data FIFO
bits

```
#define  
XDAC_FIFO_OFFSET
```

```
DAC Data  
FIFO
```

```
#define  
XDAC_GIER_GIE_MASK
```

```
Global  
Enable
```

```
#define  
XDAC_GIER_OFFSET
```

```
Global Interrupt  
Enable
```

```
#define  
XDAC_IPIER_OFFSET
```

```
IPIF interrupt  
enable
```

```
#define  
XDAC_IPISR_OFFSET
```

```
IPIF interrupt  
status
```

```
#define  
XDAC_IPISRR_OFFSET
```

```
IPIF sw reset  
register
```

```
#define  
XDAC_IPISRR_RESET_MASK
```

```
the reset bit is  
set
```

```
#define  
XDAC_IPIXR_EMPTY_MASK
```

the FIFO empty interrupt is
set

```
#define  
XDAC_IPIXR_PIRQ_MASK
```

the depth interrupt is
set

```
#define  
XDAC_OCCY_MASK
```

valid FIFO occupancy
bits

```
#define  
XDAC_OCCY_OFFSET
```

DAC Occupancy
register

```
#define  
XDAC_PIRQ_MASK
```

valid compare value
bits

```
#define  
XDAC_PIRQ_OFFSET
```

DAC Programmable Depth
register

```
#define XDsDac_mReadReg( BaseAddress,  
                        RegOffset )
```

Read value from DS DAC register.

Parameters:

BaseAddress is the base address of the DS DAC device.

RegOffset is the register offset from the base to read from.

Returns:

The data read from the device.

Note:

None.

C-style signature Xuint32 **XDsDac_mReadReg**(Xuint32 BaseAddress, Xuint32 RegOffset)

```
#define XDsDac_mWriteReg( BaseAddress,  
                        RegOffset,  
                        Data      )
```

Write value to DS DAC register.

Parameters:

BaseAddress is the base address of the DS DAC device.

RegOffset is the register offset from the base to write to.

Data is the data to be written to the device.

Returns:

None.

Note:

None.

C-style signature void XDsDac_mWriteReg(Xuint32 BaseAddress, Xuint32 RegOffset, Xuint32 Data)

dsadc/v1_00_a/src/xdsadc_selftest.c File Reference

Detailed Description

Function in this file are needed for the selftest functionality.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	05/10/04	First release

```
#include "xstatus.h"  
#include "xdsadc_1.h"  
#include "xdsadc.h"  
#include "xio.h"
```

Defines

```
#define TEST_VALUE
```

Define Documentation

```
#define  
TEST_VALUE
```

Performs a SelfTest on the DsAdc device as follows:

- Reads the current value in the FSTM register.
- Writes test value to register.
- Reads back FSTM register and compares with test value.
- Returns the original value to the FSTM. If the value does not match the test value, the test fails.

Parameters:

InstancePtr is a pointer to the **XDsAdc** instance to be worked on.

Returns:

- XST_SUCCESS if the device Passed the Self Test.
- XST_FAILURE if the data read backdose not match the test value..

Note:

None.

dsadc/v1_00_a/src/xdsadc_i.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions and macros that can be used to access the device.

The Xilinx driver component. This component supports the Xilinx Delta-Sigma Analog-to-Digital Converter (ADC).

Refer to [xdsadc.h](#) for more information.

Note:

None

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	05/10/04	First release

```
#include "xbasic_types.h"
```

```
#include "xio.h"
```

Defines

```
#define XADC_GIER_OFFSET
```

```
#define XADC_IPISR_OFFSET
```

```
#define XADC_IPIER_OFFSET
```

```
#define XADC_FSTM_OFFSET
```

```
#define XADC_DATA_OFFSET
```

```
#define XADC_GIER_GIE_MASK
#define XADC_IPIXR_READY_MASK
#define XADC_FSTM_MASK
#define XDsAdc_mWriteReg(BaseAddress, RegOffset, Data)
#define XDsAdc_mReadReg(BaseAddress, RegOffset)
```

Define Documentation

```
#define  
XADC_DATA_OFFSET
```

ADC
Data

```
#define  
XADC_FSTM_MASK
```

vaild bits in
register

```
#define  
XADC_FSTM_OFFSET
```

Filter Settle Time Mult
Reg

```
#define  
XADC_GIER_GIE_MASK
```

Global
Enable

```
#define  
XADC_GIER_OFFSET
```

Offset for the GIE
bit

```
#define  
XADC_IPIER_OFFSET
```

IPIF interrupt
enable

```
#define  
XADC_IPISR_OFFSET
```

IPIF interrupt
status

```
#define  
XADC_IPIXR_READY_MASK
```

the sample ready interrupt is
set

```
#define XDsAdc_mReadReg( BaseAddress,  
                        RegOffset )
```

Read value from DS ADC register.

Parameters:

BaseAddress is the base address of the DS ADC device.

RegOffset is the register offset from the base to read from.

Returns:

The data read from the device.

Note:

C-style signature Xuint32 [XDsAdc_mReadReg](#)(Xuint32 BaseAddress, Xuint32
RegOffset)

```
#define XDsAdc_mWriteReg( BaseAddress,  
                        RegOffset,  
                        Data )
```

Write value to DS ADC register.

Parameters:

BaseAddress is the base address of the DS ADC device.

RegOffset is the register offset from the base to write to.

Data is the data to be written to the device.

Returns:

None.

Note:

C-style signature void XDsAdc_mWriteReg(Xuint32 BaseAddress, Xuint32 RegOffset, Xuint32 Data)

dsadc/v1_00_a/src/xdsadc.h File Reference

Detailed Description

The Xilinx Delta-Sigma Analog-to-Digital Converter (ADC) driver. This driver supports the Xilinx ADC hardware.

Driver Description

The device driver enables higher layer software (e.g., an application) to communicate to the D-S ADC .

The driver supports interrupt driven mode and the default mode of operation is polled mode.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are stored in the [xdsadc_g.c](#) file. A table is defined where each entry contains configuration information for an DsAdc device. This information includes such things as the base address of the memory-mapped device.

Interrupt Processing

After [XDsAdc_Initialize\(\)](#) is called, [XDsAdc_EnableInterrupts\(\)](#) is called to enable the interrupts from the DS- ADC hardware. After initializing the Interrupt handling for the Intc Driver, such as [XIntc_SetHandler\(\)](#), and enabling the interrupt in the Intc, interrupt driven processing will begin. There is no callback handler provided by this driver.

An example is provided in the examples directory on how to construct a user defined handler.

When processing is complete, call [XDsAdc_DisableInterrupts\(\)](#) to stop the interrupts.

The SampleReady interrupt is enabled within the [XDsAdc_EnableInterrupts\(\)](#) function and disabled in the [XDsAdc_DisableInterrupts\(\)](#) function. The [XDsAdc_ReadInput\(\)](#) function does not

acknowledge the D-S ADC generated interrupts. The application must call `XDsAdc_GetInterruptStatus()` to determine which interrupts are pending and call `XDsAdc_ClearInterruptStatus()` to acknowledge the appropriate interrupt sources.

It is the application's responsibility to acknowledge any associated Interrupt Controller interrupts if it is used in the system.

Since there isn't any buffering within the hardware it is possible for the interrupt processing to dominate the processor if the designer is not careful.

Asserts

Asserts are used within all Xilinx drivers to enforce constraints on argument values. Asserts can be turned off on a system-wide basis by defining, at compile time, the `NDEBUG` identifier. By default, asserts are turned on and it is recommended that application developers leave asserts on during development.

Note:

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	05/10/04	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xdsadc_1.h"
```

Data Structures

```
struct XDsAdc  
struct XDsAdc_Config
```

Defines

```
#define XDsAdc_mIsEmpty(InstancePtr)
#define XDsAdc_mSetFstmReg(InstancePtr, Data)
#define XDsAdc_mGetFstmReg(InstancePtr)
```

Functions

```
XStatus XDsAdc_Initialize (XDsAdc *InstancePtr, Xuint16 DeviceId)
```

```
XDsAdc_Config * XDsAdc_LookupConfig (Xuint16 DeviceId)
```

```
Xuint16 XDsAdc_ReadInput (XDsAdc *InstancePtr)
```

```
void XDsAdc_EnableInterrupts (XDsAdc *InstancePtr)
```

```
void XDsAdc_DisableInterrupts (XDsAdc *InstancePtr)
```

```
Xuint32 XDsAdc_GetInterruptStatus (XDsAdc *InstancePtr)
```

```
void XDsAdc_ClearInterruptStatus (XDsAdc *InstancePtr, Xuint32 Mask)
```

Define Documentation

```
#define XDsAdc_mGetFstmReg( InstancePtr )
```

Gets the Filter Settle Time Multiplier Register.

Parameters:

InstancePtr is the instance of the device to be operated on.

Returns:

the value read from the register.

Note:

C-style signature Xuint32 [XDsAdc_mGetFstmReg](#)(XDsAdc *InstancePtr)

```
#define XDsAdc_mIsEmpty( InstancePtr )
```

Check to see if the ADC is empty.

Parameters:

InstancePtr is the instance of the device to be operated on.

Returns:

XTRUE if it is empty, or XFALSE if it is not.

Note:

C-style signature Xboolean **XDsAdc_mIsEmpty**(XDsAdc *InstancePtr)

```
#define XDsAdc_mSetFstmReg( InstancePtr,  
                           Data          )
```

Set the Filter Settle Time Multiplier Register.

Parameters:

InstancePtr is the instance of the device to be operated on.

Data is the value written to the register.

Returns:

None.

Note:

C-style signature void **XDsAdc_mSetFstmReg**(XDsAdc *InstancePtr, Xuint32 Data)

Please refer to the hardware spec to determine what the appropriate value should be.

Function Documentation

```
void  
XDsAdc_ClearInterruptStatus ( XDsAdc *  
                               InstancePtr,  
                               Xuint32  Mask  
                               )
```

Clears the interrupt status in the register as directed by the bits set in the provided Mask

Parameters:

InstancePtr contains a pointer to the D-S ADC device instance for the interrupt.

Mask contains the interrupt sources to be acknowledged/cleared.

Returns:

None.

Note:

```
void ( XDsAdc * )  
XDsAdc_DisableInterrupts InstancePtr
```

Disables the interrupt from the DS-ADC (the higher layer software is responsible for disabling interrupts at the interrupt controller).

To start using the device again, `_EnableInterrupts` must be called.

Parameters:

InstancePtr is a pointer to the `XDsAdc` instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XDsAdc * )  
XDsAdc_EnableInterrupts InstancePtr
```

Enables the DsAdc Interrupt.

This function must be called before other functions to convert data when in interrupt driven mode. The user should have connected the interrupt handler of the application to an interrupt source such as an interrupt controller or the processor interrupt prior to this function being called.

Parameters:

InstancePtr is a pointer to the **XDsAdc** instance to be worked on.

Returns:

None.

Note:

None

```
Xuint32 ( XDsAdc * )  
XDsAdc_GetInterruptStatus InstancePtr
```

Get the interrupt status from the registers to determine the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the D-S ADC device instance for the interrupt.

Returns:

The currently asserted interrupts

None.

Note:

```
XStatus ( XDsAdc *  
XDsAdc_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize a specific **XDsAdc** instance/driver. The initialization entails:

- Initialize fields of the **XDsAdc** instance structure.
- Disables interrupts.

The driver defaults to polled mode operation.

Parameters:

InstancePtr is a pointer to the **XDsAdc** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XDsAdc** instance. Passing in a device id associates the generic **XDsAdc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful.
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None

XDsAdc_Config* (**Xuint16**)
XDsAdc_LookupConfig *DeviceId*

Lookup the device configuration based on the unique device ID. The table XDsAdc_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

Xuint16 (**XDsAdc ***)
XDsAdc_ReadInput *InstancePtr*

Receive Delta-Sigma ADC data.

Parameters:

InstancePtr is a pointer to the **XDsAdc** instance to be worked on.

Returns:

The last value read from the ADC

Note:

This function call is not blocking in nature, i.e. it will not wait until the data is available in the hardware.

Generated on 24 Jun 2004 for Xilinx Device Drivers

dsadc/v1_00_a/src/xdsadc_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of D-S ADC devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	05/10/04	First release

```
#include "xdsadc.h"  
#include "xparameters.h"
```

Variables

[XDsAdc_Config](#) [XDsAdc_ConfigTable](#) [XPAR_XDSADC_NUM_INSTANCES]

Variable Documentation

[XDsAdc_Config](#) [XDsAdc_ConfigTable](#)
[XPAR_XDSADC_NUM_INSTANCES]

This table contains configuration information for each D-S ADC device in the system.

XDsAdc_Config Struct Reference

```
#include <xdsadc.h>
```

Detailed Description

This typedef contains configuration information for a device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

Field Documentation

[Xuint32 XDsAdc_Config::](#)

BaseAddress

Device base
address

[Xuint16 XDsAdc_Config::](#)

DeviceId

Unique ID of
device

The documentation for this struct was generated from the following file:

- dsadc/v1_00_a/src/[xdsadc.h](#)

dsadc/v1_00_a/src/xdsadc.c File Reference

Detailed Description

Functions in this file are the minimum required functions for the D-S ADC driver. See [xdsadc.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a ecm 05/10/04 First release

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xio.h"  
#include "xenv.h"  
#include "xdsadc_1.h"  
#include "xdsadc.h"
```

Functions

[XStatus XDsAdc_Initialize](#) ([XDsAdc](#) *InstancePtr, [Xuint16](#) DeviceId)

[Xuint16 XDsAdc_ReadInput](#) ([XDsAdc](#) *InstancePtr)

[XDsAdc_Config](#) * [XDsAdc_LookupConfig](#) ([Xuint16](#) DeviceId)

Variables

[XDsAdc_Config](#) [XDsAdc_ConfigTable](#) []

Function Documentation

```
XStatus ( XDsAdc *  
XDsAdc_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize a specific **XDsAdc** instance/driver. The initialization entails:

- Initialize fields of the **XDsAdc** instance structure.
- Disables interrupts.

The driver defaults to polled mode operation.

Parameters:

InstancePtr is a pointer to the **XDsAdc** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XDsAdc** instance. Passing in a device id associates the generic **XDsAdc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful.
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None

```
XDsAdc_Config* ( Xuint16 )  
XDsAdc_LookupConfig DeviceId
```

Lookup the device configuration based on the unique device ID. The table `XDsAdc_ConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or `XNULL` if no match is found.

Note:

None.

Xuint16 (**XDsAdc ***)
XDsAdc_ReadInput *InstancePtr*

Receive Delta-Sigma ADC data.

Parameters:

InstancePtr is a pointer to the **XDsAdc** instance to be worked on.

Returns:

The last value read from the ADC

Note:

This function call is not blocking in nature, i.e. it will not wait until the data is available in the hardware.

Variable Documentation

XDsAdc_Config `XDsAdc_ConfigTable` ()

[]

This table contains configuration information for each D-S ADC device in the system.

XDsAdc Struct Reference

```
#include <xdsadc.h>
```

Detailed Description

The XDsAdc driver instance data. The user is required to allocate a variable of this type for every DsAdc device in the system. A pointer to a variable of this type is then passed to the driver API functions.

Data Fields

[Xuint32 BaseAddress](#)

[Xuint32 IsReady](#)

Field Documentation

[Xuint32 XDsAdc::](#)

BaseAddress

Base address for device
(IPIF)

[Xuint32 XDsAdc::](#)

IsReady

Device is initialized and
ready

The documentation for this struct was generated from the following file:

- dsadc/v1_00_a/src/[xdsadc.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

common/v1_00_a/src/xbasic_types.c File Reference

Detailed Description

This file contains basic functions for Xilinx software IP.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a rpm 11/07/03 Added XNullHandler function as a stub interrupt handler

```
#include "xbasic_types.h"
```

Functions

void [XAssert](#) (char *File, int Line)
void [XAssertSetCallback](#) ([XAssertCallback](#) Routine)
void [XNullHandler](#) (void *NullParameter)

Variables

unsigned int [XAssertStatus](#)
[Xboolean](#) [XWaitInAssert](#)

Function Documentation

```
void      ( char *
XAssert   File,
          int   Line
          )
```

Implements assert. Currently, it calls a user-defined callback function if one has been set. Then, it potentially enters an infinite loop depending on the value of the XWaitInAssert variable.

Parameters:

File is the name of the filename of the source

Line is the linenumber within File

Returns:

None.

Note:

None.

```
void      ( XAssertCallback
XAssertSetCallback Routine
          )
```

Sets up a callback function to be invoked when an assert occurs. If there was already a callback installed, then it is replaced.

Parameters:

Routine is the callback to be invoked when an assert is taken

Returns:

None.

Note:

This function has no effect if NDEBUG is set

```
void      ( void *
XNullHandler NullParameter
          )
```

Null handler function. This follows the XInterruptHandler signature for interrupt handlers. It can be used to assign a null handler (a stub) to an interrupt controller vector table.

Parameters:

NullParameter is an arbitrary void pointer and not used.

Returns:

None.

Note:

None.

Variable Documentation

unsigned int **XAssertStatus**

This variable allows testing to be done easier with asserts. An assert sets this variable such that a driver can evaluate this variable to determine if an assert occurred.

Xboolean **XWaitInAssert**

This variable allows the assert functionality to be changed for testing such that it does not wait infinitely. Use the debugger to disable the waiting during testing of asserts.

XAtmc Struct Reference

```
#include <xatmc.h>
```

Detailed Description

The XAtmc driver instance data. The user is required to allocate a variable of this type for every ATMC device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [atmc/v1_00_c/src/xatmc.h](#)
-

atmc/v1_00_c/src/xatmc.c File Reference

Detailed Description

This file contains the ATM controller driver. This file contains send and receive functions as well as interrupt service routines.

There is one interrupt service routine registered with the interrupt controller. This function determines the source of the interrupt and calls an appropriate handler function.

Note:

None.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	JHL	07/31/01	First release
1.00c	rpm	01/08/03	New release supports v2.00a of packet fifo driver an v1.23b of the IPIF driver

```
#include "xatmc.h"  
#include "xatmc_i.h"  
#include "xipif_v1_23_b.h"  
#include "xpacket_fifo_v2_00_a.h"  
#include "xio.h"
```

Functions

XStatus XAtmc_SgSend (**XAtmc** *InstancePtr, XBufDescriptor *BdPtr)
XStatus XAtmc_SgRecv (**XAtmc** *InstancePtr, XBufDescriptor *BdPtr)
XStatus XAtmc_SgGetSendCell (**XAtmc** *InstancePtr, XBufDescriptor **PtrToBdPtr, int *BdCountPtr)
XStatus XAtmc_SgGetRecvCell (**XAtmc** *InstancePtr, XBufDescriptor **PtrToBdPtr, int *BdCountPtr)
XStatus XAtmc_PollSend (**XAtmc** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** ByteCount)
XStatus XAtmc_PollRecv (**XAtmc** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** *ByteCountPtr, **Xuint32** *CellStatusPtr)
void **XAtmc_InterruptHandler** (void *InstancePtr)

Function Documentation

```
void XAtmc_InterruptHandler ( void * InstancePtr )
```

Interrupt handler for the ATM controller driver. It performs the following processing:

- Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: the ATM controller, the send packet FIFO, the receive packet FIFO, the send DMA channel, or the receive DMA channel. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the ATM controller.
- Call the appropriate handler based on the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the ATMC controller instance for the interrupt.

Returns:

None.

Note:

None.

```

XStatus
XAtmc_PollRecv
    ( XAtmc * InstancePtr,
      Xuint8 * BufPtr,
      Xuint32 * ByteCountPtr,
      Xuint32 * CellStatusPtr
    )

```

Receives an ATM cell in polled mode. The device/driver must be in polled mode before calling this function. The driver receives the cell directly from the ATM controller packet FIFO. This is a non-blocking receive, in that if there is no cell ready to be received at the device, the function returns with an error. The buffer into which the cell will be received must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer into which the received Atmc cell will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to the size of the buffer on entry into the function and the size the received cell on return from the function.

CellStatusPtr is both an input and an output parameter. It is a pointer to the status of the cell which is received. It is only valid if the return value indicates success. The status is necessary when cells with errors are not being discarded. This status is a bit mask which may contain one or more of the following values with the exception of XAT_CELL_STATUS_NO_ERROR which is mutually exclusive. The status values are:

- XAT_CELL_STATUS_NO_ERROR indicates the cell was received without any errors
- XAT_CELL_STATUS_BAD_PARITY indicates the cell parity was not correct
- XAT_CELL_STATUS_BAD_HEC indicates the cell HEC was not correct
- XAT_CELL_STATUS_SHORT indicates the cell was not the correct length
- XAT_CELL_STATUS_VXI_MISMATCH indicates the cell VPI/VCI fields did not match the expected header values

Returns:

- XST_SUCCESS if the cell was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_NO_DATA if there is no cell to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the cell is too small for the cell waiting in the FIFO.

Note:

The input buffer must be big enough to hold the largest ATM cell. The buffer must also be 32-bit aligned.

```
XStatus          ( XAtmc *  
XAtmc_PollSend      InstancePtr,  
                    Xuint8 * BufPtr,  
                    Xuint32 ByteCount  
                    )
```

Sends an ATM cell in polled mode. The device/driver must be in polled mode before calling this function. The driver writes the cell directly to the ATM controller packet FIFO, then enters a loop checking the device status for completion or error. The buffer to be sent must be word-aligned.

It is assumed that the upper layer software supplies a correctly formatted ATM cell based upon the configuration of the ATM controller (attaching header or not).

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer containing the ATM cell to be sent.

ByteCount is the size of the ATM cell. An ATM cell for a 16 bit Utopia interface is 54 bytes with a 6 byte header and 48 bytes of payload. This function may be used to send short cells with or without headers depending on the configuration of the ATM controller.

Returns:

- XST_SUCCESS if the cell was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_PFIFO_NO_ROOM if there is no room in the FIFO for this cell
- XST_FIFO_ERROR if the FIFO was overrun or underrun

Note:

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PollSend thread.

The input buffer must be big enough to hold the largest ATM cell. The buffer must also be 32-bit aligned.

```

XStatus ( XAtmc *
XAtmc_SgGetRecvCell InstancePtr,
XBufDescriptor ** PtrToBdPtr,
int * BdCountPtr
)

```

Gets the first buffer descriptor of the oldest cell which was received by the scatter-gather DMA channel of the ATM controller. This function is provided to be called from a callback function such that the buffer descriptors for received cells can be processed. The function should be called by the application repetitively for the number of cells indicated as an argument in the callback function. This function may also be used when only payloads are being sent and received by the ATM controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the cell. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the cell. This input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the cell pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- XST_SUCCESS if a descriptor was successfully returned to the driver.
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode.
- XST_DMA_SG_NO_LIST if the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```

XStatus ( XAtmc *
XAtmc_SgGetSendCell InstancePtr,
XBufDescriptor ** PtrToBdPtr,
int * BdCountPtr
)

```

Gets the first buffer descriptor of the oldest cell which was sent by the scatter-gather DMA channel of the ATM controller. This function is provided to be called from a callback function such that the buffer descriptors for sent cells can be processed. The function should be called by the application repetitively for the number of cells indicated as an argument in the callback function. This function may also be used when only payloads are being sent and received by the ATM controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the cell. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the cell. this input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the cell pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- XST_SUCCESS if a descriptor was successfully returned to the driver.
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode.
- XST_DMA_SG_NO_LIST if the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```

XStatus ( XAtmc *
XAtmc_SgRecv InstancePtr,
XBufDescriptor * BdPtr
)

```

Adds this descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be word-aligned. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of cells to replace filled buffers with empty buffers. The contents of the specified buffer descriptor are copied into the scatter-gather transmit list. This function can be called when the device is started or stopped.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

Note:

None.

```
XStatus ( XAtmc *  
XAtmc_SgSend InstancePtr,  
 XBufDescriptor * BdPtr  
 )
```

Sends an ATM cell using scatter-gather DMA. The caller attaches the cell to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire ATM cell may or may not be contained within one descriptor. The contents of the buffer descriptor are copied into the scatter-gather transmit list. The caller is responsible for providing mutual exclusion to guarantee that a cell is contiguous in the transmit list. The buffer attached to the descriptor must be word-aligned.

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the cell, the inserts are committed, which means the descriptors for this cell are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted ATM cell based upon the configuration of the ATM controller (attaching header or not). The ATM controller must be started

before calling this function.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Returns:

- XST_SUCCESS if the buffer was successfully sent
- XST_DEVICE_IS_STOPPED if the ATM controller has not been started yet
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the descriptor list for the DMA channel is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

atmc/v1_00_c/src/xatmc_i.h File Reference

Detailed Description

This file contains data which is shared between files internal to the [XAtmc](#) component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a	JHL	07/31/01	First release
-------	-----	----------	---------------

1.00c	rpm	01/08/03	New release supports v2.00a of packet fifo driver
-------	-----	----------	---

an v1.23b of the IPIF driver

```
#include "xatmc.h"
```

```
#include "xatmc\_l.h"
```

Defines

```
#define XAtmc\_mIsSgDma(InstancePtr)
```

Variables

```
XAtmc\_Config XAtmc\_ConfigTable []
```

Define Documentation

```
#define XAtmc_mIsSgDma( InstancePtr )
```

This macro determines if the device is currently configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the [XAtmc](#) instance to be worked on.

Returns:

Boolean XTRUE if the device is configured for scatter-gather DMA, or XFALSE if it is not.

Note:

Signature: Xboolean [XAtmc_mIsSgDma](#)(XAtmc *InstancePtr)

Variable Documentation

```
XAtmc_Config XAtmc_ConfigTable ( )
```

```
[]
```

This table contains configuration information for each ATMC device in the system.

Generated on 24 Jun 2004 for Xilinx Device Drivers

atmc/v1_00_c/src/xatmc_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xatmc.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	rpm	05/01/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

Defines

```
#define XAtmc_mReadReg(BaseAddress, RegOffset)  
#define XAtmc_mWriteReg(BaseAddress, RegOffset, Data)  
#define XAtmc_mEnable(BaseAddress)  
#define XAtmc_mDisable(BaseAddress)  
#define XAtmc_mIsTxDone(BaseAddress)  
#define XAtmc_mIsRxEmpty(BaseAddress)
```

Functions

```
void XAtmc_SendCell (Xuint32 BaseAddress, Xuint8 *CellPtr, int Size)  
int XAtmc_RecvCell (Xuint32 BaseAddress, Xuint8 *CellPtr, Xuint32 *CellStatusPtr)
```

Define Documentation

```
#define XAtmc_mDisable( BaseAddress )
```

Disable the transmitter and receiver. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XAtmc_mEnable( BaseAddress )
```

Enable the transmitter and receiver. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XAtmc_mIsRxEmpty( BaseAddress )
```

Check to see if the receive FIFO is empty.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is empty, or XFALSE if it is not.

Note:

None.

```
#define XAtmc_mIsTxDone( BaseAddress )
```

Check to see if the transmission is complete.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is done, or XFALSE if it is not.

Note:

None.

```
#define XAtmc_mReadReg( BaseAddress,  
                        RegOffset  )
```

Read the given register.

Parameters:

BaseAddress is the base address of the
device

RegOffset is the register offset to be read

Returns:

The 32-bit value of the register

Note:

None.

```
#define XAtmc_mWriteReg( BaseAddress,  
                        RegOffset,  
                        Data      )
```

Write the given register.

Parameters:

BaseAddress is the base address of the device

RegOffset is the register offset to be written

Data is the 32-bit value to write to the register

Returns:

None.

Note:

None.

Function Documentation

```
int  
XAtmc_RecvCell ( Xuint32  
                BaseAddress,  
                Xuint8 * CellPtr,  
                Xuint32 * CellStatusPtr  
                )
```

Receive a cell. Wait for a cell to arrive.

Parameters:

BaseAddress is the base address of the device

CellPtr is a pointer to a word-aligned buffer where the cell will be stored.

CellStatusPtr is a pointer to a cell status that will be valid after this function returns.

Returns:

The size, in bytes, of the cell received.

Note:

None.

```
void
XAtmc_SendCell ( Xuint32
                 BaseAddress,
                 Xuint8 * CellPtr,
                 int      Size
               )
```

Send an ATM cell. This function blocks waiting for the cell to be transmitted.

Parameters:

BaseAddress is the base address of the device
CellPtr is a pointer to word-aligned cell
Size is the size, in bytes, of the cell

Returns:

None.

Note:

None.

XAtmc_Config Struct Reference

```
#include <xatmc.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xuint8 IpIfDmaConfig](#)

Field Documentation

[Xuint32 XAtmc_Config::](#)

BaseAddress

Base address of
device

[Xuint16 XAtmc_Config::](#)

DeviceId

Unique ID of
device

[Xuint8 XAtmc_Config::](#)

IpIfDmaConfig

IPIF/DMA hardware configuration

The documentation for this struct was generated from the following file:

- [atmc/v1_00_c/src/xatmc.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

packet_fifo/v2_00_a/src/xpacket_fifo_v2_00_a.h

File Reference

Detailed Description

This component is a common component because it's primary purpose is to prevent code duplication in drivers. A driver which must handle a packet FIFO uses this component rather than directly manipulating a packet FIFO.

A FIFO is a device which has dual port memory such that one user may be inserting data into the FIFO while another is consuming data from the FIFO. A packet FIFO is designed for use with packet protocols such as Ethernet and ATM. It is typically only used with devices when DMA and/or Scatter Gather is used. It differs from a nonpacket FIFO in that it does not provide any interrupts for thresholds of the FIFO such that it is less useful without DMA.

Note:

This component has the capability to generate an interrupt when an error condition occurs. It is the user's responsibility to provide the interrupt processing to handle the interrupt. This component provides the ability to determine if that interrupt is active, a deadlock condition, and the ability to reset the FIFO to clear the condition. In this condition, the device which is using the FIFO should also be reset to prevent other problems. This error condition could occur as a normal part of operation if the size of the FIFO is not setup correctly. See the hardware IP specification for more details.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
2.00a	ecm	12/30/02	First release
2.00a	rpm	10/22/03	Created and made use of Level 0 driver
2.00a	rmm	02/24/04	Added WriteDre function.

```
#include "xbasic_types.h"
```

```
#include "xstatus.h"
#include "xpacket_fifo_1_v2_00_a.h"
```

Data Structures

```
struct XPacketFifoV200a
```

Defines

```
#define XPF_V200A_RESET(InstancePtr)
#define XPF_V200A_GET_COUNT(InstancePtr)
#define XPF_V200A_IS_ALMOST_EMPTY(InstancePtr)
#define XPF_V200A_IS_ALMOST_FULL(InstancePtr)
#define XPF_V200A_IS_EMPTY(InstancePtr)
#define XPF_V200A_IS_FULL(InstancePtr)
#define XPF_V200A_IS_DEADLOCKED(InstancePtr)
```

Functions

```
XStatus XPacketFifoV200a_Initialize (XPacketFifoV200a *InstancePtr, Xuint32 RegBaseAddress,
    Xuint32 DataBaseAddress)
XStatus XPacketFifoV200a_SelfTest (XPacketFifoV200a *InstancePtr, Xuint32 FifoType)
XStatus XPacketFifoV200a_Read (XPacketFifoV200a *InstancePtr, Xuint8 *ReadBufferPtr,
    Xuint32 ByteCount)
XStatus XPacketFifoV200a_Write (XPacketFifoV200a *InstancePtr, Xuint8 *WriteBufferPtr,
    Xuint32 ByteCount)
XStatus XPacketFifoV200a_WriteDre (XPacketFifoV200a *InstancePtr, Xuint8 *WriteBufferPtr,
    Xuint32 ByteCount)
```

Define Documentation

```
#define XPF_V200A_GET_COUNT( InstancePtr )
```

Get the occupancy count for a read packet FIFO and the vacancy count for a write packet FIFO. These counts indicate the number of 32-bit words contained (occupancy) in the FIFO or the number of 32-bit words available to write (vacancy) in the FIFO.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

Returns:

The occupancy or vacancy count for the specified packet FIFO.

Note:

Signature: Xuint32 **XPF_V200A_GET_COUNT**(XPacketFifoV200a *InstancePtr)

```
#define XPF_V200A_IS_ALMOST_EMPTY( InstancePtr )
```

Determine if the specified packet FIFO is almost empty. Almost empty is defined for a read FIFO when there is only one data word in the FIFO.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

Returns:

XTRUE if the packet FIFO is almost empty, XFALSE otherwise.

Note:

Signature: Xboolean **XPF_V200A_IS_ALMOST_EMPTY**(XPacketFifoV200a *InstancePtr)

```
#define XPF_V200A_IS_ALMOST_FULL( InstancePtr )
```

Determine if the specified packet FIFO is almost full. Almost full is defined for a write FIFO when there is only one available data word in the FIFO.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

Returns:

XTRUE if the packet FIFO is almost full, XFALSE otherwise.

Note:

Signature: Xboolean **XPF_V200A_IS_ALMOST_FULL**(XPacketFifoV200a *InstancePtr)

```
#define XPF_V200A_IS_DEADLOCKED( InstancePtr )
```

Determine if the specified packet FIFO is deadlocked. This condition occurs when the FIFO is full and empty at the same time and is caused by a packet being written to the FIFO which exceeds the total data capacity of the FIFO. It occurs because of the mark/restore features of the packet FIFO which allow retransmission of a packet. The software should reset the FIFO and any devices using the FIFO when this condition occurs.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

Returns:

XTRUE if the packet FIFO is deadlocked, XFALSE otherwise.

Note:

This component has the capability to generate an interrupt when an error condition occurs. It is the user's responsibility to provide the interrupt processing to handle the interrupt. This function provides the ability to determine if a deadlock condition, and the ability to reset the FIFO to clear the condition.

In this condition, the device which is using the FIFO should also be reset to prevent other problems. This error condition could occur as a normal part of operation if the size of the FIFO is not setup correctly.

Signature: Xboolean **XPF_V200A_IS_DEADLOCKED**(XPacketFifoV200a *InstancePtr)

```
#define XPF_V200A_IS_EMPTY( InstancePtr )
```

Determine if the specified packet FIFO is empty. This applies only to a read FIFO.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

Returns:

XTRUE if the packet FIFO is empty, XFALSE otherwise.

Note:

Signature: Xboolean [XPF_V200A_IS_EMPTY](#)(XPacketFifoV200a *InstancePtr)

```
#define XPF_V200A_IS_FULL( InstancePtr )
```

Determine if the specified packet FIFO is full. This applies only to a write FIFO.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

Returns:

XTRUE if the packet FIFO is full, XFALSE otherwise.

Note:

Signature: Xboolean [XPF_V200A_IS_FULL](#)(XPacketFifoV200a *InstancePtr)

```
#define XPF_V200A_RESET( InstancePtr )
```

Reset the specified packet FIFO. Resetting a FIFO will cause any data contained in the FIFO to be lost.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

Returns:

None.

Note:

Signature: void [XPF_V200A_RESET](#)(XPacketFifoV200a *InstancePtr)

Function Documentation

```
XStatus ( XPacketFifoV200a * InstancePtr,  
XPacketFifoV200a_Initialize RegBaseAddress,  
 RegBaseAddress,  
 Xuint32 DataBaseAddress,  
 Xuint32 DataBaseAddress  
 )
```

This function initializes a packet FIFO. Initialization resets the FIFO such that it's empty and ready to use.

Parameters:

- InstancePtr* contains a pointer to the FIFO to operate on.
- RegBaseAddress* contains the base address of the registers for the packet FIFO.
- DataBaseAddress* contains the base address of the data for the packet FIFO.

Returns:

Always returns XST_SUCCESS.

Note:

None.

```
XStatus ( XPacketFifoV200a * InstancePtr,  
XPacketFifoV200a_Read BufferPtr,  
 BufferPtr,  
 Xuint8 * ByteCount,  
 Xuint32 ByteCount  
 )
```

Read data from a FIFO and puts it into a specified buffer. This function invokes the Level 0 driver function to read the FIFO.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

BufferPtr points to the memory buffer to write the data into. This buffer must be 32 bit aligned or an alignment exception could be generated. Since this buffer is a byte buffer, the data is assumed to be endian independent.

ByteCount contains the number of bytes to read from the FIFO. This number of bytes must be present in the FIFO or an error will be returned.

Returns:

- XST_SUCCESS if the operation was successful
- XST_PFIFO_LACK_OF_DATA if the number of bytes specified by the byte count is not present in the FIFO.

Note:

None.

```
XStatus ( XPacketFifoV200a *
XPacketFifoV200a_SelfTest InstancePtr,
                          Xuint32
                          FifoType
                          )
```

This function performs a self-test on the specified packet FIFO. The self test resets the FIFO and reads a register to determine if it is the correct reset value. This test is destructive in that any data in the FIFO will be lost.

Parameters:

InstancePtr is a pointer to the packet FIFO to be operated on.

FifoType specifies the type of FIFO, read or write, for the self test. The FIFO type is specified by the values XPF_V200A_READ_FIFO_TYPE or XPF_V200A_WRITE_FIFO_TYPE.

Returns:

XST_SUCCESS is returned if the selftest is successful, or XST_PFIFO_BAD_REG_VALUE indicating that the value readback from the occupancy/vacancy count register after a reset does not match the specified reset value.

Note:

None.

```

XStatus ( XPacketFifoV200a *
XPacketFifoV200a_Write InstancePtr,
                        BufferPtr,
                        Xuint8 *,
                        Xuint32
                        ByteCount
                        )

```

Write data into a packet FIFO. This function invokes the Level 0 driver function to read the FIFO.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

BufferPtr points to the memory buffer that data is to be read from and written into the FIFO. Since this buffer is a byte buffer, the data is assumed to be endian independent. This buffer must be 32 bit aligned or an alignment exception could be generated.

ByteCount contains the number of bytes to read from the buffer and to write to the FIFO.

Returns:

- o XST_SUCCESS is returned if the operation succeeded.
- o XST_PFIFO_NO_ROOM is returned if there is not enough room in the FIFO to hold the specified bytes.

Note:

None.

```

XStatus ( XPacketFifoV200a *
XPacketFifoV200a_WriteDre InstancePtr,
                        BufferPtr,
                        Xuint8 *,
                        Xuint32
                        ByteCount
                        )

```

Write data into a packet FIFO configured with the Data Realignment engine (DRE). There are no alignment restrictions. The FIFO can be written on any byte boundary. The FIFO must be at least 32 bits wide.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

BufferPtr points to the memory buffer that data is to be read from and written into the FIFO. Since this buffer is a byte buffer, the data is assumed to be endian independent.

ByteCount contains the number of bytes to read from the buffer and to write to the FIFO.

Returns:

XST_SUCCESS is returned if the operation succeeded. If there is not enough room in the FIFO to hold the specified bytes, XST_PFIFO_NO_ROOM is returned.

Note:

This function assumes that if the device inserting data into the FIFO is a byte device, the order of the bytes in each 32/64 bit word is from the most significant byte to the least significant byte.

packet_fifo/v2_00_a/src/xpacket_fifo_l_v2_00_a.h File Reference

Detailed Description

This header file contains identifiers and low-level (Level 0) driver functions (or macros) that can be used to access the FIFO. High-level driver (Level 1) functions are defined in [xpacket_fifo_v2_00_a.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

2.00a	rpm	10/22/03	First release. Moved most of Level 1 driver functions
-------	-----	----------	---

into this layer.

2.00a	rmm	02/24/04	Added L0WriteDre function.
-------	-----	----------	----------------------------

```
#include "xbasic_types.h"
```

```
#include "xstatus.h"
```

XPacketFifoV200a Struct Reference

```
#include <xpacket_fifo_v2_00_a.h>
```

Detailed Description

The XPacketFifo driver instance data. The driver is required to allocate a variable of this type for every packet FIFO in the device.

The documentation for this struct was generated from the following file:

- [packet_fifo/v2_00_a/src/xpacket_fifo_v2_00_a.h](#)
-

XAtmc_Stats Struct Reference

```
#include <xatmc.h>
```

Detailed Description

ATM controller statistics

Data Fields

[Xuint32 XmitCells](#)

[Xuint32 RecvCells](#)

[Xuint32 RecvUnexpectedHeaders](#)

[Xuint32 RecvShortCells](#)

[Xuint32 RecvLongCells](#)

[Xuint32 RecvHecErrors](#)

[Xuint32 RecvParityErrors](#)

[Xuint32 DmaErrors](#)

[Xuint32 FifoErrors](#)

[Xuint32 RecvInterrupts](#)

[Xuint32 XmitInterrupts](#)

[Xuint32 AtmcInterrupts](#)

Field Documentation

[Xuint32 XAtmc_Stats::](#)

[AtmcInterrupts](#)

Number of ATMC
interrupts

Xuint32 XAtmc_Stats::

DmaErrors

Number of DMA errors since
init

Xuint32 XAtmc_Stats::

FifoErrors

Number of FIFO errors since
init

Xuint32 XAtmc_Stats::

RecvCells

Number of cells
received

Xuint32 XAtmc_Stats::

RecvHecErrors

Number of HEC
errors

Xuint32 XAtmc_Stats::

RecvInterrupts

Number of receive
interrupts

Xuint32 XAtmc_Stats::

RecvLongCells

Number of long
cells

Xuint32 XAtmc_Stats::

RecvParityErrors

Number of parity
errors

Xuint32 XAtmc_Stats::

RecvShortCells

Number of short
cells

Xuint32 XAtmc_Stats::

RecvUnexpectedHeaders

Number of cells with unexpected
headers

Xuint32 XAtmc_Stats::

XmitCells

Number of cells
transmitted

Xuint32 XAtmc_Stats::

XmitInterrupts

Number of transmit
interrupts

The documentation for this struct was generated from the following file:

- [atmc/v1_00_c/src/xatmc.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

atmc/v1_00_c/src/xatmc_cfg.c File Reference

Detailed Description

Functions in this file handle configuration (including initialization, reset, and self-test) of the Xilinx ATM driver component.

Note:

None.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	
1.00a	JHL	07/31/01	First release
1.00b	rpm	12/12/02	Changed location of IsStarted assignment in XAtmc_Start to be sure the flag is set before the device and interrupts are enabled.
1.00c	rpm	01/08/03	New release supports v2.00a of packet fifo driver an v1.23b of the IPIF driver

```
#include "xatmc.h"  
#include "xatmc_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Data Structures

struct **Mapping**

Functions

XStatus **XAtmc_Initialize** (**XAtmc** *InstancePtr, **Xuint16** DeviceId)

XStatus **XAtmc_Start** (**XAtmc** *InstancePtr)

XStatus **XAtmc_Stop** (**XAtmc** *InstancePtr)

void **XAtmc_Reset** (**XAtmc** *InstancePtr)

XStatus **XAtmc_SelfTest** (**XAtmc** *InstancePtr)

XStatus **XAtmc_SetOptions** (**XAtmc** *InstancePtr, **Xuint32** OptionsFlag)

Xuint32 **XAtmc_GetOptions** (**XAtmc** *InstancePtr)

XStatus **XAtmc_SetPhyAddress** (**XAtmc** *InstancePtr, **Xuint8** Address)

Xuint8 **XAtmc_GetPhyAddress** (**XAtmc** *InstancePtr)

XStatus **XAtmc_SetHeader** (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint32** Header)

Xuint32 **XAtmc_GetHeader** (**XAtmc** *InstancePtr, **Xuint32** Direction)

XStatus **XAtmc_SetUserDefined** (**XAtmc** *InstancePtr, **Xuint8** UserDefined)

Xuint8 **XAtmc_GetUserDefined** (**XAtmc** *InstancePtr)

XStatus **XAtmc_SetPktThreshold** (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint8** Threshold)

XStatus **XAtmc_GetPktThreshold** (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint8** *ThreshPtr)

XStatus **XAtmc_SetPktWaitBound** (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint32** TimerValue)

XStatus **XAtmc_GetPktWaitBound** (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint32** *WaitPtr)

void **XAtmc_GetStats** (**XAtmc** *InstancePtr, **XAtmc_Stats** *StatsPtr)

void **XAtmc_ClearStats** (**XAtmc** *InstancePtr)

XStatus **XAtmc_SetSgRecvSpace** (**XAtmc** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)

XStatus **XAtmc_SetSgSendSpace** (**XAtmc** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)

void **XAtmc_SetSgRecvHandler** (**XAtmc** *InstancePtr, void *CallbackRef, **XAtmc_SgHandler** FuncPtr)

void **XAtmc_SetSgSendHandler** (**XAtmc** *InstancePtr, void *CallbackRef, **XAtmc_SgHandler** FuncPtr)

void **XAtmc_SetErrorHandler** (**XAtmc** *InstancePtr, void *CallbackRef, **XAtmc_ErrorHandler** FuncPtr)

XAtmc_Config * **XAtmc_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
void ( XAtmc * )  
XAtmc_ClearStats InstancePtr
```

Clears the [XAtmc_Stats](#) structure for this driver.

Parameters:

InstancePtr is a pointer to the [XAtmc](#) instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint32 ( XAtmc *  
XAtmc_GetHeader InstancePtr,  
Xuint32 Direction  
)
```

Gets the send or receive ATM header in the ATM controller. The ATM controller attaches the send header to cells which are to be sent but contain only the payload.

If the ATM controller is configured appropriately, it will compare the header of received cells against the receive header and discard cells which don't match in the VCI and VPI fields of the header.

Parameters:

InstancePtr is a pointer to the [XAtmc](#) instance to be worked on.

Direction indicates whether we're retrieving the send header or the receive header.

Returns:

The ATM header currently being used by the ATM controller for attachment to transmitted cells or the header which is being compared against received cells. An invalid specified direction will cause this function to return a value of 0.

Note:

None.

Xuint32 (**XAtmc ***)

XAtmc_GetOptions *InstancePtr*

Gets Atmc driver/device options. The value returned is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

The 32-bit value of the Atmc options. The value is a bit-mask representing all options that are currently enabled. See **xatmc.h** for a detailed description of the options.

Note:

None.

Xuint8 (**XAtmc ***)

XAtmc_GetPhyAddress *InstancePtr*

Gets the PHY address for this driver/device.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

The 5-bit PHY address (0 - 31) currently being used by the ATM controller.

Note:

None.

XStatus (**XAtmc ***

XAtmc_GetPktThreshold *InstancePtr,*

Xuint32 *Direction,*

Xuint8 * *ThreshPtr*

)

Gets the value of the packet threshold register for this driver/device. The packet threshold is used for interrupt coalescing when the ATM controller is configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

ThreshPtr is a pointer to the byte into which the current value of the packet threshold register will be copied. An output parameter. A value of 0 indicates the use of packet threshold by the hardware is disabled.

Returns:

- XST_SUCCESS if the packet threshold was retrieved successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
XStatus ( XAtmc *  
XAtmc_GetPktWaitBound InstancePtr,  
 Xuint32 Direction,  
 Xuint32 * WaitPtr  
 )
```

Gets the packet wait bound register for this driver/device. The packet wait bound is used for interrupt coalescing when the ATM controller is configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

WaitPtr is a pointer to the byte into which the current value of the packet wait bound register will be copied. An output parameter. Units are in milliseconds in the range 0 - 1023. A value of 0 indicates the packet wait bound timer is disabled.

Returns:

- XST_SUCCESS if the packet wait bound was retrieved successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
void  
XAtmc_GetStats ( XAtmc *  
                InstancePtr,  
                XAtmc_Stats * StatsPtr  
                )
```

Gets a copy of the [XAtmc_Stats](#) structure, which contains the current statistics for this driver.

Parameters:

InstancePtr is a pointer to the [XAtmc](#) instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None. Although the output parameter will contain a copy of the statistics upon return from this function.

Note:

None.

```
Xuint8  
XAtmc_GetUserDefined ( XAtmc *  
                      InstancePtr
```

Gets the 2nd byte of the User Defined data in the ATM controller for the channel which is sending data. The ATM controller will attach the header to all cells which are being sent and do not have a header. The header of a 16 bit Utopia interface contains the User Defined data which is two bytes. The first byte contains the HEC field and the second byte is available for user data. This function only allows the second byte to be retrieved.

Parameters:

InstancePtr is a pointer to the [XAtmc](#) instance to be worked on.

Returns:

The second byte of the User Defined data.

Note:

None.

```
XStatus ( XAtmc *  
XAtmc_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific ATM controller instance/driver. The initialization entails:

- Initialize fields of the **XAtmc** structure
- Clear the ATM statistics for this device
- Initialize the IPIF component with its register base address
- Configure the FIFO components with their register base addresses.
- Configure the DMA channel components with their register base addresses. At some later time, memory pools for the scatter-gather descriptor lists will be passed to the driver.
- Reset the ATM controller

The only driver function that should be called before this Initialize function is called is GetInstance.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XAtmc** instance. Passing in a device id associates the generic **XAtmc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started

Note:

None.

```
XAtmc_Config* ( Xuint16 )  
XAtmc_LookupConfig DeviceId
```

Looks up the device configuration based on the unique device ID. The table `AtmcConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId contains the unique device ID that for the device. This ID is used to lookup the configuration.

Returns:

A pointer to the configuration for the specified device, or `XNULL` if the device could not be found.

Note:

None.

```
void ( XAtmc * )  
XAtmc_Reset InstancePtr
```

Resets the ATM controller. It resets the the DMA channels, the FIFOs, and the ATM controller. The reset does not remove any of the buffer descriptors from the scatter-gather list for DMA. Reset must only be called after the driver has been initialized.

The configuration after this reset is as follows:

- Disabled transmitter and receiver
- Default packet threshold and packet wait bound register values for scatter-gather DMA operation
- PHY address of 0

The upper layer software is responsible for re-configuring (if necessary) and restarting the ATM controller after the reset.

When a reset is required due to an internal error, the driver notifies the upper layer software of this need through the `ErrorHandler` callback and specific status codes. The upper layer software is responsible for calling this `Reset` function and then re-configuring the device.

Parameters:

InstancePtr is a pointer to the `XAtmc` instance to be worked on.

Returns:

None.

Note:

The reset is accomplished by setting the IPIF reset register. This takes care of resetting all hardware blocks, including the ATM controller.

XStatus (**XAtmc** *)
XAtmc_SelfTest *InstancePtr*

Performs a self-test on the ATM controller device. The test includes:

- Run self-test on DMA channel, FIFO, and IPIF components
- Reset the ATM controller device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run.

It should be noted that data caching must be disabled when this function is called because the DMA self-test uses two local buffers (on the stack) for the transfer test.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

- XST_SUCCESS if self-test was successful
- XST_PFIPO_BAD_REG_VALUE if the FIFO failed register self-test
- XST_DMA_TRANSFER_ERROR if DMA failed data transfer self-test
- XST_DMA_RESET_REGISTER_ERROR if DMA control register value was incorrect after a reset
- XST_REGISTER_ERROR if the ATM controller failed register reset test
- XST_LOOPBACK_ERROR if the ATM controller internal loopback failed
- XST_IPIF_REG_WIDTH_ERROR if an invalid register width was passed into the function
- XST_IPIF_RESET_REGISTER_ERROR if the value of a register at reset was invalid
- XST_IPIF_DEVICE_STATUS_ERROR if a write to the device status register did not read back correctly
- XST_IPIF_DEVICE_ACK_ERROR if a bit in the device status register did not reset when acked
- XST_IPIF_DEVICE_ENABLE_ERROR if the device interrupt enable register was not updated correctly by the hardware when other registers were written to
- XST_IPIF_IP_STATUS_ERROR if a write to the IP interrupt status register did not

- read back correctly
- XST_IPIF_IP_ACK_ERROR if one or more bits in the IP status register did not reset when acked
- XST_IPIF_IP_ENABLE_ERROR if the IP interrupt enable register was not updated correctly when other registers were written to

Note:

Because this test uses the PollSend function for its loopback testing, there is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the self-test thread.

```

void
XAtmc_SetErrorHandler ( XAtmc *           InstancePtr,
                       void *           CallbackRef,
                       XAtmc_ErrorHandler FuncPtr
                       )

```

Sets the callback function for handling errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback which should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- XST_DMA_ERROR indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- XST_FIFO_ERROR indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- XST_RESET_ERROR indicates an unrecoverable ATM controller error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- XST_ATMC_ERROR_COUNT_MAX indicates the counters of the ATM controller have reached the maximum value and that the statistics of the ATM controller should be cleared.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus ( XAtmc *  
XAtmc_SetHeader InstancePtr,  
 Xuint32 Direction,  
 Xuint32 Header  
 )
```

Sets the send or receive ATM header in the ATM controller. If cells with only payloads are given to the controller to be sent, it will attach the header to the cells. If the ATM controller is configured appropriately, it will compare the header of received cells against the receive header and discard cells which don't match in the VCI and VPI fields of the header.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the direction, send(transmit) or receive, for the header to set.

Header contains the ATM header to be attached to each transmitted cell for cells with only payloads or the expected header for cells which are received.

Returns:

- XST_SUCCESS if the PHY address was set successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been stopped
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
XStatus ( XAtmc *  
XAtmc_SetOptions InstancePtr,  
 Xuint32 OptionsFlag  
 )
```

Set Atmc driver/device options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option. A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off. See [xatmc.h](#) for a detailed description of the available options.

Parameters:

InstancePtr is a pointer to the [XAtmc](#) instance to be worked on.

OptionsFlag is a bit-mask representing the Atmc options to turn on or off

Returns:

- XST_SUCCESS if options were set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

```
XStatus ( XAtmc *  
XAtmc_SetPhyAddress InstancePtr,  
 Xuint8 Address  
 )
```

Sets the PHY address for this driver/device. The address is a 5-bit value. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the [XAtmc](#) instance to be worked on.

Address contains the 5-bit PHY address (0 - 31).

Returns:

- XST_SUCCESS if the PHY address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```

XStatus ( XAtmc *
XAtmc_SetPktThreshold InstancePtr,
Xuint32 Direction,
Xuint8 Threshold
)

```

Sets the packet count threshold register for this driver/device. The device must be stopped before setting the threshold. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

Threshold is the value of the packet threshold count used during interrupt coalescing. A value of 0 disables the use of packet threshold by the hardware.

Returns:

- XST_SUCCESS if the threshold was successfully set
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_DMA_SG_COUNT_EXCEEDED if the threshold must be equal to or less than the number of descriptors in the list
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```

XStatus ( XAtmc *
XAtmc_SetPktWaitBound InstancePtr,
Xuint32 Direction,
Xuint32 TimerValue
)

```

Sets the packet wait bound register for this driver/device. The device must be stopped before setting the timer value. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold.

The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

TimerValue is the value of the packet wait bound used during interrupt coalescing. It is in milliseconds in the range 0 - 1023. A value of 0 disables the packet wait bound timer.

Returns:

- XST_SUCCESS if the packet wait bound was set successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
void XAtmc_SetSgRecvHandler ( XAtmc * InstancePtr, void * CallbackRef, XAtmc_SgHandler FuncPtr )
```

Sets the callback function for handling received cells in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of cells, determined by the DMA scatter-gather packet threshold, are received. The number of received cells is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each received cell from the list and should attach a new buffer to each descriptor. It is important that the specified number of cells passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XAtmc_SetSgRecvSpace ( XAtmc *  
                                InstancePtr,  
                                Xuint32 *  
                                MemoryPtr,  
                                Xuint32 ByteCount  
                                )
```

Gives the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the Atmc driver. The memory space must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the XAtmc_Initialize function because the DMA channel components must be initialized before the memory space is set.

```
void XAtmc_SetSgSendHandler ( XAtmc * InstancePtr, void * CallbackRef, XAtmc_SgHandler FuncPtr )
```

Sets the callback function for handling confirmation of transmitted cells in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of cells, determined by the DMA scatter-gather packet threshold, are sent. The number of sent cells is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each sent cell from the list and should also free the buffers attached to the descriptors if necessary. It is important that the specified number of cells passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XAtmc_SetSgSendSpace(XAtmc *InstancePtr,  
Xuint32 *MemoryPtr,  
Xuint32 ByteCount)
```

Gives the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the Atmc driver. The memory space must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the XAtmc_Initialize function because the DMA channel components must be initialized before the memory space is set.

```
XStatus ( XAtmc *  
XAtmc_SetUserDefined InstancePtr,  
Xuint8 UserDefined  
)
```

Sets the 2nd byte of the User Defined data in the ATM controller for the channel which is sending data. The ATM controller will attach the header to all cells which are being sent and do not have a header. The header of a 16 bit Utopia interface contains the User Defined data which is two bytes. The first byte contains the HEC field and the second byte is available for user data. This function only allows the second byte to be set.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

UserDefined contains the second byte of the User Defined data.

Returns:

- XST_SUCCESS if the user-defined data was set successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been stopped

Note:

None.

```
XStatus ( XAtmc *  
XAtmc_Start InstancePtr  
)
```

Starts the ATM controller as follows:

- If not in polled mode enable interrupts
- Enable the transmitter
- Enable the receiver
- Start the DMA channels if the descriptor lists are not empty

It is necessary for the caller to connect the interrupt service routine of the ATM controller to the interrupt source, typically an interrupt controller, and enable the interrupt in the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully

- XST_DEVICE_IS_STARTED if the device is already started
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.
- XST_DMA_SG_LIST_EMPTY iff configured for scatter-gather DMA and no buffer descriptors have been put into the list for the receive channel.

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XStatus (**XAtmc** *)
XAtmc_Stop *InstancePtr*

Stops the ATM controller as follows:

- Stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver
- Disable interrupts if not in polled mode

It is the callers responsibility to disconnect the interrupt handler of the ATM controller from the interrupt source, typically an interrupt controller, and disable the interrupt in the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to stop the device, the user is required to provide protection of this shared data (typically using a semaphore).

atmc/v1_00_c/src/xatmc_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of ATMC devices in the system. Each ATMC device should have an entry in this table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	

1.00a	JHL	07/31/01	First release
1.00b	rpm	05/01/02	Condensed base addresses into one
1.00c	rpm	01/08/03	New release supports v2.00a of packet fifo driver
			an v1.23b of the IPIF driver

```
#include "xatmc.h"  
#include "xparameters.h"
```

Variables

[XAtmc_Config](#) [XAtmc_ConfigTable](#) [XPAR_XATMC_NUM_INSTANCES]

Variable Documentation

[XAtmc_Config](#) [XAtmc_ConfigTable](#)
[XPAR_XATMC_NUM_INSTANCES]

This table contains configuration information for each ATMC device in the system.

Generated on 24 Jun 2004 for Xilinx Device Drivers

atmc/v1_00_c/src/xatmc_l.c File Reference

Detailed Description

This file contains low-level polled functions to send and receive ATM cells.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	05/01/02	First release

```
#include "xatmc_l.h"
```

Functions

```
void XAtmc_SendCell (Xuint32 BaseAddress, Xuint8 *CellPtr, int Size)  
int XAtmc_RecvCell (Xuint32 BaseAddress, Xuint8 *CellPtr, Xuint32 *CellStatusPtr)
```

Function Documentation

```
int  
XAtmc_RecvCell ( Xuint32  
                 BaseAddress,  
                 Xuint8 * CellPtr,  
                 Xuint32 * CellStatusPtr  
                )
```

Receive a cell. Wait for a cell to arrive.

Parameters:

BaseAddress is the base address of the device

CellPtr is a pointer to a word-aligned buffer where the cell will be stored.

CellStatusPtr is a pointer to a cell status that will be valid after this function returns.

Returns:

The size, in bytes, of the cell received.

Note:

None.

```
void XAtmc_SendCell ( Xuint32 BaseAddress,  
                    Xuint8 * CellPtr,  
                    int Size  
                    )
```

Send an ATM cell. This function blocks waiting for the cell to be transmitted.

Parameters:

BaseAddress is the base address of the device

CellPtr is a pointer to word-aligned cell

Size is the size, in bytes, of the cell

Returns:

None.

Note:

None.

ddr/v1_00_b/src/xddr_stats.c File Reference

Detailed Description

The implementation of the [XDdr](#) component's functionality that is related to statistics. See [xddr.h](#) for more information about the component.

Note:

None

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	jhl	12/24/03	First release

```
#include "xddr.h"
```

Functions

```
void XDdr\_GetStats (XDdr *InstancePtr, XDdr\_Stats *StatsPtr)
```

```
void XDdr\_ClearStats (XDdr *InstancePtr)
```

Function Documentation

```
void  
XDdr\_ClearStats ( XDdr *  
InstancePtr )
```

Clear the statistics of the DDR device including the Single Error Count, Double Error Count, and Parity Field Error Count. The counts are all contained in registers of the DDR device.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Returns:

None.

Note:

None.

```
void          ( XDdr *
XDdr_GetStats      InstancePtr,
                  XDdr_Stats * StatsPtr
                  )
```

Get the statistics of the DDR device including the Single Error Count, Double Error Count, and Parity Field Error Count. The counts are all contained in registers of the DDR device.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

StatsPtr contains a pointer to a **XDdr_Stats** data type. The function puts the statistics of the device into the specified data structure.

Returns:

The statistics data type pointed to by input StatsPtr is modified.

Note:

None.

XDdr Struct Reference

```
#include <xddr.h>
```

Detailed Description

The XDdr driver instance data. The user is required to allocate a variable of this type for every DDR device in the system. A pointer to a variable of this type is then passed to the driver API functions.

Data Fields

[Xuint32 BaseAddress](#)

[Xuint32 IsReady](#)

Field Documentation

Xuint32 XDdr::

BaseAddress

Base address of registers

Xuint32 XDdr::

IsReady

Device is initialized and ready

The documentation for this struct was generated from the following file:

- [ddr/v1_00_b/src/xddr.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

ddr/v1_00_b/src/xddr.h File Reference

Detailed Description

This header file contains interface for the DDR device driver. This device driver is not necessary for the DDR device unless ECC is being used.

ECC (Error Correction Code) is a mode that detects and corrects some memory errors. This device driver provides the following abilities.

- Enable and disable ECC mode
- Enable and disable interrupts for specific ECC errors
- Error injection for testing of ECC mode
- Statistics for specific ECC errors detected

The Xilinx DDR controller is a soft IP core designed for Xilinx FPGAs on the OPB or PLB bus. The OPB DDR device does not currently support ECC such that there would be no reason to use this driver for the device.

Hardware Parameters Needed

In order for the driver to be used with the hardware device, ECC registers must be enabled in the hardware.

The interrupt capability for the device must be enabled in the hardware if interrupts are to be used with the driver. The interrupt functions of the device driver will assert when called if interrupt support is not present in the hardware.

The ability to force errors is a test mode and it must be enabled in the hardware if the control register is to be used to force ECC errors.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a jhl 12/24/03 First release

```
#include "xddr_1.h"  
#include "xbasic_types.h"  
#include "xio.h"  
#include "xstatus.h"
```

Data Structures

```
struct XDdr  
struct XDdr_Config  
struct XDdr_Stats
```

Functions

XStatus XDdr_Initialize (**XDdr** *InstancePtr, **Xuint16** DeviceId)

void **XDdr_EnableEcc** (**XDdr** *InstancePtr)

void **XDdr_DisableEcc** (**XDdr** *InstancePtr)

void **XDdr_SetControl** (**XDdr** *InstancePtr, **Xuint32** Control)

Xuint32 XDdr_GetControl (**XDdr** *InstancePtr)

Xuint32 XDdr_GetStatus (**XDdr** *InstancePtr)

void **XDdr_ClearStatus** (**XDdr** *InstancePtr)

XDdr_Config * **XDdr_LookupConfig** (**Xuint16** DeviceId)

void **XDdr_GetStats** (**XDdr** *InstancePtr, **XDdr_Stats** *StatsPtr)

void **XDdr_ClearStats** (**XDdr** *InstancePtr)

XStatus XDdr_SelfTest (**XDdr** *InstancePtr)

void **XDdr_InterruptGlobalEnable** (**XDdr** *InstancePtr)

void **XDdr_InterruptGlobalDisable** (**XDdr** *InstancePtr)

void **XDdr_InterruptEnable** (**XDdr** *InstancePtr, **Xuint32** Mask)

void **XDdr_InterruptDisable** (**XDdr** *InstancePtr, **Xuint32** Mask)

void **XDdr_InterruptClear** (**XDdr** *InstancePtr, **Xuint32** Mask)

Xuint32 XDdr_InterruptGetEnabled (**XDdr** *InstancePtr)

Xuint32 XDdr_InterruptGetStatus (**XDdr** *InstancePtr)

Function Documentation

```
void ( XDdr * )  
XDdr_ClearStats InstancePtr
```

Clear the statistics of the DDR device including the Single Error Count, Double Error Count, and Parity Field Error Count. The counts are all contained in registers of the DDR device.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XDdr * )  
XDdr_ClearStatus InstancePtr
```

Clear the ECC Status Register contents of the DDR device. This function can be used to clear errors in the status that have been processed.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XDdr * )  
XDdr_DisableEcc InstancePtr
```

Disable the ECC mode for both read and write operations in the DDR ECC device.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XDdr * )  
XDdr_EnableEcc InstancePtr
```

Enable the ECC mode for both read and write operations in the DDR ECC device.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint32 ( XDdr * )  
XDdr_GetControl InstancePtr
```

Get the ECC Control Register contents of the DDR device. This function can be used to determine which features are enabled in the device.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Returns:

The value read from the register which consists of constants named XDDR_ECCCR* for each bit field as specified in [xddr_1.h](#).

Note:

None.

```
void  
XDDR_GetStats ( XDDR *  
               InstancePtr,  
               XDDR_Stats * StatsPtr  
             )
```

Get the statistics of the DDR device including the Single Error Count, Double Error Count, and Parity Field Error Count. The counts are all contained in registers of the DDR device.

Parameters:

InstancePtr is a pointer to an **XDDR** instance to be worked on.

StatsPtr contains a pointer to a **XDDR_Stats** data type. The function puts the statistics of the device into the specified data structure.

Returns:

The statistics data type pointed to by input StatsPtr is modified.

Note:

None.

```
Xuint32  
XDDR_GetStatus ( XDDR *  
                InstancePtr
```

Get the ECC Status Register contents of the DDR device. This function can be used to determine which errors have occurred for ECC mode.

Parameters:

InstancePtr is a pointer to an **XDDR** instance to be worked on.

Returns:

The value read from the register which consists of constants named XDDR_ECCSR* for each bit field as specified in [xDDR_1.h](#).

None.

Note:

None.

```
XStatus ( XDdr *  
XDdr_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize the **XDdr** instance provided by the caller based on the given DeviceID.

Nothing is done except to initialize the InstancePtr.

Parameters:

InstancePtr is a pointer to an **XDdr** instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the **XDdr** API must be made with this pointer.

DeviceId is the unique id of the device controlled by this **XDdr** component. Passing in a device id associates the generic **XDdr** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS Initialization was successful.
- XST_DEVICE_NOT_FOUND Device configuration data was not found for a device with the supplied device ID.

Note:

None.

```
void ( XDdr *  
XDdr_InterruptClear InstancePtr,  
 Xuint32 Mask  
 )
```

Clear pending interrupts with the provided mask. An interrupt must be cleared after software has serviced it or it can cause another interrupt.

Parameters:

InstancePtr is the DDR component to operate on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XDDR_IPIXR_*_MASK.

Note:

This function will assert if the hardware device has not been built with interrupt capabilities.

```
void XDDR_InterruptDisable ( XDdr * InstancePtr, Xuint32 Mask )
```

Disable ECC interrupts so that ECC errors will not cause an interrupt.

Parameters:

InstancePtr is the DDR component to operate on.

Mask is the mask to disable. Bits set to 1 are disabled. The mask is formed by OR'ing bits from XDDR_IPIXR_*_MASK.

Note:

This function will assert if the hardware device has not been built with interrupt capabilities.

```
void XDDR_InterruptEnable ( XDdr * InstancePtr, Xuint32 Mask )
```

Enable ECC interrupts so that specific ECC errors will cause an interrupt. The function `XDdr_InterruptGlobalEnable` must also be called to enable any interrupt to occur.

Parameters:

InstancePtr is the DDR component to operate on.

Mask is the mask to enable. Bit positions of 1 are enabled. The mask is formed by OR'ing bits from `XDDR_IPIXR_*_MASK`.

Note:

This function will assert if the hardware device has not been built with interrupt capabilities.

Xuint32 (**XDdr ***)
XDdr_InterruptGetEnabled *InstancePtr*

Returns the interrupt enable mask as set by `XDdr_InterruptEnable()` which indicates which ECC interrupts are enabled or disabled.

Parameters:

InstancePtr is the DDR component to operate on.

Returns:

Mask of bits made from `XDDR_IPIXR_*_MASK`.

Note:

This function will assert if the hardware device has not been built with interrupt capabilities.

Xuint32 (**XDdr ***)
XDdr_InterruptGetStatus *InstancePtr*

Returns the status of interrupts which indicates which ECC interrupts are pending.

Parameters:

InstancePtr is the DDR component to operate on.

Returns:

Mask of bits made from XDDR_IPIXR_*_MASK.

Note:

The interrupt status indicates the status of the device irregardless if the interrupts from the devices have been enabled or not through [XDDR_InterruptEnable\(\)](#).

This function will assert if the hardware device has not been built with interrupt capabilities.

```
void ( XDDR * )  
XDDR_InterruptGlobalDisable InstancePtr
```

Disable the core's interrupt output signal. Interrupts enabled through [XDDR_InterruptEnable\(\)](#) will not occur until the global enable bit is set by [XDDR_InterruptGlobalEnable\(\)](#). This function is designed to allow all interrupts to be disabled easily for entering a critical section.

Parameters:

InstancePtr is the DDR component to operate on.

Note:

This function will assert if the hardware device has not been built with interrupt capabilities.

```
void ( XDDR * )  
XDDR_InterruptGlobalEnable InstancePtr
```

Enable the core's interrupt output signal. Interrupts enabled through [XDDR_InterruptEnable\(\)](#) will not occur until the global enable bit is set by this function. This function is designed to allow all interrupts to be enabled easily for exiting a critical section.

Parameters:

InstancePtr is the DDR component to operate on.

Note:

This function will assert if the hardware device has not been built with interrupt capabilities.

XDdr_Config* (**Xuint16**)
XDdr_LookupConfig *DeviceId*

Lookup the device configuration based on the unique device ID. The table XDdr_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceID is the device identifier to lookup.

Returns:

- **XDdr** configuration structure pointer if DeviceID is found.
- XNULL if DeviceID is not found.

Note:

None.

XStatus (**XDdr ***)
XDdr_SelfTest *InstancePtr*

Perform a self-test on the DDR device. Self-test will read, write and verify that some of the registers of the device are functioning correctly. This function will restore the state of the device to state it was in prior to the function call.

Parameters:

InstancePtr is the DDR component to operate on.

Returns:

- XST_SUCCESS The self-test was successful.
- XST_FAILURE The self-test failed.

Note:

None

void (**XDdr ***
XDdr_SetControl *InstancePtr,*
Xuint32 *Control*
)

Set the ECC Control Register of the DDR device to the specified value. This function can be used to individually enable/disable read or write ECC and force specific types of ECC errors to occur.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Control contains the value to be written to the register and consists of constants named XDDR_ECCCR* for each bit field as specified in **xddr_1.h**.

Returns:

None.

Note:

None.

ddr/v1_00_b/src/xddr_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions for the DDR device driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	jhl	12/24/03	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

Register offsets

```
#define XDDR_ECCCR_OFFSET  
#define XDDR_ECCSR_OFFSET  
#define XDDR_ECCSEC_OFFSET  
#define XDDR_ECCDEC_OFFSET  
#define XDDR_ECCPEC_OFFSET  
#define XDDR_ECCSETR_OFFSET  
#define XDDR_ECCPETR_OFFSET  
#define XDDR_IPIF_OFFSET  
#define XDDR_DGIE_OFFSET  
#define XDDR_IPISR_OFFSET  
#define XDDR_IPIER_OFFSET
```

ECC Control Register bitmaps and masks

```
#define XDDR_ECCCR_FORCE_PE_MASK  
#define XDDR_ECCCR_FORCE_DE_MASK  
#define XDDR_ECCCR_FORCE_SE_MASK  
#define XDDR_ECCCR_RE_MASK  
#define XDDR_ECCCR_WE_MASK
```

ECC Status Register bitmaps and masks

```
#define XDDR_ECCSR_SE_SYND_MASK  
#define XDDR_ECCSR_PE_MASK  
#define XDDR_ECCSR_DE_MASK  
#define XDDR_ECCSR_SE_MASK
```

Device Global Interrupt Enable Register bitmaps and masks

Bit definitions for the global interrupt enable register.

```
#define XDDR_DGIE_GIE_MASK
```

Interrupt Status and Enable Register bitmaps and masks

Bit definitions for the interrupt status register and interrupt enable registers.

```
#define XDDR_IPIXR_PE_IX_MASK  
#define XDDR_IPIXR_DE_IX_MASK  
#define XDDR_IPIXR_SE_IX_MASK
```

Defines

```
#define XDDR_mWriteReg(BaseAddress, RegOffset, Data)  
#define XDDR_mReadReg(BaseAddress, RegOffset)
```

Define Documentation

```
#define  
XDDR_DGIE_GIE_MASK
```

Global interrupt
enable

```
#define  
XDDR_DGIE_OFFSET
```

Device Global Interrupt Enable
Register

```
#define  
XDDR_ECCCR_FORCE_DE_MASK
```

Force double bit
error

```
#define  
XDDR_ECCCR_FORCE_PE_MASK
```

Force parity
error

```
#define  
XDDR_ECCCR_FORCE_SE_MASK
```

Force single bit
error

```
#define  
XDDR_ECCCR_OFFSET
```

Control
Register

```
#define  
XDDR_ECCCR_RE_MASK
```

ECC read
enable

```
#define  
XDDR_ECCCR_WE_MASK
```

ECC write
enable

```
#define  
XDDR_ECCDEC_OFFSET
```

Double Error Count
Register

```
#define  
XDDR_ECCPEC_OFFSET
```

Parity Field Error Count
Register

```
#define  
XDDR_ECCPETR_OFFSET
```

Parity Error Interrupt Threshold
Register

```
#define  
XDDR_ECCSEC_OFFSET
```

Single Error Count
Register

```
#define  
XDDR_ECCSETR_OFFSET
```

Single Error Interrupt Threshold
Register

```
#define  
XDDR_ECCSR_DE_MASK
```

Double bit
error

```
#define  
XDDR_ECCSR_OFFSET
```

Status
Register

```
#define  
XDDR_ECCSR_PE_MASK
```

Parity field bit
error

```
#define  
XDDR_ECCSR_SE_MASK
```

Single bit
error

```
#define  
XDDR_ECCSR_SE_SYND_MASK
```

Single bit error
syndrome

```
#define  
XDDR_IPIER_OFFSET
```

IP Interrupt Enable
Register

```
#define  
XDDR_IPIF_OFFSET
```

IPIF
Registers

```
#define  
XDDR_IPISR_OFFSET
```

IP Interrupt Status
Register

```
#define  
XDDR_IPIXR_DE_IX_MASK
```

Double bit error
interrupt

Write a value to a DDR register. A 32 bit write is performed.

Parameters:

BaseAddress is the base address of the DDR device.

RegOffset is the register offset from the base to write to.

Data is the data written to the register.

Returns:

None.

Note:

None.

C-style signature: void XDdr_mWriteReg(Xuint32 BaseAddress, unsigned RegOffset, Xuint32 Data)

Generated on 24 Jun 2004 for Xilinx Device Drivers

XDdr_Stats Struct Reference

```
#include <xddr.h>
```

Detailed Description

The **XDdr** driver stats data. A pointer to a variable of this type is passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [ddr/v1_00_b/src/xddr.h](#)
-

ddr/v1_00_b/src/xddr.c File Reference

Detailed Description

The implementation of the **XDdr** component's basic functionality. See [xddr.h](#) for more information about the component.

Note:

None

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	jhl	12/24/03	First release

```
#include "xparameters.h"  
#include "xddr.h"  
#include "xstatus.h"
```

Functions

```
XStatus XDdr_Initialize (XDdr *InstancePtr, Xuint16 DeviceId)  
XDdr_Config * XDdr_LookupConfig (Xuint16 DeviceId)  
void XDdr_EnableEcc (XDdr *InstancePtr)  
void XDdr_DisableEcc (XDdr *InstancePtr)  
void XDdr_SetControl (XDdr *InstancePtr, Xuint32 Control)  
Xuint32 XDdr_GetControl (XDdr *InstancePtr)  
Xuint32 XDdr_GetStatus (XDdr *InstancePtr)  
void XDdr_ClearStatus (XDdr *InstancePtr)
```

Function Documentation

```
void ( XDdr * )  
XDdr_ClearStatus InstancePtr
```

Clear the ECC Status Register contents of the DDR device. This function can be used to clear errors in the status that have been processed.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XDdr * )  
XDdr_DisableEcc InstancePtr
```

Disable the ECC mode for both read and write operations in the DDR ECC device.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XDdr * )  
XDdr_EnableEcc InstancePtr
```

Enable the ECC mode for both read and write operations in the DDR ECC device.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Returns:

None.

Note:

None.

Xuint32 (**XDdr ***)
XDdr_GetControl *InstancePtr*

Get the ECC Control Register contents of the DDR device. This function can be used to determine which features are enabled in the device.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Returns:

The value read from the register which consists of constants named XDDR_ECCCR* for each bit field as specified in **xddr_1.h**.

Note:

None.

Xuint32 (**XDdr ***)
XDdr_GetStatus *InstancePtr*

Get the ECC Status Register contents of the DDR device. This function can be used to determine which errors have occurred for ECC mode.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Returns:

The value read from the register which consists of constants named XDDR_ECCSR* for each bit field as specified in [xddr_l.h](#).

None.

Note:

None.

```
XStatus          ( XDdr *  
XDdr_Initialize      InstancePtr,  
                      Xuint16 DeviceId  
                      )
```

Initialize the **XDdr** instance provided by the caller based on the given DeviceID.

Nothing is done except to initialize the InstancePtr.

Parameters:

InstancePtr is a pointer to an **XDdr** instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the **XDdr** API must be made with this pointer.

DeviceId is the unique id of the device controlled by this **XDdr** component. Passing in a device id associates the generic **XDdr** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS Initialization was successful.
- XST_DEVICE_NOT_FOUND Device configuration data was not found for a device with the supplied device ID.

Note:

None.

```
XDdr_Config*                ( Xuint16                )  
XDdr_LookupConfig         DeviceId
```

Lookup the device configuration based on the unique device ID. The table XDdr_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceID is the device identifier to lookup.

Returns:

- o **XDdr** configuration structure pointer if DeviceID is found.
- o XNULL if DeviceID is not found.

Note:

None.

```
void                ( XDdr *  
XDdr_SetControl    InstancePtr,  
                    Xuint32 Control  
                    )
```

Set the ECC Control Register of the DDR device to the specified value. This function can be used to individually enable/disable read or write ECC and force specific types of ECC errors to occur.

Parameters:

InstancePtr is a pointer to an **XDdr** instance to be worked on.

Control contains the value to be written to the register and consists of constants named XDDR_ECCCR* for each bit field as specified in **xddr_1.h**.

Returns:

None.

Note:

None.

ddr/v1_00_b/src/xddr_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of DDR devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	jhl	01/06/04	First release

```
#include "xddr.h"  
#include "xparameters.h"
```

Variables

XDdr_Config [XDdr_ConfigTable](#) []

Variable Documentation

XDdr_Config XDdr_ConfigTable
[]

This table contains configuration information for each DDR device in the system.

ddr/v1_00_b/src/xddr_intr.c File Reference

Detailed Description

The implementation of the **XDdr** component's functionality that is related to interrupts. See [xddr.h](#) for more information about the component. The functions that are contained in this file require that the hardware device is built with interrupt support.

Note:

None

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	jhl	12/24/03	First release

```
#include "xddr.h"  
#include "xipif_v1_23_b.h"
```

Functions

```
void XDdr_InterruptGlobalEnable (XDdr *InstancePtr)  
void XDdr_InterruptGlobalDisable (XDdr *InstancePtr)  
void XDdr_InterruptEnable (XDdr *InstancePtr, Xuint32 Mask)  
void XDdr_InterruptDisable (XDdr *InstancePtr, Xuint32 Mask)  
void XDdr_InterruptClear (XDdr *InstancePtr, Xuint32 Mask)
```

```
Xuint32 XDdr_InterruptGetEnabled (XDdr *InstancePtr)
```

```
Xuint32 XDdr_InterruptGetStatus (XDdr *InstancePtr)
```

Function Documentation

```
void  
XDDR_InterruptClear ( XDDR *  
                    InstancePtr,  
                    Xuint32 Mask  
                    )
```

Clear pending interrupts with the provided mask. An interrupt must be cleared after software has serviced it or it can cause another interrupt.

Parameters:

InstancePtr is the DDR component to operate on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XDDR_IPIXR_*_MASK.

Note:

This function will assert if the hardware device has not been built with interrupt capabilities.

```
void  
XDDR_InterruptDisable ( XDDR *  
                      InstancePtr,  
                      Xuint32 Mask  
                      )
```

Disable ECC interrupts so that ECC errors will not cause an interrupt.

Parameters:

InstancePtr is the DDR component to operate on.

Mask is the mask to disable. Bits set to 1 are disabled. The mask is formed by OR'ing bits from XDDR_IPIXR_*_MASK.

Note:

This function will assert if the hardware device has not been built with interrupt capabilities.

```
void  
XDDR_InterruptEnable ( XDDR *  
                     InstancePtr,  
                     Xuint32 Mask  
                     )
```

Enable ECC interrupts so that specific ECC errors will cause an interrupt. The function `XDdr_InterruptGlobalEnable` must also be called to enable any interrupt to occur.

Parameters:

InstancePtr is the DDR component to operate on.

Mask is the mask to enable. Bit positions of 1 are enabled. The mask is formed by OR'ing bits from `XDDR_IPIXR_*_MASK`.

Note:

This function will assert if the hardware device has not been built with interrupt capabilities.

Xuint32 (**XDdr ***)
XDdr_InterruptGetEnabled *InstancePtr*

Returns the interrupt enable mask as set by `XDdr_InterruptEnable()` which indicates which ECC interrupts are enabled or disabled.

Parameters:

InstancePtr is the DDR component to operate on.

Returns:

Mask of bits made from `XDDR_IPIXR_*_MASK`.

Note:

This function will assert if the hardware device has not been built with interrupt capabilities.

Xuint32 (**XDdr ***)
XDdr_InterruptGetStatus *InstancePtr*

Returns the status of interrupts which indicates which ECC interrupts are pending.

Parameters:

InstancePtr is the DDR component to operate on.

Returns:

Mask of bits made from XDDR_IPIXR_*_MASK.

Note:

The interrupt status indicates the status of the device irregardless if the interrupts from the devices have been enabled or not through [XDdr_InterruptEnable\(\)](#).

This function will assert if the hardware device has not been built with interrupt capabilities.

```
void ( XDdr * )  
XDdr_InterruptGlobalDisable InstancePtr
```

Disable the core's interrupt output signal. Interrupts enabled through [XDdr_InterruptEnable\(\)](#) will not occur until the global enable bit is set by [XDdr_InterruptGlobalEnable\(\)](#). This function is designed to allow all interrupts to be disabled easily for entering a critical section.

Parameters:

InstancePtr is the DDR component to operate on.

Note:

This function will assert if the hardware device has not been built with interrupt capabilities.

```
void ( XDdr * )  
XDdr_InterruptGlobalEnable InstancePtr
```

Enable the core's interrupt output signal. Interrupts enabled through [XDdr_InterruptEnable\(\)](#) will not occur until the global enable bit is set by this function. This function is designed to allow all interrupts to be enabled easily for exiting a critical section.

Parameters:

InstancePtr is the DDR component to operate on.

Note:

This function will assert if the hardware device has not been built with interrupt capabilities.

ddr/v1_00_b/src/xddr_selftest.c File Reference

Detailed Description

The implementation of the [XDdr](#) component's functionality that is related to selftest. See [xddr.h](#) for more information about the component.

Note:

None

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	jhl	12/24/03	First release

```
#include "xddr.h"
```

Functions

[XStatus XDdr_SelfTest](#) ([XDdr](#) *InstancePtr)

Function Documentation

```
XStatus (XDdr *  
XDdr\_SelfTest InstancePtr)
```

Perform a self-test on the DDR device. Self-test will read, write and verify that some of the registers of the device are functioning correctly. This function will restore the state of the device to state it was in prior to the function call.

Parameters:

InstancePtr is the DDR component to operate on.

Returns:

- XST_SUCCESS The self-test was successful.
- XST_FAILURE The self-test failed.

Note:

None

dmacentral/v1_00_a/src/xdmacentral.h File Reference

Detailed Description

This header file contains prototypes of high-level driver function that can be used to access the Central DMA device.

DESCRIPTION

Central DMA is a standalone IP core, as opposed to a DMA channel component distributed in devices.

Central DMA allows a CPU to minimize the CPU interaction required to move data between a memory and a device. The CPU requests the Central DMA to perform a DMA operation and typically continues performing other processing until the DMA operation completes. DMA could be considered a primitive form of multiprocessing such that caching and address translation can be an issue.

Central DMA device does not support Scatter Gather operation at current stage.

A feature the Central DMA device contains is supporting different data sizes used in each data transfer on the bus. Currently data sizes of Byte, Half word and Word are supported.

The user should refer to the hardware device specification for more details of the device operation.

CONFIGURATION

The Central DMA device supports a parameter, `C_READ_OPTIONAL_REGS`, at hardware design time. The Source Address register, Destination Address register, DMA Control register and Interrupt Enable register are:

- write only if `C_READ_OPTIONAL_REGS = 0`
- readable/writable if `C_READ_OPTIONAL_REGS = 1`

If the application needs to read the contents of these register, it is the responsibility of the application developer to make sure the hardware parameter mentioned above is set to 1. This could be done by checking data member, `ReadOptionalRegs`, in the device configuration structure `XDmaCentral_Config` defined in `xdmacentral.h`. Note that the registers above are readable/writable if `ReadOptionalRegs` equals `XTRUE`, write only if `ReadOptionalRegs` equals `XFALSE`.

This driver does not maintain the current states of these registers. It is the responsibility of the application to manage those states if needed.

INTERRUPTS

Central DMA has the ability to generate an interrupt. It is the responsibility of the caller of DMA functions to manage the interrupt including connecting to the interrupt and enabling/disabling the interrupt.

CRITICAL SECTIONS

This driver does not use critical sections and it does access registers using read-modify-write operations. Calls to DMA functions from a main thread and from an interrupt context could produce unpredictable behavior such that the caller must provide the appropriate critical sections.

ADDRESS TRANSLATION

All addresses of data structures which are passed to DMA functions must be physical (real) addresses as opposed to logical (virtual) addresses.

CACHING

The caller of DMA functions is responsible for ensuring that any data buffers which are passed to the Central DMA have been flushed from the cache.

The caller of DMA functions is responsible for ensuring that the cache is invalidated prior to using any data buffers which are the result of a DMA operation.

MEMORY ALIGNMENT

The addresses of data buffers which are passed to DMA functions must be multiples of the data size currently used by the Central DMA hardware. Data size could be set by using DMA functions provided in this driver. The caller of this driver should refer to the device specification for more details.

MUTUAL EXCLUSION

The functions of the Central DMA are not thread safe such that the caller of all DMA functions is responsible for ensuring mutual exclusion for a Central DMA device. Mutual exclusion across multiple Central DMA device is not necessary.

The user should refer to the hardware device specification for more details of the device operation.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

-----	-----	-----	
-------	-------	-------	--

1.00a	xd	03/11/04	First release.

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xdmacentral_1.h"
```

Data Structures

```
struct XDmaCentral  
struct XDmaCentral_Config
```

Functions

```
XStatus XDmaCentral_Initialize (XDmaCentral *InstancePtr, Xuint16 DeviceId)  
void XDmaCentral_Reset (XDmaCentral *InstancePtr)  
XDmaCentral_Config * XDmaCentral_LookupConfig (Xuint16 DeviceId)  
void XDmaCentral_SetControl (XDmaCentral *InstancePtr, Xuint32 Value)  
Xuint32 XDmaCentral_GetControl (XDmaCentral *InstancePtr)  
Xuint32 XDmaCentral_GetStatus (XDmaCentral *InstancePtr)  
Xuint32 XDmaCentral_GetSrcAddress (XDmaCentral *InstancePtr)  
Xuint32 XDmaCentral_GetDestAddress (XDmaCentral *InstancePtr)  
void XDmaCentral_Transfer (XDmaCentral *InstancePtr, void *SourcePtr,  
void *DestinationPtr, Xuint32 ByteCount)  
XStatus XDmaCentral_SelfTest (XDmaCentral *InstancePtr)
```

```
void XDmaCentral_InterruptEnableSet (XDmaCentral *InstancePtr, Xuint32
Mask)
Xuint32 XDmaCentral_InterruptEnableGet (XDmaCentral *InstancePtr)
Xuint32 XDmaCentral_InterruptStatusGet (XDmaCentral *InstancePtr)
void XDmaCentral_InterruptClear (XDmaCentral *InstancePtr, Xuint32
Mask)
```

Function Documentation

```
Xuint32 ( XDmaCentral * )
XDmaCentral_GetControl InstancePtr
```

Get the contents of DMA Control register. Use the XDMC_DMACR_* constants defined in [xdmcentral_1.h](#) to interpret the value.

Parameters:

InstancePtr is a pointer to the [XDmaCentral](#) instance to be worked on.

Returns:

A 32-bit value representing the contents of DMA Control register.

Note:

The hardware parameter C_READ_OPTIONAL_REGS must be set to 1 for this function to work. This function asserts if the hardware parameter is set to 0. Please read the device specification to get more detailed information.

```
Xuint32 ( XDmaCentral * )
XDmaCentral_GetDestAddress InstancePtr
```

Get the contents of the Destination Address register.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Returns:

A 32-bit value representing the contents of the Destination Address register.

Note:

The hardware parameter C_READ_OPTIONAL_REGS must be set to 1 for this function to work. This function asserts if the hardware parameter is set to 0. Please read the device specification to get more detailed information.

```
Xuint32 ( XDmaCentral * )  
XDmaCentral_GetSrcAddress InstancePtr
```

Get the contents of the Source Address register.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Returns:

A 32-bit value representing the contents of the Source Address register.

Note:

The hardware parameter C_READ_OPTIONAL_REGS must be set to 1 for this function to work. This function asserts if the hardware parameter is set to 0. Please read the device specification to get more detailed information.

```
Xuint32 ( XDmaCentral * )  
XDmaCentral_GetStatus InstancePtr
```

Get the contents of the DMA Status register. Use the `XDMC_DMASR_*` constants defined in [xdmacentral_1.h](#) to interpret the value.

Parameters:

InstancePtr is a pointer to the [XDmaCentral](#) instance to be worked on.

Returns:

A 32-bit value representing the contents of the Status register.

Note:

None.

```
XStatus ( XDmaCentral *  
XDmaCentral_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific [XDmaCentral](#) instance. This function must be called prior to using a Central DMA device. Initialization of a device includes looking up the configuration for the given device instance, initializing the instance structure, and resetting the device such that it is in a known state.

Parameters:

InstancePtr is a pointer to the [XDmaCentral](#) instance to be worked on.

DeviceId is the unique id of the device controlled by this [XDmaCentral](#) instance.

Returns:

- XST_SUCCESS if everything initializes as expected.
- XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

```
void ( XDmaCentral *  
XDmaCentral_InterruptClear InstancePtr,  
 Xuint32 Mask  
 )
```

Clear pending interrupts with the provided mask. This function should be called after the software has serviced the interrupts that are pending.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XDMC_IXR_* bits which are contained in **xdmacentral_1.h**.

Returns:

None.

Note:

None.

```
Xuint32 ( XDmaCentral * InstancePtr )  
XDmaCentral_InterruptEnableGet
```

Get the contents of the Interrupt Enable Register. Use the XDMC_IXR_* constants defined in **xdmacentral_1.h** to interpret the value.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Returns:

A 32-bit value representing the contents of the Interrupt Enable register.

Note:

The hardware parameter C_READ_OPTIONAL_REGS must be set to 1 for this function to work. This function asserts if the hardware parameter is set to 0. Please read the device specification to get more detailed information.

```
void ( XDmaCentral * InstancePtr,  
XDmaCentral_InterruptEnableSet Xuint32 Mask )
```

Set the contents of Interrupt Enable Register. Use the XDMC_IXR_* constants defined in [xdmacentral_1.h](#) to create the bit-mask to enable interrupts.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Mask is the mask to enable. Bit positions of 1 are enabled. Bit positions of 0 are disabled. This mask is formed by OR'ing bits from XDMC_IXR_* bits which are contained in [xdmacentral_1.h](#).

Returns:

None.

Note:

None.

Xuint32 (**XDmaCentral ***)
XDmaCentral_InterruptStatusGet *InstancePtr*

Get the contents of the Interrupt Status register. Use the XDMC_IXR_* constants defined in [xdmacentral_1.h](#) to interpret the value.

The Interrupt Status register indicates which interrupts are active for the Central DMA device. The definitions of each bit in the register match the definitions of the bits in the Interrupt Enable register.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Returns:

A 32-bit value representing the contents of the Interrupt Status register.

Note:

None.

XDmaCentral_Config* (**Xuint16**)
XDmaCentral_LookupConfig *DeviceId*

Looks up the device configuration given an unique device ID. The table `XDmaCentral_ConfigTable`, defined in [xdmacentral_g.c](#), contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for.

Returns:

A pointer to the configuration data for the device, or XNULL if no match is found.

Note:

None.

```
void ( XDmaCentral * )  
XDmaCentral_Reset InstancePtr
```

Forces a software reset to occur in the device.

Parameters:

InstancePtr is a pointer to the [XDmaCentral](#) instance to be worked on.

Returns:

None.

Note:

This function is a destructive operation such that it should not be called while a dma transfer is ongoing. Please read the device specification for the device status after this reset operation is executed.

```
XStatus ( XDmaCentral * )  
XDmaCentral_SelfTest InstancePtr
```

Run a self-test on the driver/device. The test resets the device, starts a DMA transfer, compares the contents of destination buffer and source buffer after the the DMA transfer is finished, and resets the device again before the function returns.

Note that this is a destructive test in that resets of the device are performed. Please refer to the device specification for the device status after the reset operation.

If the hardware system is not built correctly, this function may never return to the caller.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Returns:

- XST_SUCCESS if the DMA transfer could get finished and the contents of destination buffer were the same as the source buffer after the transfer.
- XST_FAILURE if a Bus error/Bus timeout occurred, or the contents of the destination buffer were different from the source buffer after the transfer was finished.

Note:

Caching must be turned off for this function to work.

```
void          ( XDmaCentral *
XDmaCentral_SetControl      InstancePtr,
                             Xuint32      Mask
                             )
```

Set the contents of DMA Control register. Use the XDMC_DMACR_* constants defined in [xdmacentral_1.h](#) to create the bit-mask to be written to the register.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Mask is the 32-bit value to write to the DMA Control register.

Returns:

None.

Note:

The caller is responsible to make sure different XDMC_DMACR_DATASIZE_*_MASKs are NOT used at the same time. This function asserts if multiple

XDMC_DMOCR_DATASIZE_*_MASKs are used at the same time.

```
void
XDmaCentral_Transfer
( XDmaCentral * InstancePtr,
  void * SourcePtr,
  void * DestinationPtr,
  Xuint32 ByteCount
)
```

This function starts the DMA transferring data from a memory source to a memory destination. This function only starts the operation and returns before the operation may be complete. If the interrupt is enabled, an interrupt will be generated when the operation is complete, otherwise it is necessary to poll the Status register to determine when it's complete. It is the responsibility of the caller to determine when the operation is complete by handling the generated interrupt or polling the DMA Status register. (See [XDmaCentral_GetStatus\(\)](#))

Padding

If the input transfer length is not a multiple of data size, The DMA device will pad the destination buffer with extra bytes needed to reach a full data size at the end of the transfer. For example, assume the DMA transfer length parameter passed into this function equals X,

- If current data size is word, then $((X+3)/4)*4$ bytes will be actually transferred.
- If the current data size is half word, then $((X+1)/2)*2$ bytes will be actually transferred.
- If current data size is byte, then X bytes will be actually transferred.

Parameters:

InstancePtr is a pointer to the [XDmaCentral](#) instance to be worked on.

SourcePtr contains a pointer to the source memory where the data is to be transferred from and must be aligned to the data size currently used by the Central DMA device.

DestinationPtr contains a pointer to the destination memory where the data is to be transferred to and must be aligned to the data size currently used by the Central DMA device.

ByteCount contains the number of bytes to transfer during the DMA operation. Please refer to the padding note above.

Returns:

None.

Note:

It is the responsibility of the caller to ensure that the cache is flushed and invalidated both before the DMA operation is started and after the DMA operation completes if the memory pointed to is cached. The caller must also ensure that the pointers contain a physical address rather than a virtual address if address translation is being used.

The caller is also responsible to set up the device by writing the correct value to the Control register of the device before this function is called.

Generated on 24 Jun 2004 for Xilinx Device Drivers

XDmaCentral_Config Struct Reference

```
#include <xdmacentral.h>
```

Detailed Description

This typedef contains configuration information for the Central DMA device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xboolean SupportReadRegs](#)

Field Documentation

[Xuint32 XDmaCentral_Config::](#)

BaseAddress

Register base
address

[Xuint16 XDmaCentral_Config::](#)

DeviceId

Unique ID of
device

[Xboolean XDmaCentral_Config::](#)

SupportReadRegs

parameter

C_READ_OPTIONAL_REGS

The documentation for this struct was generated from the following file:

- [dmacentral/v1_00_a/src/xdmacentral.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

dmacentral/v1_00_a/src/xdmacentral_1.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the Central DMA device.

Note:

All provided functions which are register accessors do not provide a lot of error detection. This is to minimize the overhead in this low-level driver. The caller is expected to understand the impact of a function call based upon the current state of the Central DMA.

Refer to the device specification and [xdmacentral.h](#) for more information about this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	xd	03/11/04	First release.

```
#include "xbasic_types.h"  
#include "xio.h"
```

Defines

```
#define XDMC_RST_OFFSET  
#define XDMC_MIR_OFFSET
```

```
#define XDMC_DMACR_OFFSET
#define XDMC_SA_OFFSET
#define XDMC_DA_OFFSET
#define XDMC_LENGTH_OFFSET
#define XDMC_DMASR_OFFSET
#define XDMC_ISR_OFFSET
#define XDMC_IER_OFFSET
#define XDMC_RST_MASK
#define XDMC_DMACR_SOURCE_INCR_MASK
#define XDMC_DMACR_DEST_INCR_MASK
#define XDMC_DMACR_DATASIZE_4_MASK
#define XDMC_DMACR_DATASIZE_2_MASK
#define XDMC_DMACR_DATASIZE_1_MASK
#define XDMC_DMACR_DATASIZE_MASK
#define XDMC_DMASR_BUSY_MASK
#define XDMC_DMASR_BUS_ERROR_MASK
#define XDMC_DMASR_BUS_TIMEOUT_MASK
#define XDMC_IXR_DMA_DONE_MASK
#define XDMC_IXR_DMA_ERROR_MASK
#define XDmaCentral_mReadReg(BaseAddress, RegOffset)
#define XDmaCentral_mWriteReg(BaseAddress, RegOffset, Data)
```

Define Documentation

```
#define XDmaCentral_mReadReg( BaseAddress,
                             RegOffset  )
```

Read a register of the Central DMA. This macro provides register access to all registers using the register offsets defined above.

Parameters:

BaseAddress contains the base address of the device.

RegOffset is the offset of the register to read.

Returns:

The contents of the register.

Note:

C-style Signature: Xuint32 XDmaCentral_mReadReg(Xuint32 BaseAddress, Xuint32 RegOffset)

```
#define XDmaCentral_mWriteReg( BaseAddress,  
                               RegOffset,  
                               Data          )
```

Write a register of the Central DMA. This macro provides register access to all registers using the register offsets defined above.

Parameters:

BaseAddress contains the base address of the device.

RegOffset is the offset of the register to write.

Data is the value to write to the register.

Returns:

None.

Note:

C-style Signature: void XDmaCentral_mWriteReg(Xuint32 BaseAddress, Xuint32 RegOffset, Xuint32 Data)

```
#define  
XDMC_DA_OFFSET
```

Destination Address
register

```
#define  
XDMC_DMACR_DATASIZE_1_MASK
```

transfer Dsize = 1
byte

```
#define  
XDMC_DMACR_DATASIZE_2_MASK
```

transfer Dsize = 2
bytes

```
#define  
XDMC_DMACR_DATASIZE_4_MASK
```

```
transfer Dsize = 4  
bytes
```

```
#define  
XDMC_DMACR_DATASIZE_MASK
```

```
transfer data size  
mask
```

```
#define  
XDMC_DMACR_DEST_INCR_MASK
```

```
increment dest  
address
```

```
#define  
XDMC_DMACR_OFFSET
```

```
DMA Control  
register
```

```
#define  
XDMC_DMACR_SOURCE_INCR_MASK
```

```
increment source  
address
```

```
#define  
XDMC_DMASR_BUS_ERROR_MASK
```

```
bus error  
occurred
```

```
#define  
XDMC_DMASR_BUS_TIMEOUT_MASK
```

```
bus timeout  
occurred
```

```
#define  
XDMC_DMASR_BUSY_MASK
```

```
device is  
busy
```

```
#define  
XDMC_DMASR_OFFSET
```

```
DMA Status  
register
```

```
#define  
XDMC_IER_OFFSET
```

```
Interrupt Enable  
register
```

```
#define  
XDMC_ISR_OFFSET
```

```
Interrupt Status  
register
```

```
#define  
XDMC_IXR_DMA_DONE_MASK
```

```
DMA operation  
done
```

```
#define  
XDMC_IXR_DMA_ERROR_MASK
```

```
DMA operation  
error
```

```
#define  
XDMC_LENGTH_OFFSET
```

```
Length  
register
```

```
#define  
XDMC_MIR_OFFSET
```

Module Information
register

#define
XDMC_RST_MASK

Value used to reset the
device

#define
XDMC_RST_OFFSET

Reset
register

#define
XDMC_SA_OFFSET

Source Address
register

XDmaCentral Struct Reference

```
#include <xdmacentral.h>
```

Detailed Description

The driver's instance data. The user is required to allocate a variable of this type for every Central DMA device in the system. A pointer to a variable of this type is then passed to the driver API functions.

Data Fields

[Xuint32 BaseAddress](#)

[Xuint32 IsReady](#)

[Xboolean SupportReadRegs](#)

Field Documentation

[Xuint32 XDmaCentral::](#)

BaseAddress

Base address of
device

[Xuint32 XDmaCentral::](#)

IsReady

Device is initialized and
ready

Xboolean XDmaCentral::

SupportReadRegs

Supports reading IER, CA, SA and
DA ?

The documentation for this struct was generated from the following file:

- `dmacentral/v1_00_a/src/xdmacentral.h`
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

dmacentral/v1_00_a/src/xdmacentral_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of Central DMA devices in the system.

See [xdmacentral.h](#) for more information about this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	xd	03/11/04	First release.

```
#include "xdmacentral.h"  
#include "xparameters.h"
```

dmacentral/v1_00_a/src/xdmacentral.c File Reference

Detailed Description

This file contains high-level driver API functions that can be used to access the central dma device.

Please refer to the [xdmacentral.h](#) header file for more information about this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	xd	03/11/04	First release

```
#include "xio.h"  
#include "xdmacentral.h"  
#include "xparameters.h"
```

Functions

XStatus **XDmaCentral_Initialize** (**XDmaCentral** *InstancePtr, **Xuint16** DeviceId)

void **XDmaCentral_Reset** (**XDmaCentral** *InstancePtr)

XDmaCentral_Config * **XDmaCentral_LookupConfig** (**Xuint16** DeviceId)

void **XDmaCentral_SetControl** (**XDmaCentral** *InstancePtr, **Xuint32** Mask)

Xuint32 **XDmaCentral_GetControl** (**XDmaCentral** *InstancePtr)

Xuint32 **XDmaCentral_GetStatus** (**XDmaCentral** *InstancePtr)

Xuint32 **XDmaCentral_GetSrcAddress** (**XDmaCentral** *InstancePtr)

Xuint32 **XDmaCentral_GetDestAddress** (**XDmaCentral** *InstancePtr)

void **XDmaCentral_Transfer** (**XDmaCentral** *InstancePtr, void *SourcePtr,
void *DestinationPtr, **Xuint32** ByteCount)

Function Documentation

Xuint32 (**XDmaCentral** *)
XDmaCentral_GetControl *InstancePtr*

Get the contents of DMA Control register. Use the XDMC_DMACR_* constants defined in [xdmacentral_1.h](#) to interpret the value.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Returns:

A 32-bit value representing the contents of DMA Control register.

Note:

The hardware parameter C_READ_OPTIONAL_REGS must be set to 1 for this function to work. This function asserts if the hardware parameter is set to 0. Please read the device specification to get more detailed information.

Xuint32 (**XDmaCentral** *)
XDmaCentral_GetDestAddress *InstancePtr*

Get the contents of the Destination Address register.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Returns:

A 32-bit value representing the contents of the Destination Address register.

Note:

The hardware parameter C_READ_OPTIONAL_REGS must be set to 1 for this function to work. This function asserts if the hardware parameter is set to 0. Please read the device specification to get more detailed information.

Xuint32 (**XDmaCentral ***)

XDmaCentral_GetSrcAddress *InstancePtr*

Get the contents of the Source Address register.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Returns:

A 32-bit value representing the contents of the Source Address register.

Note:

The hardware parameter C_READ_OPTIONAL_REGS must be set to 1 for this function to work. This function asserts if the hardware parameter is set to 0. Please read the device specification to get more detailed information.

Xuint32 (**XDmaCentral ***)

XDmaCentral_GetStatus *InstancePtr*

Get the contents of the DMA Status register. Use the XDMC_DMASR_* constants defined in [xdmcentral_1.h](#) to interpret the value.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Returns:

A 32-bit value representing the contents of the Status register.

Note:

None.

XStatus (**XDmaCentral ***
XDmaCentral_Initialize *InstancePtr*,
Xuint16 *DeviceId*
)

Initializes a specific **XDmaCentral** instance. This function must be called prior to using a Central DMA device. Initialization of a device includes looking up the configuration for the given device instance, initializing the instance structure, and resetting the device such that it is in a known state.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XDmaCentral** instance.

Returns:

- o XST_SUCCESS if everything initializes as expected.
- o XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

```
XDmaCentral_Config* ( Xuint16 )  
XDmaCentral_LookupConfig DeviceId
```

Looks up the device configuration given an unique device ID. The table `XDmaCentral_ConfigTable`, defined in `xdmacentral_g.c`, contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for.

Returns:

A pointer to the configuration data for the device, or XNULL if no match is found.

Note:

None.

```
void ( XDmaCentral * )  
XDmaCentral_Reset InstancePtr
```

Forces a software reset to occur in the device.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Returns:

None.

Note:

This function is a destructive operation such that it should not be called while a dma transfer is ongoing. Please read the device specification for the device status after this reset operation is executed.

```
void          ( XDmaCentral *
XDmaCentral_SetControl      InstancePtr,
                             Xuint32      Mask
                             )
```

Set the contents of DMA Control register. Use the XDMC_DMACR_* constants defined in [xdmacentral_1.h](#) to create the bit-mask to be written to the register.

Parameters:

InstancePtr is a pointer to the **XDmaCentral** instance to be worked on.

Mask is the 32-bit value to write to the DMA Control register.

Returns:

None.

Note:

The caller is responsible to make sure different XDMC_DMACR_DATASIZE_*_MASKs are NOT used at the same time. This function asserts if multiple XDMC_DMACR_DATASIZE_*_MASKs are used at the same time.

```

void
XDmaCentral_Transfer
    ( XDmaCentral *
      InstancePtr,
      void *
      SourcePtr,
      void *
      DestinationPtr,
      Xuint32
      ByteCount
    )

```

This function starts the DMA transferring data from a memory source to a memory destination. This function only starts the operation and returns before the operation may be complete. If the interrupt is enabled, an interrupt will be generated when the operation is complete, otherwise it is necessary to poll the Status register to determine when it's complete. It is the responsibility of the caller to determine when the operation is complete by handling the generated interrupt or polling the DMA Status register. (See [XDmaCentral_GetStatus\(\)](#))

Padding

If the input transfer length is not a multiple of data size, The DMA device will pad the destination buffer with extra bytes needed to reach a full data size at the end of the transfer. For example, assume the DMA transfer length parameter passed into this function equals X,

- If current data size is word, then $((X+3)/4)*4$ bytes will be actually transferred.
- If the current data size is half word, then $((X+1)/2)*2$ bytes will be actually transferred.
- If current data size is byte, then X bytes will be actually transferred.

Parameters:

InstancePtr is a pointer to the [XDmaCentral](#) instance to be worked on.

SourcePtr contains a pointer to the source memory where the data is to be transferred from and must be aligned to the data size currently used by the Central DMA device.

DestinationPtr contains a pointer to the destination memory where the data is to be transferred to and must be aligned to the data size currently used by the Central DMA device.

ByteCount contains the number of bytes to transfer during the DMA operation. Please refer to the padding note above.

Returns:

None.

Note:

It is the responsibility of the caller to ensure that the cache is flushed and invalidated both before the DMA operation is started and after the DMA operation completes if the memory pointed to is cached. The caller must also ensure that the pointers contain a physical address

rather than a virtual address if address translation is being used.

The caller is also responsible to set up the device by writing the correct value to the Control register of the device before this function is called.

Generated on 24 Jun 2004 for Xilinx Device Drivers

dmacentral/v1_00_a/src/xdmacentral_intr.c File Reference

Detailed Description

This file contains interrupt handling API functions of the Central DMA device.

Please refer to [xdmacentral.h](#) header file for more information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	xd	03/11/04	First release

```
#include "xio.h"  
#include "xdmacentral.h"
```

Functions

```
void XDmaCentral_InterruptEnableSet (XDmaCentral *InstancePtr, Xuint32 Mask)  
Xuint32 XDmaCentral_InterruptEnableGet (XDmaCentral *InstancePtr)  
Xuint32 XDmaCentral_InterruptStatusGet (XDmaCentral *InstancePtr)  
void XDmaCentral_InterruptClear (XDmaCentral *InstancePtr, Xuint32 Mask)
```

Function Documentation

```
void  
XDmaCentral_InterruptClear ( XDmaCentral *  
                             InstancePtr,  
                             Xuint32      Mask  
                             )
```

Clear pending interrupts with the provided mask. This function should be called after the software has serviced the interrupts that are pending.

Parameters:

InstancePtr is a pointer to the **[XDmaCentral](#)** instance to be worked on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XDMC_IXR_* bits which are contained in **[xdmcentral_1.h](#)**.

Returns:

None.

Note:

None.

```
Xuint32 ( XDmaCentral *  
XDmaCentral_InterruptEnableGet InstancePtr
```

Get the contents of the Interrupt Enable Register. Use the XDMC_IXR_* constants defined in **[xdmcentral_1.h](#)** to interpret the value.

Parameters:

InstancePtr is a pointer to the **[XDmaCentral](#)** instance to be worked on.

Returns:

A 32-bit value representing the contents of the Interrupt Enable register.

Note:

The hardware parameter C_READ_OPTIONAL_REGS must be set to 1 for this function to work. This function asserts if the hardware parameter is set to 0. Please read the device specification to get more detailed information.

```
void ( XDmaCentral *
XDmaCentral_InterruptEnableSet InstancePtr,
Xuint32 Mask
)
```

Set the contents of Interrupt Enable Register. Use the XDMC_IXR_* constants defined in [xdmcentral_1.h](#) to create the bit-mask to enable interrupts.

Parameters:

InstancePtr is a pointer to the [XDmaCentral](#) instance to be worked on.

Mask is the mask to enable. Bit positions of 1 are enabled. Bit positions of 0 are disabled. This mask is formed by OR'ing bits from XDMC_IXR_* bits which are contained in [xdmcentral_1.h](#).

Returns:

None.

Note:

None.

```
Xuint32 ( XDmaCentral *
XDmaCentral_InterruptStatusGet InstancePtr
)
```

Get the contents of the Interrupt Status register. Use the XDMC_IXR_* constants defined in [xdmcentral_1.h](#) to interpret the value.

The Interrupt Status register indicates which interrupts are active for the Central DMA device. The definitions of each bit in the register match the definitions of the bits in the Interrupt Enable register.

Parameters:

InstancePtr is a pointer to the [XDmaCentral](#) instance to be worked on.

Returns:

A 32-bit value representing the contents of the Interrupt Status register.

Note:

None.

dmacentral/v1_00_a/src/xdmacentral_selftest.c

File Reference

Detailed Description

Contains a diagnostic self-test function for the **XDmaCentral** driver. This code assumes the caching is turned off.

See [xdmacentral.h](#) for more information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	xd	03/12/04	First release

```
#include "xstatus.h"  
#include "xdmacentral.h"
```

Defines

```
#define XDMC_SELFTEST_BUFSIZE
```

Functions

XStatus XDmaCentral_SelfTest (**XDmaCentral** *InstancePtr)

Define Documentation

```
#define
XDMC_SELFTEST_BUFSIZE
```

size of transfer test buffer in
bytes

Function Documentation

```
XStatus ( XDmaCentral * )
XDmaCentral_SelfTest InstancePtr
```

Run a self-test on the driver/device. The test resets the device, starts a DMA transfer, compares the contents of destination buffer and source buffer after the the DMA transfer is finished, and resets the device again before the function returns.

Note that this is a destructive test in that resets of the device are performed. Please refer to the device specification for the device status after the reset operation.

If the hardware system is not built correctly, this function may never return to the caller.

Parameters:

InstancePtr is a pointer to the **[XDmaCentral](#)** instance to be worked on.

Returns:

- XST_SUCCESS if the DMA transfer could get finished and the contents of destination buffer were the same as the source buffer after the transfer.
- XST_FAILURE if a Bus error/Bus timeout occurred, or the contents of the destination buffer were different from the source buffer after the transfer was finished.

Note:

Caching must be turned off for this function to work.

dma/v1_00_a/src/xdma_multi_sg.c File Reference

Detailed Description

Description

This file contains the implementation of the **XDmaMulti** component which is related to scatter gather operations.

Scatter Gather Operations

The multichannel DMA may support scatter gather operations. A scatter gather operation automates the DMA transfer such that multiple buffers can be sent or received with minimal software interaction with the hardware. Buffer descriptors, contained in the XBufDescriptor component, are used by the scatter gather operations of the DMA to describe the buffers to be processed.

Scatter Gather List Operations

A scatter gather list may be supported by each DMA channel. The scatter gather list allows buffer descriptors to be put into the list by a device driver which requires scatter gather. The hardware processes the buffer descriptors which are contained in the list and modifies the buffer descriptors to reflect the status of the DMA operations. The device driver is notified by interrupt that specific DMA events occur including scatter gather events. The device driver removes the completed buffer descriptors from the scatter gather list to evaluate the status of each DMA operation.

The scatter gather list is created and buffer descriptors are inserted into the list. Buffer descriptors are never removed from the list after it's creation such that a put operation copies from a temporary buffer descriptor to a buffer descriptor in the list. Get operations don't copy from the list to a temporary, but return a pointer to the buffer descriptor in the list. A buffer descriptor in the list may be locked to prevent it from being overwritten by a put operation. This allows the device driver to get a descriptor from a scatter gather list and prevent it from being overwritten until the buffer associated with the buffer descriptor has been processed.

The get and put functions only operate on the list and are asynchronous from the hardware which may

be using the list of descriptors. This is important because there are no checks in the get and put functions to ensure that the hardware has processed the descriptors. This must be handled by the driver using the DMA scatter gather channel through the use of the other functions. When a scatter gather operation is started, the start function does ensure that the descriptor to start has not already been processed by the hardware and is not the first of a series of descriptors that have not been committed yet.

Descriptors are put into the list but not marked as ready to use by the hardware until a commit operation is done. This allows multiple descriptors which may contain a single packet of information for a protocol to be guaranteed not to cause any underflow conditions during transmission. The hardware design only allows descriptors to cause it to stop after a descriptor has been processed rather than before it is processed. A series of descriptors are put into the list followed by a commit operation, or each descriptor may be committed. A commit operation is performed by changing a single descriptor, the first of the series of puts, to indicate that the hardware may now use all descriptors after it. The last descriptor in the list is always set to cause the hardware to stop after it is processed.

Typical Scatter Gather Processing

The following steps illustrate the typical processing to use the scatter gather features of the multichannel DMA.

1. Create a scatter gather list for the each channel of DMA which puts empty buffer descriptors into the list.
2. Create buffer descriptors which describe the buffers to be filled with receive data or the buffers which contain data to be sent.
3. Put buffer descriptors into the scatter list such that scatter gather operations are requested.
4. Commit the buffer descriptors in the list such that they are ready to be used by the DMA hardware.
5. Start the scatter gather operations of the DMA channel.
6. Process any interrupts which occur as a result of the scatter gather operations or poll the DMA channel to determine the status. This may be accomplished by getting the packet count for the channel and then getting the appropriate number of descriptors from the list for that number of packets.

Minimizing Interrupts

The Scatter Gather operating mode is designed to reduce the amount of CPU throughput necessary to manage the hardware for devices. A key to the CPU throughput is the number and rate of interrupts that the CPU must service. Devices with higher data rates can cause larger numbers of interrupts and higher frequency interrupts. Ideally the number of interrupts can be reduced by only generating an interrupt when a specific amount of data has been received from the interface. This design suffers from a lack of interrupts when the amount of data received is less than the specified amount of data to generate an interrupt. In order to help minimize the number of interrupts which the CPU must service, an algorithm referred to as "interrupt coalescing" is utilized. In the case of the multichannel DMA, each channel of DMA is independent and the interrupts from each channel can have independent Packet Wait and Packet Count values allowing for traffic specific values for each channel.

Interrupt Coalescing

The principle of interrupt coalescing is to wait before generating an interrupt until a certain number of packets have been received or sent. An interrupt is also generated if a smaller number of packets have been received followed by a certain period of time with no packet reception. This is a trade-off of latency for bandwidth and is accomplished using several mechanisms of the hardware including a counter for packets received or transmitted and a packet timer. These two hardware mechanisms work in combination to allow a reduction in the number of interrupts processed by the CPU for packet reception.

Unserviced Packet Count

The purpose of the packet counter is to count the number of packets received or transmitted and provide an interrupt when a specific number of packets have been processed by the hardware. An interrupt is generated whenever the counter is greater than or equal to the Packet Count Threshold. This counter contains an accurate count of the number of packets that the hardware has processed, either received or transmitted, and the software has not serviced.

The packet counter allows the number of interrupts to be reduced by waiting to generate an interrupt until enough packets are received. For packet reception, packet counts of less than the number to generate an interrupt would not be serviced without the addition of a packet timer. This counter is continuously updated by the hardware, not latched to the value at the time the interrupt occurred.

The packet counter can be used within the interrupt service routine for the device to reduce the number of interrupts. The interrupt service routine loops while performing processing for each packet which has been received or transmitted and decrements the counter by a specified value. At the same time, the hardware is possibly continuing to receive or transmit more packets such that the software may choose, based upon the value in the packet counter, to remain in the interrupt service routine rather than exiting and immediately returning. This feature should be used with caution as reducing the number of interrupts is beneficial, but unbounded interrupt processing is not desirable.

Since the hardware may be incrementing the packet counter simultaneously with the software decrementing the counter, there is a need for atomic operations. The hardware ensures that the operation is atomic such that simultaneous accesses are properly handled.

Packet Wait Bound

The purpose of the packet wait bound is to augment the unserviced packet count. Whenever there is no pending interrupt for the channel and the unserviced packet count is non-zero, a timer starts counting timeout at the value contained in the packet wait bound register. If the timeout is reached, an interrupt is generated such that the software may service the data which was buffered.

Asserts

Asserts are used within all Xilinx drivers to enforce constraints on argument values. Asserts can be turned off on a system-wide basis by defining, at compile time, the NDEBUG identifier. By default, asserts are turned on and it is recommended that application developers leave asserts on during development.

Note:

Special Test Conditions

The Put Pointer points to the next location in the descriptor list to copy in a new descriptor. The Get Pointer points to the next location in the list to get a descriptor from. The Get Pointer only allows software to have a traverse the list after the hardware has finished processing some number of descriptors. The Commit Pointer points to the descriptor in the list which is to be committed. It is also used to determine that no descriptor is waiting to be committed (NULL). The Last Pointer points to the last descriptor that was put into the list. It typically points to the previous descriptor to the one pointed to by the Put Pointer. Comparisons are done between these pointers to determine when the following special conditions exist.

Single Put And Commit

The buffer descriptor is ready to be used by the hardware so it is important for the descriptor to not appear to be waiting to be committed. The commit pointer is reset when a commit is done indicating there are no descriptors waiting to be committed. In all cases but this one, the descriptor is changed to cause the hardware to go to the next descriptor after processing this one. But in this case, this is the last descriptor in the list such that it must not be changed.

3 Or More Puts And Commit

A series of 3 or more puts followed by a single commit is different in that only the 1st descriptor put into the list is changed when the commit is done. This requires each put starting on the 3rd to change the previous descriptor so that it allows the hardware to continue to the next descriptor in the list.

The 1st Put Following A Commit

The commit caused the commit pointer to be NULL indicating that there are no descriptors waiting to be committed. It is necessary for the next put to set the commit pointer so that a commit must follow the put for the hardware to use the descriptor.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	09/16/03	First release

```
#include "xdma_multi.h"
#include "xbasic_types.h"
#include "xio.h"
#include "xbuf_descriptor.h"
#include "xstatus.h"
```

Functions

XStatus XDmaMulti_SgStart (**XDmaMulti** *InstancePtr, unsigned Channel)

XStatus XDmaMulti_SgStop (**XDmaMulti** *InstancePtr, unsigned Channel, XBufDescriptor
**BufDescriptorPtr)

XStatus XDmaMulti_CreateSgList (**XDmaMulti** *InstancePtr, unsigned Channel, **Xuint32**
*BdMemoryPtr, **Xuint32** ByteCount)

Xboolean XDmaMulti_IsSgListEmpty (**XDmaMulti** *InstancePtr, unsigned Channel)

XStatus XDmaMulti_PutDescriptor (**XDmaMulti** *InstancePtr, unsigned Channel,
XBufDescriptor *BufferDescriptorPtr)

XStatus XDmaMulti_CommitPuts (**XDmaMulti** *InstancePtr, unsigned Channel)

XStatus XDmaMulti_GetDescriptor (**XDmaMulti** *InstancePtr, unsigned Channel,
XBufDescriptor **BufDescriptorPtr)

Function Documentation

XStatus	(XDmaMulti *
XDmaMulti_CommitPuts	<i>InstancePtr,</i>
	unsigned <i>Channel</i>
)

This function commits the buffer descriptors which have been put into the scatter list for a DMA channel since the last commit operation was performed. This enables the calling functions to put several buffer descriptors into the list (e.g., a packet's worth) before allowing the scatter gather operations to start. This prevents the DMA channel hardware from starting to use the buffer descriptors in the list before they are ready to be used (multiple buffer descriptors for a single packet).

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.

Channel is the particular channel of interest.

Returns:

- XST_SUCCESS if the buffer descriptors of the list were successfully committed.
- XST_DMA_SG_NOTHING_TO_COMMIT indicates that the buffer descriptors were not committed because there was nothing to commit in the list. All the buffer descriptors which are in the list are committed.

Note:

None.

```
XStatus  
XDmaMulti_CreateSgList  
  
    ( XDmaMulti *  
      unsigned  
      Xuint32 *  
      Xuint32  
    )  
      InstancePtr,  
      Channel,  
      BdMemoryPtr,  
      ByteCount
```

This function creates a scatter gather list for a channel of DMA. A scatter gather list consists of a list of buffer descriptors that are available to be used for scatter gather operations. Buffer descriptors are put into the list to request a scatter gather operation to be performed.

A number of buffer descriptors are created from the specified memory and put into a buffer descriptor list as empty buffer descriptors. This function must be called before non-empty buffer descriptors may be put into the DMA channel to request scatter gather operations.

Parameters:

- InstancePtr* contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.
- Channel* is the particular channel of interest.
- BdMemoryPtr* contains a pointer to the memory which is to be used for buffer descriptors and must not be cached.
- ByteCount* contains the number of bytes for the specified memory to be used for buffer descriptors.

Returns:

- XST_SUCCESS if the scatter gather list was successfully created.
- XST_DMA_SG_LIST_EXISTS indicates that the scatter gather list was not created because the list has already been created.

Note:

None.

```
XStatus  
XDmaMulti_GetDescriptor ( XDmaMulti * InstancePtr,  
                          unsigned Channel,  
                          XBufDescriptor ** BufDescriptorPtr  
                          )
```

This function gets a buffer descriptor from the scatter gather list of the DMA channel. The buffer descriptor is retrieved from the scatter gather list and the scatter gather list is updated to not include the retrieved buffer descriptor. This is typically done after a scatter gather operation completes indicating that a data buffer has been successfully sent or data has been received into the data buffer. The purpose of this function is to allow the device using the scatter gather operation to get the results of the operation.

Parameters:

- InstancePtr* contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.
- Channel* is the particular channel of interest.
- BufDescriptorPtr* is a pointer to a pointer to the buffer descriptor which was retrieved from the list. The buffer descriptor is not really removed from the list, but it is changed to a state such that the hardware will not use it again until it is put into the scatter gather list of a DMA channel.

Returns:

- XST_SUCCESS if a buffer descriptor was retrieved
- XST_DMA_SG_NO_LIST indicates the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY indicates no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.
- BufDescriptorPtr is updated to point to the buffer descriptor which was retrieved from the list if the status indicates success.

Note:

None.

```
Xboolean ( XDmaMulti *  
XDmaMulti_IsSgListEmpty unsigned InstancePtr,  
Channel )
```

This function determines if the scatter gather list of a DMA channel is empty with regard to buffer descriptors which are pointing to buffers to be used for scatter gather operations.

Channel is the particular channel of interest.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.

Channel is the particular channel of interest.

Returns:

- o XTRUE if the scatter gather list is empty, otherwise a value of XFALSE.

Note:

None.

```
XStatus  
XDmaMulti_PutDescriptor ( XDmaMulti * InstancePtr,  
                          unsigned Channel,  
                          XBufDescriptor * BufferDescriptorPtr  
                          )
```

DESCRIPTION:

This function puts a buffer descriptor into the DMA channel scatter gather list. A DMA channel maintains a list of buffer descriptors which are to be processed. This function puts the specified buffer descriptor at the next location in the list. Note that since the list is already intact, the information in the parameter is copied into the list (rather than modify list pointers on the fly).

After buffer descriptors are put into the list, they must also be committed by calling another function. This allows multiple buffer descriptors which span a single packet to be put into the list while preventing the hardware from starting the first buffer descriptor of the packet.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.

Channel is the particular channel of interest.

BufferDescriptorPtr is a pointer to the buffer descriptor to be put into the next available location of the scatter gather list.

Returns:

- XST_SUCCESS if the buffer descriptor was successfully put into the scatter gather list.
- XST_DMA_SG_NO_LIST indicates the scatter gather list has not been created.
- XST_DMA_SG_LIST_FULL indicates the buffer descriptor was not put into the list because the list was full.
- XST_DMA_SG_BD_LOCKED indicates the buffer descriptor was not put into the list because the buffer descriptor in the list which is to be overwritten was locked. A locked buffer descriptor indicates the higher layered software is still using the buffer descriptor.

Note:

It is necessary to create a scatter gather list for a DMA channel before putting buffer descriptors into it.

```
XStatus ( XDmaMulti *  
XDmaMulti_SgStart InstancePtr,  
 unsigned Channel  
 )
```

This function starts a scatter gather operation for a scatter gather DMA channel. The first buffer descriptor in the buffer descriptor list will be started with the scatter gather operation. A scatter gather list should have previously been created for the channel of DMA and buffer descriptors put into the scatter gather list such that there are scatter operations ready to be performed.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.

Channel is the particular channel of interest.

Returns:

- XST_SUCCESS if scatter gather was started successfully
- XST_DMA_SG_NO_LIST indicates the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY indicates scatter gather was not started because the scatter gather list of the DMA channel does not contain any buffer descriptors that are ready to be

processed by the hardware.

- XST_DMA_SG_IS_STARTED indicates scatter gather was not started because the scatter gather was not stopped, but was already started.
- XST_DMA_SG_BD_NOT_COMMITTED indicates the buffer descriptor of scatter gather list which was to be started is not committed to the list. This status is more likely if this function is being called from an ISR and non-ISR processing is putting descriptors into the list.
- XST_DMA_SG_NO_DATA indicates that the buffer descriptor of the scatter gather list which was to be started had already been used by the hardware for a DMA transfer that has been completed.

Note:

It is the responsibility of the caller to get all the buffer descriptors after performing a stop operation and before performing a start operation. If buffer descriptors are not retrieved between stop and start operations, buffer descriptors may be processed by the hardware more than once.

```
XStatus ( XDmaMulti * InstancePtr,  
XDmaMulti_SgStop unsigned Channel,  
XBufDescriptor ** BufDescriptorPtr )
```

This function stops a scatter gather operation for a scatter gather DMA channel. This function starts the process of stopping a scatter gather operation that is in progress and waits for the stop to be completed. Since it waits for the operation to be stopped before returning, this function could take an amount of time relative to the size of the DMA scatter gather operation which is in progress. The scatter gather list of the DMA channel is not modified by this function such that starting the scatter gather channel after stopping it will cause it to resume. This operation is considered to be a graceful stop in that the scatter gather operation completes the current buffer descriptor before stopping.

If the interrupt is enabled, an interrupt will be generated when the operation is stopped and the caller is responsible for handling the interrupt.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on. The DMA channel should be configured to use scatter gather in order for this function to be called.

Channel is the particular channel of interest.

BufDescriptorPtr is also a return value which contains a pointer to the buffer descriptor which the scatter gather operation completed when it was stopped.

Returns:

- XST_SUCCESS if scatter gather was stopped successfully
- XST_DMA_SG_IS_STOPPED indicates scatter gather was not stopped because the scatter gather is not started, but was already stopped.
- BufDescriptorPtr contains a pointer to the buffer descriptor which was completed when the operation was stopped.

Note:

This function implements a loop which polls the hardware for the Stop indication. If the hardware is hung or malfunctioning, this function will loop for an infinite amount of time and this function may never return.

dma/v1_00_a/src/xdma_multi.c File Reference

Detailed Description

Description

This file contains the multichannel DMA component. This component supports a multichannel DMA design in which each device can have it's own dedicated multiple channel DMA, as opposed to a centralized DMA design. This component performs processing for multichannel DMA on all devices.

See [xdma_multi.h](#) for more information about this component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	09/16/03	First release

```
#include "xdma_multi.h"  
#include "xbasic_types.h"  
#include "xio.h"
```

Functions

XStatus XDmaMulti_Initialize (**XDmaMulti** *InstancePtr, **Xuint32** BaseAddress, **Xuint32** *UserMemoryPtr, **Xuint16** ChannelCount)

XStatus XDmaMulti_SelfTest (**XDmaMulti** *InstancePtr)

void **XDmaMulti_Reset** (**XDmaMulti** *InstancePtr, unsigned Channel)

void **XDmaMulti_Transfer** (**XDmaMulti** *InstancePtr, unsigned Channel, **Xuint32** *SourcePtr, **Xuint32** *DestinationPtr, **Xuint32** ByteCount)

Function Documentation

```
XStatus
XDmaMulti_Initialize
    ( XDmaMulti *
      InstancePtr,
      Xuint32
      BaseAddress,
      Xuint32 *
      UserMemoryPtr,
      Xuint16
      ChannelCount
    )
```

This function initializes the multichannel DMA. This function must be called prior to using the multichannel DMA. Initialization of the instance includes setting up the registers base address, register access tables, and resetting the channels such that they are in a known state. Interrupts for the channels are disabled when the channels are reset.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on.

BaseAddress contains the base address of the registers for the multichannel DMA.

UserMemoryPtr is a pointer to user allocated memory for the register access tables and the buffer descriptor tables. The **XDmaMulti_mSizeNeeded()** macro returns the amount of memory needed for the number of channels configured.

ChannelCount is the total number of transmit and receive channels associated with the device

Returns:

- o XST_SUCCESS indicating initialization was successful.

Note:

None.

```
void
XDmaMulti_Reset
    ( XDmaMulti *
      InstancePtr,
      unsigned
      Channel
    )
```

This function resets a particular channel of the multichannel DMA. This is a destructive operation such that it should not be done while a channel is being used. If the channel of the DMA is transferring data into other blocks, such as a FIFO, it may be necessary to reset other blocks. This function does not modify the contents of the scatter gather list for the specified channel such that the user is responsible for getting buffer descriptors from the list if necessary.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on.
Channel is the particular channel that is to be reset.

Returns:

None.

Note:

The registers are set to zero because they are in BRAM in this device. There is no initial state for BRAM and therefore it must be initialized to actually accomplish the reset state for the device..

XStatus (**XDmaMulti** *)
XDmaMulti_SelfTest *InstancePtr*

DESCRIPTION:

This function performs a self test on the multichannel DMA. This self test is destructive in that channel 0 of the DMA is reset and DMACR defaults are verified.

Parameters:

InstancePtr is a pointer to the multichannel DMA to be operated on.

Returns:

- XST_SUCCESS is returned if the self test is successful, or one of the following errors.
- XST_FAILURE Indicates the test failed

Note:

This test does not perform a DMA transfer to test the channel because the DMA hardware will not currently allow a non-local memory transfer to non-local memory (memory copy), but only allows a non-local memory to or from the device memory (typically a FIFO).

```

void
XDmaMulti_Transfer
    ( XDmaMulti *
      InstancePtr,
      unsigned
      Xuint32 *
      Channel,
      Xuint32 *
      SourcePtr,
      Xuint32 *
      DestinationPtr,
      Xuint32
      ByteCount
    )

```

This function starts the specified channel of DMA transferring data from a memory source to a memory destination. This function only starts the operation and returns before the operation may be complete. If the interrupt is enabled, an interrupt will be generated when the operation is complete, otherwise it is necessary to poll the channel status to determine when it's complete. It is the responsibility of the caller to determine when the operation is complete by handling the generated interrupt or polling the status. It is also the responsibility of the caller to ensure that the DMA channel specified is not busy with another transfer before calling this function.

Parameters:

InstancePtr contains a pointer to the multichannel DMA to operate on.

Channel is the particular channel of interest.

SourcePtr contains a pointer to the source memory where the data is to be transferred from and must be 32 bit aligned.

DestinationPtr contains a pointer to the destination memory where the data is to be transferred and must be 32 bit aligned.

ByteCount contains the number of bytes to transfer during the DMA operation.

Returns:

None.

Note:

The DMA h/w will not currently allow a non-local memory transfer to non-local memory (memory copy), but only allows a non-local memory to or from the device memory (typically a FIFO).

It is the responsibility of the caller to ensure that the cache is flushed and invalidated both before and after the DMA operation completes if the memory pointed to is cached. The caller must also ensure that the pointers contain a physical address rather than a virtual address if address translation is being used.

dsadc/v1_00_a/src/xdsadc_intr.c File Reference

Detailed Description

Functions in this file are for the interrupt driven processing functionality. See [xdsadc.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a ecm 05/10/04 First release

```
#include "xstatus.h"  
#include "xdsadc_1.h"  
#include "xio.h"  
#include "xdsadc.h"
```

Functions

void [XDsAdc_EnableInterrupts](#) ([XDsAdc](#) *InstancePtr)

void [XDsAdc_DisableInterrupts](#) ([XDsAdc](#) *InstancePtr)

[Xuint32](#) [XDsAdc_GetInterruptStatus](#) ([XDsAdc](#) *InstancePtr)

void [XDsAdc_ClearInterruptStatus](#) ([XDsAdc](#) *InstancePtr, [Xuint32](#) Mask)

Function Documentation

```
void ( XDsAdc *
XDsAdc_ClearInterruptStatus InstancePtr,
Xuint32 Mask
)
```

Clears the interrupt status in the register as directed by the bits set in the provided Mask

Parameters:

InstancePtr contains a pointer to the D-S ADC device instance for the interrupt.

Mask contains the interrupt sources to be acknowledged/cleared.

Returns:

None.

Note:

```
void ( XDsAdc *
XDsAdc_DisableInterrupts InstancePtr
)
```

Disables the interrupt from the DS-ADC (the higher layer software is responsible for disabling interrupts at the interrupt controller).

To start using the device again, `_EnableInterrupts` must be called.

Parameters:

InstancePtr is a pointer to the **XDsAdc** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XDsAdc *
XDsAdc_EnableInterrupts InstancePtr
)
```

Enables the DsAdc Interrupt.

This function must be called before other functions to convert data when in interrupt driven mode. The user should have connected the interrupt handler of the application to an interrupt source such as an interrupt controller or the processor interrupt prior to this function being called.

Parameters:

InstancePtr is a pointer to the **XDsAdc** instance to be worked on.

Returns:

None.

Note:

None

Xuint32 (**XDsAdc** *)
XDsAdc_GetInterruptStatus *InstancePtr*

Get the interrupt status from the registers to determine the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the D-S ADC device instance for the interrupt.

Returns:

The currently asserted interrupts

None.

Note:

dsdac/v1_00_a/src/xdsdac_intr.c File Reference

Detailed Description

Functions in this file are for the interrupt driven processing functionality. See [xdsdac.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a ecm 03/30/04 First release

```
#include "xstatus.h"  
#include "xio.h"  
#include "xdsdac.h"
```

Functions

void [XDsdac_EnableInterrupts](#) ([XDsdac](#) *InstancePtr)

void [XDsdac_DisableInterrupts](#) ([XDsdac](#) *InstancePtr)

[Xuint32](#) [XDsdac_GetInterruptStatus](#) ([XDsdac](#) *InstancePtr)

void [XDsdac_ClearInterruptStatus](#) ([XDsdac](#) *InstancePtr, [Xuint32](#) Mask)

void [XDsdac_SetInterruptDepth](#) ([XDsdac](#) *InstancePtr, [Xuint32](#) Depth)

Function Documentation

```
void ( XDsDac *
XDsDac_ClearInterruptStatus InstancePtr,
Xuint32 Mask
)
```

Clears the interrupt status in the register as directed by the bits set in the provided Mask

Parameters:

InstancePtr contains a pointer to the D-S DAC device instance for the interrupt.

Mask contains the interrupt sources to be cleared. Bits are defined in [xdsdac_1.h](#).

Returns:

None.

Note:

```
void ( XDsDac *
XDsDac_DisableInterrupts InstancePtr
)
```

Disables the interrupts from the device(the higher layer software is responsible for disabling interrupts at the interrupt controller).

To start using the device again in interrupt driven mode, [XDsDac_EnableInterrupts\(\)](#) must be called.

Parameters:

InstancePtr is a pointer to the [XDsDac](#) instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XDsDac *
XDsDac_EnableInterrupts InstancePtr
)
```

Enable the DsDac Interrupts.

This function must be called before other functions to transmit data in interrupt driven mode. The user should have connected the interrupt handler within the application to an interrupt source such as an interrupt controller or the processor interrupt prior to this function being called.

Parameters:

InstancePtr is a pointer to the **XDsDac** instance to be worked on.

Returns:

None.

Note:

None

```
Xuint32 ( XDsDac * )  
XDsDac_GetInterruptStatus InstancePtr
```

Get the interrupt status from the registers to determine the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the D-S DAC device instance for the interrupt.

Returns:

The currently asserted interrupts. Bits are defined in **xdsdac_1.h**.

None.

Note:

```
void ( XDsDac *  
XDsDac_SetInterruptDepth InstancePtr,  
 Xuint32 Depth  
 )
```

Set the level of the FIFO which triggers the interrupt.

Parameters:

InstancePtr is the instance of the device to operate on
Depth is the intended interrupt trigger depth (0 through
XDAC_MAX_DATA_LENGTH-1 samples)

Returns:

none.

Note:

Generated on 24 Jun 2004 for Xilinx Device Drivers

emac/v1_00_e/src/xemac.h File Reference

Detailed Description

The Xilinx Ethernet driver component. This component supports the Xilinx Ethernet 10/100 MAC (EMAC).

The Xilinx Ethernet 10/100 MAC supports the following features:

- Simple and scatter-gather DMA operations, as well as simple memory mapped direct I/O interface (FIFOs).
- Media Independent Interface (MII) for connection to external 10/100 Mbps PHY transceivers.
- MII management control reads and writes with MII PHYs
- Independent internal transmit and receive FIFOs
- CSMA/CD compliant operations for half-duplex modes
- Programmable PHY reset signal
- Unicast, broadcast, and promiscuous address filtering
- Reception of all multicast addresses (no multicast filtering yet) (NOTE: EMAC core may not support this - check the specification)
- Internal loopback
- Automatic source address insertion or overwrite (programmable)
- Automatic FCS insertion and stripping (programmable)
- Automatic pad insertion and stripping (programmable)
- Pause frame (flow control) detection in full-duplex mode
- Programmable interframe gap
- VLAN frame support.
- Pause frame support

The device driver supports all the features listed above.

Driver Description

The device driver enables higher layer software (e.g., an application) to communicate to the EMAC. The driver handles transmission and reception of Ethernet frames, as well as configuration of the controller. It does not handle protocol stack functionality such as Link Layer Control (LLC) or the Address Resolution Protocol (ARP). The protocol stack that makes use of the driver handles this functionality. This implies that the driver is simply a pass-through mechanism between a protocol stack and the EMAC. A single device driver can support multiple EMACs.

The driver is designed for a zero-copy buffer scheme. That is, the driver will not copy buffers. This avoids potential throughput bottlenecks within the driver.

Since the driver is a simple pass-through mechanism between a protocol stack and the EMAC, no assembly or disassembly of Ethernet frames is done at the driver-level. This assumes that the protocol stack passes a correctly formatted Ethernet frame to the driver for transmission, and that the driver does not validate the contents of an incoming frame

Buffer Alignment

It is important to note that when using direct FIFO communication (either polled or interrupt-driven), packet buffers must be 32-bit aligned. When using DMA and the OPB 10/100 Ethernet core, packet buffers must be 32-bit aligned. When using DMA and the PLB 10/100 Ethernet core, packet buffers must be 64-bit aligned. When using scatter-gather DMA, the buffer descriptors must be 32-bit aligned (for either the OPB or the PLB core). The driver may not enforce this alignment and it is up to the user to guarantee the proper alignment.

PHY Communication

The driver provides rudimentary read and write functions to allow the higher layer software to access the PHY. The EMAC provides MII registers for the driver to access. This management interface can be parameterized away in the FPGA implementation process. If this is the case, the PHY read and write functions of the driver return XST_NO_FEATURE.

External loopback is usually supported at the PHY. It is up to the user to turn external loopback on or off at the PHY. The driver simply provides pass-through functions for configuring the PHY. The driver does not read, write, or reset the PHY on its own. All control of the PHY must be done by the user.

Asynchronous Callbacks

The driver services interrupts and passes Ethernet frames to the higher layer software through asynchronous callback functions. When using the driver directly (i.e., not with the RTOS protocol stack), the higher layer software must register its callback functions during initialization. The driver requires callback functions for received frames, for confirmation of transmitted frames, and for asynchronous errors.

Interrupts

The driver has no dependencies on the interrupt controller. The driver provides two interrupt handlers.

[XEmac_IntrHandlerDma\(\)](#) handles interrupts when the EMAC is configured with scatter-gather DMA.

[XEmac_IntrHandlerFifo\(\)](#) handles interrupts when the EMAC is configured for direct FIFO I/O or simple DMA. Either of these routines can be connected to the system interrupt controller by the user.

Interrupt Frequency

When the EMAC is configured with scatter-gather DMA, the frequency of interrupts can be controlled with the interrupt coalescing features of the scatter-gather DMA engine. The frequency of interrupts can be adjusted using the driver API functions for setting the packet count threshold and the packet wait bound values.

The scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

The packet wait bound is a timer value used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold.

These values can be tuned by the user to meet their needs. If there appear to be interrupt latency problems or delays in packet arrival that are longer than might be expected, the user should verify that the packet count threshold is set low enough to receive interrupts before the wait bound timer goes off.

Device Reset

Some errors that can occur in the device require a device reset. These errors are listed in the [XEmac_SetErrorHandler\(\)](#) function header. The user's error handler is responsible for resetting the device and re-configuring it based on its needs (the driver does not save the current configuration). When integrating into an RTOS, these reset and re-configure obligations are

taken care of by the Xilinx adapter software if it exists for that RTOS.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are stored in the `xemac_g.c` files. A table is defined where each entry contains configuration information for an EMAC device. This information includes such things as the base address of the memory-mapped device, the base addresses of IPIF, DMA, and FIFO modules within the device, and whether the device has DMA, counter registers, multicast support, MII support, and flow control.

The driver tries to use the features built into the device. So if, for example, the hardware is configured with scatter-gather DMA, the driver expects to start the scatter-gather channels and expects that the user has set up the buffer descriptor lists already. If the user expects to use the driver in a mode different than how the hardware is configured, the user should modify the configuration table to reflect the mode to be used. Modifying the configuration table is a workaround for now until we get some experience with how users are intending to use the hardware in its different configurations. For example, if the hardware is built with scatter-gather DMA but the user is intending to use only simple DMA, the user either needs to modify the config table as a workaround or rebuild the hardware with only simple DMA. The recommendation at this point is to build the hardware with the features you intend to use. If you're inclined to modify the table, do so before the call to `XEmac_Initialize()`. Here is a snippet of code that changes a device to simple DMA (the hardware needs to have DMA for this to work of course):

```
XEmac_Config *ConfigPtr;

ConfigPtr = XEmac_LookupConfig(DeviceId);
ConfigPtr->IpIfDmaConfig = XEM_CFG_SIMPLE_DMA;
```

Simple DMA

Simple DMA is supported through the FIFO functions, `FifoSend` and `FifoRecv`, of the driver (i.e., there is no separate interface for it). The driver makes use of the DMA engine for a simple DMA transfer if the device is configured with DMA, otherwise it uses the FIFOs directly. While the simple DMA interface is therefore transparent to the user, the caching of network buffers is not. If the device is configured with DMA and the FIFO interface is used, the user must ensure that the network buffers are not cached or are cache coherent, since DMA will be used to transfer to and from the Emac device. If the device is configured with DMA and the user really wants to use the FIFOs directly, the user should rebuild the hardware without DMA. If unable to do this, there is a workaround (described above in Device Configuration) to modify the configuration table of the driver to fake the driver into thinking the device has no DMA. A code snippet follows:

```
XEmac_Config *ConfigPtr;

ConfigPtr = XEmac_LookupConfig(DeviceId);
ConfigPtr->IpIfDmaConfig = XEM_CFG_NO_DMA;
```

Asserts

Asserts are used within all Xilinx drivers to enforce constraints on argument values. Asserts can be turned off on a system-wide basis by defining, at compile time, the `NDEBUG` identifier. By default, asserts are turned on and it is recommended that users leave asserts on during development.

Building the driver

The **XEmac** driver is composed of several source files. Why so many? This allows the user to build and link only those parts of the driver that are necessary. Since the EMAC hardware can be configured in various ways (e.g., with or without DMA), the driver too can be built with varying features. For the most part, this means that besides always linking in **xemac.c**, you link in only the driver functionality you want. Some of the choices you have are polled vs. interrupt, interrupt with FIFOs only vs. interrupt with DMA, self-test diagnostics, and driver statistics. Note that currently the DMA code must be linked in, even if you don't have DMA in the device.

Note:

Xilinx drivers are typically composed of two components, one is the driver and the other is the adapter. The driver is independent of OS and processor and is intended to be highly portable. The adapter is OS-specific and facilitates communication between the driver and an OS.

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00b	rpm	10/08/02	Replaced HasSgDma boolean with IpifDmaConfig enumerated configuration parameter
1.00c	rpm	12/05/02	New version includes support for simple DMA and the delay argument to SgSend
1.00c	rpm	02/03/03	The XST_DMA_SG_COUNT_EXCEEDED return code was removed from SetPktThreshold in the internal DMA driver. Also avoided compiler warnings by initializing Result in the DMA interrupt service routines.
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and v2.00a of the packet fifo driver. Also supports multicast option.
1.00e	rmm	04/06/04	SGEND option added, Zero instance memory on init. Changed SG DMA callback invocation from once per packet to once for all packets received for an interrupt event. Added XEmac_GetSgRecvFreeDesc() and GetSgSendFreeDesc() functions. Moved some IFG and PHY constants to xemac_1.h .

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xparameters.h"
#include "xpacket_fifo_v2_00_a.h"
#include "xdma_channel.h"
```

Data Structures

```
struct XEmac
struct XEmac_Config
struct XEmac_Stats
```

Configuration options

Device configuration options (see the [XEmac_SetOptions\(\)](#) and [XEmac_GetOptions\(\)](#) for information on how to use these options)

```
#define XEM_UNICAST_OPTION
#define XEM_BROADCAST_OPTION
#define XEM_PROMISC_OPTION
#define XEM_FDUPLEX_OPTION
#define XEM_POLLED_OPTION
#define XEM_LOOPBACK_OPTION
#define XEM_MULTICAST_OPTION
#define XEM_FLOW_CONTROL_OPTION
#define XEM_INSERT_PAD_OPTION
#define XEM_INSERT_FCS_OPTION
#define XEM_INSERT_ADDR_OPTION
#define XEM_OVWRT_ADDR_OPTION
#define XEM_STRIP_PAD_FCS_OPTION
#define XEM_NO_SGEND_INT_OPTION
```

Typedefs for callbacks

Callback functions.

```
typedef void(* XEmac_SgHandler )(void *CallBackRef, XBufDescriptor *BdPtr, Xuint32 NumBds)
typedef void(* XEmac_FifoHandler )(void *CallBackRef)
typedef void(* XEmac_ErrorHandler )(void *CallBackRef, XStatus ErrorCode)
```

Defines

```
#define XEmac_mIsSgDma(InstancePtr)
#define XEmac_mIsSimpleDma(InstancePtr)
#define XEmac_mIsDma(InstancePtr)
```

Functions

```
XStatus XEmac_Initialize (XEmac *InstancePtr, Xuint16 DeviceId)
XStatus XEmac_Start (XEmac *InstancePtr)
XStatus XEmac_Stop (XEmac *InstancePtr)
void XEmac_Reset (XEmac *InstancePtr)
XEmac_Config * XEmac_LookupConfig (Xuint16 DeviceId)
XStatus XEmac_SelfTest (XEmac *InstancePtr)
XStatus XEmac_PollSend (XEmac *InstancePtr, Xuint8 *BufPtr, Xuint32 ByteCount)
XStatus XEmac_PollRecv (XEmac *InstancePtr, Xuint8 *BufPtr, Xuint32 *ByteCountPtr)
XStatus XEmac_SgSend (XEmac *InstancePtr, XBufDescriptor *BdPtr, int Delay)
XStatus XEmac_SgRecv (XEmac *InstancePtr, XBufDescriptor *BdPtr)
XStatus XEmac_SetPktThreshold (XEmac *InstancePtr, Xuint32 Direction, Xuint8 Threshold)
```

XStatus XEmac_GetPktThreshold (XEmac *InstancePtr, **Xuint32** Direction, **Xuint8** *ThreshPtr)
XStatus XEmac_SetPktWaitBound (XEmac *InstancePtr, **Xuint32** Direction, **Xuint32** TimerValue)
XStatus XEmac_GetPktWaitBound (XEmac *InstancePtr, **Xuint32** Direction, **Xuint32** *WaitPtr)
XStatus XEmac_SetSgRecvSpace (XEmac *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)
XStatus XEmac_SetSgSendSpace (XEmac *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)
void **XEmac_SetSgRecvHandler** (XEmac *InstancePtr, void *CallBackRef, **XEmac_SgHandler** FuncPtr)
void **XEmac_SetSgSendHandler** (XEmac *InstancePtr, void *CallBackRef, **XEmac_SgHandler** FuncPtr)
unsigned **XEmac_GetSgSendFreeDesc** (XEmac *InstancePtr)
unsigned **XEmac_GetSgRecvFreeDesc** (XEmac *InstancePtr)
void **XEmac_IntrHandlerDma** (void *InstancePtr)
XStatus XEmac_FifoSend (XEmac *InstancePtr, **Xuint8** *BufPtr, **Xuint32** ByteCount)
XStatus XEmac_FifoRecv (XEmac *InstancePtr, **Xuint8** *BufPtr, **Xuint32** *ByteCountPtr)
void **XEmac_SetFifoRecvHandler** (XEmac *InstancePtr, void *CallBackRef, **XEmac_FifoHandler** FuncPtr)
void **XEmac_SetFifoSendHandler** (XEmac *InstancePtr, void *CallBackRef, **XEmac_FifoHandler** FuncPtr)
void **XEmac_IntrHandlerFifo** (void *InstancePtr)
void **XEmac_SetErrorHandler** (XEmac *InstancePtr, void *CallBackRef, **XEmac_ErrorHandler** FuncPtr)
XStatus XEmac_SetOptions (XEmac *InstancePtr, **Xuint32** OptionFlag)
Xuint32 XEmac_GetOptions (XEmac *InstancePtr)
XStatus XEmac_SetMacAddress (XEmac *InstancePtr, **Xuint8** *AddressPtr)
void **XEmac_GetMacAddress** (XEmac *InstancePtr, **Xuint8** *BufferPtr)
XStatus XEmac_SetInterframeGap (XEmac *InstancePtr, **Xuint8** Part1, **Xuint8** Part2)
void **XEmac_GetInterframeGap** (XEmac *InstancePtr, **Xuint8** *Part1Ptr, **Xuint8** *Part2Ptr)
XStatus XEmac_MulticastAdd (XEmac *InstancePtr, **Xuint8** *AddressPtr)
XStatus XEmac_MulticastClear (XEmac *InstancePtr)
XStatus XEmac_PhyRead (XEmac *InstancePtr, **Xuint32** PhyAddress, **Xuint32** RegisterNum, **Xuint16** *PhyDataPtr)
XStatus XEmac_PhyWrite (XEmac *InstancePtr, **Xuint32** PhyAddress, **Xuint32** RegisterNum, **Xuint16** PhyData)
void **XEmac_GetStats** (XEmac *InstancePtr, **XEmac_Stats** *StatsPtr)
void **XEmac_ClearStats** (XEmac *InstancePtr)

Define Documentation

```
#define
XEM_BROADCAST_OPTION
```

Broadcast addressing (defaults on)

```
#define
XEM_FDUPLEX_OPTION
```

Full duplex mode (defaults off)

```
#define
XEM_FLOW_CONTROL_OPTION
```

Interpret pause frames in full duplex mode (defaults off)

#define**XEM_INSERT_ADDR_OPTION**

Insert source address on transmit (defaults on)

#define**XEM_INSERT_FCS_OPTION**

Insert FCS (CRC) on transmit (defaults on)

#define**XEM_INSERT_PAD_OPTION**

Pad short frames on transmit (defaults on)

#define**XEM_LOOPBACK_OPTION**

Internal loopback mode (defaults off)

#define**XEM_MULTICAST_OPTION**

Multicast address reception (defaults off)

#define**XEM_NO_SGEND_INT_OPTION**

Disables the SGEND interrupt with SG DMA. Setting this option to ON may help bulk data transfer performance when utilizing higher packet threshold counts on slower systems (default is off)

#define**XEM_OVWRT_ADDR_OPTION**

Overwrite source address on transmit. This is only used only used if source address insertion is on (defaults on)

#define**XEM_POLLED_OPTION**

Polled mode (defaults off)

#define**XEM_PROMISC_OPTION**

Promiscuous addressing (defaults off)

#define
XEM_STRIP_PAD_FCS_OPTION

Strip FCS and padding from received frames (defaults off)

#define
XEM_UNICAST_OPTION

Unicast addressing (defaults on)

#define XEmac_mIsDma(InstancePtr)

This macro determines if the device is currently configured with DMA (either simple DMA or scatter-gather DMA)

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

Boolean XTRUE if the device is configured with DMA, or XFALSE otherwise

Note:

Signature: Xboolean **XEmac_mIsDma**(XEmac *InstancePtr)

#define XEmac_mIsSgDma(InstancePtr)

This macro determines if the device is currently configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

Boolean XTRUE if the device is configured for scatter-gather DMA, or XFALSE if it is not.

Note:

Signature: Xboolean **XEmac_mIsSgDma**(XEmac *InstancePtr)

#define XEmac_mIsSimpleDma(InstancePtr)

This macro determines if the device is currently configured for simple DMA.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

Returns:

Boolean XTRUE if the device is configured for simple DMA, or XFALSE otherwise

Note:

Signature: Xboolean [XEmac_mIsSimpleDma](#)(XEmac *InstancePtr)

Typedef Documentation

```
typedef void(* XEmac_ErrorHandler)(void *CallbackRef, XStatus  
ErrorCode)
```

Callback when an asynchronous error occurs.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

ErrorCode is a Xilinx error code defined in [xstatus.h](#). Also see [XEmac_SetErrorHandler\(\)](#) for a description of possible errors.

```
typedef void(* XEmac_FifoHandler)(void  
*CallbackRef)
```

Callback when data is sent or received with direct FIFO communication or simple DMA. The user typically defines two callacks, one for send and one for receive.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

```
typedef void(* XEmac_SgHandler)(void *CallbackRef, XBufDescriptor *BdPtr, Xuint32  
NumBds)
```

Callback when data is sent or received with scatter-gather DMA.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

BdPtr is a pointer to the first buffer descriptor in a list of buffer descriptors.

NumBds is the number of buffer descriptors in the list pointed to by BdPtr.

Function Documentation

```
void XEmac_ClearStats ( XEmac * InstancePtr )
```

Clear the XEmacStats structure for this driver.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus XEmac_FifoRecv ( XEmac * InstancePtr, Xuint8 * BufPtr, Xuint32 * ByteCountPtr )
```

Receive an Ethernet frame into the buffer passed as an argument. This function is called in response to the callback function for received frames being called by the driver. The callback function is set up using SetFifoRecvHandler, and is invoked when the driver receives an interrupt indicating a received frame. The driver expects the upper layer software to call this function, `FifoRecv`, to receive the frame. The buffer supplied should be large enough to hold a maximum-size Ethernet frame.

The buffer into which the frame will be received must be 32-bit aligned. If using simple DMA and the PLB 10/100 Ethernet core, the buffer must be 64-bit aligned.

If the device is configured with DMA, simple DMA will be used to transfer the buffer from the Emac to memory. This means that this buffer should not be cached. See the comment section "Simple DMA" in [xemac.h](#) for more information.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a aligned buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_INTERRUPT if the device is not in interrupt mode
- XST_NO_DATA if there is no frame to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.
- XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

The input buffer must be big enough to hold the largest Ethernet frame.

```
XStatus          ( XEmac *
XEmac_FifoSend   InstancePtr,
                   Xuint8 * BufPtr,
                   Xuint32 ByteCount
                   )
```

Send an Ethernet frame using direct FIFO I/O or simple DMA with interrupts. The caller provides a contiguous-memory buffer and its length. The buffer must be 32-bit aligned. If using simple DMA and the PLB 10/100 Ethernet core, the buffer must be 64-bit aligned. The callback function set by using `SetFifoSendHandler` is invoked when the transmission is complete.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field.

If the device is configured with DMA, simple DMA will be used to transfer the buffer from memory to the Emac. This means that this buffer should not be cached. See the comment section "Simple DMA" in [xemac.h](#) for more information.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a aligned buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was successfully sent. An interrupt is generated when the EMAC transmits the frame and the driver calls the callback set with **XEmac_SetFifoSendHandler()**
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_INTERRUPT if the device is not in interrupt mode
- XST_FIFO_NO_ROOM if there is no room in the FIFO for this frame
- XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

```
void              ( XEmac *
XEmac_GetInterframeGap InstancePtr,
                   Xuint8 * Part1Ptr,
                   Xuint8 * Part2Ptr
                   )
```

Get the interframe gap, parts 1 and 2. See the description of interframe gap above in [XEmac_SetInterframeGap\(\)](#).

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

Part1Ptr is a pointer to an 8-bit buffer into which the interframe gap part 1 value will be copied.

Part2Ptr is a pointer to an 8-bit buffer into which the interframe gap part 2 value will be copied.

Returns:

None. The values of the interframe gap parts are copied into the output parameters.

```
void  
XEmac_GetMacAddress ( XEmac *  
                    InstancePtr,  
                    Xuint8 * BufferPtr  
                    )
```

Get the MAC address for this driver/device.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

BufferPtr is an output parameter, and is a pointer to a buffer into which the current MAC address will be copied. The buffer must be at least 6 bytes.

Returns:

None.

Note:

None.

```
Xuint32  
XEmac_GetOptions ( XEmac *  
                  InstancePtr  
                  )
```

Get Ethernet driver/device options. The 32-bit value returned is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

Returns:

The 32-bit value of the Ethernet options. The value is a bit-mask representing all options that are currently enabled. See [xemac.h](#) for a description of the available options.

Note:

None.

```
XStatus ( XEmac *
XEmac_GetPktThreshold InstancePtr,
Xuint32 Direction,
Xuint8 * ThreshPtr
)
```

Get the value of the packet count threshold for this driver/device. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

ThreshPtr is a pointer to the byte into which the current value of the packet threshold register will be copied. An output parameter. A value of 0 indicates the use of packet threshold by the hardware is disabled.

Returns:

- o XST_SUCCESS if the packet threshold was retrieved successfully
- o XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- o XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
XStatus ( XEmac *
XEmac_GetPktWaitBound InstancePtr,
Xuint32 Direction,
Xuint32 * WaitPtr
)
```

Get the packet wait bound timer for this driver/device. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case. The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

WaitPtr is a pointer to the byte into which the current value of the packet wait bound register will be copied. An output parameter. Units are in milliseconds in the range 0 - 1023. A value of 0 indicates the packet wait bound timer is disabled.

Returns:

- o XST_SUCCESS if the packet wait bound was retrieved successfully
- o XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- o XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
unsigned ( XEmac * )
XEmac_GetSgRecvFreeDesc InstancePtr
```

Return the number of free buffer descriptor slots that can be added to the receive descriptor ring with [XEmac_SgRecv\(\)](#) before filling it up.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

Returns:

- The number of descriptors that can be given to the HW with [XEmac_SgRecv\(\)](#)
- 0 if no room is left or the device is not configured for SG DMA

Note:

None.

```
unsigned ( XEmac * )
XEmac_GetSgSendFreeDesc InstancePtr
```

Return the number of free buffer descriptor slots that can be added to the send descriptor ring with [XEmac_SgSend\(\)](#) before filling it up.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

Returns:

- The number of descriptors that can be given to the HW with [XEmac_SgSend\(\)](#)
- 0 if no room is left or the device is not configured for SG DMA

Note:

None.

```
void ( XEmac *
XEmac_GetStats InstancePtr,
XEmac_Stats * StatsPtr
)
```

Get a copy of the XEmacStats structure, which contains the current statistics for this driver. The statistics are only cleared at initialization or on demand using the [XEmac_ClearStats\(\)](#) function.

The DmaErrors and FifoErrors counts indicate that the device has been or needs to be reset. Reset of the device is the responsibility of the upper layer software.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None.

Note:

None.

```
XStatus ( XEmac *  
XEmac_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize a specific [XEmac](#) instance/driver. The initialization entails:

- Initialize fields of the [XEmac](#) structure
- Clear the Ethernet statistics for this device
- Initialize the IPIF component with its register base address
- Configure the FIFO components with their register base addresses.
- If the device is configured with DMA, configure the DMA channel components with their register base addresses. At some later time, memory pools for the scatter-gather descriptor lists may be passed to the driver.
- Reset the Ethernet MAC

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

DeviceId is the unique id of the device controlled by this [XEmac](#) instance. Passing in a device id associates the generic [XEmac](#) instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None.

```
void ( void *  
XEmac_IntrHandlerDma InstancePtr
```

The interrupt handler for the Ethernet driver when configured with scatter- gather DMA.

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, Send Packet FIFO, Recv DMA channel, or Send DMA channel. The packet FIFOs only interrupt during "deadlock" conditions.

Parameters:

InstancePtr is a pointer to the **XEmac** instance that just interrupted.

Returns:

None.

Note:

None.

```
void XEmac_IntrHandlerFifo ( void * InstancePtr )
```

The interrupt handler for the Ethernet driver when configured for direct FIFO communication or simple DMA.

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, or Send Packet FIFO. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the MAC.

Parameters:

InstancePtr is a pointer to the **XEmac** instance that just interrupted.

Returns:

None.

Note:

None.

```
XEmac_Config* XEmac_LookupConfig ( Xuint16 DeviceId )
```

Lookup the device configuration based on the unique device ID. The table EmacConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

```
XStatus ( XEmac *  
XEmac_MulticastAdd InstancePtr,  
 Xuint8 * AddressPtr  
 )
```

Add a multicast address to the list of multicast addresses from which the EMAC accepts frames. The EMAC uses a hash table for multicast address filtering. Obviously, the more multicast addresses that are added reduces the accuracy of the address filtering. The upper layer software that receives multicast frames should perform additional filtering when accuracy must be guaranteed. There is no way to retrieve a multicast address or the multicast address list once added. The upper layer software should maintain its own list of multicast addresses. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

AddressPtr is a pointer to a 6-byte multicast address.

Returns:

- o XST_SUCCESS if the multicast address was added successfully
- o XST_NO_FEATURE if the device is not configured with multicast support
- o XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

Not currently supported.

```
XStatus ( XEmac *  
XEmac_MulticastClear InstancePtr  
 )
```

Clear the hash table used by the EMAC for multicast address filtering. The entire hash table is cleared, meaning no multicast frames will be accepted after this function is called. If this function is used to delete one or more multicast addresses, the upper layer software is responsible for adding back those addresses still needed for address filtering. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- o XST_SUCCESS if the multicast address list was cleared
- o XST_NO_FEATURE if the device is not configured with multicast support
- o XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

Not currently supported.

```

XStatus
XEmac_PhyRead
    ( XEmac * InstancePtr,
      Xuint32 PhyAddress,
      Xuint32 RegisterNum,
      Xuint16 * PhyDataPtr
    )

```

Read the current value of the PHY register indicated by the *PhyAddress* and the *RegisterNum* parameters. The MAC provides the driver with the ability to talk to a PHY that adheres to the Media Independent Interface (MII) as defined in the IEEE 802.3 standard.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

PhyAddress is the address of the PHY to be read (supports multiple PHYs)

RegisterNum is the register number, 0-31, of the specific PHY register to read

PhyDataPtr is an output parameter, and points to a 16-bit buffer into which the current value of the register will be copied.

Returns:

- o XST_SUCCESS if the PHY was read from successfully
- o XST_NO_FEATURE if the device is not configured with MII support
- o XST_EMAC_MII_BUSY if there is another PHY operation in progress
- o XST_EMAC_MII_READ_ERROR if a read error occurred between the MAC and the PHY

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that the read is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the *PhyRead* thread.

```

XStatus
XEmac_PhyWrite
    ( XEmac * InstancePtr,
      Xuint32 PhyAddress,
      Xuint32 RegisterNum,
      Xuint16 PhyData
    )

```

Write data to the specified PHY register. The Ethernet driver does not require the device to be stopped before writing to the PHY. Although it is probably a good idea to stop the device, it is the responsibility of the application to deem this necessary. The MAC provides the driver with the ability to talk to a PHY that adheres to the Media Independent Interface (MII) as defined in the IEEE 802.3 standard.

Parameters:

- InstancePtr* is a pointer to the **XEmac** instance to be worked on.
- PhyAddress* is the address of the PHY to be written (supports multiple PHYs)
- RegisterNum* is the register number, 0-31, of the specific PHY register to write
- PhyData* is the 16-bit value that will be written to the register

Returns:

- o XST_SUCCESS if the PHY was written to successfully. Since there is no error status from the MAC on a write, the user should read the PHY to verify the write was successful.
- o XST_NO_FEATURE if the device is not configured with MII support
- o XST_EMAC_MII_BUSY if there is another PHY operation in progress

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that the write is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PhyWrite thread.

```
XStatus ( XEmac * InstancePtr,  
XEmac_PollRecv Xuint8 * BufPtr,  
 Xuint32 * ByteCountPtr )
```

Receive an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver receives the frame directly from the MAC's packet FIFO. This is a non-blocking receive, in that if there is no frame ready to be received at the device, the function returns with an error. The MAC's error status is not checked, so statistics are not updated for polled receive. The buffer into which the frame will be received must be 32-bit aligned.

Parameters:

- InstancePtr* is a pointer to the **XEmac** instance to be worked on.
- BufPtr* is a pointer to a aligned buffer into which the received Ethernet frame will be copied.
- ByteCountPtr* is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- o XST_SUCCESS if the frame was sent successfully
- o XST_DEVICE_IS_STOPPED if the device has not yet been started
- o XST_NOT_POLLED if the device is not in polled mode
- o XST_NO_DATA if there is no frame to be received from the FIFO
- o XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.

- XST_FIFO_ERROR if the FIFO was overrun or underrun. This error is critical and requires the caller to reset the device.

Note:

Input buffer must be big enough to hold the largest Ethernet frame.

```

XStatus          ( XEmac *
XEmac_PollSend   InstancePtr,
                   Xuint8 * BufPtr,
                   Xuint32 ByteCount
                   )

```

Send an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver writes the frame directly to the MAC's packet FIFO, then enters a loop checking the device status for completion or error. Statistics are updated if an error occurs. The buffer to be sent must be 32-bit aligned.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a aligned buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_FIFO_NO_ROOM if there is no room in the EMAC's length FIFO for this frame
- XST_FIFO_ERROR if the FIFO was overrun or underrun. This error is critical and requires the caller to reset the device.
- XST_EMAC_COLLISION if the send failed due to excess deferral or late collision

Note:

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PollSend thread. On a 10Mbps MAC, it takes about 1.21 msec to transmit a maximum size Ethernet frame (1518 bytes). On a 100Mbps MAC, it takes about 121 usecs to transmit a maximum size Ethernet frame.

```

void              ( XEmac *
XEmac_Reset      InstancePtr
                   )

```

Reset the Ethernet MAC. This is a graceful reset in that the device is stopped first. Resets the DMA channels, the FIFOs, the transmitter, and the receiver. The PHY is not reset. Any frames in the scatter-gather descriptor lists will remain in the lists. The side effect of doing this is that after a reset and following a restart of the device, frames that were in the list before the reset may be transmitted or received. Reset must only be called after the driver has been initialized.

The driver is also taken out of polled mode if polled mode was set. The user is responsible for re-configuring the driver into polled mode after the reset if desired.

The configuration after this reset is as follows:

- Half duplex
- Disabled transmitter and receiver
- Enabled PHY (the PHY is not reset)
- MAC transmitter does pad insertion, FCS insertion, and source address overwrite.
- MAC receiver does not strip padding or FCS
- Interframe Gap as recommended by IEEE Std. 802.3 (96 bit times)
- Unicast addressing enabled
- Broadcast addressing enabled
- Multicast addressing disabled (addresses are preserved)
- Promiscuous addressing disabled
- Default packet threshold and packet wait bound register values for scatter-gather DMA operation
- MAC address of all zeros
- Non-polled mode

The upper layer software is responsible for re-configuring (if necessary) and restarting the MAC after the reset. Note that the PHY is not reset. PHY control is left to the upper layer software. Note also that driver statistics are not cleared on reset. It is up to the upper layer software to clear the statistics if needed.

When a reset is required due to an internal error, the driver notifies the upper layer software of this need through the ErrorHandler callback and specific status codes. The upper layer software is responsible for calling this Reset function and then re-configuring the device.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( XEmac * )  
XEmac_SelfTest InstancePtr
```

Performs a self-test on the Ethernet device. The test includes:

- Run self-test on DMA channel, FIFO, and IPIF components
- Reset the Ethernet device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run, and starting it when ready to send and receive frames.

It should be noted that data caching must be disabled when this function is called because the DMA self-test uses two local buffers (on the stack) for the transfer test.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

Returns:

XST_SUCCESS	Self-test was successful
XST_PFIPO_BAD_REG_VALUE	FIFO failed register self-test
XST_DMA_TRANSFER_ERROR	DMA failed data transfer self-test
XST_DMA_RESET_REGISTER_ERROR	DMA control register value was incorrect after a reset
XST_REGISTER_ERROR	Ethernet failed register reset test
XST_LOOPBACK_ERROR	Internal loopback failed
XST_IPIF_REG_WIDTH_ERROR	An invalid register width was passed into the function
XST_IPIF_RESET_REGISTER_ERROR	The value of a register at reset was invalid
XST_IPIF_DEVICE_STATUS_ERROR	A write to the device status register did not read back correctly
XST_IPIF_DEVICE_ACK_ERROR	A bit in the device status register did not reset when acked
XST_IPIF_DEVICE_ENABLE_ERROR	The device interrupt enable register was not updated correctly by the hardware when other registers were written to
XST_IPIF_IP_STATUS_ERROR	A write to the IP interrupt status register did not read back correctly
XST_IPIF_IP_ACK_ERROR	One or more bits in the IP status register did not reset when acked
XST_IPIF_IP_ENABLE_ERROR	The IP interrupt enable register was not updated correctly when other registers were written to

Note:

This function makes use of options-related functions, and the [XEmac_PollSend\(\)](#) and [XEmac_PollRecv\(\)](#) functions.

Because this test uses the PollSend function for its loopback testing, there is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the self-test thread.

```

void
XEmac_SetErrorHandler ( XEmac *
                        InstancePtr,
                        void *
                        CallbackRef,
                        XEmac_ErrorHandler FuncPtr
                      )

```

Set the callback function for handling asynchronous errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- XST_DMA_ERROR indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- XST_FIFO_ERROR indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- XST_RESET_ERROR indicates an unrecoverable MAC error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- XST_DMA_SG_NO_LIST indicates an attempt was made to access a scatter-gather DMA list that has not yet been created.
- XST_DMA_SG_LIST_EMPTY indicates the driver tried to get a descriptor from the receive descriptor list, but the list was empty.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```

void
XEmac_SetFifoRecvHandler ( XEmac *
                           InstancePtr,
                           void *
                           CallbackRef,
                           XEmac_FifoHandler FuncPtr
                         )

```

Set the callback function for handling confirmation of transmitted frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called by the driver once per frame sent. The callback is responsible for freeing the transmitted buffer if necessary.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallBackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void XEmac_SetFifoSendHandler ( XEmac * InstancePtr, void * CallBackRef, XEmac_FifoHandler FuncPtr )
```

Set the callback function for handling received frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called once per frame received. During the callback, the upper layer software should call `FifoRecv` to retrieve the received frame.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallBackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus ( XEmac *  
XEmac_SetInterframeGap InstancePtr,  
 Xuint8 Part1,  
 Xuint8 Part2  
 )
```

Set the Interframe Gap (IFG), which is the time the MAC delays between transmitting frames. There are two parts required. The total interframe gap is the total of the two parts. The values provided for the Part1 and Part2 parameters are multiplied by 4 to obtain the bit-time interval. The first part should be the first 2/3 of the total interframe gap. The MAC will reset the interframe gap timer if carrier sense becomes true during the period defined by interframe gap Part1. Part1 may be shorter than 2/3 the total and can be as small as zero. The second part should be the last 1/3 of the total interframe gap, but can be as large as the total interframe gap. The MAC will not reset the interframe gap timer if carrier sense becomes true during the period defined by interframe gap Part2.

The device must be stopped before setting the interframe gap.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Part1 is the interframe gap part 1 (which will be multiplied by 4 to get the bit-time interval).

Part2 is the interframe gap part 2 (which will be multiplied by 4 to get the bit-time interval).

Returns:

- o XST_SUCCESS if the interframe gap was set successfully
- o XST_DEVICE_IS_STARTED if the device has not been stopped

Note:

None.

```
XStatus ( XEmac *  
XEmac_SetMacAddress InstancePtr,  
 Xuint8 * AddressPtr  
 )
```

Set the MAC address for this driver/device. The address is a 48-bit value. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

AddressPtr is a pointer to a 6-byte MAC address.

Returns:

- o XST_SUCCESS if the MAC address was set successfully
- o XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```
XStatus ( XEmac *  
XEmac_SetOptions InstancePtr,  
 Xuint32 OptionsFlag  
 )
```

Set Ethernet driver/device options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option (i.e., you can OR the options together). A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

OptionsFlag is a bit-mask representing the Ethernet options to turn on or off. See [xemac.h](#) for a description of the available options.

Returns:

- o XST_SUCCESS if the options were set successfully
- o XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

```
XStatus ( XEmac *  
XEmac_SetPktThreshold InstancePtr,  
 Xuint32 Direction,  
 Xuint8 Threshold  
 )
```

Set the packet count threshold for this device. The device must be stopped before setting the threshold. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

Threshold is the value of the packet threshold count used during interrupt coalescing. A value of 0 disables the use of packet threshold by the hardware.

Returns:

- o XST_SUCCESS if the threshold was successfully set
- o XST_NOT_SGDMMA if the MAC is not configured for scatter-gather DMA
- o XST_DEVICE_IS_STARTED if the device has not been stopped
- o XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

The packet threshold could be set to larger than the number of descriptors allocated to the DMA channel. In this case, the wait bound will take over and always indicate data arrival. There was a check in this function that returned

an error if the threshold was larger than the number of descriptors, but that was removed because users would then have to set the threshold only after they set descriptor space, which is an order dependency that caused confusion.

```
XStatus ( XEmac *  
XEmac_SetPktWaitBound InstancePtr,  
 Xuint32 Direction,  
 Xuint32 TimerValue  
 )
```

Set the packet wait bound timer for this driver/device. The device must be stopped before setting the timer value. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case. The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

TimerValue is the value of the packet wait bound used during interrupt coalescing. It is in milliseconds in the range 0 - 1023. A value of 0 disables the packet wait bound timer.

Returns:

- o XST_SUCCESS if the packet wait bound was set successfully
- o XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- o XST_DEVICE_IS_STARTED if the device has not been stopped
- o XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
void ( XEmac *  
XEmac_SetSgRecvHandler InstancePtr,  
 void *  
 XEmac_SgHandler FuncPtr  
 )
```

Set the callback function for handling received frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame received. The head of a descriptor list is passed in along with the number of descriptors in the list. Before leaving the callback, the upper layer software should attach a new buffer to each descriptor in the list.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XEmac_SetSgRecvSpace ( XEmac * InstancePtr,  
                               Xuint32 * MemoryPtr,  
                               Xuint32 ByteCount  
                               )
```

Give the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The [xemac.h](#) file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be 32-bit aligned. An assert will occur if asserts are turned on and the memory is not aligned.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

MemoryPtr is a pointer to the aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- o XST_SUCCESS if the space was initialized successfully
- o XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- o XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XEmac_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```
void XEmac_SetSgSendHandler ( XEmac * InstancePtr,  
                              void * CallBackRef,  
                              XEmac_SgHandler FuncPtr  
                              )
```

Set the callback function for handling confirmation of transmitted frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame sent. The head of a descriptor list is passed in along with the number of descriptors in the list. The callback is responsible for freeing buffers attached to these descriptors.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallBackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XEmac_SetSgSendSpace(XEmac *InstancePtr,  
Xuint32 *MemoryPtr,  
Xuint32 ByteCount)
```

Give the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The **xemac.h** file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be 32-bit aligned. An assert will occur if asserts are turned on and the memory is not aligned.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

MemoryPtr is a pointer to the aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- o XST_SUCCESS if the space was initialized successfully
- o XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- o XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XEmac_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```

XStatus          ( XEmac *
XEmac_SgRecv    InstancePtr,
                   XBufDescriptor * BdPtr
                   )

```

Add a descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be 32-bit aligned if using the OPB Ethernet core and 64-bit aligned if using the PLB Ethernet core. This function is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of frames to replace filled buffers with empty buffers. This function can be called when the device is started or stopped. Note that it does start the scatter-gather DMA engine. Although this is not necessary during initialization, it is not a problem during initialization because the MAC receiver is not yet started.

The buffer attached to the descriptor must be aligned on both the front end and the back end.

Notification of received frames are done asynchronously through the receive callback function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- o XST_SUCCESS if a descriptor was successfully returned to the driver
- o XST_NOT_SG_DMA if the device is not in scatter-gather DMA mode
- o XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- o XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- o XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

```

XStatus          ( XEmac *
XEmac_SgSend    InstancePtr,
                   XBufDescriptor * BdPtr,
                   int              Delay
                   )

```

Send an Ethernet frame using scatter-gather DMA. The caller attaches the frame to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire Ethernet frame may or may not be contained within one descriptor. This function simply inserts the descriptor into the scatter-gather engine's transmit list. The caller is responsible for providing mutual exclusion to guarantee that a frame is contiguous in the transmit list. The buffer attached to the descriptor must be 32-bit aligned if using the OPB Ethernet core and 64-bit aligned if using the PLB Ethernet core.

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the frame, the inserts are committed, which means the descriptors for this frame are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

This call is non-blocking. Notification of error or successful transmission is done asynchronously through the send or error callback function.

Parameters:

- InstancePtr* is a pointer to the **XEmac** instance to be worked on.
- BdPtr* is the address of a descriptor to be inserted into the transmit ring.
- Delay* indicates whether to start the scatter-gather DMA channel immediately, or whether to wait. This allows the user to build up a list of more than one descriptor before starting the transmission of the packets, which allows the application to keep up with DMA and have a constant stream of frames being transmitted. Use XEM_SGDMA_NODELAY or XEM_SGDMA_DELAY, defined in **xemac.h**, as the value of this argument. If the user chooses to delay and build a list, the user must call this function with the XEM_SGDMA_NODELAY option or call **XEmac_Start()** to kick off the transmissions.

Returns:

- XST_SUCCESS if the buffer was successful sent
- XST_DEVICE_IS_STOPPED if the Ethernet MAC has not been started yet
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the descriptor list for the DMA channel is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- XST_DMA_SG_NOHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

```
XStatus ( XEmac * )  
XEmac_Start InstancePtr
```

Start the Ethernet controller as follows:

- If not in polled mode
 - Set the internal interrupt enable registers appropriately
 - Enable interrupts within the device itself. Note that connection of the driver's interrupt handler to the interrupt source (typically done using the interrupt controller component) is done by the higher layer software.
 - If the device is configured with scatter-gather DMA, start the DMA channels if the descriptor lists are not empty
- Enable the transmitter
- Enable the receiver

The PHY is enabled after driver initialization. We assume the upper layer software has configured it and the EMAC appropriately before this function is called.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully
- XST_NO_CALLBACK if a callback function has not yet been registered using the SetxxxHandler function.

This is required if in interrupt mode.

- XST_DEVICE_IS_STARTED if the device is already started
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.

Note:

The driver tries to match the hardware configuration. So if the hardware is configured with scatter-gather DMA, the driver expects to start the scatter-gather channels and expects that the user has set up the buffer descriptor lists already. If the user expects to use the driver in a mode different than how the hardware is configured, the user should modify the configuration table to reflect the mode to be used. Modifying the config table is a workaround for now until we get some experience with how users are intending to use the hardware in its different configurations. For example, if the hardware is built with scatter-gather DMA but the user is intending to use only simple DMA, the user either needs to modify the config table as a workaround or rebuild the hardware with only simple DMA.

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

```
XStatus ( XEmac * )  
XEmac_Stop InstancePtr
```

Stop the Ethernet MAC as follows:

- If the device is configured with scatter-gather DMA, stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver
- Disable interrupts if not in polled mode (the higher layer software is responsible for disabling interrupts at the interrupt controller)

The PHY is left enabled after a Stop is called.

If the device is configured for scatter-gather DMA, the DMA engine stops at the next buffer descriptor in its list. The remaining descriptors in the list are not removed, so anything in the list will be transmitted or received when the device is restarted. The side effect of doing this is that the last buffer descriptor processed by the DMA engine before stopping may not be the last descriptor in the Ethernet frame. So when the device is restarted, a partial frame (i.e., a bad frame) may be transmitted/received. This is only a concern if a frame can span multiple buffer descriptors, which is dependent on the size of the network buffers.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

emac/v1_00_e/src/xemac_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of EMAC devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and
v2.00a	of		the packet fifo driver.

```
#include "xemac.h"  
#include "xparameters.h"
```

Variables

[XEmac_Config](#) [XEmac_ConfigTable](#) [XPAR_XEMAC_NUM_INSTANCES]

Variable Documentation

[XEmac_Config](#) [XEmac_ConfigTable](#)
[XPAR_XEMAC_NUM_INSTANCES]

This table contains configuration information for each EMAC device in the system.

Generated on 24 Jun 2004 for Xilinx Device Drivers

XEmac_Config Struct Reference

```
#include <xemac.h>
```

Detailed Description

This typedef contains configuration information for a device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xboolean HasCounters](#)

[Xuint8 IpIfDmaConfig](#)

[Xboolean HasMii](#)

Field Documentation

[Xuint32 XEmac_Config::](#)

BaseAddress

Register base
address

[Xuint16 XEmac_Config::](#)

DeviceId

Unique ID of
device

Xboolean XEmac_Config::

HasCounters

Does device have counters?

Xboolean XEmac_Config::

HasMii

Does device support MII?

Xuint8 XEmac_Config::

IpIfDmaConfig

IPIF/DMA hardware configuration

The documentation for this struct was generated from the following file:

- [emac/v1_00_e/src/xemac.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

emac/v1_00_e/src/xemac.c File Reference

Detailed Description

The [XEmac](#) driver. Functions in this file are the minimum required functions for this driver. See [xemac.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00b	rpm	07/23/02	Removed the PHY reset from Initialize()
1.00b	rmm	09/23/02	Removed commented code in Initialize().
Recycled as			
			XEmac_mPhyReset macro in xemac_1.h .
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00c	rpm	12/12/02	Changed location of IsStarted assignment in XEmac_Start
			to be sure the flag is set before the device and interrupts are enabled.
1.00c	rpm	02/03/03	SelfTest was not clearing polled mode. Take driver out
			of polled mode in XEmac_Reset() to fix this problem.
1.00c	rmm	05/13/03	Fixed diab compiler warnings relating to asserts.
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and v2.00a of
			the packet fifo driver.
1.00e	rmm	04/06/04	Changed XEmac_Initialize() to clear the instance data.

Added XEM_NO_SGEND_INT_OPTION processing to

XEmac_Start().

```
#include "xbasic_types.h"
#include "xenv.h"
#include "xemac_i.h"
#include "xio.h"
#include "xipif_v1_23_b.h"
```

Functions

XStatus XEmac_Initialize (**XEmac** *InstancePtr, **Xuint16** DeviceId)

XStatus XEmac_Start (**XEmac** *InstancePtr)

XStatus XEmac_Stop (**XEmac** *InstancePtr)

void **XEmac_Reset** (**XEmac** *InstancePtr)

XStatus XEmac_SetMacAddress (**XEmac** *InstancePtr, **Xuint8** *AddressPtr)

void **XEmac_GetMacAddress** (**XEmac** *InstancePtr, **Xuint8** *BufferPtr)

XEmac_Config * **XEmac_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
void
XEmac_GetMacAddress ( XEmac *
                    InstancePtr,
                    Xuint8 * BufferPtr
                    )
```

Get the MAC address for this driver/device.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufferPtr is an output parameter, and is a pointer to a buffer into which the current MAC address will be copied. The buffer must be at least 6 bytes.

Returns:

None.

Note:

None.

```
XStatus ( XEmac *  
XEmac_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize a specific **XEmac** instance/driver. The initialization entails:

- Initialize fields of the **XEmac** structure
- Clear the Ethernet statistics for this device
- Initialize the IPIF component with its register base address
- Configure the FIFO components with their register base addresses.
- If the device is configured with DMA, configure the DMA channel components with their register base addresses. At some later time, memory pools for the scatter-gather descriptor lists may be passed to the driver.
- Reset the Ethernet MAC

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XEmac** instance. Passing in a device id associates the generic **XEmac** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None.

```
XEmac_Config* ( Xuint16 )  
XEmac_LookupConfig DeviceId
```

Lookup the device configuration based on the unique device ID. The table `EmacConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or `XNULL` if no match is found.

Note:

None.

```
void ( XEmac * )  
XEmac_Reset InstancePtr
```

Reset the Ethernet MAC. This is a graceful reset in that the device is stopped first. Resets the DMA channels, the FIFOs, the transmitter, and the receiver. The PHY is not reset. Any frames in the scatter-gather descriptor lists will remain in the lists. The side effect of doing this is that after a reset and following a restart of the device, frames that were in the list before the reset may be transmitted or received. Reset must only be called after the driver has been initialized.

The driver is also taken out of polled mode if polled mode was set. The user is responsible for re-configuring the driver into polled mode after the reset if desired.

The configuration after this reset is as follows:

- Half duplex
- Disabled transmitter and receiver
- Enabled PHY (the PHY is not reset)
- MAC transmitter does pad insertion, FCS insertion, and source address overwrite.
- MAC receiver does not strip padding or FCS
- Interframe Gap as recommended by IEEE Std. 802.3 (96 bit times)
- Unicast addressing enabled
- Broadcast addressing enabled
- Multicast addressing disabled (addresses are preserved)
- Promiscuous addressing disabled
- Default packet threshold and packet wait bound register values for scatter-gather DMA operation
- MAC address of all zeros
- Non-polled mode

The upper layer software is responsible for re-configuring (if necessary) and restarting the MAC after the reset. Note that the PHY is not reset. PHY control is left to the upper layer software. Note also that driver statistics are not cleared on reset. It is up to the upper layer software to clear the statistics if needed.

When a reset is required due to an internal error, the driver notifies the upper layer software of this need through the ErrorHandler callback and specific status codes. The upper layer software is responsible for calling this Reset function and then re-configuring the device.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( XEmac *  
XEmac_SetMacAddress InstancePtr,  
 Xuint8 * AddressPtr  
 )
```

Set the MAC address for this driver/device. The address is a 48-bit value. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

AddressPtr is a pointer to a 6-byte MAC address.

Returns:

- XST_SUCCESS if the MAC address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

XStatus (**XEmac** *)
XEmac_Start *InstancePtr*

Start the Ethernet controller as follows:

- If not in polled mode
 - Set the internal interrupt enable registers appropriately
 - Enable interrupts within the device itself. Note that connection of the driver's interrupt handler to the interrupt source (typically done using the interrupt controller component) is done by the higher layer software.
 - If the device is configured with scatter-gather DMA, start the DMA channels if the descriptor lists are not empty
- Enable the transmitter
- Enable the receiver

The PHY is enabled after driver initialization. We assume the upper layer software has configured it and the EMAC appropriately before this function is called.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully
- XST_NO_CALLBACK if a callback function has not yet been registered using the SetxxxHandler function. This is required if in interrupt mode.
- XST_DEVICE_IS_STARTED if the device is already started
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.

Note:

The driver tries to match the hardware configuration. So if the hardware is configured with scatter-gather DMA, the driver expects to start the scatter-gather channels and expects that the user has set up the buffer descriptor lists already. If the user expects to use the driver in a mode different than how the hardware is configured, the user should modify the configuration table to reflect the mode to be used. Modifying the config table is a workaround for now until we get some experience with how users are intending to use the hardware in its different configurations. For example, if the hardware is built with scatter-gather DMA but the user is intending to use only simple DMA, the user either needs to modify the config table as a workaround or rebuild the hardware with only simple DMA.

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a

semaphore).

```
XStatus          ( XEmac *          )  
XEmac_Stop      InstancePtr
```

Stop the Ethernet MAC as follows:

- If the device is configured with scatter-gather DMA, stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver
- Disable interrupts if not in polled mode (the higher layer software is responsible for disabling interrupts at the interrupt controller)

The PHY is left enabled after a Stop is called.

If the device is configured for scatter-gather DMA, the DMA engine stops at the next buffer descriptor in its list. The remaining descriptors in the list are not removed, so anything in the list will be transmitted or received when the device is restarted. The side effect of doing this is that the last buffer descriptor processed by the DMA engine before stopping may not be the last descriptor in the Ethernet frame. So when the device is restarted, a partial frame (i.e., a bad frame) may be transmitted/received. This is only a concern if a frame can span multiple buffer descriptors, which is dependent on the size of the network buffers.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XEmac Struct Reference

```
#include <xemac.h>
```

Detailed Description

The XEmac driver instance data. The user is required to allocate a variable of this type for every EMAC device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `emac/v1_00_e/src/xemac.h`
-

emac/v1_00_e/src/xemac_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xemac.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	rpm	04/26/02	First release
1.00b	rmm	09/23/02	Added XEmac_mPhyReset macro
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and
v2.00a	of		the packet fifo driver.
1.00e	rmm	04/06/04	Relocated IFG and MGT max values from 'c' files to this one.

```
#include "xbasic_types.h"
```

```
#include "xio.h"
```

Defines

```
#define XEmac_mReadReg(BaseAddress, RegOffset)
```

```
#define XEmac_mWriteReg(BaseAddress, RegOffset, Data)
```

```
#define XEmac_mSetControlReg(BaseAddress, Mask)
```

```
#define XEmac_mSetMacAddress(BaseAddress, AddressPtr)
```

```
#define XEmac_mEnable(BaseAddress)
```

```
#define XEmac_mDisable(BaseAddress)
#define XEmac_mIsTxDone(BaseAddress)
#define XEmac_mIsRxEmpty(BaseAddress)
#define XEmac_mPhyReset(BaseAddress)
```

Functions

```
void XEmac_SendFrame (Xuint32 BaseAddress, Xuint8 *FramePtr, int Size)
int XEmac_RecvFrame (Xuint32 BaseAddress, Xuint8 *FramePtr)
```

Define Documentation

```
#define XEmac_mDisable( BaseAddress )
```

Disable the transmitter and receiver. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XEmac_mEnable( BaseAddress )
```

Enable the transmitter and receiver. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XEmac_mIsRxEmpty( BaseAddress )
```

Check to see if the receive FIFO is empty.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is empty, or XFALSE if it is not.

Note:

None.

```
#define XEmac_mIsTxDone( BaseAddress )
```

Check to see if the transmission is complete.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is done, or XFALSE if it is not.

Note:

None.

```
#define XEmac_mPhyReset( BaseAddress )
```

Reset MII compliant PHY

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XEmac_mReadReg( BaseAddress,  
                        RegOffset  )
```

Read the given register.

Parameters:

BaseAddress is the base address of the device

RegOffset is the register offset to be read

Returns:

The 32-bit value of the register

Note:

None.

```
#define XEmac_mSetControlReg( BaseAddress,  
                             Mask      )
```

Set the contents of the control register. Use the XEM_ECR_* constants defined above to create the bit-mask to be written to the register.

Parameters:

BaseAddress is the base address of the device

Mask is the 16-bit value to write to the control register

Returns:

None.

Note:

None.

```
#define XEmac_mSetMacAddress( BaseAddress,  
                             AddressPtr )
```

Set the station address of the EMAC device.

Parameters:

BaseAddress is the base address of the device

AddressPtr is a pointer to a 6-byte MAC address

Returns:

None.

Note:

None.

```
#define XEmac_mWriteReg( BaseAddress,  
                        RegOffset,  
                        Data      )
```

Write the given register.

Parameters:

BaseAddress is the base address of the device

RegOffset is the register offset to be written

Data is the 32-bit value to write to the register

Returns:

None.

Note:

None.

Function Documentation

```
int  
XEmac_RecvFrame ( Xuint32  
                 BaseAddress,  
                 Xuint8 * FramePtr  
                 )
```

Receive a frame. Wait for a frame to arrive.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to a 32-bit aligned buffer where the frame will be stored

Returns:

The size, in bytes, of the frame received.

Note:

None.

```
void XEmac_SendFrame ( Xuint32 BaseAddress,  
                      Xuint8 * FramePtr,  
                      int Size  
                      )
```

Send an Ethernet frame. This size is the total frame size, including header. This function blocks waiting for the frame to be transmitted.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to a 32-bit aligned frame

Size is the size, in bytes, of the frame

Returns:

None.

Note:

An early return may occur if there is no room in the FIFO for the requested frame.

emac/v1_00_e/src/xemac_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between **XEmac** components. The identifiers in this file are not intended for use external to the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00b	rpm	04/29/02	Moved register definitions to xemac_1.h
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and
v2.00a	of		the packet fifo driver.

```
#include "xemac.h"  
#include "xemac\_1.h"
```

Variables

[XEmac_Config](#) [XEmac_ConfigTable](#) []

Variable Documentation

XEmac_Config XEmac_ConfigTable ()

[]

This table contains configuration information for each EMAC device in the system.

Generated on 24 Jun 2004 for Xilinx Device Drivers

XEmac_Stats Struct Reference

```
#include <xemac.h>
```

Detailed Description

Ethernet statistics (see [XEmac_GetStats\(\)](#) and [XEmac_ClearStats\(\)](#))

Data Fields

[Xuint32 XmitFrames](#)

[Xuint32 XmitBytes](#)

[Xuint32 XmitLateCollisionErrors](#)

[Xuint32 XmitExcessDeferral](#)

[Xuint32 XmitOverrunErrors](#)

[Xuint32 XmitUnderrunErrors](#)

[Xuint32 RecvFrames](#)

[Xuint32 RecvBytes](#)

[Xuint32 RecvFcsErrors](#)

[Xuint32 RecvAlignmentErrors](#)

[Xuint32 RecvOverrunErrors](#)

[Xuint32 RecvUnderrunErrors](#)

[Xuint32 RecvMissedFrameErrors](#)

[Xuint32 RecvCollisionErrors](#)

[Xuint32 RecvLengthFieldErrors](#)

[Xuint32 RecvShortErrors](#)

[Xuint32 RecvLongErrors](#)

[Xuint32 DmaErrors](#)

[Xuint32 FifoErrors](#)

[Xuint32 RecvInterrupts](#)

Xuint32 XmitInterrupts

Xuint32 EmacInterrupts

Xuint32 TotalIntrs

Field Documentation

Xuint32 XEmac_Stats::

DmaErrors

Number of DMA errors since
init

Xuint32 XEmac_Stats::

EmacInterrupts

Number of MAC (device)
interrupts

Xuint32 XEmac_Stats::

FifoErrors

Number of FIFO errors since
init

Xuint32 XEmac_Stats::

RecvAlignmentErrors

Number of frames received with alignment
errors

Xuint32 XEmac_Stats::

RecvBytes

Number of bytes
received

Xuint32 XEmac_Stats::

RecvCollisionErrors

Number of frames discarded due to
collisions

Xuint32 XEmac_Stats::

RecvFcsErrors

Number of frames discarded due to FCS errors

Xuint32 XEmac_Stats::

RecvFrames

Number of frames received

Xuint32 XEmac_Stats::

RecvInterrupts

Number of receive interrupts

Xuint32 XEmac_Stats::

RecvLengthFieldErrors

Number of frames discarded with invalid length field

Xuint32 XEmac_Stats::

RecvLongErrors

Number of long frames discarded

Xuint32 XEmac_Stats::

RecvMissedFrameErrors

Number of frames missed by MAC

Xuint32 XEmac_Stats::

RecvOverrunErrors

Number of frames discarded due to overrun errors

Xuint32 XEmac_Stats::

RecvShortErrors

Number of short frames
discarded

Xuint32 XEmac_Stats::

RecvUnderrunErrors

Number of recv underrun
errors

Xuint32 XEmac_Stats::

TotalIntrs

Total
interrupts

Xuint32 XEmac_Stats::

XmitBytes

Number of bytes
transmitted

Xuint32 XEmac_Stats::

XmitExcessDeferral

Number of transmission failures due o excess collision
deferrals

Xuint32 XEmac_Stats::

XmitFrames

Number of frames
transmitted

Xuint32 XEmac_Stats::

XmitInterrupts

Number of transmit
interrupts

Xuint32 XEmac_Stats::

XmitLateCollisionErrors

Number of transmission failures due to late collisions

Xuint32 XEmac_Stats::

XmitOverrunErrors

Number of transmit overrun errors

Xuint32 XEmac_Stats::

XmitUnderrunErrors

Number of transmit underrun errors

The documentation for this struct was generated from the following file:

- [emac/v1_00_e/src/xemac.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

emac/v1_00_e/src/xemac_stats.c File Reference

Detailed Description

Contains functions to get and clear the **XEmac** driver statistics.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and
v2.00a	of		the packet fifo driver.

```
#include "xbasic_types.h"  
#include "xemac_i.h"
```

Functions

```
void XEmac_GetStats (XEmac *InstancePtr, XEmac_Stats *StatsPtr)  
void XEmac_ClearStats (XEmac *InstancePtr)
```

Function Documentation

```
void  
XEmac_ClearStats ( XEmac *  
                  InstancePtr )
```

Clear the XEmacStats structure for this driver.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

None.

Note:

None.

```
void XEmac_GetStats ( XEmac * InstancePtr, XEmac_Stats * StatsPtr )
```

Get a copy of the XEmacStats structure, which contains the current statistics for this driver. The statistics are only cleared at initialization or on demand using the **XEmac_ClearStats()** function.

The DmaErrors and FifoErrors counts indicate that the device has been or needs to be reset. Reset of the device is the responsibility of the upper layer software.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None.

Note:

None.

emac/v1_00_e/src/xemac_intr_fifo.c File Reference

Detailed Description

Contains functions related to interrupt mode using direct FIFO I/O or simple DMA. The driver uses simple DMA if the device is configured with DMA, otherwise it uses direct FIFO access.

The interrupt handler, [XEmac_IntrHandlerFifo\(\)](#), must be connected by the user to the interrupt controller.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00c	rpm	04/01/03	Added check in FifoSend for room in the data FIFO before starting a simple DMA transfer.
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and
v2.00a	of		the packet fifo driver.

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

```
XStatus XEmac_FifoSend (XEmac *InstancePtr, Xuint8 *BufPtr, Xuint32 ByteCount)
XStatus XEmac_FifoRecv (XEmac *InstancePtr, Xuint8 *BufPtr, Xuint32 *ByteCountPtr)
void XEmac_IntrHandlerFifo (void *InstancePtr)
void XEmac_SetFifoRecvHandler (XEmac *InstancePtr, void *CallBackRef,
    XEmac_FifoHandler FuncPtr)
void XEmac_SetFifoSendHandler (XEmac *InstancePtr, void *CallBackRef,
    XEmac_FifoHandler FuncPtr)
```

Function Documentation

```
XStatus
XEmac_FifoRecv
    ( XEmac *
      InstancePtr,
      Xuint8 * BufPtr,
      Xuint32 * ByteCountPtr
    )
```

Receive an Ethernet frame into the buffer passed as an argument. This function is called in response to the callback function for received frames being called by the driver. The callback function is set up using SetFifoRecvHandler, and is invoked when the driver receives an interrupt indicating a received frame. The driver expects the upper layer software to call this function, FifoRecv, to receive the frame. The buffer supplied should be large enough to hold a maximum-size Ethernet frame.

The buffer into which the frame will be received must be 32-bit aligned. If using simple DMA and the PLB 10/100 Ethernet core, the buffer must be 64-bit aligned.

If the device is configured with DMA, simple DMA will be used to transfer the buffer from the Emac to memory. This means that this buffer should not be cached. See the comment section "Simple DMA" in [xemac.h](#) for more information.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a aligned buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_INTERRUPT if the device is not in interrupt mode
- XST_NO_DATA if there is no frame to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.
- XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

The input buffer must be big enough to hold the largest Ethernet frame.

```

XStatus          ( XEmac *
XEmac_FifoSend   InstancePtr,
                   Xuint8 * BufPtr,
                   Xuint32 ByteCount
                   )

```

Send an Ethernet frame using direct FIFO I/O or simple DMA with interrupts. The caller provides a contiguous-memory buffer and its length. The buffer must be 32-bit aligned. If using simple DMA and the PLB 10/100 Ethernet core, the buffer must be 64-bit aligned. The callback function set by using SetFifoSendHandler is invoked when the transmission is complete.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field.

If the device is configured with DMA, simple DMA will be used to transfer the buffer from memory to the Emac. This means that this buffer should not be cached. See the comment section "Simple DMA" in [xemac.h](#) for more information.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a aligned buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was successfully sent. An interrupt is generated when

the EMAC transmits the frame and the driver calls the callback set with **XEmac_SetFifoSendHandler()**

- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_INTERRUPT if the device is not in interrupt mode
- XST_FIFO_NO_ROOM if there is no room in the FIFO for this frame
- XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA).
The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

```
void XEmac_IntrHandlerFifo ( void * InstancePtr )
```

The interrupt handler for the Ethernet driver when configured for direct FIFO communication or simple DMA.

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, or Send Packet FIFO. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the MAC.

Parameters:

InstancePtr is a pointer to the **XEmac** instance that just interrupted.

Returns:

None.

Note:

None.

```
void XEmac_SetFifoRecvHandler ( XEmac * InstancePtr, void * CallbackRef, XEmac_FifoHandler FuncPtr )
```

Set the callback function for handling confirmation of transmitted frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called by the driver once per frame sent. The callback is responsible for freeing the transmitted buffer if necessary.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void XEmac_SetFifoSendHandler ( XEmac * InstancePtr, void * CallbackRef, XEmac_FifoHandler FuncPtr )
```

Set the callback function for handling received frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called once per frame received. During the callback, the upper layer software should call `FifoRecv` to retrieve the received frame.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

Generated on 24 Jun 2004 for Xilinx Device Drivers

emac/v1_00_e/src/xemac_options.c File Reference

Detailed Description

Functions in this file handle configuration of the **XEmac** driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and
v2.00a	of		the packet fifo driver.
1.00e	rmm	04/06/04	Added XEM_NO_SGEND_INT_OPTION processing.
Relocated			XEM_MAX_IFG definition from here to xemac_1.h
as XEM_IFGP_			PART1_MAX and XEM_IFGP_PART2_MAX.

```
#include "xbasic\_types.h"  
#include "xemac\_i.h"  
#include "xio.h"
```

Data Structures

struct **OptionMap**

Functions

XStatus **XEmac_SetOptions** (**XEmac** *InstancePtr, **Xuint32** OptionsFlag)

Xuint32 **XEmac_GetOptions** (**XEmac** *InstancePtr)

XStatus **XEmac_SetInterframeGap** (**XEmac** *InstancePtr, **Xuint8** Part1, **Xuint8** Part2)

void **XEmac_GetInterframeGap** (**XEmac** *InstancePtr, **Xuint8** *Part1Ptr, **Xuint8** *Part2Ptr)

Function Documentation

```
void  
XEmac_GetInterframeGap ( XEmac *  
                        InstancePtr,  
                        Xuint8 * Part1Ptr,  
                        Xuint8 * Part2Ptr  
                        )
```

Get the interframe gap, parts 1 and 2. See the description of interframe gap above in [XEmac_SetInterframeGap\(\)](#).

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Part1Ptr is a pointer to an 8-bit buffer into which the interframe gap part 1 value will be copied.

Part2Ptr is a pointer to an 8-bit buffer into which the interframe gap part 2 value will be copied.

Returns:

None. The values of the interframe gap parts are copied into the output parameters.

```
Xuint32  
XEmac_GetOptions ( XEmac *  
                  InstancePtr
```

Get Ethernet driver/device options. The 32-bit value returned is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

The 32-bit value of the Ethernet options. The value is a bit-mask representing all options that are currently enabled. See **xemac.h** for a description of the available options.

Note:

None.

```
XStatus ( XEmac *  
XEmac_SetInterframeGap InstancePtr,  
 Xuint8 Part1,  
 Xuint8 Part2  
 )
```

Set the Interframe Gap (IFG), which is the time the MAC delays between transmitting frames. There are two parts required. The total interframe gap is the total of the two parts. The values provided for the Part1 and Part2 parameters are multiplied by 4 to obtain the bit-time interval. The first part should be the first 2/3 of the total interframe gap. The MAC will reset the interframe gap timer if carrier sense becomes true during the period defined by interframe gap Part1. Part1 may be shorter than 2/3 the total and can be as small as zero. The second part should be the last 1/3 of the total interframe gap, but can be as large as the total interframe gap. The MAC will not reset the interframe gap timer if carrier sense becomes true during the period defined by interframe gap Part2.

The device must be stopped before setting the interframe gap.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Part1 is the interframe gap part 1 (which will be multiplied by 4 to get the bit-time interval).

Part2 is the interframe gap part 2 (which will be multiplied by 4 to get the bit-time interval).

Returns:

- o XST_SUCCESS if the interframe gap was set successfully

- XST_DEVICE_IS_STOPPED if the device has not been stopped

Note:

None.

```
XStatus ( XEmac *  
XEmac_SetOptions (InstancePtr,  
 Xuint32 OptionsFlag  
 )
```

Set Ethernet driver/device options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option (i.e., you can OR the options together). A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

OptionsFlag is a bit-mask representing the Ethernet options to turn on or off. See [xemac.h](#) for a description of the available options.

Returns:

- XST_SUCCESS if the options were set successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

emac/v1_00_e/src/xemac_intr_dma.c File Reference

Detailed Description

Contains functions used in interrupt mode when configured with scatter-gather DMA.

The interrupt handler, [XEmac_IntrHandlerDma\(\)](#), must be connected by the user to the interrupt controller.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA and the delay argument to SgSend
1.00c	rpm	02/03/03	The XST_DMA_SG_COUNT_EXCEEDED return code was removed from SetPktThreshold in the internal DMA driver. Also avoided compiler warnings by initializing Result in the interrupt service routines.
1.00c	rpm	03/26/03	Fixed a problem in the interrupt service routines where the interrupt status was toggled clear after a call to ErrorHandler, but if ErrorHandler reset the device the toggle actually asserted the interrupt because

the

reset had cleared it.

1.00d rpm 09/26/03 New version includes support PLB Ethernet and v2.00a of

the packet fifo driver.

1.00e rmm 04/06/04 Instead of invoking once for each packet received,

send/recv callbacks are invoked once for all

packets.

Added functions **XEmac_GetSgRecvFreeDesc()** and

XEmac_Get-

SgSendFreeDesc().

```
#include "xbasic_types.h"
#include "xemac_i.h"
#include "xio.h"
#include "xbuf_descriptor.h"
#include "xdma_channel.h"
#include "xipif_v1_23_b.h"
```

Functions

XStatus XEmac_SgSend (**XEmac** *InstancePtr, **XBufDescriptor** *BdPtr, int Delay)

XStatus XEmac_SgRecv (**XEmac** *InstancePtr, **XBufDescriptor** *BdPtr)

void **XEmac_IntrHandlerDma** (void *InstancePtr)

XStatus XEmac_SetPktThreshold (**XEmac** *InstancePtr, **Xuint32** Direction, **Xuint8** Threshold)

XStatus XEmac_GetPktThreshold (**XEmac** *InstancePtr, **Xuint32** Direction, **Xuint8** *ThreshPtr)

XStatus XEmac_SetPktWaitBound (**XEmac** *InstancePtr, **Xuint32** Direction, **Xuint32** TimerValue)

XStatus XEmac_GetPktWaitBound (**XEmac** *InstancePtr, **Xuint32** Direction, **Xuint32** *WaitPtr)

XStatus XEmac_SetSgRecvSpace (**XEmac** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)

XStatus XEmac_SetSgSendSpace (**XEmac** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)

unsigned **XEmac_GetSgSendFreeDesc** (**XEmac** *InstancePtr)

unsigned **XEmac_GetSgRecvFreeDesc** (**XEmac** *InstancePtr)

void **XEmac_SetSgRecvHandler** (**XEmac** *InstancePtr, void *CallBackRef,

XEmac_SgHandler FuncPtr)

void **XEmac_SetSgSendHandler** (**XEmac** *InstancePtr, void *CallBackRef,

XEmac_SgHandler FuncPtr)

Function Documentation

```
XStatus ( XEmac *  
XEmac_GetPktThreshold InstancePtr,  
 Xuint32 Direction,  
 Xuint8 * ThreshPtr  
 )
```

Get the value of the packet count threshold for this driver/device. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

ThreshPtr is a pointer to the byte into which the current value of the packet threshold register will be copied. An output parameter. A value of 0 indicates the use of packet threshold by the hardware is disabled.

Returns:

- XST_SUCCESS if the packet threshold was retrieved successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
XStatus ( XEmac *  
XEmac_GetPktWaitBound InstancePtr,  
 Xuint32 Direction,  
 Xuint32 * WaitPtr  
 )
```

Get the packet wait bound timer for this driver/device. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case. The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

WaitPtr is a pointer to the byte into which the current value of the packet wait bound register will be copied. An output parameter. Units are in milliseconds in the range 0 - 1023. A value of 0 indicates the packet wait bound timer is disabled.

Returns:

- XST_SUCCESS if the packet wait bound was retrieved successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

unsigned (**XEmac** *)
XEmac_GetSgRecvFreeDesc *InstancePtr*

Return the number of free buffer descriptor slots that can be added to the receive descriptor ring with **XEmac_SgRecv()** before filling it up.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- The number of descriptors that can be given to the HW with **XEmac_SgRecv()**
- 0 if no room is left or the device is not configured for SG DMA

Note:

None.

```
unsigned ( XEmac * )
XEmac_GetSgSendFreeDesc InstancePtr
```

Return the number of free buffer descriptor slots that can be added to the send descriptor ring with **XEmac_SgSend()** before filling it up.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- o The number of descriptors that can be given to the HW with **XEmac_SgSend()**
- o 0 if no room is left or the device is not configured for SG DMA

Note:

None.

```
void ( void * )
XEmac_IntrHandlerDma InstancePtr
```

The interrupt handler for the Ethernet driver when configured with scatter- gather DMA.

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, Send Packet FIFO, Recv DMA channel, or Send DMA channel. The packet FIFOs only interrupt during "deadlock" conditions.

Parameters:

InstancePtr is a pointer to the **XEmac** instance that just interrupted.

Returns:

None.

Note:

None.

```
XStatus ( XEmac *
XEmac_SetPktThreshold InstancePtr,
Xuint32 Direction,
Xuint8 Threshold
)
```

Set the packet count threshold for this device. The device must be stopped before setting the threshold. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

Threshold is the value of the packet threshold count used during interrupt coalescing. A value of 0 disables the use of packet threshold by the hardware.

Returns:

- XST_SUCCESS if the threshold was successfully set
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

The packet threshold could be set to larger than the number of descriptors allocated to the DMA channel. In this case, the wait bound will take over and always indicate data arrival. There was a check in this function that returned an error if the threshold was larger than the number of descriptors, but that was removed because users would then have to set the threshold only after they set descriptor space, which is an order dependency that caused confusion.

```
XStatus XEmac_SetPktWaitBound (XEmac *InstancePtr,  
Xuint32 Direction,  
Xuint32 TimerValue)
```

Set the packet wait bound timer for this driver/device. The device must be stopped before setting the timer value. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case. The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

TimerValue is the value of the packet wait bound used during interrupt coalescing. It is in milliseconds in the range 0 - 1023. A value of 0 disables the packet wait bound timer.

Returns:

- XST_SUCCESS if the packet wait bound was set successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
void
XEmac_SetSgRecvHandler ( XEmac * InstancePtr,
                          void * CallbackRef,
                          XEmac_SgHandler FuncPtr
                        )
```

Set the callback function for handling received frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame received. The head of a descriptor list is passed in along with the number of descriptors in the list. Before leaving the callback, the upper layer software should attach a new buffer to each descriptor in the list.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus ( XEmac *  
XEmac_SetSgRecvSpace InstancePtr,  
 Xuint32 * MemoryPtr,  
 Xuint32 ByteCount  
 )
```

Give the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The **xemac.h** file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be 32-bit aligned. An assert will occur if asserts are turned on and the memory is not aligned.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

MemoryPtr is a pointer to the aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XEmac_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```
void  
XEmac_SetSgSendHandler ( XEmac * InstancePtr,  
  
void * CallbackRef,  
XEmac_SgHandler FuncPtr  
)
```

Set the callback function for handling confirmation of transmitted frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame sent. The head of a descriptor list is passed in along with the number of descriptors in the list. The callback is responsible for freeing buffers attached to these descriptors.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```

XStatus
XEmac_SetSgSendSpace
    ( XEmac *
      InstancePtr,
      Xuint32 *
      MemoryPtr,
      Xuint32 ByteCount
    )

```

Give the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The [xemac.h](#) file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be 32-bit aligned. An assert will occur if asserts are turned on and the memory is not aligned.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

MemoryPtr is a pointer to the aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XEmac_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```

XStatus
XEmac_SgRecv
    ( XEmac *
      InstancePtr,
      XBufDescriptor * BdPtr
    )

```

Add a descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be 32-bit aligned if using the OPB Ethernet core and 64-bit aligned if using the PLB Ethernet core. This function is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of frames to replace filled buffers with empty buffers. This function can be called when the device is started or stopped. Note that it does start the scatter-gather DMA engine. Although this is not necessary during initialization, it is not a problem during initialization because the MAC receiver is not yet started.

The buffer attached to the descriptor must be aligned on both the front end and the back end.

Notification of received frames are done asynchronously through the receive callback function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

```
XStatus ( XEmac *  
XEmac_SgSend InstancePtr,  
 XBufDescriptor * BdPtr,  
 int Delay  
 )
```

Send an Ethernet frame using scatter-gather DMA. The caller attaches the frame to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire Ethernet frame may or may not be contained within one descriptor. This function simply inserts the descriptor into the scatter-gather engine's transmit list. The caller is responsible for providing mutual exclusion to guarantee that a frame is contiguous in the transmit list. The buffer attached to the descriptor must be 32-bit aligned if using the OPB Ethernet core and 64-bit aligned if using the PLB Ethernet core.

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the frame, the inserts are committed, which means the descriptors for this frame are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

This call is non-blocking. Notification of error or successful transmission is done asynchronously through the send or error callback function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Delay indicates whether to start the scatter-gather DMA channel immediately, or whether to wait. This allows the user to build up a list of more than one descriptor before starting the transmission of the packets, which allows the application to keep up with DMA and have a constant stream of frames being transmitted. Use `XEM_SGDMA_NODELAY` or `XEM_SGDMA_DELAY`, defined in **xemac.h**, as the value of this argument. If the user chooses to delay and build a list, the user must call this function with the `XEM_SGDMA_NODELAY` option or call **XEmac_Start()** to kick off the transmissions.

Returns:

- `XST_SUCCESS` if the buffer was successful sent
- `XST_DEVICE_IS_STOPPED` if the Ethernet MAC has not been started yet
- `XST_NOT_SGDMA` if the device is not in scatter-gather DMA mode
- `XST_DMA_SG_LIST_FULL` if the descriptor list for the DMA channel is full
- `XST_DMA_SG_BD_LOCKED` if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- `XST_DMA_SG_NOTHING_TO_COMMIT` if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

Generated on 24 Jun 2004 for Xilinx Device Drivers

emac/v1_00_e/src/xemac_multicast.c File Reference

Detailed Description

Contains functions to configure multicast addressing in the Ethernet MAC.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and
v2.00a	of		the packet fifo driver.

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"
```

Functions

XStatus XEmac_MulticastAdd (XEmac *InstancePtr, Xuint8 *AddressPtr)
XStatus XEmac_MulticastClear (XEmac *InstancePtr)

Function Documentation

```
XStatus ( XEmac *
XEmac_MulticastAdd InstancePtr,
Xuint8 * AddressPtr
)
```

Add a multicast address to the list of multicast addresses from which the EMAC accepts frames. The EMAC uses a hash table for multicast address filtering. Obviously, the more multicast addresses that are added reduces the accuracy of the address filtering. The upper layer software that receives multicast frames should perform additional filtering when accuracy must be guaranteed. There is no way to retrieve a multicast address or the multicast address list once added. The upper layer software should maintain its own list of multicast addresses. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

AddressPtr is a pointer to a 6-byte multicast address.

Returns:

- XST_SUCCESS if the multicast address was added successfully
- XST_NO_FEATURE if the device is not configured with multicast support
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

Not currently supported.

```
XStatus ( XEmac *
XEmac_MulticastClear InstancePtr
)
```

Clear the hash table used by the EMAC for multicast address filtering. The entire hash table is cleared, meaning no multicast frames will be accepted after this function is called. If this function is used to delete one or more multicast addresses, the upper layer software is responsible for adding back those addresses still needed for address filtering. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- XST_SUCCESS if the multicast address list was cleared
- XST_NO_FEATURE if the device is not configured with multicast support

- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

Not currently supported.

Generated on 24 Jun 2004 for Xilinx Device Drivers

emac/v1_00_e/src/xemac_phy.c File Reference

Detailed Description

Contains functions to read and write the PHY through the Ethernet MAC MII registers. These assume an MII-compliant PHY.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and
v2.00a	of		the packet fifo driver.
1.00e	rmm	04/06/04	Moved XEM_MAX_PHY constants to xemac_1.h

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"
```

Functions

XStatus XEmac_PhyRead (**XEmac** *InstancePtr, **Xuint32** PhyAddress, **Xuint32** RegisterNum, **Xuint16** *PhyDataPtr)

XStatus XEmac_PhyWrite (**XEmac** *InstancePtr, **Xuint32** PhyAddress, **Xuint32** RegisterNum, **Xuint16** PhyData)

Function Documentation

```
XStatus
XEmac_PhyRead
( XEmac * InstancePtr,
  Xuint32 PhyAddress,
  Xuint32 RegisterNum,
  Xuint16 * PhyDataPtr
)
```

Read the current value of the PHY register indicated by the *PhyAddress* and the *RegisterNum* parameters. The MAC provides the driver with the ability to talk to a PHY that adheres to the Media Independent Interface (MII) as defined in the IEEE 802.3 standard.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

PhyAddress is the address of the PHY to be read (supports multiple PHYs)

RegisterNum is the register number, 0-31, of the specific PHY register to read

PhyDataPtr is an output parameter, and points to a 16-bit buffer into which the current value of the register will be copied.

Returns:

- XST_SUCCESS if the PHY was read from successfully
- XST_NO_FEATURE if the device is not configured with MII support
- XST_EMAC_MII_BUSY if there is another PHY operation in progress
- XST_EMAC_MII_READ_ERROR if a read error occurred between the MAC and the PHY

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that the read is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the *PhyRead* thread.

```

XStatus          ( XEmac * InstancePtr,
XEmac_PhyWrite  Xuint32 PhyAddress,
                  Xuint32 RegisterNum,
                  Xuint16 PhyData
                  )

```

Write data to the specified PHY register. The Ethernet driver does not require the device to be stopped before writing to the PHY. Although it is probably a good idea to stop the device, it is the responsibility of the application to deem this necessary. The MAC provides the driver with the ability to talk to a PHY that adheres to the Media Independent Interface (MII) as defined in the IEEE 802.3 standard.

Parameters:

- InstancePtr* is a pointer to the **XEmac** instance to be worked on.
- PhyAddress* is the address of the PHY to be written (supports multiple PHYs)
- RegisterNum* is the register number, 0-31, of the specific PHY register to write
- PhyData* is the 16-bit value that will be written to the register

Returns:

- XST_SUCCESS if the PHY was written to successfully. Since there is no error status from the MAC on a write, the user should read the PHY to verify the write was successful.
- XST_NO_FEATURE if the device is not configured with MII support
- XST_EMAC_MII_BUSY if there is another PHY operation in progress

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that the write is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PhyWrite thread.

emac/v1_00_e/src/xemac_polled.c File Reference

Detailed Description

Contains functions used when the driver is in polled mode. Use the [XEmac_SetOptions\(\)](#) function to put the driver into polled mode.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and
v2.00a	of		the packet fifo driver.

```
#include "xbasic_types.h"
#include "xemac_i.h"
#include "xio.h"
#include "xipif_v1_23_b.h"
```

Functions

[XStatus XEmac_PollSend](#) ([XEmac](#) *InstancePtr, [Xuint8](#) *BufPtr, [Xuint32](#) ByteCount)

[XStatus XEmac_PollRecv](#) ([XEmac](#) *InstancePtr, [Xuint8](#) *BufPtr, [Xuint32](#) *ByteCountPtr)

Function Documentation

```

XStatus
XEmac_PollRecv
    ( XEmac *
      InstancePtr,
      Xuint8 * BufPtr,
      Xuint32 * ByteCountPtr
    )

```

Receive an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver receives the frame directly from the MAC's packet FIFO. This is a non-blocking receive, in that if there is no frame ready to be received at the device, the function returns with an error. The MAC's error status is not checked, so statistics are not updated for polled receive. The buffer into which the frame will be received must be 32-bit aligned.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a aligned buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_NO_DATA if there is no frame to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.
- XST_FIFO_ERROR if the FIFO was overrun or underrun. This error is critical and requires the caller to reset the device.

Note:

Input buffer must be big enough to hold the largest Ethernet frame.

```

XStatus
XEmac_PollSend
    ( XEmac *
      InstancePtr,
      Xuint8 * BufPtr,
      Xuint32 ByteCount
    )

```

Send an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver writes the frame directly to the MAC's packet FIFO, then enters a loop checking the device status for completion or error. Statistics are updated if an error occurs. The buffer to be sent must be 32-bit aligned.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a aligned buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_FIFO_NO_ROOM if there is no room in the EMAC's length FIFO for this frame
- XST_FIFO_ERROR if the FIFO was overrun or underrun. This error is critical and requires the caller to reset the device.
- XST_EMAC_COLLISION if the send failed due to excess deferral or late collision

Note:

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PollSend thread. On a 10Mbps MAC, it takes about 1.21 msecs to transmit a maximum size Ethernet frame (1518 bytes). On a 100Mbps MAC, it takes about 121 usecs to transmit a maximum size Ethernet frame.

emac/v1_00_e/src/xemac_l.c File Reference

Detailed Description

This file contains low-level polled functions to send and receive Ethernet frames.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	rpm	04/29/02	First release
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and v2.00a of the packet fifo driver.
1.00d	rpm	10/22/03	Fixed the Level 0 functions to work with the PLB EMAC. These functions now make use of the packet fifo driver.

```
#include "xemac_l.h"  
#include "xpacket_fifo_l_v2_00_a.h"
```

Functions

```
void XEmac_SendFrame (Xuint32 BaseAddress, Xuint8 *FramePtr, int Size)  
int XEmac_RecvFrame (Xuint32 BaseAddress, Xuint8 *FramePtr)
```

Function Documentation

```
int
XEmac_RecvFrame ( Xuint32
                  BaseAddress,
                  Xuint8 * FramePtr
                  )
```

Receive a frame. Wait for a frame to arrive.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to a 32-bit aligned buffer where the frame will be stored

Returns:

The size, in bytes, of the frame received.

Note:

None.

```
void
XEmac_SendFrame ( Xuint32
                  BaseAddress,
                  Xuint8 * FramePtr,
                  int      Size
                  )
```

Send an Ethernet frame. This size is the total frame size, including header. This function blocks waiting for the frame to be transmitted.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to a 32-bit aligned frame

Size is the size, in bytes, of the frame

Returns:

None.

Note:

An early return may occur if there is no room in the FIFO for the requested frame.

emac/v1_00_e/src/xemac_selftest.c File Reference

Detailed Description

Self-test and diagnostic functions of the **XEmac** driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and v2.00a of the packet fifo driver.

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

XStatus **XEmac_SelfTest** (**XEmac** *InstancePtr)

Function Documentation

```
XStatus                    ( XEmac *                    )  
XEmac_SelfTest                    InstancePtr
```

Performs a self-test on the Ethernet device. The test includes:

- Run self-test on DMA channel, FIFO, and IPIF components
- Reset the Ethernet device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run, and starting it when ready to send and receive frames.

It should be noted that data caching must be disabled when this function is called because the DMA self-test uses two local buffers (on the stack) for the transfer test.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

Returns:

XST_SUCCESS	Self-test was successful
XST_PFIPO_BAD_REG_VALUE	FIFO failed register self-test
XST_DMA_TRANSFER_ERROR	DMA failed data transfer self-test
XST_DMA_RESET_REGISTER_ERROR	DMA control register value was incorrect after a reset
XST_REGISTER_ERROR	Ethernet failed register reset test
XST_LOOPBACK_ERROR	Internal loopback failed
XST_IPIF_REG_WIDTH_ERROR	An invalid register width was passed into the function
XST_IPIF_RESET_REGISTER_ERROR	The value of a register at reset was invalid
XST_IPIF_DEVICE_STATUS_ERROR	A write to the device status register did not read back correctly
XST_IPIF_DEVICE_ACK_ERROR	A bit in the device status register did not reset when acked
XST_IPIF_DEVICE_ENABLE_ERROR	The device interrupt enable register was not updated correctly by the hardware when other registers were written to
XST_IPIF_IP_STATUS_ERROR	A write to the IP interrupt status register did not read back correctly
XST_IPIF_IP_ACK_ERROR	One or more bits in the IP status register did not reset when acked
XST_IPIF_IP_ENABLE_ERROR	The IP interrupt enable register was not updated correctly when other registers were written to

Note:

This function makes use of options-related functions, and the [XEmac_PollSend\(\)](#) and [XEmac_PollRecv\(\)](#) functions.

Because this test uses the PollSend function for its loopback testing, there is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the self-test thread.

emac/v1_00_e/src/xemac_intr.c File Reference

Detailed Description

This file contains general interrupt-related functions of the **XEmac** driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00c	rpm	03/31/03	Added comment to indicate that no Receive Length FIFO overrun interrupts occur in v1.001 and later of the EMAC device. This avoids the need to reset the device on receive overruns.
1.00d	rpm	09/26/03	New version includes support PLB Ethernet and v2.00a of the packet fifo driver.

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

void **XEmac_SetErrorHandler** (**XEmac** *InstancePtr, void *CallbackRef, **XEmac_ErrorHandler** FuncPtr)

Function Documentation

```
void  
XEmac_SetErrorHandler ( XEmac *  
  
InstancePtr,  
  
void *  
CallbackRef,  
XEmac_ErrorHandler FuncPtr  
)
```

Set the callback function for handling asynchronous errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- **XST_DMA_ERROR** indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- **XST_FIFO_ERROR** indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- **XST_RESET_ERROR** indicates an unrecoverable MAC error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- **XST_DMA_SG_NO_LIST** indicates an attempt was made to access a scatter-gather DMA list that has not yet been created.
- **XST_DMA_SG_LIST_EMPTY** indicates the driver tried to get a descriptor from the receive descriptor list, but the list was empty.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:
None.

Generated on 24 Jun 2004 for Xilinx Device Drivers

emac_lite/v1_01_a/src/xemac_lite_l.c File Reference

Detailed Description

This file contains the minimal, polled functions to send and receive Ethernet frames.

Refer to [xemac_lite.h](#) for more details.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	06/01/02	First release
1.01a	ecm	03/31/04	Additional functionality and the <code>_AlignedRead</code> and <code>_AlignedWrite</code> functions.

```
#include "xbasic_types.h"  
#include "xemac_lite_l.h"  
#include "xemac_lite_i.h"
```

Functions

```
void XEmaLite_AlignedWrite (void *SrcPtr, Xuint32 *DestPtr, unsigned ByteCount)  
void XEmaLite_AlignedRead (Xuint32 *SrcPtr, void *DestPtr, unsigned ByteCount)  
void XEmaLite_SendFrame (Xuint32 BaseAddress, Xuint8 *FramePtr, unsigned ByteCount)  
Xuint16 XEmaLite_RecvFrame (Xuint32 BaseAddress, Xuint8 *FramePtr)
```

Function Documentation

```
void  
XEmacLite_AlignedRead ( Xuint32 * SrcPtr,  
                        void * DestPtr,  
                        unsigned ByteCount  
                        )
```

This function reads from a 32-bit aligned source address range and aligns the writes to the provided destination pointer alignment.

Parameters:

SrcPtr is a pointer to incoming data of 32-bit alignment.

DestPtr is a pointer to outgoing data of any alignment.

ByteCount is the number of bytes to read.

Returns:

None.

Note:

None.

```
void  
XEmacLite_AlignedWrite ( void * SrcPtr,  
                          Xuint32 * DestPtr,  
                          unsigned ByteCount  
                          )
```

This function aligns the incoming data and writes it out to a 32-bit aligned destination address range.

Parameters:

SrcPtr is a pointer to incoming data of any alignment.

DestPtr is a pointer to outgoing data of 32-bit alignment.

ByteCount is the number of bytes to write.

Returns:

None.

Note:

None.

```
Xuint16 ( Xuint32  
XEmacLite_RecvFrame BaseAddress,  
Xuint8 * FramePtr  
)
```

Receive a frame. Wait for a frame to arrive.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to a buffer where the frame will be stored.

Returns:

The type/length field of the frame received. When the type/length field contains the type , XEL_MAX_FRAME_SIZE bytes will be copied out of the buffer and it is up to the higher layers to sort out the frame.

Note:

This function call is blocking in nature, i.e. it will wait until a frame arrives.

If the ping buffer is the source of the data, the argument should be DeviceAddress + XEL_RXBUFF_OFFSET. If the pong buffer is the source of the data, the argument should be DeviceAddress + XEL_RXBUFF_OFFSET + XEL_BUFFER_OFFSET. The function does not take the different buffers into consideration.

```
void ( Xuint32  
XEmacLite_SendFrame BaseAddress,  
Xuint8 * FramePtr,  
unsigned ByteCount  
)
```

Send an Ethernet frame. The size is the total frame size, including header. This function blocks waiting for the frame to be transmitted.

Parameters:

BaseAddress is the base address of the device
FramePtr is a pointer to frame
ByteCount is the size, in bytes, of the frame

Returns:

None.

Note:

This function call is blocking in nature, i.e. it will wait until the frame is transmitted. This function can hang and not exit if the hardware is not configured properly.

If the ping buffer is the destination of the data, the argument should be DeviceAddress + XEL_TXBUFF_OFFSET. If the pong buffer is the destination of the data, the argument should be DeviceAddress + XEL_TXBUFF_OFFSET + XEL_BUFFER_OFFSET. The function does not take the different buffers into consideration.

emacLite/v1_01_a/src/xemacLite_intr.c File Reference

Detailed Description

Functions in this file are for the interrupt driven processing functionality. See [xemacLite.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.01a	ecm	03/31/04	First release

```
#include "xstatus.h"  
#include "xemacLite_l.h"  
#include "xemacLite_i.h"  
#include "xio.h"  
#include "xemacLite.h"
```

Functions

[XStatus](#) [XEmacLite_EnableInterrupts](#) ([XEmacLite](#) *InstancePtr)
void [XEmacLite_DisableInterrupts](#) ([XEmacLite](#) *InstancePtr)
void [XEmacLite_InterruptHandler](#) (void *InstancePtr)
void [XEmacLite_SetRecvHandler](#) ([XEmacLite](#) *InstancePtr, void *CallbackRef,
XEmacLite_Handler FuncPtr)
void [XEmacLite_SetSendHandler](#) ([XEmacLite](#) *InstancePtr, void *CallbackRef,
XEmacLite_Handler FuncPtr)

Function Documentation

```
void ( XEmacLite * )  
XEmacLite_DisableInterrupts InstancePtr
```

Disables the interrupts from the device(the higher layer software is responsible for disabling interrupts at the interrupt controller).

To start using the device again, `_EnableInterrupts` must be called.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( XEmacLite * )  
XEmacLite_EnableInterrupts InstancePtr
```

Enable the EmacLite Interrupts.

This function must be called before other functions to send or receive data in interrupt driven mode. The user should have connected the interrupt handler of the driver to an interrupt source such as an interrupt controller or the processor interrupt prior to this function being called.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

Returns:

- XST_SUCCESS if the device interrupts were enabled successfully.
- XST_NO_CALLBACK if the callbacks were not set.

Note:

None

```
void XEmacLite_InterruptHandler ( void * InstancePtr )
```

Interrupt handler for the EMACLite driver. It performs the following processing:

- Get the interrupt status from the registers to determine the source of the interrupt.
- Call the appropriate handler based on the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the EMACLite device instance for the interrupt.

Returns:

None.

Note:

```
void XEmacLite_SetRecvHandler ( XEmacLite * InstancePtr, void * CallbackRef, XEmacLite_Handler FuncPtr )
```

Sets the callback function for handling received frames in interrupt mode. The upper layer software should call this function during initialization. The callback is called when a frame is received. The callback function should communicate the data to a thread such that the processing is not performed in an interrupt context.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void  
XEmacLite_SetSendHandler ( XEmacLite * InstancePtr,  
void * CallbackRef,  
XEmacLite_Handler FuncPtr  
)
```

Sets the callback function for handling transmitted frames in interrupt mode. The upper layer software should call this function during initialization. The callback is called when a frame is transmitted. The callback function should communicate the data to a thread such that the processing is not performed in an interrupt context.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

emacLite/v1_01_a/src/xemacLite_selftest.c File Reference

Detailed Description

Function(s) in this file are the required functions for the EMAC Lite driver selftest for the hardware. See [xemacLite.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.01a	ecm	01/31/04	First release

```
#include "xstatus.h"  
#include "xemacLite_1.h"  
#include "xio.h"  
#include "xemacLite.h"  
#include "xemacLite_i.h"
```

Functions

[XStatus XEmacLite_SelfTest \(XEmacLite *InstancePtr\)](#)

Function Documentation

XStatus (**XEmacLite***)
XEmacLite_SelfTest *InstancePtr*

Performs a SelfTest on the EmacLite device as follows:

- Writes to the mandatory TX buffer and reads back to verify.
- If configured, writes to the secondary TX buffer and reads back to verify.
- Writes to the mandatory RX buffer and reads back to verify.
- If configured, writes to the secondary RX buffer and reads back to verify.

Parameters:

InstancePtr is a pointer to the **XEmacLite** instance to be worked on.

Returns:

- XST_SUCCESS if the device Passed the Self Test.
- XST_FAILURE if any of the data read backs fail.

Note:

None.

emc/v1_00_a/src/xemc.h File Reference

Detailed Description

This file contains the software API definition of the Xilinx External Memory Controller (**XEmc**) component. This controller can be attached to host OPB or PLB buses to control multiple banks of supported memory devices. The type of host bus is transparent to software.

This driver allows the user to access the device's registers to support fast/slow access to the memory devices as well as enabling/disabling paged mode access.

The Xilinx OPB/PLB External memory controller is a soft IP core designed for Xilinx FPGAs and contains the following general features:

- Support for 128, 64, 32, 16, and 8 bit bus interfaces.
- Controls up to 8 banks of supported memory devices.
- Separate control register for each bank of memory.
- Selectable wait state control (fast or slow). (See note 1)
- Supports page mode accesses. Page size is 8 bytes.
- System clock frequency of up to 133 MHz.

OPB features:

- OPB V2.0 bus interface with byte-enable support.
- Memory width of connected devices is the same as or smaller than OPB bus width.

Note:

(1) The number of wait states inserted for fast and slow mode is determined by the HW designer and is hard-coded into the IP. Each bank has its own settings.

(2) For read accesses, fast/slow access mode is ignored when page mode is enabled. For write accesses, page mode does not apply. (3) This driver is not thread-safe. Thread safety must be guaranteed by the layer above this driver if there is a need to access the device from multiple threads. MODIFICATION HISTORY: Ver Who Date Changes ----- 1.00a rmm
01/29/02 First release 1.00a rpm 05/14/02 Made configuration table/lookup public

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

Data Structures

```
struct XEmc  
struct XEmc_Config
```

Functions

```
XStatus XEmc_Initialize (XEmc *InstancePtr, Xuint16 DeviceId)  
XEmc_Config * XEmc_LookupConfig (Xuint16 DeviceId)  
XStatus XEmc_SetPageMode (XEmc *InstancePtr, unsigned Bank, unsigned Mode)  
XStatus XEmc_SetAccessSpeed (XEmc *InstancePtr, unsigned Bank, unsigned Speed)  
unsigned XEmc_GetPageMode (XEmc *InstancePtr, unsigned Bank)  
unsigned XEmc_GetAccessSpeed (XEmc *InstancePtr, unsigned Bank)  
XStatus XEmc_SelfTest (XEmc *InstancePtr)
```

Function Documentation

```
unsigned XEmc_GetAccessSpeed ( XEmc * InstancePtr,  
                             unsigned Bank  
                             )
```

Gets current access speed setting for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to retrieve the setting for. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in **xparameters.h** (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Returns:

Current access speed of bank. XEMC_ACCESS_SPEED_FAST or XEMC_ACCESS_SPEED_SLOW.

Note:

none

```
unsigned          ( XEmc *  
XEmc_GetPageMode      InstancePtr,  
                      unsigned Bank  
                      )
```

Gets the current page mode setting for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to retrieve the setting for. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in **xparameters.h** (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Returns:

Current mode of bank. XEMC_PAGE_MODE_ENABLE or XEMC_PAGE_MODE_DISABLE.

Note:

none

```
XStatus ( XEmc *  
XEmc_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes the **XEmc** instance provided by the caller based on the given DeviceID.

Parameters:

InstancePtr is a pointer to an **XEmc** instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the **XEmc** API must be made with this pointer.

DeviceId is the unique id of the device controlled by this **XEmc** component. Passing in a device id associates the generic **XEmc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS Initialization was successful.
- XST_DEVICE_NOT_FOUND Device configuration data was not found for a device with the supplied device ID.

Note:

The control registers for each bank are not modified because it is possible that they have been setup during bootstrap processing prior to "C" runtime support.

```
XEmc_Config* ( Xuint16 )  
XEmc_LookupConfig DeviceId
```

Looks up the device configuration based on the unique device ID. The table XEmc_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceID is the device identifier to lookup.

Returns:

XEmc configuration structure pointer if DeviceID is found.

XNULL if DeviceID is not found.

```
XStatus ( XEmc * )  
XEmc_SelfTest InstancePtr
```

Runs a self-test on the driver/device. This includes the following tests:

- Control register read/write access for each bank.

Memory devices controlled by this component are not changed. However access speeds are toggled which could possibly have undesirable effects.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on. This parameter must have been previously initialized with **XEmc_Initialize()**.

Returns:

XST_SUCCESS If test passed

XST_FAILURE If test failed

Note:

- Control register contents are restored to their original state when the test completes.
- This test does not abort if an error is detected.

```
XStatus                            ( XEmc *  
XEmc_SetAccessSpeed                            InstancePtr,  
                                                 unsigned Bank,  
                                                 unsigned Speed  
                                                 )
```

Sets the access speed for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to change. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in **xparameters.h** (XPAR EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Speed is the new access speed. Valid speeds are XEMC_ACCESS_SPEED_SLOW and XEMC_ACCESS_SPEED_FAST.

Returns:

- XST_SUCCESS Access speed successfully set.
- XST_INVALID_PARAM Speed parameter is invalid.

Note:

none

```
XStatus ( XEmc *  
XEmc_SetPageMode InstancePtr,  
                unsigned Bank,  
                unsigned Mode  
                )
```

Sets the page mode for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to change. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in **xparameters.h** (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Mode is the new mode to set. Valid modes are XEMC_PAGE_MODE_ENABLE and XEMC_PAGE_MODE_DISABLE.

Returns:

- XST_SUCCESS Mode successfully set.
- XST_INVALID_PARAM Mode parameter is invalid.

Note:

none

XEmc Struct Reference

```
#include <xemc.h>
```

Detailed Description

The XEmc driver instance data. The user is required to allocate a variable of this type for every EMC device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [emc/v1_00_a/src/xemc.h](#)
-

XEmc_Config Struct Reference

```
#include <xemc.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 RegBaseAddr](#)

[Xuint8 NumBanks](#)

Field Documentation

[Xuint16 XEmc_Config::](#)

DeviceId

Unique ID of device

[Xuint8 XEmc_Config::](#)

NumBanks

Number of devices controlled by this component

[Xuint32 XEmc_Config::](#)

RegBaseAddr

Register base
address

The documentation for this struct was generated from the following file:

- [emc/v1_00_a/src/xemc.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

emc/v1_00_a/src/xemc.c File Reference

Detailed Description

The implementation of the **XEmc** component. See [xemc.h](#) for more information about the component.

Note:

None

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	01/29/02	First release
1.00a	rpm	05/14/02	Made configuration table/lookup public

```
#include "xemc.h"  
#include "xemc_i.h"  
#include "xparameters.h"
```

Functions

[XStatus XEmc_Initialize](#) (**XEmc** *InstancePtr, **Xuint16** DeviceId)
XEmc_Config * [XEmc_LookupConfig](#) (**Xuint16** DeviceId)
[XStatus XEmc_SetPageMode](#) (**XEmc** *InstancePtr, unsigned Bank, unsigned Mode)
[XStatus XEmc_SetAccessSpeed](#) (**XEmc** *InstancePtr, unsigned Bank, unsigned Speed)
unsigned [XEmc_GetPageMode](#) (**XEmc** *InstancePtr, unsigned Bank)
unsigned [XEmc_GetAccessSpeed](#) (**XEmc** *InstancePtr, unsigned Bank)

Function Documentation

```
unsigned  
XEmc_GetAccessSpeed      ( XEmc *  
                          InstancePtr,  
                          unsigned Bank  
                          )
```

Gets current access speed setting for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to retrieve the setting for. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in [xparameters.h](#) (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Returns:

Current access speed of bank. XEMC_ACCESS_SPEED_FAST or XEMC_ACCESS_SPEED_SLOW.

Note:

none

```
unsigned  
XEmc_GetPageMode        ( XEmc *  
                          InstancePtr,  
                          unsigned Bank  
                          )
```

Gets the current page mode setting for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to retrieve the setting for. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in [xparameters.h](#) (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Returns:

Current mode of bank. XEMC_PAGE_MODE_ENABLE or XEMC_PAGE_MODE_DISABLE.

Note:

none

```
XStatus          ( XEmc *  
XEmc_Initialize      InstancePtr,  
                      Xuint16 DeviceId  
                      )
```

Initializes the **XEmc** instance provided by the caller based on the given DeviceID.

Parameters:

InstancePtr is a pointer to an **XEmc** instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the **XEmc** API must be made with this pointer.

DeviceId is the unique id of the device controlled by this **XEmc** component. Passing in a device id associates the generic **XEmc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS Initialization was successful.
- XST_DEVICE_NOT_FOUND Device configuration data was not found for a device with the supplied device ID.

Note:

The control registers for each bank are not modified because it is possible that they have been setup during bootstrap processing prior to "C" runtime support.

```
XEmc_Config*          ( Xuint16          )  
XEmc_LookupConfig      DeviceId
```

Looks up the device configuration based on the unique device ID. The table XEmc_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceID is the device identifier to lookup.

Returns:

XEmc configuration structure pointer if DeviceID is found.

XNULL if DeviceID is not found.

```
XStatus ( XEmc *  
XEmc_SetAccessSpeed InstancePtr,  
                unsigned Bank,  
                unsigned Speed  
                )
```

Sets the access speed for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to change. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in [xparameters.h](#) (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Speed is the new access speed. Valid speeds are XEMC_ACCESS_SPEED_SLOW and XEMC_ACCESS_SPEED_FAST.

Returns:

- XST_SUCCESS Access speed successfully set.
- XST_INVALID_PARAM Speed parameter is invalid.

Note:

none

```
XStatus ( XEmc *  
XEmc_SetPageMode InstancePtr,  
                unsigned Bank,  
                unsigned Mode  
                )
```

Sets the page mode for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to change. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in **xparameters.h** (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Mode is the new mode to set. Valid modes are XEMC_PAGE_MODE_ENABLE and XEMC_PAGE_MODE_DISABLE.

Returns:

- o XST_SUCCESS Mode successfully set.
- o XST_INVALID_PARAM Mode parameter is invalid.

Note:

none

emc/v1_00_a/src/xemc_i.h File Reference

Detailed Description

This header file contains register offsets and bit definitions for the external memory controller (EMC). The definitions here are meant to be used for internal xemc driver purposes.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

-----	-----	-----	-----
1.00a	rmm	02/04/02	First release
1.00a	rpm	05/14/02	Moved identifiers to xemc_1.h

```
#include "xemc\_1.h"
```

emc/v1_00_a/src/xemc_i.h File Reference

Detailed Description

Contains identifiers and low-level macros that can be used to access the device directly. High-level functions are defined in [xemc.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	05/14/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

Defines

```
#define XEmc_mGetOffset(Bank)  
#define XEmc_mGetControlReg(Base, Bank)  
#define XEmc_mSetControlReg(Base, Bank, Data)  
#define XEmc_mEnablePageMode(BaseAddress, Bank)  
#define XEmc_mDisablePageMode(BaseAddress, Bank)  
#define XEmc_mEnableFastAccess(BaseAddress, Bank)  
#define XEmc_mDisableFastAccess(BaseAddress, Bank)
```

Define Documentation

```
#define XEmc_mDisableFastAccess( BaseAddress,  
                                Bank          )
```

Disable fast access in the given memory bank.

Parameters:

BaseAddress is the base address of the device

Bank is the memory bank to set.

Returns:

None.

Note:

None.

```
#define XEmc_mDisablePageMode( BaseAddress,  
                               Bank          )
```

Disable page mode in the given memory bank.

Parameters:

BaseAddress is the base address of the device

Bank is the memory bank to set.

Returns:

None.

Note:

None.

```
#define XEmc_mEnableFastAccess( BaseAddress,  
                                Bank          )
```

Enable fast access in the given memory bank.

Parameters:

BaseAddress is the base address of the device

Bank is the memory bank to set.

Returns:

None.

Note:

None.

```
#define XEmc_mEnablePageMode( BaseAddress,  
                             Bank      )
```

Enable page mode in the given memory bank.

Parameters:

BaseAddress is the base address of the device

Bank is the memory bank to set.

Returns:

None.

Note:

None.

```
#define XEmc_mGetControlReg( Base,  
                             Bank )
```

Reads the contents of a bank's control register.

Parameters:

Base is the base address of the EMC component.

Bank identifies the control register to read.

Returns:

Value of the Bank's control register

Note:

- Macro signature: Xuint32 **XEmc_mGetControlReg**(Xuint32 Base, unsigned Bank)

```
#define XEmc_mGetOffset( Bank )
```

Calculate the offset of a control register based on its bank. This macro is used internally.

Parameters:

Bank is the bank number of the control register offset to calculate

Returns:

Offset to control register associated with Bank parameter.

Note:

- To compute the physical address of the register add the base address of the component to the result of this macro.
- Does not test for validity of Bank.
- Macro signature: unsigned **XEmc_mGetOffset**(unsigned Bank)

```
#define XEmc_mSetControlReg( Base,  
                             Bank,  
                             Data )
```

Writes to a bank's control register.

Parameters:

Base is the base address of the EMC component.

Bank identifies the control register to modify.

Data is the data to write to the control register.

Returns:

None.

Note:

- o Macro signature: void **XEmc_mSetControlReg**(Xuint32 Base, unsigned Bank, Xuint32 Data)

Generated on 24 Jun 2004 for Xilinx Device Drivers

emc/v1_00_a/src/xemc_selftest.c File Reference

Detailed Description

The implementation of the **XEmc** component for the self-test functions.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	02/08/02	First release

```
#include "xemc.h"  
#include "xemc_i.h"
```

Functions

XStatus **XEmc_SelfTest** (**XEmc** *InstancePtr)

Function Documentation

```
XStatus ( XEmc *  
XEmc_SelfTest InstancePtr )
```

Runs a self-test on the driver/device. This includes the following tests:

- Control register read/write access for each bank.

Memory devices controlled by this component are not changed. However access speeds are toggled which could possibly have undesirable effects.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on. This parameter must have been previously initialized with **XEmc_Initialize()**.

Returns:

XST_SUCCESS If test passed

XST_FAILURE If test failed

Note:

- Control register contents are restored to their original state when the test completes.
- This test does not abort if an error is detected.

common/v1_00_a/src/xenv_vxworks.h File Reference

Detailed Description

Defines common services that are typically found in a VxWorks target environment.

Note:

This file is not intended to be included directly by driver code. Instead, the generic [xenv.h](#) file is intended to be included by driver code.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
	rmm	09/13/03	CR 177068: Fix compiler warning in XENV_MEM_FILL
	rmm	10/24/02	Added XENV_USLEEP macro
1.00a	rmm	07/16/01	First release

```
#include "xbasic_types.h"  
#include "vxWorks.h"  
#include "vxLib.h"  
#include <string.h>
```

Data Structures

struct [XENV_TIME_STAMP](#)

Defines

Fills an area of memory with constant data.

Parameters:

DestPtr is the destination address to set.

Data contains the value to set.

Bytes is the number of bytes to set.

Returns:

None.

Note:

Signature: void **XENV_MEM_FILL**(void *DestPtr, char Data, unsigned Bytes)

```
#define XENV_TIME_STAMP_DELTA_MS( Stamp1Ptr,  
                                Stamp2Ptr )
```

This macro is not yet implemented and always returns 0.

Parameters:

Stamp1Ptr is the first sampled time stamp.

Stamp2Ptr is the second sampled time stamp.

Returns:

0

Note:

None.

```
#define XENV_TIME_STAMP_DELTA_US( Stamp1Ptr,  
                                Stamp2Ptr )
```

This macro is not yet implemented and always returns 0.

Parameters:

Stamp1Ptr is the first sampled time stamp.

Stamp2Ptr is the second sampled time stamp.

Returns:

0

Note:

None.

```
#define XENV_TIME_STAMP_GET( StampPtr )
```

Time is derived from the 64 bit PPC timebase register

Parameters:

StampPtr is the storage for the retrieved time stamp.

Returns:

None.

Note:

Signature: void [XENV_TIME_STAMP_GET](#)(XTIME_STAMP *StampPtr)

```
#define XENV_USLEEP( delay )
```

[XENV_USLEEP](#)(unsigned delay)

Delay the specified number of microseconds.

Parameters:

delay is the number of microseconds to delay.

Returns:

None

XENV_TIME_STAMP Struct Reference

```
#include <xenv_vxworks.h>
```

Detailed Description

A structure that contains a time stamp used by other time stamp macros defined below. This structure is processor dependent.

The documentation for this struct was generated from the following file:

- [common/v1_00_a/src/xenv_vxworks.h](#)
-

common/v1_00_a/src/xenv_linux.h File Reference

Detailed Description

Defines common services specified by [xenv.h](#). Some of these services are defined as not performing any action. The implementation of these services are left to the user.

Note:

This file is not intended to be included directly by driver code. Instead, the generic [xenv.h](#) file is intended to be included by driver code.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ch	10/24/02	First release

```
#include "sleep.h"
```

Defines

```
#define XENV\_MEM\_COPY(DestPtr, SrcPtr, Bytes)  
#define XENV\_MEM\_FILL(DestPtr, Data, Bytes)  
#define XENV\_TIME\_STAMP\_GET(StampPtr)  
#define XENV\_TIME\_STAMP\_DELTA\_US(Stamp1Ptr, Stamp2Ptr)  
#define XENV\_TIME\_STAMP\_DELTA\_MS(Stamp1Ptr, Stamp2Ptr)  
#define XENV\_USLEEP(delay)
```

Typedefs

typedef int [XENV_TIME_STAMP](#)

Define Documentation

```
#define XENV_MEM_COPY( DestPtr,  
                      SrcPtr,  
                      Bytes  )
```

Copies a non-overlapping block of memory.

Parameters:

DestPtr is the destination address to copy data to.

SrcPtr is the source address to copy data from.

Bytes is the number of bytes to copy.

Returns:

None.

Note:

Signature: void [XENV_MEM_COPY](#)(void *DestPtr, void *SrcPtr, unsigned Bytes)

```
#define XENV_MEM_FILL( DestPtr,  
                     Data,  
                     Bytes  )
```

Fills an area of memory with constant data.

Parameters:

DestPtr is the destination address to set.

Data contains the value to set.

Bytes is the number of bytes to set.

Returns:

None.

Note:

Signature: void [XENV_MEM_FILL](#)(void *DestPtr, char Data, unsigned

Bytes)

```
#define XENV_TIME_STAMP_DELTA_MS( Stamp1Ptr,  
                                Stamp2Ptr )
```

This macro is not yet implemented and always returns 0.

Parameters:

Stamp1Ptr is the first sampled time stamp.

Stamp2Ptr is the second sampled time stamp.

Returns:

0

Note:

This macro must be implemented by the user

```
#define XENV_TIME_STAMP_DELTA_US( Stamp1Ptr,  
                                Stamp2Ptr )
```

This macro is not yet implemented and always returns 0.

Parameters:

Stamp1Ptr is the first sampled time stamp.

Stamp2Ptr is the second sampled time stamp.

Returns:

0

Note:

This macro must be implemented by the user

```
#define XENV_TIME_STAMP_GET( StampPtr )
```

Time is derived from the 64 bit PPC timebase register

Parameters:

StampPtr is the storage for the retrieved time stamp.

Returns:

None.

Note:

Signature: void **XENV_TIME_STAMP_GET**(XTIME_STAMP *StampPtr)

Note:

This macro must be implemented by the user

```
#define XENV_USLEEP( delay )
```

```
XENV_USLEEP(unsigned delay)
```

Delay the specified number of microseconds.

Parameters:

delay is the number of microseconds to delay.

Returns:

None

Typedef Documentation

```
typedef int
```

```
XENV_TIME_STAMP
```

A structure that contains a time stamp used by other time stamp macros defined below. This structure is processor dependent.

flash/v1_00_a/src/xflash_cfi.h File Reference

Detailed Description

This is a helper component for XFlash. It contains methods used to extract and interpret Common Flash Interface (CFI) from a flash memory part that supports the CFI query command.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	07/16/01	First release

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xflash.h"
#include "xflash_geometry.h"
#include "xflash_properties.h"
```

Defines

```
#define XFL_CFI_POSITION_PTR(Ptr, BaseAddr, Interleave, ByteAddr)
#define XFL_CFI_READ8(Ptr, Interleave)
#define XFL_CFI_READ16(Ptr, Interleave, Data)
#define XFL_CFI_ADVANCE_PTR8(Ptr, Interleave)
#define XFL_CFI_ADVANCE_PTR16(Ptr, Interleave)
```

Functions

Define Documentation

```
#define XFL_CFI_ADVANCE_PTR16( Ptr,  
                               Interleave )
```

Advances the CFI pointer to the next 16-bit quantity.

Parameters:

Ptr is the pointer to advance. Can be a pointer to any type.

Interleave is the byte interleaving (based on part layout)

Returns:

Adjusted Ptr.

```
#define XFL_CFI_ADVANCE_PTR8( Ptr,  
                               Interleave )
```

Advances the CFI pointer to the next byte

Parameters:

Ptr is the pointer to advance. Can be a pointer to any type.

Interleave is the byte interleaving (based on part layout)

Returns:

Adjusted Ptr.

```
#define XFL_CFI_POSITION_PTR( Ptr,  
                               BaseAddr,  
                               Interleave,  
                               ByteAddr )
```

Moves the CFI data pointer to a physical address that corresponds to a specific CFI byte offset.

Parameters:

Ptr is the pointer to modify. Can be of any type

BaseAddr is the base address of flash part

Interleave is the byte interleaving (based on part layout)

ByteAddr is the byte offset within CFI data to read

Returns:

The *Ptr* argument is set to point at the the CFI byte specified by the *ByteAddr* parameter.

```
#define XFL_CFI_READ16( Ptr,  
                      Interleave,  
                      Data      )
```

Reads 16-bits of data from the CFI data location into a local variable.

Parameters:

Ptr is the pointer to read. Can be a pointer to any type.

Interleave is the byte interleaving (based on part layout)

Data is the 16-bit storage location for the data to be read.

Returns:

The 16-bit value at *Ptr* adjusted for the interleave factor.

```
#define XFL_CFI_READ8( Ptr,  
                     Interleave )
```

Reads 8-bits of data from the CFI data location into a local variable.

Parameters:

Ptr is the pointer to read. Can be a pointer to any type.
Interleave is the byte interleaving (based on part layout)

Returns:

The byte at *Ptr* adjusted for the interleave factor.

Function Documentation

```
XStatus ( XFlashGeometry *  
XFlashCFI_ReadCommon GeometryPtr,  
 XFlashProperties * PropertiesPtr  
 )
```

Retrieves the standard CFI data from the part(s), interpret the data, and update the provided geometry and properties structures.

Extended CFI data is part specific and ignored here. This data must be read by the specific part component driver.

Parameters:

GeometryPtr is an input/output parameter. This function expects the BaseAddress and MemoryLayout attributes to be correctly initialized. All other attributes of this structure will be setup using translated CFI data read from the part.
PropertiesPtr is an output parameter. Timing, identification, and programming CFI data will be translated and written to this structure.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_CFI_QUERY_ERROR if an error occurred interpreting the data.
- XST_FLASH_PART_NOT_SUPPORTED if invalid Layout parameter

Note:

None.

flash/v1_00_a/src/xflash.h File Reference

Detailed Description

This is the base component for XFlash. It provides the public interface which upper layers use to communicate with specific flash hardware.

This driver supports "Common Flash Interface" (CFI) enabled flash hardware. CFI allows entire families of flash parts to be supported with a single driver sub-component.

This driver is designed for devices external to the FPGA. As a result interfaces such as IPIF and versioning are not incorporated into its design. A more detailed description of the driver operation can be found in [XFlash.c](#)

Note:

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads, mutual exclusion, virtual memory, cache control, or HW write protection management must be satisfied by the layer above this driver.

All writes to flash occur in units of bus-width bytes. If more than one part exists on the data bus, then the parts are written in parallel. Reads from flash are performed in any width up to the width of the data bus. It is assumed that the flash bus controller or local bus supports these types of accesses.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	07/16/01	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xflash_geometry.h"  
#include "xflash_properties.h"
```

Data Structures

struct [XFlash_Config](#)

struct **XFlashTag**

Configuration options

#define **XFL_NON_BLOCKING_ERASE_OPTION**

#define **XFL_NON_BLOCKING_WRITE_OPTION**

Defines

#define **XFL_MANUFACTURER_ID_INTEL**

Typedefs

typedef **XFlashTag XFlash**

Functions

XStatus XFlash_Initialize (**XFlash** *InstancePtr, **Xuint16** DeviceId)

XStatus XFlash_SelfTest (**XFlash** *InstancePtr)

XStatus XFlash_Reset (**XFlash** *InstancePtr)

XFlash_Config * **XFlash_LookupConfig** (**Xuint16** DeviceId)

XStatus XFlash_SetOptions (**XFlash** *InstancePtr, **Xuint32** OptionsFlag)

Xuint32 XFlash_GetOptions (**XFlash** *InstancePtr)

XFlashProperties * **XFlash_GetProperties** (**XFlash** *InstancePtr)

XFlashGeometry * **XFlash_GetGeometry** (**XFlash** *InstancePtr)

XStatus XFlash_DeviceControl (**XFlash** *InstancePtr, **Xuint32** Command, **Xuint32** Param, **Xuint32** *ReturnPtr)

XStatus XFlash_Read (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes, void *DestPtr)

XStatus XFlash_Write (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes, void *SrcPtr)

XStatus XFlash_WriteSuspend (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlash_WriteResume (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlash_Erase (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)

XStatus XFlash_EraseSuspend (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlash_EraseResume (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlash_Lock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)

XStatus XFlash_Unlock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)

XStatus XFlash_GetStatus (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlash_ReadBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset, **Xuint32** Bytes, void *DestPtr)

XStatus XFlash_WriteBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset, **Xuint32** Bytes, void *SrcPtr)

XStatus XFlash_WriteBlockSuspend (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block, Xuint32 Offset)

XStatus XFlash_WriteBlockResume (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block, Xuint32 Offset)

XStatus XFlash_EraseBlock (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block, Xuint16 NumBlocks)

XStatus XFlash_EraseBlockSuspend (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block)

XStatus XFlash_EraseBlockResume (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block)

XStatus XFlash_LockBlock (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block, Xuint16 NumBlocks)

XStatus XFlash_UnlockBlock (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block, Xuint16 NumBlocks)

XStatus XFlash_GetBlockStatus (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block)

XStatus XFlash_EraseChip (XFlash *InstancePtr)

Define Documentation

#define

XFL_MANUFACTURER_ID_INTEL

Supported manufacturer IDs. Note, that not all parts from these listed vendors are supported.

#define

XFL_NON_BLOCKING_ERASE_OPTION

XFL_NON_BLOCKING_ERASE_OPTION Controls whether the interface blocks on device erase until the operation is completed 1=noblock, 0=block

XFL_NON_BLOCKING_WRITE_OPTION Controls whether the interface blocks on device program until the operation is completed 1=noblock, 0=block

#define

XFL_NON_BLOCKING_WRITE_OPTION

XFL_NON_BLOCKING_ERASE_OPTION Controls whether the interface blocks on device erase until the operation is completed 1=noblock, 0=block

XFL_NON_BLOCKING_WRITE_OPTION Controls whether the interface blocks on device program until the operation is completed 1=noblock, 0=block

Typedef Documentation

typedef struct **XFlashTag**

XFlash

The XFlash driver instance data. The user is required to allocate a variable of this type for every flash device in the system. A pointer to a variable of this type is then passed to the driver API functions.

Function Documentation

```
XStatus ( XFlash *  
XFlash_DeviceControl InstancePtr,  
 Xuint32 Command,  
 Xuint32 Param,  
 Xuint32 * ReturnPtr  
 )
```

Accesses device specific data or commands. For a list of commands, see derived component documentation.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Command is the device specific command to issue

Param is the command parameter

ReturnPtr is the result of command (if any)

Returns:

- XST_SUCCESS if successful
- XST_FLASH_NOT_SUPPORTED if the command is not recognized/supported by the device (s).

Note:

None.

```
XStatus ( XFlash *  
XFlash_Erase InstancePtr,  
 Xuint32 Offset,  
 Xuint32 Bytes  
 )
```

Erases the specified address range.

Returns immediately if the `XFL_NON_BLOCKING_ERASE_OPTION` option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

Erase the specified range of the device(s). Returns immediately if the `XFL_NON_BLOCKING_ERASE_OPTION` option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The `XFL_NON_BLOCKING_ERASE_OPTION` option has an impact on the number of bytes that can be erased in a single call to this function:

- If clear, then the number of bytes to erase can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the the block size of the device at the provided offset and whether the device(s) contains a block erase queue.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin erasure.

Bytes is the number of bytes to erase.

Returns:

If `XFL_NON_BLOCKING_ERASE_OPTION` option is set, then the return value is one of the following:

- `XST_SUCCESS` if successful.
- `XST_FLASH_ADDRESS_ERROR` if the destination address range is not completely within the addressable areas of the device(s).
- `XST_FLASH_BLOCKING_CALL_ERROR` if the amount of data to be erased exceeds the erase queue capacity of the device(s).

If `XFL_NON_BLOCKING_ERASE_OPTION` option is clear, then the following additional codes can be returned:

- `XST_FLASH_ERROR` if an erase error occurred. This error is usually device specific. Use [XFlash_DeviceControl\(\)](#) to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially erased.

Note:

Due to flash memory design, the range actually erased may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

```
XStatus ( XFlash *  
XFlash_EraseBlock InstancePtr,  
                Xuint16 Region,  
                Xuint16 Block,  
                Xuint16 NumBlocks  
                )
```

Erases the specified block.

Returns immediately if the XFL_NON_BLOCKING_ERASE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_ERASE_OPTION option has an impact on the number of bytes that can be erased in a single call to this function:

- If clear, then the number of bytes to erase can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the the block size of the device at the provided offset and whether the device(s) contains a block erase queue.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

NumBlocks is the the number of blocks to erase.

Returns:

If XFL_NON_BLOCKING_ERASE_OPTION option is set, then the return value is one of the below.

- XST_SUCCESS if successfull.
- XST_FLASH_ADDRESS_ERROR if region and/or block do not specify a valid block within the device.

If XFL_NON_BLOCKING_ERASE_OPTION option is clear, then the following additional codes can be returned

- XST_FLASH_ERROR if an erase error occured. This error is usually device specific. Use [XFlash_DeviceControl\(\)](#) to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially erased.

Note:

The arguments point to a starting Region and Block. The NumBlocks parameter may cross over Region boundaries as long as the entire range lies within the part(s) address range and the XFL_NON_BLOCKING_ERASE_OPTION option is not set.

```

XStatus ( XFlash *
XFlash_EraseBlockResume InstancePtr,
Xuint16 Region,
Xuint16 Block
)

```

Resumes an erase operation that was suspended with XFlash_EraseBlockSuspend.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Region* is the region containing block
- Block* is the block that is being erased

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored.

```

XStatus ( XFlash *
XFlash_EraseBlockSuspend InstancePtr,
Xuint16 Region,
Xuint16 Block
)

```

Suspends a currently in progress erase operation and place the device(s) in read mode. When suspended, any block not being erased can be read.

This function should be used only when the XFL_NON_BLOCKING_ERASE_OPTION option is set and a previous call to **XFlash_EraseBlock()** has been made. Otherwise, undetermined results may occur.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Region* is the region containing block
- Block* is the block that is being erased

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored.

```
XStatus ( XFlash * )  
XFlash_EraseChip InstancePtr
```

Erases the entire device(s).

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

If XFL_NON_BLOCKING_ERASE_OPTION option is set, then the return value is always XST_SUCCESS. If XFL_NON_BLOCKING_ERASE_OPTION option is clear, then the following additional codes can be returned:

- XST_FLASH_NOT_SUPPORTED if the chip erase is not supported by the device(s).
- XST_FLASH_ERROR if the device(s) have experienced an internal error during the operation. [XFlash_DeviceControl\(\)](#) must be used to access the cause of the device specific error condition.

Note:

None.

```
XStatus ( XFlash *  
XFlash_EraseResume InstancePtr,  
 Xuint32 Offset  
 )
```

Resumes an erase operation that was suspended with XFlash_EraseSuspend.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where erase resumption should be Resumed.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

```

XStatus ( XFlash *
XFlash_EraseSuspend InstancePtr,
Xuint32 Offset
)

```

Suspends a currently in progress erase operation and place the device(s) in read mode. When suspended, any block not being programmed can be read.

This function should be used only when the XFL_NON_BLOCKING_ERASE_OPTION option is set and a previous call to **XFlash_Erase()** has been made. Otherwise, undetermined results may occur.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where suspension should occur.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

```

XStatus ( XFlash *
XFlash_GetBlockStatus InstancePtr,
Xuint16 Region,
Xuint16 Block
)

```

Returns the status of the device. This function is intended to be used to poll the device in the following circumstances:

- After calling XFlash_WriteBlock with XFL_NON_BLOCKING_WRITE_OPTION option set.
- After calling XFlash_EraseBlock with XFL_NON_BLOCKING_ERASE_OPTION option set.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

Returns:

- XST_FLASH_READY if the device(s) have completed the previous operation without error.
- XST_FLASH_BUSY if the device(s) are currently performing an erase or write operation.
- XST_FLASH_ERROR if the device(s) have experienced an internal error occurring during

another operation such as write, erase, or block locking. [XFlash_DeviceControl\(\)](#) must be used to access the cause of the device specific error condition.

Note:
With some types of flash devices, there may be no difference between using `XFlash_GetBlockStatus` or `XFlash_GetStatus`. See your part data sheet for more information.

XFlashGeometry* (**XFlash ***)
XFlash_GetGeometry *InstancePtr*

Gets the instance's geometry data

Parameters:
InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:
Instance's Geometry structure

Note:
None.

Xuint32 (**XFlash ***)
XFlash_GetOptions *InstancePtr*

Gets interface options for this device instance.

Parameters:
InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:
Current options flag

Note:
None.

XFlashProperties* (**XFlash ***)
XFlash_GetProperties *InstancePtr*

Gets the instance's property data

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

Instance's Properties structure

Note:

None.

```
XStatus ( XFlash *  
XFlash_GetStatus InstancePtr,  
 Xuint32 Offset  
 )
```

Returns the status of the device. This function is intended to be used to poll the device in the following circumstances:

- After calling XFlash_Write with XFL_NON_BLOCKING_WRITE_OPTION option set.
- After calling XFlash_Erase or XFlash_EraseChip with XFL_NON_BLOCKING_ERASE_OPTION option set.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the part.

Returns:

- XST_FLASH_READY if the device(s) have completed the previous operation without error.
- XST_FLASH_BUSY if the device(s) are currently performing an erase or write operation.
- XST_FLASH_ERROR if the device(s) have experienced an internal error occurring during another operation such as write, erase, or block locking. **XFlash_DeviceControl()** must be used to access the cause of the device specific error condition.

Note:

With some types of flash devices, there may be no difference between using XFlash_GetBlockStatus or XFlash_GetStatus. See your part data sheet for more information.

```
XStatus ( XFlash *  
XFlash_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific XFlash instance. The initialization entails:

- Issuing the CFI query command
- Get and translate relevant CFI query information.
- Set default options for the instance.
- Setup the VTable.
- Call the initialize function of the instance, which does the following:
 - Get and translate extended vendor CFI query information.
 - Some VTable functions may be replaced with more efficient ones based on data extracted from the extended CFI query. A replacement example would be a buffered XFlash_WriteBlock replacing a non-buffered XFlash_WriteBlock.
 - Reset the device by clearing any status information and placing the device in read mode.

Parameters:

InstancePtr is a pointer to the XFlash instance to be worked on.

DeviceId is the unique id of the device controlled by this component. Passing in a device id associates the generic component to a specific device, as chosen by the caller or application developer.

Returns:

The return value is XST_SUCCESS if successful. On error, a code indicating the specific error is returned. Possible error codes are:

- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.
- XST_FLASH_PART_NOT_SUPPORTED if the command set algorithm or Layout is not supported by any derived component compiled into the system.
- XST_FLASH_TOO_MANY_REGIONS if the part contains too many erase regions. This can be fixed by increasing the value of XFL_MAX_ERASE_REGIONS then re-compiling the driver.
- XST_FLASH_CFI_QUERY_ERROR if the device would not enter CFI query mode. Either the device(s) do not support CFI, the wrong BaseAddress param was used, an unsupported part layout exists, or a hardware problem exists with the part.

Note:

None.

```
XStatus ( XFlash *  
XFlash_Lock InstancePtr,  
           Xuint32 Offset,  
           Xuint32 Bytes  
)
```

Locks the blocks in the specified range of the device(s).

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin block locking.

Bytes is the number of bytes to lock.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

Due to flash memory design, the range actually locked may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

```
XStatus          ( XFlash *
XFlash_LockBlock      InstancePtr,
                      Xuint16  Region,
                      Xuint16  Block,
                      Xuint16  NumBlocks
                      )
```

Locks the specified block. Prevents it from being erased or written.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

NumBlocks is the the number of blocks to erase. The number may extend into a different region.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if region and/or block do not specify a valid block within the device.
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

None.

```
XFlash_Config* ( Xuint16 )  
XFlash_LookupConfig DeviceId
```

Looks up the device configuration based on the unique device ID.

Parameters:

DeviceId is the unique device ID to be searched for in the table

Returns:

Returns a pointer to the configuration data for the device, or XNULL if not device is found.

```
XStatus ( XFlash *  
XFlash_Read InstancePtr,  
 Xuint32 Offset,  
 Xuint32 Bytes,  
 void * DestPtr  
 )
```

Copies data from the device(s) memory space to a user buffer. The source and destination addresses can be on any alignment supported by the processor.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to read.

Bytes is the number of bytes to copy.

DestPtr is the destination address to copy data to.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the source address does not start within the addressable areas of the device(s).

Note:

This function allows the transfer of data past the end of the device's address space. If this occurs, then results are undefined.

```

XStatus                ( XFlash *
XFlash_ReadBlock      InstancePtr,
                          Xuint16 Region,
                          Xuint16 Block,
                          Xuint32 Offset,
                          Xuint32 Bytes,
                          void * DestPtr
                          )

```

Copy data from a specific device block to a user buffer. The source and destination addresses can be on any byte alignment supported by the target processor.

Parameters:

- InstancePtr* is a pointer to the XFlash instance to be worked on.
- Region* is the erase region the block appears in.
- Block* is the block number within the erase region.
- Offset* is the starting offset in the block where reading will begin.
- Bytes* is the number of bytes to copy.
- DestPtr* is the destination address to copy data to.

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_ADDRESS_ERROR if Region, Block, and Offset parameters do not point to a valid block.

Note:

The arguments point to a starting Region, Block, and Offset within that block. The Bytes parameter may cross over Region and Block boundaries. If Bytes extends past the end of the device's address space, then results are undefined.

```

XStatus                ( XFlash *
XFlash_Reset          InstancePtr
                          )

```

Clears the device(s) status register(s) and place the device(s) into read mode.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_BUSY if the flash devices were in the middle of an operation and could not be reset.
- o XST_FLASH_ERROR if the device(s) have experienced an internal error during the operation. [XFlash_DeviceControl\(\)](#) must be used to access the cause of the device specific error

condition.

Note:

None.

```
XStatus ( XFlash * )  
XFlash_SelfTest InstancePtr
```

Runs a self-test on the driver/device. This is a destructive test. Tests performed include:

- Address bus test
- Data bus test

When the tests are complete, the device is reset back into read mode.

Parameters:

InstancePtr is a pointer to the XComponent instance to be worked on.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ERROR if any test fails.

Note:

None.

```
XStatus ( XFlash *  
XFlash_SetOptions InstancePtr,  
 Xuint32 OptionsFlag  
 )
```

Sets interface options for this device instance.

Here are the currently available options: <pre XFL_NON_BLOCKING_WRITE_OPTION Blocking write on or off XFL_NON_BLOCKING_ERASE_OPTION Blocking erase on or off To set multiple options, OR the option constants together.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

OptionsFlag is the options to set. 1=set option, 0=clear option.

Returns:

- XST_SUCCESS if options successfully set.

- XST_FLASH_NOT_SUPPORTED if option is not supported.

Note:

None.

```

XStatus          ( XFlash *
XFlash_Unlock   InstancePtr,
                   Xuint32 Offset,
                   Xuint32 Bytes
                   )

```

Unlocks the blocks in the specified range of the device(s).

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin block unlocking.

Bytes is the number of bytes to unlock.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

Due to flash memory design, the range actually unlocked may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

```

XStatus          ( XFlash *
XFlash_UnlockBlock InstancePtr,
                   Xuint16 Region,
                   Xuint16 Block,
                   Xuint16 NumBlocks
                   )

```

Unlocks the specified block.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

NumBlocks is the the number of blocks to erase. The number may extend into a different region.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if region and/or block do not specify a valid block within the device.
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

None.

```
XStatus ( XFlash *  
XFlash_Write InstancePtr,  
 Xuint32 Offset,  
 Xuint32 Bytes,  
 void * SrcPtr  
 )
```

Programs the devices with data stored in the user buffer. The source and destination address must be aligned to the width of the flash's data bus.

Returns immediately if the XFL_NON_BLOCKING_WRITE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_WRITE_OPTION option has an impact on the number of bytes that can be written:

- If clear, then the number of bytes to write can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the alignment and size of the device's write buffer. The rule is that the number of bytes being written cannot cross over an alignment boundary. Alignment information is obtained in the InstancePtr->Properties.ProgCap attribute.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin programming. Must be aligned to the width of the flash's data bus.

Bytes is the number of bytes to program.
SrcPtr is the source address containing data to be programmed. Must be aligned to the width of the flash's data bus.

Returns:

If XFL_NON_BLOCKING_WRITE_OPTION option is set, then the return value is one of the following:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_ALIGNMENT_ERROR if the Offset or SrcPtr is not aligned to the width of the flash's data bus.
- XST_BLOCKING_CALL_ERROR if the write would cross a write buffer boundary.

If XFL_NON_BLOCKING_WRITE_OPTION option is clear, then the following additional codes can be returned:

- XST_FLASH_ERROR if a write error occurred. This error is usually device specific. Use [XFlash_DeviceControl\(\)](#) to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially programmed.

Note:

None.

```
XStatus ( XFlash *  
XFlash_WriteBlock InstancePtr,  
                Xuint16 Region,  
                Xuint16 Block,  
                Xuint32 Offset,  
                Xuint32 Bytes,  
                void * SrcPtr  
                )
```

Programs the devices with data stored in the user buffer. The source and destination address can be on any alignment supported by the processor. This function will block until the operation completes or an error is detected.

Returns immediately if the XFL_NON_BLOCKING_WRITE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_WRITE_OPTION option has an impact on the number of bytes that can be written:

- If clear, then the number of bytes to write can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the alignment and size of the device's write buffer. The rule is that the number of bytes being written cannot cross over an alignment boundary. Alignment information is obtained in the InstancePtr->Properties.ProgCap attribute.

Parameters:

InstancePtr is a pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

Offset is the starting offset in the block where writing will begin.

Bytes is the number of bytes to write.

SrcPtr is the source address containing data to be programmed

Returns:

If XFL_NON_BLOCKING_WRITE_OPTION option is set, then the return value is XST_SUCCESS if successful. On error, a code indicating the specific error is returned. Possible error codes are:

- XST_FLASH_ADDRESS_ERROR if Region, Block, and Offset parameters do not point to a valid block. Or, the Bytes parameter causes the read to go past the last addressable byte in the device(s).
- XST_FLASH_ALIGNMENT_ERROR if the Offset or SrcPtr is not aligned to the width of the flash's data bus.
- XST_BLOCKING_CALL_ERROR if the write would cross a write buffer boundary.

If XFL_NON_BLOCKING_WRITE_OPTION option is clear, then the following additional codes can be returned:

- XST_FLASH_ERROR if a write error occurred. This error is usually device specific. Use [XFlash_DeviceControl\(\)](#) to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially programmed.

Note:

The arguments point to a starting Region, Block, and Offset within that block. The Bytes parameter may cross over Region and Block boundaries as long as the entire range lies within the part(s) address range and the XFL_NON_BLOCKING_WRITE_OPTION option is not set.

```
XStatus ( XFlash *  
XFlash_WriteBlockResume InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block,  
 Xuint32 Offset  
 )
```

Resumes a write operation that was suspended with XFlash_WriteBlockSuspend.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Region* is the region containing block
- Block* is the block that is being erased
- Offset* is the offset in the device where resumption should occur

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored.

```
XStatus  
XFlash_WriteBlockSuspend ( XFlash *  
                           InstancePtr,  
                           Xuint16 Region,  
                           Xuint16 Block,  
                           Xuint32 Offset  
                           )
```

Suspends a currently in progress write operation and place the device(s) in read mode. When suspended, any block not being programmed can be read.

This function should be used only when the XFL_NON_BLOCKING_WRITE_OPTION option is set and a previous call to XFlash_WriteBlock() has been made. Otherwise, undetermined results may occur.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Region* is the region containing block
- Block* is the block that is being written
- Offset* is the offset in the device where suspension should occur

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored in those cases.

```
XStatus ( XFlash *  
XFlash_WriteResume InstancePtr,  
 Xuint32 Offset  
 )
```

Resumes a write operation that was suspended with XFlash_WriteSuspend.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where write resumption should occur.

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

```
XStatus ( XFlash *  
XFlash_WriteSuspend InstancePtr,  
 Xuint32 Offset  
 )
```

Suspends a currently in progress write operation and place the device(s) in read mode. When suspended, any block not being programmed can be read.

This function should be used only when the XFL_NON_BLOCKING_ERASE_OPTION option is set and a previous call to **XFlash_Write()** has been made. Otherwise, undetermined results may occur.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where suspension should occur.

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

flash/v1_00_a/src/xflash.c File Reference

Detailed Description

This module implements the base component for flash memory devices that conform to the "Common Flash Interface" (CFI) standard. CFI allows a single flash driver to be used for an entire family of parts.

This is not a driver for a specific device, but for a set of command read/write/erase algorithms. CFI allows us to determine which algorithm to utilize at runtime. It is this set of command algorithms that will be implemented as the derived component.

Flash memory space is segmented into areas called blocks. The size of each block is based on a power of 2. A region is defined as a contiguous set of blocks of the same size. Some parts have several regions while others have one. The arrangement of blocks and regions is referred to by this module as the part's geometry.

The cells within the part can be programmed from a logic 1 to a logic 0 and not the other way around. To change a cell back to a logic 1, the entire block containing that cell must be erased. When a block is erased all bytes contain the value 0xFF. The number of times a block can be erased is finite. Eventually the block will wear out and will no longer be capable of erasure. As of this writing, the typical flash block can be erased 100,000 or more times.

Features provided by this module:

- Part timing, geometry, features, and command algorithm determined by CFI query.
- Supported architectures include:
 - 16-bit data bus: Single x16 part in word mode
 - 32-bit data bus: Two x16 parts in word mode.
- Two read/write/erase APIs.
- Erase/write suspension.
- Self test diagnostics.
- Block locking (if supported by specific part).
- Chip erase (if supported by specific part).
- Non-blocking write & erase function calls.

- Part specific control. Supported features of individual parts are listed within the driver module for that part.

Features listed above not currently implemented include:

- Non-blocking write & erase function calls.
- Block locking.
- Support of more architectures.
- Self test diagnostics.

This component exports two differing types of read/write/erase APIs. The geometry un-aware API allows the user to ignore the geometry of the underlying flash device. The geometry aware API operates on specific blocks. The former API is designed for casual use while the latter may prove useful for designers wishing to use this driver under a flash file system. Both APIs can be used interchangeably.

Write and erase function calls can be set to return immediately (non-blocking) even though the intended operation is incomplete. This is useful for systems that utilize a watchdog timer. It also facilitates the use of an interrupt driven programming algorithm. This feature is dependent upon the capabilities of the flash devices.

If the geometry un-aware API is used, then the user requires no knowledge of the underlying hardware. Usage of this API along with non-blocking write/erase should be done carefully because non-blocking write/erase assumes some knowledge of the device(s) geometry.

If part specific advanced features are required, then the XFlash_DeviceControl function is available provided the feature has been implemented by the part driver module.

Note:

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads, mutual exclusion, virtual memory, cache control, or HW write protection management must be satisfied by the layer above this driver.

Use of this driver by multiple threads must be carefully thought out taking into consideration the underlying flash devices in use. This driver does not use mutual exclusion or critical region control.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	07/16/01	First release

```
#include "xflash.h"  
#include "xflash_cfi.h"  
#include "xparameters.h"
```

Functions

XStatus XFlash_Initialize (XFlash *InstancePtr, Xuint16 DeviceId)

XFlash_Config * XFlash_LookupConfig (Xuint16 DeviceId)

XStatus XFlash_ReadBlock (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block, Xuint32 Offset, Xuint32 Bytes, void *DestPtr)

XStatus XFlash_WriteBlock (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block, Xuint32 Offset, Xuint32 Bytes, void *SrcPtr)

XStatus XFlash_WriteBlockSuspend (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block, Xuint32 Offset)

XStatus XFlash_WriteBlockResume (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block, Xuint32 Offset)

XStatus XFlash_EraseBlock (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block, Xuint16 NumBlocks)

XStatus XFlash_EraseBlockSuspend (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block)

XStatus XFlash_EraseBlockResume (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block)

XStatus XFlash_LockBlock (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block, Xuint16 NumBlocks)

XStatus XFlash_UnlockBlock (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block, Xuint16 NumBlocks)

XStatus XFlash_GetBlockStatus (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block)

XStatus XFlash_Read (XFlash *InstancePtr, Xuint32 Offset, Xuint32 Bytes, void *DestPtr)

XStatus XFlash_Write (XFlash *InstancePtr, Xuint32 Offset, Xuint32 Bytes, void *SrcPtr)

XStatus XFlash_WriteSuspend (XFlash *InstancePtr, Xuint32 Offset)

XStatus XFlash_WriteResume (XFlash *InstancePtr, Xuint32 Offset)

XStatus XFlash_Erase (XFlash *InstancePtr, Xuint32 Offset, Xuint32 Bytes)

XStatus XFlash_EraseSuspend (XFlash *InstancePtr, Xuint32 Offset)

XStatus XFlash_EraseResume (XFlash *InstancePtr, Xuint32 Offset)

XStatus XFlash_Lock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)

XStatus XFlash_Unlock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)

XStatus XFlash_GetStatus (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlash_EraseChip (**XFlash** *InstancePtr)

XStatus XFlash_SelfTest (**XFlash** *InstancePtr)

XStatus XFlash_Reset (**XFlash** *InstancePtr)

XStatus XFlash_SetOptions (**XFlash** *InstancePtr, **Xuint32** OptionsFlag)

Xuint32 XFlash_GetOptions (**XFlash** *InstancePtr)

XFlashGeometry * **XFlash_GetGeometry** (**XFlash** *InstancePtr)

XFlashProperties * **XFlash_GetProperties** (**XFlash** *InstancePtr)

XStatus XFlash_DeviceControl (**XFlash** *InstancePtr, **Xuint32** Command, **Xuint32** Param, **Xuint32** *ReturnPtr)

Xboolean XFlash_IsReady (**XFlash** *InstancePtr)

Function Documentation

```
XStatus
XFlash_DeviceControl
    ( XFlash *
      InstancePtr,
      Xuint32 Command,
      Xuint32 Param,
      Xuint32 * ReturnPtr
    )
```

Accesses device specific data or commands. For a list of commands, see derived component documentation.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Command is the device specific command to issue

Param is the command parameter

ReturnPtr is the result of command (if any)

Returns:

- XST_SUCCESS if successful
- XST_FLASH_NOT_SUPPORTED if the command is not recognized/supported by the device(s).

Note:

None.

```

XStatus          ( XFlash *
XFlash_Erase    InstancePtr,
                  Xuint32 Offset,
                  Xuint32 Bytes
                  )

```

Erases the specified address range.

Returns immediately if the XFL_NON_BLOCKING_ERASE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

Erase the specified range of the device(s). Returns immediately if the XFL_NON_BLOCKING_ERASE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_ERASE_OPTION option has an impact on the number of bytes that can be erased in a single call to this function:

- If clear, then the number of bytes to erase can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the the block size of the device at the provided offset and whether the device(s) contains a block erase queue.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin erasure.

Bytes is the number of bytes to erase.

Returns:

If XFL_NON_BLOCKING_ERASE_OPTION option is set, then the return value is one of the following:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_BLOCKING_CALL_ERROR if the amount of data to be erased exceeds the erase queue capacity of the device(s).

If XFL_NON_BLOCKING_ERASE_OPTION option is clear, then the following additional codes can be returned:

- XST_FLASH_ERROR if an erase error occurred. This error is usually device specific. Use **XFlash_DeviceControl()** to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially erased.

Note:

Due to flash memory design, the range actually erased may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

```
XStatus ( XFlash *
XFlash_EraseBlock InstancePtr,
Xuint16 Region,
Xuint16 Block,
Xuint16 NumBlocks
)
```

Erases the specified block.

Returns immediately if the XFL_NON_BLOCKING_ERASE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_ERASE_OPTION option has an impact on the number of bytes that can be erased in a single call to this function:

- If clear, then the number of bytes to erase can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the the block size of the device at the provided offset and whether the device(s) contains a block erase queue.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

NumBlocks is the the number of blocks to erase.

Returns:

If XFL_NON_BLOCKING_ERASE_OPTION option is set, then the return value is one of the below.

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if region and/or block do not specify a valid block within the device.

If XFL_NON_BLOCKING_ERASE_OPTION option is clear, then the following additional codes can be returned

- XST_FLASH_ERROR if an erase error occurred. This error is usually device specific. Use [XFlash_DeviceControl\(\)](#) to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially erased.

Note:

The arguments point to a starting Region and Block. The NumBlocks parameter may cross over Region boundaries as long as the entire range lies within the part(s) address range and the XFL_NON_BLOCKING_ERASE_OPTION option is not set.

```

XStatus ( XFlash *
XFlash_EraseBlockResume           InstancePtr,
                                     Xuint16 Region,
                                     Xuint16 Block
                                     )

```

Resumes an erase operation that was suspended with XFlash_EraseBlockSuspend.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the region containing block

Block is the block that is being erased

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device (s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored.

```

XStatus ( XFlash *
XFlash_EraseBlockSuspend InstancePtr,
Xuint16 Region,
Xuint16 Block
)

```

Suspends a currently in progress erase operation and place the device(s) in read mode. When suspended, any block not being erased can be read.

This function should be used only when the `XFL_NON_BLOCKING_ERASE_OPTION` option is set and a previous call to `XFlash_EraseBlock()` has been made. Otherwise, undetermined results may occur.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the region containing block

Block is the block that is being erased

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored.

```

XStatus ( XFlash * )
XFlash_EraseChip InstancePtr

```

Erases the entire device(s).

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

If XFL_NON_BLOCKING_ERASE_OPTION option is set, then the return value is always XST_SUCCESS. If XFL_NON_BLOCKING_ERASE_OPTION option is clear, then the following additional codes can be returned:

- XST_FLASH_NOT_SUPPORTED if the chip erase is not supported by the device (s).
- XST_FLASH_ERROR if the device(s) have experienced an internal error during the operation. **XFlash_DeviceControl()** must be used to access the cause of the device specific error condition.

Note:

None.

```
XStatus ( XFlash *  
XFlash_EraseResume InstancePtr,  
 Xuint32 Offset  
 )
```

Resumes an erase operation that was suspended with XFlash_EraseSuspend.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where erase resumption should be Resumed.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device (s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

```
XStatus ( XFlash *  
XFlash_EraseSuspend InstancePtr,  
 Xuint32 Offset  
 )
```

Suspends a currently in progress erase operation and place the device(s) in read mode. When suspended, any block not being programmed can be read.

This function should be used only when the `XFL_NON_BLOCKING_ERASE_OPTION` option is set and a previous call to `XFlash_Erase()` has been made. Otherwise, undetermined results may occur.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where suspension should occur.

Returns:

- `XST_SUCCESS` if successful.
- `XST_FLASH_NOT_SUPPORTED` if write suspension is not supported by the device (s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

```
XStatus ( XFlash *  
XFlash_GetBlockStatus InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block  
 )
```

Returns the status of the device. This function is intended to be used to poll the device in the following circumstances:

- After calling XFlash_WriteBlock with XFL_NON_BLOCKING_WRITE_OPTION option set.
- After calling XFlash_EraseBlock with XFL_NON_BLOCKING_ERASE_OPTION option set.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

Returns:

- XST_FLASH_READY if the device(s) have completed the previous operation without error.
- XST_FLASH_BUSY if the device(s) are currently performing an erase or write operation.
- XST_FLASH_ERROR if the device(s) have experienced an internal error occurring during another operation such as write, erase, or block locking.
[XFlash_DeviceControl\(\)](#) must be used to access the cause of the device specific error condition.

Note:

With some types of flash devices, there may be no difference between using XFlash_GetBlockStatus or XFlash_GetStatus. See your part data sheet for more information.

XFlashGeometry* (**XFlash ***)
XFlash_GetGeometry *InstancePtr*

Gets the instance's geometry data

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

Instance's Geometry structure

Note:

None.

Xuint32 (**XFlash ***)
XFlash_GetOptions *InstancePtr*

Gets interface options for this device instance.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

Current options flag

Note:

None.

XFlashProperties* (**XFlash ***)
XFlash_GetProperties *InstancePtr*

Gets the instance's property data

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

Instance's Properties structure

Note:

None.

```
XStatus ( XFlash *  
XFlash_GetStatus InstancePtr,  
 Xuint32 Offset  
 )
```

Returns the status of the device. This function is intended to be used to poll the device in the following circumstances:

- After calling XFlash_Write with XFL_NON_BLOCKING_WRITE_OPTION option set.
- After calling XFlash_Erase or XFlash_EraseChip with XFL_NON_BLOCKING_ERASE_OPTION option set.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the part.

Returns:

- XST_FLASH_READY if the device(s) have completed the previous operation without error.
- XST_FLASH_BUSY if the device(s) are currently performing an erase or write operation.
- XST_FLASH_ERROR if the device(s) have experienced an internal error occurring during another operation such as write, erase, or block locking.
XFlash_DeviceControl() must be used to access the cause of the device specific error condition.

Note:

With some types of flash devices, there may be no difference between using XFlash_GetBlockStatus or XFlash_GetStatus. See your part data sheet for more information.

```
XStatus ( XFlash *  
XFlash_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific XFlash instance. The initialization entails:

- Issuing the CFI query command
- Get and translate relevant CFI query information.
- Set default options for the instance.
- Setup the VTable.
- Call the initialize function of the instance, which does the following:
 - Get and translate extended vendor CFI query information.
 - Some VTable functions may be replaced with more efficient ones based on data extracted from the extended CFI query. A replacement example would be a buffered XFlash_WriteBlock replacing a non-buffered XFlash_WriteBlock.
 - Reset the device by clearing any status information and placing the device in read mode.

Parameters:

InstancePtr is a pointer to the XFlash instance to be worked on.

DeviceId is the unique id of the device controlled by this component. Passing in a device id associates the generic component to a specific device, as chosen by the caller or application developer.

Returns:

The return value is XST_SUCCESS if successful. On error, a code indicating the specific error is returned. Possible error codes are:

- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.
- XST_FLASH_PART_NOT_SUPPORTED if the command set algorithm or Layout is not supported by any derived component compiled into the system.
- XST_FLASH_TOO_MANY_REGIONS if the part contains too many erase regions. This can be fixed by increasing the value of XFL_MAX_ERASE_REGIONS then re-compiling the driver.
- XST_FLASH_CFI_QUERY_ERROR if the device would not enter CFI query mode. Either the device(s) do not support CFI, the wrong BaseAddress param was used, an unsupported part layout exists, or a hardware problem exists with the part.

Note:

None.

Xboolean (**XFlash ***)
XFlash_IsReady *InstancePtr*

Checks the readiness of the device, which means it has been successfully initialized.

Parameters:

InstancePtr is a pointer to the XFlash instance to be worked on.

Returns:

XTRUE if the device has been initialized (but not necessarily started), and XFALSE otherwise.

Note:

This function only exists in the base component since it is common across all derived components. Asserts based on the IsReady flag exist only in the base component.

```
XStatus          ( XFlash *  
XFlash_Lock      InstancePtr,  
                  Xuint32  Offset,  
                  Xuint32  Bytes  
                  )
```

Locks the blocks in the specified range of the device(s).

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin block locking.

Bytes is the number of bytes to lock.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

Due to flash memory design, the range actually locked may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

```

XStatus ( XFlash *
XFlash_LockBlock InstancePtr,
Xuint16 Region,
Xuint16 Block,
Xuint16 NumBlocks
)

```

Locks the specified block. Prevents it from being erased or written.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

NumBlocks is the the number of blocks to erase. The number may extend into a different region.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if region and/or block do not specify a valid block within the device.
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

None.

```

XFlash_Config* ( Xuint16 )
XFlash_LookupConfig DeviceId

```

Looks up the device configuration based on the unique device ID.

Parameters:

DeviceId is the unique device ID to be searched for in the table

Returns:

Returns a pointer to the configuration data for the device, or XNULL if not device is found.

```

XStatus          ( XFlash *
XFlash_Read      InstancePtr,
                   Xuint32 Offset,
                   Xuint32 Bytes,
                   void * DestPtr
                   )

```

Copies data from the device(s) memory space to a user buffer. The source and destination addresses can be on any alignment supported by the processor.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Offset* is the offset into the device(s) address space from which to read.
- Bytes* is the number of bytes to copy.
- DestPtr* is the destination address to copy data to.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the source address does not start within the addressable areas of the device(s).

Note:

This function allows the transfer of data past the end of the device's address space. If this occurs, then results are undefined.

```

XStatus          ( XFlash *
XFlash_ReadBlock InstancePtr,
                   Xuint16 Region,
                   Xuint16 Block,
                   Xuint32 Offset,
                   Xuint32 Bytes,
                   void * DestPtr
                   )

```

Copy data from a specific device block to a user buffer. The source and destination addresses can be on any byte alignment supported by the target processor.

Parameters:

InstancePtr is a pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

Offset is the starting offset in the block where reading will begin.

Bytes is the number of bytes to copy.

DestPtr is the destination address to copy data to.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if Region, Block, and Offset parameters do not point to a valid block.

Note:

The arguments point to a starting Region, Block, and Offset within that block. The Bytes parameter may cross over Region and Block boundaries. If Bytes extends past the end of the device's address space, then results are undefined.

XStatus (**XFlash ***)
XFlash_Reset *InstancePtr*

Clears the device(s) status register(s) and place the device(s) into read mode.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_BUSY if the flash devices were in the middle of an operation and could not be reset.
- XST_FLASH_ERROR if the device(s) have experienced an internal error during the operation. **XFlash_DeviceControl()** must be used to access the cause of the device specific error condition.

Note:

None.

XStatus (**XFlash ***)
XFlash_SelfTest *InstancePtr*

Runs a self-test on the driver/device. This is a destructive test. Tests performed include:

- Address bus test
- Data bus test

When the tests are complete, the device is reset back into read mode.

Parameters:

InstancePtr is a pointer to the XComponent instance to be worked on.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ERROR if any test fails.

Note:

None.

XStatus (**XFlash ***
XFlash_SetOptions *InstancePtr*,
Xuint32 *OptionsFlag*
)

Sets interface options for this device instance.

Here are the currently available options: <pre XFL_NON_BLOCKING_WRITE_OPTION
Blocking write on or off XFL_NON_BLOCKING_ERASE_OPTION Blocking erase on or off To
set multiple options, OR the option constants together.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

OptionsFlag is the options to set. 1=set option, 0=clear option.

Returns:

- XST_SUCCESS if options successfully set.

- XST_FLASH_NOT_SUPPORTED if option is not supported.

Note:

None.

```

XStatus          ( XFlash *
XFlash_Unlock   InstancePtr,
                   Xuint32 Offset,
                   Xuint32 Bytes
                   )

```

Unlocks the blocks in the specified range of the device(s).

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin block unlocking.

Bytes is the number of bytes to unlock.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

Due to flash memory design, the range actually unlocked may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

```

XStatus          ( XFlash *
XFlash_UnlockBlock InstancePtr,
                   Xuint16 Region,
                   Xuint16 Block,
                   Xuint16 NumBlocks
                   )

```

Unlocks the specified block.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

NumBlocks is the the number of blocks to erase. The number may extend into a different region.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if region and/or block do not specify a valid block within the device.
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

None.

```
XStatus ( XFlash *  
XFlash_Write InstancePtr,  
             Xuint32 Offset,  
             Xuint32 Bytes,  
             void * SrcPtr  
             )
```

Programs the devices with data stored in the user buffer. The source and destination address must be aligned to the width of the flash's data bus.

Returns immediately if the XFL_NON_BLOCKING_WRITE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_WRITE_OPTION option has an impact on the number of bytes that can be written:

- If clear, then the number of bytes to write can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the alignment and size of the device's write buffer. The rule is that the number of bytes being written cannot cross over an alignment boundary. Alignment information is obtained in the InstancePtr->Properties.ProgCap attribute.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin programming. Must be aligned to the width of the flash's data bus.

Bytes is the number of bytes to program.

SrcPtr is the source address containing data to be programmed. Must be aligned to the width of the flash's data bus.

Returns:

If XFL_NON_BLOCKING_WRITE_OPTION option is set, then the return value is one of the following:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_ALIGNMENT_ERROR if the Offset or SrcPtr is not aligned to the width of the flash's data bus.
- XST_BLOCKING_CALL_ERROR if the write would cross a write buffer boundary.

If XFL_NON_BLOCKING_WRITE_OPTION option is clear, then the following additional codes can be returned:

- XST_FLASH_ERROR if a write error occurred. This error is usually device specific. Use [XFlash_DeviceControl\(\)](#) to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially programmed.

Note:

None.

```
XStatus ( XFlash *  
XFlash_WriteBlock InstancePtr,  
                Xuint16 Region,  
                Xuint16 Block,  
                Xuint32 Offset,  
                Xuint32 Bytes,  
                void * SrcPtr  
                )
```

Programs the devices with data stored in the user buffer. The source and destination address can be on any alignment supported by the processor. This function will block until the operation completes or an error is detected.

Returns immediately if the `XFL_NON_BLOCKING_WRITE_OPTION` option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The `XFL_NON_BLOCKING_WRITE_OPTION` option has an impact on the number of bytes that can be written:

- If clear, then the number of bytes to write can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the alignment and size of the device's write buffer. The rule is that the number of bytes being written cannot cross over an alignment boundary. Alignment information is obtained in the `InstancePtr->Properties.ProgCap` attribute.

Parameters:

InstancePtr is a pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

Offset is the starting offset in the block where writing will begin.

Bytes is the number of bytes to write.

SrcPtr is the source address containing data to be programmed

Returns:

If `XFL_NON_BLOCKING_WRITE_OPTION` option is set, then the return value is `XST_SUCCESS` if successful. On error, a code indicating the specific error is returned.

Possible error codes are:

- `XST_FLASH_ADDRESS_ERROR` if *Region*, *Block*, and *Offset* parameters do not point to a valid block. Or, the *Bytes* parameter causes the read to go past the last addressable byte in the device(s).
- `XST_FLASH_ALIGNMENT_ERROR` if the *Offset* or *SrcPtr* is not aligned to the width of the flash's data bus.
- `XST_BLOCKING_CALL_ERROR` if the write would cross a write buffer boundary.

If `XFL_NON_BLOCKING_WRITE_OPTION` option is clear, then the following additional codes can be returned:

- `XST_FLASH_ERROR` if a write error occurred. This error is usually device specific. Use [XFlash_DeviceControl\(\)](#) to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially programmed.

Note:

The arguments point to a starting Region, Block, and Offset within that block. The Bytes parameter may cross over Region and Block boundaries as long as the entire range lies within the part(s) address range and the XFL_NON_BLOCKING_WRITE_OPTION option is not set.

```

XStatus ( XFlash *
XFlash_WriteBlockResume InstancePtr,
Xuint16 Region,
Xuint16 Block,
Xuint32 Offset
)

```

Resumes a write operation that was suspended with XFlash_WriteBlockSuspend.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the region containing block

Block is the block that is being erased

Offset is the offset in the device where resumption should occur

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device (s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored.

```

XStatus ( XFlash *
XFlash_WriteBlockSuspend InstancePtr,
Xuint16 Region,
Xuint16 Block,
Xuint32 Offset
)

```

Suspends a currently in progress write operation and place the device(s) in read mode. When suspended, any block not being programmed can be read.

This function should be used only when the XFL_NON_BLOCKING_WRITE_OPTION option is set and a previous call to [XFlash_WriteBlock\(\)](#) has been made. Otherwise, undetermined results may occur.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Region* is the region containing block
- Block* is the block that is being written
- Offset* is the offset in the device where suspension should occur

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device (s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored in those cases.

```
XStatus ( XFlash *  
XFlash_WriteResume InstancePtr,  
 Xuint32 Offset  
 )
```

Resumes a write operation that was suspended with XFlash_WriteSuspend.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Offset* is the offset with the device where write resumption should occur.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device (s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the

Offset parameter is ignored.

```
XStatus          ( XFlash *  
XFlash_WriteSuspend      InstancePtr,  
                          Xuint32 Offset  
                          )
```

Suspends a currently in progress write operation and place the device(s) in read mode. When suspended, any block not being programmed can be read.

This function should be used only when the `XFL_NON_BLOCKING_ERASE_OPTION` option is set and a previous call to `XFlash_Write()` has been made. Otherwise, undetermined results may occur.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where suspension should occur.

Returns:

- `XST_SUCCESS` if successful.
- `XST_FLASH_NOT_SUPPORTED` if write suspension is not supported by the device (s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

flash/v1_00_a/src/xflash_geometry.h File Reference

Detailed Description

This is a helper component for XFlash. It contains the geometry information for an XFlash instance with utilities to translate from absolute to block coordinate systems.

Absolute coordinates

This coordinate system is simply an offset into the address space of the flash instance.

Block coordinates

This coordinate system is dependent on the device's block geometry. All flash devices are divisible by independent erase blocks which can be addressed with this coordinate system. The coordinates are defined as:

- BlockOffset - The offset within an erase block
- Block - An erase block
- Region - An area containing erase blocks of the same size

The picture below shows the differences between the coordinate systems. The sample part has two regions of three blocks each.

	Absolute	Region	Block	Offset
	-----	-----	-----	-----
+-----+	0	0	0	0
		0	0	
blocksizeA-1				
+-----+		0	1	0
		0	1	
blocksizeA-1				


```
#include "xbasic_types.h"
#include "xstatus.h"
```

Data Structures

```
struct XFlashGeometry
```

Defines

```
#define XFL_MAX_ERASE_REGIONS
#define XFL_GEOMETRY_IS_BLOCK_VALID(GeometryPtr, Region, Block, BlockOffset)
#define XFL_GEOMETRY_IS_ABSOLUTE_VALID(GeometryPtr, Offset)
#define XFL_GEOMETRY_BLOCKS_LEFT(GeometryPtr, Region, Block)
#define XFL_GEOMETRY_BLOCK_DIFF(GeometryPtr, StartRegion, StartBlock, EndRegion,
    EndBlock)
#define XFL_GEOMETRY_INCREMENT(GeometryPtr, Region, Block)
```

Functions

```
XStatus XFlashGeometry_ToBlock (XFlashGeometry *InstancePtr, Xuint32 AbsoluteOffset,
    Xuint16 *Region, Xuint16 *Block, Xuint32 *BlockOffset)
XStatus XFlashGeometry_ToAbsolute (XFlashGeometry *InstancePtr, Xuint16 Region, Xuint16
    Block, Xuint32 BlockOffset, Xuint32 *AbsoluteOffsetPtr)
Xuint32 XFlashGeometry_ConvertLayout (Xuint8 NumParts, Xuint8 PartWidth, Xuint8 PartMode)
```

Define Documentation

```
#define XFL_GEOMETRY_BLOCK_DIFF( GeometryPtr,
                                StartRegion,
                                StartBlock,
                                EndRegion,
                                EndBlock    )
```

Calculates the number of blocks between the given start and end coordinates.

Parameters:

GeometryPtr is the geometry instance that defines flash addressing

StartRegion is the starting region

StartBlock is the starting block.

EndRegion is the ending region

EndBlock is the ending block.

Returns:

The number of blocks between start Region/Block and end Region/Block (inclusive)

```
#define XFL_GEOMETRY_BLOCKS_LEFT( GeometryPtr,  
                                Region,  
                                Block      )
```

Calculates the number of blocks between the given coordinates and the end of the device.

Parameters:

GeometryPtr is the geometry instance that defines flash addressing

Region is the starting region

Block is the starting block.

Returns:

The number of blocks between Region/Block and the end of the device (inclusive)

```
#define XFL_GEOMETRY_INCREMENT( GeometryPtr,  
                                Region,  
                                Block      )
```

Increments the given Region and Block to the next block address.

Parameters:

GeometryPtr is the geometry instance that defines flash addressing

Region is the starting region.

Block is the starting block.

Returns:

Region parameter is incremented if the next block starts in a new region. Block parameter is set to zero if the next block starts in a new region, otherwise it is incremented by one.

```
#define XFL_GEOMETRY_IS_ABSOLUTE_VALID( GeometryPtr,  
                                         Offset      )
```

Tests the given absolute Offset to verify it lies within the bounds of the address space defined by a geometry instance.

Parameters:

GeometryPtr is the geometry instance that defines flash addressing

Offset is the offset to test

Returns:

- o 0 if Offset do not lie within the address space described by GeometryPtr.
- o 1 if Offset are within the address space

```
#define XFL_GEOMETRY_IS_BLOCK_VALID( GeometryPtr,  
                                     Region,  
                                     Block,  
                                     BlockOffset  )
```

Tests the given Region, Block, and Offset to verify they lie within the address space defined by a geometry instance.

Parameters:

GeometryPtr is the geometry instance that defines flash addressing

Region is the region to test

Block is the block to test

BlockOffset is the offset within block

Returns:

- 0 if Region, Block, & BlockOffset do not lie within the address space described by GeometryPtr.
- 1 if Region, Block, & BlockOffset are within the address space

#define

XFL_MAX_ERASE_REGIONS

A block region is defined as a set of consecutive erase blocks of the same size. Most flash devices only have a handful of regions. If a part has more regions than defined by this constant, then the constant must be modified to accomodate the part. The minimum value of this constant is 1 and there is no maximum value. Note that increasing this value also increases the amount of memory used by the geometry structure approximately 12 bytes per increment.

Function Documentation

```
Xuint32 XFlashGeometry_ConvertLayout ( Xuint8 NumParts,  
                                     Xuint8 PartWidth,  
                                     Xuint8 PartMode  
                                     )
```

Converts array layout into an XFL_LAYOUT_Xa_Xb_Xc constant. This function is typically called during initialization to convert ordinal values delivered by a system generator into the XFL constants which are optimized for use by the flash driver.

Parameters:

- NumParts* - Number of parts in the array.
- PartWidth* - Width of each part in bytes.
- PartMode* - Operation mode of each part in bytes.

Returns:

- XFL_LAYOUT_* - One of the supported layouts
- XNULL if a layout cannot be found that supports the given arguments

Note:

None.

```
XStatus  
XFlashGeometry_ToAbsolute  
    ( XFlashGeometry *  
      InstancePtr,  
      Xuint16  
      Xuint16  
      Xuint32  
      Xuint32 *  
    )  
    Region,  
    Block,  
    BlockOffset,  
    AbsoluteOffsetPtr
```

Converts block coordinates to a part offset. Region, Block, & BlockOffset are converted to PartOffset

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Region* is the erase region the physical address appears in.
- Block* is the block within Region the physical address appears in.
- BlockOffset* is the offset within Block where the physical address appears.
- AbsoluteOffsetPtr* is the returned offset value

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the block coordinates are invalid.

Note:

None.

```
XStatus
XFlashGeometry_ToBlock
    ( XFlashGeometry *
      InstancePtr,
      Xuint32
      AbsoluteOffset,
      Xuint16 *
      RegionPtr,
      Xuint16 *
      BlockPtr,
      Xuint32 *
      BlockOffsetPtr
    )
```

Converts part offset block coordinates. PartOffset is converted to Region, Block, & BlockOffset

Parameters:

InstancePtr is the pointer to the **XFlashGeometry** instance to be worked on.

AbsoluteOffset is the offset within part to find block coordinates for.

RegionPtr is the the region that corresponds to AbsoluteOffset. This is a return parameter.

BlockPtr is the the block within Region that corresponds to AbsoluteOffset. This is a return parameter.

BlockOffsetPtr is the the offset within Block that corresponds to AbsoluteOffset. This is a return parameter.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the block coordinates are invalid.

Note:

None.

XFlashGeometry Struct Reference

```
#include <xflash_geometry.h>
```

Detailed Description

Flash geometry

Data Fields

[Xuint32 BaseAddress](#)

[Xuint32 MemoryLayout](#)

[Xuint32 DeviceSize](#)

[Xuint32 NumEraseRegions](#)

[Xuint16 NumBlocks](#)

[Xuint32 AbsoluteOffset](#)

[Xuint16 AbsoluteBlock](#)

[Xuint16 Number](#)

[Xuint32 Size](#)

Field Documentation

[Xuint16 XFlashGeometry::](#)

[AbsoluteBlock](#)

Block number where region
begins

[Xuint32 XFlashGeometry::](#)

[AbsoluteOffset](#)

Offset within part where region begins

Xuint32 XFlashGeometry::

BaseAddress

Base address of part
(s)

Xuint32 XFlashGeometry::

DeviceSize

Total device size in
bytes

Xuint32 XFlashGeometry::

MemoryLayout

How multiple parts are connected on the data bus. Choices are limited to
XFL_LAYOUT_Xa_Xb_Xc constants

Xuint16 XFlashGeometry::

Number

Number of blocks in this
region

Xuint16 XFlashGeometry::

NumBlocks

Total number of blocks in
device

Xuint32 XFlashGeometry::

NumEraseRegions

Number of erase
regions

Xuint32 XFlashGeometry::

Size

Size of the block in
bytes

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash_geometry.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

flash/v1_00_a/src/xflash_properties.h File Reference

Detailed Description

This is a helper component for XFlash. It contains various datum common to flash devices most of which can be derived from the CFI query.

Note:

There is no implementation file with this component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes

1.00a	rmm	07/16/01	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

Data Structures

```
struct XFlashPartID  
struct XFlashProgCap  
struct XFlashProperties  
struct XFlashTiming
```

XFlashPartID Struct Reference

```
#include <xflash_properties.h>
```

Detailed Description

Flash identification

Data Fields

[Xuint8 ManufacturerID](#)

[Xuint8 DeviceID](#)

[Xuint16 CommandSet](#)

Field Documentation

[Xuint16 XFlashPartID::](#)

CommandSet

Command algorithm used by part. Choices are defined in XFL_CMDSET constants

[Xuint8 XFlashPartID::](#)

DeviceID

Part number of manufacturer

[Xuint8 XFlashPartID::](#)

ManufacturerID

Manufacturer of
parts

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash_properties.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

XFlashProgCap Struct Reference

```
#include <xflash_properties.h>
```

Detailed Description

Programming parameters

Data Fields

[Xuint32 WriteBufferSize](#)

[Xuint32 WriteBufferAlignmentMask](#)

[Xuint32 EraseQueueSize](#)

Field Documentation

Xuint32 XFlashProgCap::

EraseQueueSize

Number of erase blocks that can be queued up at once

Xuint32 XFlashProgCap::

WriteBufferAlignmentMask

Alignment of the write buffer

Xuint32 XFlashProgCap::

WriteBufferSize

Number of bytes that can be programmed at once

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash_properties.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

XFlashProperties Struct Reference

```
#include <xflash_properties.h>
```

Detailed Description

Consolidated parameters

Data Fields

[XFlashPartID](#) **PartID**

[XFlashTiming](#) **TimeTypical**

[XFlashTiming](#) **TimeMax**

[XFlashProgCap](#) **ProgCap**

Field Documentation

[XFlashPartID](#) **XFlashProperties::**

PartID

Uniquely identifies the
part

[XFlashProgCap](#) **XFlashProperties::**

ProgCap

Programming
capabilities

XFlashTiming XFlashProperties::

TimeMax

Worst case timing
data

XFlashTiming XFlashProperties::

TimeTypical

Typical timing
data

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash_properties.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

XFlashTiming Struct Reference

```
#include <xflash_properties.h>
```

Detailed Description

Flash timing

Data Fields

[Xuint16 WriteSingle_Us](#)

[Xuint16 WriteBuffer_Us](#)

[Xuint16 EraseBlock_Ms](#)

[Xuint16 EraseChip_Ms](#)

Field Documentation

[Xuint16 XFlashTiming::](#)

[EraseBlock_Ms](#)

Time to erase a single block Units are in milliseconds

[Xuint16 XFlashTiming::](#)

[EraseChip_Ms](#)

Time to perform a chip erase Units are in milliseconds

Xuint16 XFlashTiming::

WriteBuffer_Us

Time to program the contents of the write buffer. Units are in microseconds If the part does not support write buffers, then this value should be zero

Xuint16 XFlashTiming::

WriteSingle_Us

Time to program a single word unit Units are in microseconds

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash_properties.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

XFlash_Config Struct Reference

```
#include <xflash.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddr](#)

[Xuint8 NumParts](#)

[Xuint8 PartWidth](#)

[Xuint8 PartMode](#)

Field Documentation

[Xuint32 XFlash_Config::](#)

BaseAddr

Base address of
array

[Xuint16 XFlash_Config::](#)

DeviceId

Unique ID of
device

Xuint8 XFlash_Config::

NumParts

Number of parts in the array

Xuint8 XFlash_Config::

PartMode

Operation mode of each part in bytes

Xuint8 XFlash_Config::

PartWidth

Width of each part in bytes

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

XFlashTag Struct Reference

```
#include <xflash.h>
```

Detailed Description

The XFlash driver instance data. The user is required to allocate a variable of this type for every flash device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash.h](#)
-

flash/v1_00_a/src/xflash_g.c File Reference

Detailed Description

This file contains a table that specifies the configuration of devices in the system. In addition, there is a lookup function used by the driver to access its configuration information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

-----	-----	-----	-----
1.00a	rmm	01/18/01	First release
1.00a	rpm	05/05/02	Added include of xparameters.h

```
#include "xflash.h"  
#include "xparameters.h"
```

Variables

[XFlash_Config](#) [XFlash_ConfigTable](#) [XPAR_XFLASH_NUM_INSTANCES]

Variable Documentation

[XFlash_Config](#) [XFlash_ConfigTable](#)
[XPAR_XFLASH_NUM_INSTANCES]

This table contains configuration information for each flash device in the system.

flash/v1_00_a/src/xflash_cfi.c File Reference

Detailed Description

This module implements the helper component XFlashCFI.

The helper component is responsible for retrieval and translation of CFI data from a compliant flash device. CFI contains data that defines part geometry, write/erase timing, and programming data.

Data is retrieved using macros defined in this component's header file. The macros simplify data extraction because they have been written to take into account the layout of parts on the data bus. To the driver, CFI data appears as if it were always being read from a single 8-bit part (XFL_LAYOUT_X8_X8_X1) Otherwise, the retrieval code would have to contend with all the formats illustrated below. The illustration shows how the first three bytes of the CFI query data "QRY" appear in flash memory space for various part layouts.

```
                Byte Offset (big-Endian)
                0123456789ABCDEF
                -----
XFL_LAYOUT_X16_X16_X1   Q R Y
XFL_LAYOUT_X16_X16_X2   Q Q R R Y Y
```

Where the holes between Q, R, and Y are NULL (all bits 0)

Note:

This helper component is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads, mutual exclusion, virtual memory, or cache control management must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	03/15/02	Added 64 bit access
1.00a	rmm	07/16/01	First release

```
#include "xbasic_types.h"
#include "xflash.h"
#include "xflash_cfi.h"
```

Functions

XStatus XFlashCFI_ReadCommon (**XFlashGeometry** *GeometryPtr, **XFlashProperties** *PropertiesPtr)

Function Documentation

```
XStatus
XFlashCFI_ReadCommon
    ( XFlashGeometry *
      GeometryPtr,
      XFlashProperties * PropertiesPtr
    )
```

Retrieves the standard CFI data from the part(s), interpret the data, and update the provided geometry and properties structures.

Extended CFI data is part specific and ignored here. This data must be read by the specific part component driver.

Parameters:

GeometryPtr is an input/output parameter. This function expects the BaseAddress and MemoryLayout attributes to be correctly initialized. All other attributes of this structure will be setup using translated CFI data read from the part.

PropertiesPtr is an output parameter. Timing, identification, and programming CFI data will be translated and written to this structure.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_CFI_QUERY_ERROR if an error occurred interpreting the data.
- XST_FLASH_PART_NOT_SUPPORTED if invalid Layout parameter

Note:
None.

Generated on 24 Jun 2004 for Xilinx Device Drivers

flash/v1_00_a/src/xflash_geometry.c File Reference

Detailed Description

This module implements the helper component [XFlashGeometry](#).

The helper component is responsible for containing the geometry information for the flash part(s) and for converting from an absolute part offset to region/block/blockOffset coordinates.

[XFlashGeometry](#) describes the geometry of the entire instance, not the individual parts of that instance. For example, if the user's board architecture uses two 16-bit parts in parallel for a 32-bit data path, then the size of erase blocks and device sizes are multiplied by a factor of two.

Note:

This helper component is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads, mutual exclusion, virtual memory, or cache control management must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	07/16/01	First release

```
#include "xbasic_types.h"  
#include "xflash.h"  
#include "xflash_geometry.h"
```

Functions

XStatus XFlashGeometry_ToAbsolute (**XFlashGeometry** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** BlockOffset, **Xuint32** *AbsoluteOffsetPtr)

XStatus XFlashGeometry_ToBlock (**XFlashGeometry** *InstancePtr, **Xuint32** AbsoluteOffset, **Xuint16** *RegionPtr, **Xuint16** *BlockPtr, **Xuint32** *BlockOffsetPtr)

Xuint32 XFlashGeometry_ConvertLayout (**Xuint8** NumParts, **Xuint8** PartWidth, **Xuint8** PartMode)

Function Documentation

```
Xuint32 XFlashGeometry_ConvertLayout (Xuint8 NumParts,  
                                     Xuint8 PartWidth,  
                                     Xuint8 PartMode  
                                     )
```

Converts array layout into an XFL_LAYOUT_Xa_Xb_Xc constant. This function is typically called during initialization to convert ordinal values delivered by a system generator into the XFL constants which are optimized for use by the flash driver.

Parameters:

NumParts - Number of parts in the array.

PartWidth - Width of each part in bytes.

PartMode - Operation mode of each part in bytes.

Returns:

- XFL_LAYOUT_* - One of the supported layouts
- XNULL if a layout cannot be found that supports the given arguments

Note:

None.

```

XStatus
XFlashGeometry_ToAbsolute
    ( XFlashGeometry *
      InstancePtr,
      Xuint16
      Xuint16
      Xuint32
      Xuint32 *
    )

```

Converts block coordinates to a part offset. Region, Block, & BlockOffset are converted to PartOffset

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Region* is the erase region the physical address appears in.
- Block* is the block within Region the physical address appears in.
- BlockOffset* is the offset within Block where the physical address appears.
- AbsoluteOffsetPtr* is the returned offset value

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_ADDRESS_ERROR if the block coordinates are invalid.

Note:

None.

```

XStatus
XFlashGeometry_ToBlock
    ( XFlashGeometry *
      InstancePtr,
      Xuint32
      Xuint16 *
      Xuint16 *
      Xuint32 *
    )

```

Converts part offset block coordinates. PartOffset is converted to Region, Block, & BlockOffset

Parameters:

InstancePtr is the pointer to the **XFlashGeometry** instance to be worked on.

AbsoluteOffset is the offset within part to find block coordinates for.

RegionPtr is the the region that corresponds to AbsoluteOffset. This is a return parameter.

BlockPtr is the the block within Region that corresponds to AbsoluteOffset. This is a return parameter.

BlockOffsetPtr is the the offset within Block that corresponds to AbsoluteOffset. This is a return parameter.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the block coordinates are invalid.

Note:

None.

flash/v1_00_a/src/xflash_intel.h File Reference

Detailed Description

This is an Intel specific Flash memory component driver for CFI enabled parts.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rrm	07/16/01	First release

```
#include "xbasic_types.h"  
#include "xflash.h"
```

Functions

- XStatus XFlashIntel_Initialize** (**XFlash** *InstancePtr)
- XStatus XFlashIntel_SelfTest** (**XFlash** *InstancePtr)
- XStatus XFlashIntel_Reset** (**XFlash** *InstancePtr)
- XStatus XFlashIntel_SetOptions** (**XFlash** *InstancePtr, **Xuint32** OptionsFlag)
- Xuint32 XFlashIntel_GetOptions** (**XFlash** *InstancePtr)
- XFlashProperties** * **XFlashIntel_GetProperties** (**XFlash** *InstancePtr)
- XFlashGeometry** * **XFlashIntel_GetGeometry** (**XFlash** *InstancePtr)
- XStatus XFlashIntel_DeviceControl** (**XFlash** *InstancePtr, **Xuint32** Command, **Xuint32** Param, **Xuint32** *ReturnPtr)
- XStatus XFlashIntel_Read** (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes, void *DestPtr)
- XStatus XFlashIntel_Write** (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes, void *SrcPtr)
- XStatus XFlashIntel_WriteSuspend** (**XFlash** *InstancePtr, **Xuint32** Offset)
- XStatus XFlashIntel_WriteResume** (**XFlash** *InstancePtr, **Xuint32** Offset)
- XStatus XFlashIntel_Erase** (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)
- XStatus XFlashIntel_EraseSuspend** (**XFlash** *InstancePtr, **Xuint32** Offset)
- XStatus XFlashIntel_EraseResume** (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlashIntel_Lock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)
XStatus XFlashIntel_Unlock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)
XStatus XFlashIntel_GetStatus (**XFlash** *InstancePtr, **Xuint32** Offset)
XStatus XFlashIntel_ReadBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset, **Xuint32** Bytes, void *DestPtr)
XStatus XFlashIntel_WriteBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset, **Xuint32** Bytes, void *SrcPtr)
XStatus XFlashIntel_WriteBlockSuspend (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset)
XStatus XFlashIntel_WriteBlockResume (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset)
XStatus XFlashIntel_EraseBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)
XStatus XFlashIntel_EraseBlockSuspend (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)
XStatus XFlashIntel_EraseBlockResume (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)
XStatus XFlashIntel_LockBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)
XStatus XFlashIntel_UnlockBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)
XStatus XFlashIntel_GetBlockStatus (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)
XStatus XFlashIntel_EraseChip (**XFlash** *InstancePtr)

Function Documentation

```

XStatus
XFlashIntel_DeviceControl
    ( XFlash *
      InstancePtr,
      Xuint32 Command,
      Xuint32 Param,
      Xuint32 * ReturnPtr
    )
  
```

See the base component for a description of this function, its return values, and arguments.

Note:

Intel specific commands:

Command: XFL_INTEL_DEVCTL_SET_RYBY

Description:

Set the mode of the RYBY signal.

Param:

One of XFL_INTEL_RYBY_PULSE_OFF, XFL_INTEL_RYBY_PULSE_ON_ERASE, XFL_INTEL_RYBY_PULSE_ON_PROG, XFL_INTEL_RYBY_PULSE_ON_ERASE_PROG

Return:

None

Command: XFL_INTEL_DEVCTL_GET_LAST_ERROR

Description:

Retrieve the last error condition. The data is in the form of the status register(s) read from the device(s) at the time the error was detected. The registers are formatted verbatim as they are seen on the data bus.

Param:

None

Return:

The contents of the Status registers at the time the last error was detected.

```
XStatus ( XFlash *  
XFlashIntel_Erase InstancePtr,  
                Xuint32 Offset,  
                Xuint32 Bytes  
                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus ( XFlash *  
XFlashIntel_EraseBlock InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block,  
 Xuint16 NumBlocks  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_EraseBlockResume InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:
Region & Block parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus ( XFlash *  
XFlashIntel_EraseBlockSuspend InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:
Region & Block parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

XStatus (**XFlash ***)
XFlashIntel_EraseChip *InstancePtr*

See the base component for a description of this function, its return values, and arguments.

Note:
None.

XStatus (**XFlash ***
XFlashIntel_EraseResume *InstancePtr,*
Xuint32 *Offset*
)

See the base component for a description of this function, its return values, and arguments.

Note:
Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

XStatus (**XFlash ***
XFlashIntel_EraseSuspend *InstancePtr,*
Xuint32 *Offset*
)

See the base component for a description of this function, its return values, and arguments.

Note:
Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

XStatus (**XFlash ***
XFlashIntel_GetBlockStatus *InstancePtr,*
Xuint16 *Region,*
Xuint16 *Block*
)

See the base component for a description of this function, its return values, and arguments.

Note:
The Region & Block parameters are not used because the device's status register appears at every addressible location.

XFlashGeometry* (**XFlash ***)
XFlashIntel_GetGeometry *InstancePtr*

See the base component for a description of this function, its return values, and arguments.

Note:
None.

Xuint32 (**XFlash ***)
XFlashIntel_GetOptions *InstancePtr*

See the base component for a description of this function, its return values, and arguments.

Note:
None.

XFlashProperties* (**XFlash ***)
XFlashIntel_GetProperties *InstancePtr*

See the base component for a description of this function, its return values, and arguments.

Note:
None.

XStatus (**XFlash ***
XFlashIntel_GetStatus *InstancePtr,*
Xuint32 *Offset*
)

See the base component for a description of this function, its return values, and arguments.

Note:

The Offset parameter is not used because the device's status register appears at every addressible location.

```
XStatus ( XFlash * )  
XFlashIntel_Initialize InstancePtr
```

See the base component for a description of this function, its return values, and arguments.

Parameters:

InstancePtr is a pointer to the flash instance to be worked on.

Returns:

- XST_SUCCESS if successful
- XST_FLASH_PART_NOT_SUPPORTED if the part is not supported

Note:

Two geometry attributes MUST be defined prior to invoking this function:

- BaseAddress
- MemoryLayout

```
XStatus ( XFlash *  
XFlashIntel_Lock InstancePtr,  
 Xuint32 Offset,  
 Xuint32 Bytes  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus ( XFlash *  
XFlashIntel_LockBlock InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block,  
 Xuint16 NumBlocks  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_Read InstancePtr,  
                Xuint32 Offset,  
                Xuint32 Bytes,  
                void * DestPtr  
            )
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_ReadBlock InstancePtr,  
                    Xuint16 Region,  
                    Xuint16 Block,  
                    Xuint32 Offset,  
                    Xuint32 Bytes,  
                    void * DestPtr  
                )
```

See the base component for a description of this function, its return values, and arguments.

Note:
The part is assumed to be in read-array mode.

```
XStatus ( XFlash * )  
XFlashIntel_Reset InstancePtr
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

XStatus (**XFlash ***)
XFlashIntel_SelfTest *InstancePtr*

See the base component for a description of this function, its return values, and arguments.

Note:
None.

XStatus (**XFlash ***
XFlashIntel_SetOptions *InstancePtr,*
Xuint32 *OptionsFlag*
)

See the base component for a description of this function, its return values, and arguments.

Note:
None.

XStatus (**XFlash ***
XFlashIntel_Unlock *InstancePtr,*
Xuint32 *Offset,*
Xuint32 *Bytes*
)

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_UnlockBlock InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block,  
 Xuint16 NumBlocks  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_Write InstancePtr,  
 Xuint32 Offset,  
 Xuint32 Bytes,  
 void * SrcPtr  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_WriteBlock InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block,  
 Xuint32 Offset,  
 Xuint32 Bytes,  
 void * SrcPtr  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_WriteBlockResume InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block,  
 Xuint32 Offset  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region, Block, & Offset parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus ( XFlash *  
XFlashIntel_WriteBlockSuspend InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block,  
 Xuint32 Offset  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region, Block, & Offset parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus ( XFlash *  
XFlashIntel_WriteResume InstancePtr,  
 Xuint32 Offset  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus ( XFlash *  
XFlashIntel_WriteSuspend InstancePtr,  
 Xuint32 Offset  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

flash/v1_00_a/src/xflash_intel.c File Reference

Detailed Description

The implementation of the Intel CFI Version of the XFlash component.

This module utilizes the XFlash base component, whose attributes have been defined mostly from a CFI data query. This data is used to define the geometry of the part(s), timeout values for write & erase operations, and optional features.

Note:

- Special consideration has to be given to varying data bus widths. To boost performance, multiple devices in parallel on the data bus are accessed in parallel. Therefore to reduce complexity and increase performance, many local primitive functions are duplicated with the only difference being the width of writes to the devices.

Even with the performance boosting optimizations, the overhead associated with this component is rather high due to the general purpose nature of its design.

Flash block erasing is a time consuming operation with nearly all latency occurring due to the devices' themselves. It takes on the order of 1 second to erase each block.

Writes by comparison are much quicker so driver overhead becomes an issue. The write algorithm has been optimized for bulk data programming and should provide relatively better performance.

- The code/comments refers to WSM frequently. This stands for Write State Machine. WSM is the internal programming engine of the devices.
- This driver and the underlying Intel flash memory does not allow re- programming while code is executing from the same memory.
- If hardware is flakey or fails, then this driver could hang a thread of execution.
- This module has some dependencies on whether it is being unit tested. These areas are noted with conditional compilation based on whether XENV_UNITTEST is defined. This is required because unit testing occurs without real flash devices.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a rmm 07/16/01 First release
1.00a rmm 03/14/02 Added 64 bit array support
1.00a rmm 05/13/03 Fixed diab compiler warnings relating to asserts.

```
#include "xflash_intel.h"  
#include "xflash_intel_l.h"  
#include "xflash_cfi.h"  
#include "xflash_geometry.h"  
#include "xenv.h"
```

Data Structures

```
union StatReg  
struct XFlashVendorData_IntelTag
```

Functions

XStatus XFlashIntel_Initialize (**XFlash** *InstancePtr)
XStatus XFlashIntel_ReadBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset, **Xuint32** Bytes, void *DestPtr)
XStatus XFlashIntel_WriteBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset, **Xuint32** Bytes, void *SrcPtr)
XStatus XFlashIntel_WriteBlockSuspend (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset)
XStatus XFlashIntel_WriteBlockResume (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset)
XStatus XFlashIntel_EraseBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)
XStatus XFlashIntel_EraseBlockSuspend (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)
XStatus XFlashIntel_EraseBlockResume (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)
XStatus XFlashIntel_LockBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)
XStatus XFlashIntel_UnlockBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)
XStatus XFlashIntel_GetBlockStatus (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)
XStatus XFlashIntel_Read (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes, void *DestPtr)
XStatus XFlashIntel_Write (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes, void *SrcPtr)
XStatus XFlashIntel_WriteSuspend (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlashIntel_WriteResume (**XFlash** *InstancePtr, **Xuint32** Offset)
XStatus XFlashIntel_Erase (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)
XStatus XFlashIntel_EraseSuspend (**XFlash** *InstancePtr, **Xuint32** Offset)
XStatus XFlashIntel_EraseResume (**XFlash** *InstancePtr, **Xuint32** Offset)
XStatus XFlashIntel_Lock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)
XStatus XFlashIntel_Unlock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)
XStatus XFlashIntel_GetStatus (**XFlash** *InstancePtr, **Xuint32** Offset)
XStatus XFlashIntel_EraseChip (**XFlash** *InstancePtr)
XStatus XFlashIntel_SelfTest (**XFlash** *InstancePtr)
XStatus XFlashIntel_Reset (**XFlash** *InstancePtr)
XStatus XFlashIntel_SetOptions (**XFlash** *InstancePtr, **Xuint32** OptionsFlag)
Xuint32 XFlashIntel_GetOptions (**XFlash** *InstancePtr)
XFlashGeometry * **XFlashIntel_GetGeometry** (**XFlash** *InstancePtr)
XFlashProperties * **XFlashIntel_GetProperties** (**XFlash** *InstancePtr)
XStatus XFlashIntel_DeviceControl (**XFlash** *InstancePtr, **Xuint32** Command, **Xuint32** Param,
Xuint32 *ReturnPtr)

Function Documentation

```

XStatus
XFlashIntel_DeviceControl
    ( XFlash *
      InstancePtr,
      Xuint32 Command,
      Xuint32 Param,
      Xuint32 * ReturnPtr
    )

```

See the base component for a description of this function, its return values, and arguments.

Note:

Intel specific commands:

Command: XFL_INTEL_DEVCTL_SET_RYBY

Description:

Set the mode of the RYBY signal.

Param:

One of XFL_INTEL_RYBY_PULSE_OFF, XFL_INTEL_RYBY_PULSE_ON_ERASE, XFL_INTEL_RYBY_PULSE_ON_PROG, XFL_INTEL_RYBY_PULSE_ON_ERASE_PROG

Return:

None

Command: XFL_INTEL_DEVCTL_GET_LAST_ERROR

Description:

Retrieve the last error condition. The data is in the form of the status register(s) read from the device(s) at the time the error was detected. The registers are formatted verbatim as they are seen on the data bus.

Param:

None

Return:

The contents of the Status registers at the time the last error was detected.

```
XStatus ( XFlash *  
XFlashIntel_Erase InstancePtr,  
                Xuint32 Offset,  
                Xuint32 Bytes  
                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus ( XFlash *  
XFlashIntel_EraseBlock InstancePtr,  
                Xuint16 Region,  
                Xuint16 Block,  
                Xuint16 NumBlocks  
                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus ( XFlash *  
XFlashIntel_EraseBlockResume InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region & Block parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus ( XFlash *  
XFlashIntel_EraseBlockSuspend InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region & Block parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus ( XFlash * )  
XFlashIntel_EraseChip InstancePtr
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus ( XFlash *  
XFlashIntel_EraseResume InstancePtr,  
 Xuint32 Offset  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus ( XFlash *  
XFlashIntel_EraseSuspend InstancePtr,  
 Xuint32 Offset  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus ( XFlash *  
XFlashIntel_GetBlockStatus InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

The Region & Block parameters are not used because the device's status register appears at every addressable location.

```
XFlashGeometry* ( XFlash * )  
XFlashIntel_GetGeometry InstancePtr
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

Xuint32 (**XFlash ***)

XFlashIntel_GetOptions *InstancePtr*

See the base component for a description of this function, its return values, and arguments.

Note:
None.

XFlashProperties* (**XFlash ***)

XFlashIntel_GetProperties *InstancePtr*

See the base component for a description of this function, its return values, and arguments.

Note:
None.

XStatus (**XFlash ***

XFlashIntel_GetStatus *InstancePtr,*
Xuint32 *Offset*
)

See the base component for a description of this function, its return values, and arguments.

Note:
The Offset parameter is not used because the device's status register appears at every addressible location.

XStatus (**XFlash ***)

XFlashIntel_Initialize *InstancePtr*

See the base component for a description of this function, its return values, and arguments.

Parameters:

InstancePtr is a pointer to the flash instance to be worked on.

Returns:

- XST_SUCCESS if successful
- XST_FLASH_PART_NOT_SUPPORTED if the part is not supported

Note:

Two geometry attributes MUST be defined prior to invoking this function:

- BaseAddress
- MemoryLayout

```
XStatus ( XFlash *  
XFlashIntel_Lock InstancePtr,  
 Xuint32 Offset,  
 Xuint32 Bytes  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus ( XFlash *  
XFlashIntel_LockBlock InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block,  
 Xuint16 NumBlocks  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus ( XFlash *
XFlashIntel_Read InstancePtr,
Xuint32 Offset,
Xuint32 Bytes,
void * DestPtr
)
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *
XFlashIntel_ReadBlock InstancePtr,
Xuint16 Region,
Xuint16 Block,
Xuint32 Offset,
Xuint32 Bytes,
void * DestPtr
)
```

See the base component for a description of this function, its return values, and arguments.

Note:
The part is assumed to be in read-array mode.

```
XStatus ( XFlash * )
XFlashIntel_Reset InstancePtr
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash * )
XFlashIntel_SelfTest InstancePtr
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_SetOptions InstancePtr,  
 Xuint32 OptionsFlag  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_Unlock InstancePtr,  
 Xuint32 Offset,  
 Xuint32 Bytes  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_UnlockBlock InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block,  
 Xuint16 NumBlocks  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_Write InstancePtr,  
 Xuint32 Offset,  
 Xuint32 Bytes,  
 void * SrcPtr  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_WriteBlock InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block,  
 Xuint32 Offset,  
 Xuint32 Bytes,  
 void * SrcPtr  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:
None.

```
XStatus ( XFlash *  
XFlashIntel_WriteBlockResume InstancePtr,  
 Xuint16 Region,  
 Xuint16 Block,  
 Xuint32 Offset  
 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region, Block, & Offset parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus ( XFlash *  
XFlashIntel_WriteBlockSuspend InstancePtr,  
                                Xuint16 Region,  
                                Xuint16 Block,  
                                Xuint32 Offset  
                                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region, Block, & Offset parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus ( XFlash *  
XFlashIntel_WriteResume InstancePtr,  
                          Xuint32 Offset  
                          )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus ( XFlash *  
XFlashIntel_WriteSuspend InstancePtr,  
                          Xuint32 Offset  
                          )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

Generated on 24 Jun 2004 for Xilinx Device Drivers

flash/v1_00_a/src/xflash_intel_i.h File Reference

Detailed Description

Contains identifiers and low-level macros/functions for the Intel 28FxxxJ3A StrataFlash driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes

1.00a	rpm	05/06/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

Defines

```
#define XFlashIntel_mSendCmd(BaseAddress, Offset, Command)
```

Functions

```
int XFlashIntel_WaitReady (Xuint32 BaseAddress, Xuint32 Offset)  
int XFlashIntel_WriteAddr (Xuint32 BaseAddress, Xuint32 Offset, Xuint8 *BufferPtr,  
    unsigned int Length)  
int XFlashIntel_ReadAddr (Xuint32 BaseAddress, Xuint32 Offset, Xuint8 *BufferPtr,  
    unsigned int Length)  
int XFlashIntel_EraseAddr (Xuint32 BaseAddress, Xuint32 Offset)  
int XFlashIntel_LockAddr (Xuint32 BaseAddress, Xuint32 Offset)  
int XFlashIntel_UnlockAddr (Xuint32 BaseAddress, Xuint32 Offset)
```

Define Documentation

```
#define XFlashIntel_mSendCmd( BaseAddress,  
                             Offset,  
                             Command  )
```

Send the specified command to the flash device.

Parameters:

BaseAddress is the base address of the device
Offset is the offset address from the base address.
Command is the command to send.

Returns:

None.

Note:

None.

Function Documentation

```
int  
XFlashIntel_EraseAddr ( Xuint32  
                       BaseAddress,  
                       Xuint32 Offset  
                       )
```

Erase the block beginning at the specified address. The user is assumed to know the block boundaries and pass in an address/offset that is block aligned.

Parameters:

BaseAddress is the base address of the device.
Offset is the offset address from the base address to begin erasing. This offset is assumed to be a block boundary.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```
int  
XFlashIntel_LockAddr ( Xuint32  
                      BaseAddress,  
                      Xuint32 Offset  
                      )
```

Lock the block beginning at the specified address. The user is assumed to know the block boundaries and pass in an address/offset that is block aligned.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to lock. This offset is assumed to be a block boundary.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```
int  
XFlashIntel_ReadAddr ( Xuint32  
                      BaseAddress,  
                      Xuint32 Offset,  
                      Xuint8 * BufferPtr,  
                      unsigned int Length  
                      )
```

Read some number of bytes from the specified address.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to begin reading.

BufferPtr is the buffer used to store the bytes that are read.

Length is the number of bytes to read from flash.

Returns:

The number of bytes actually read.

Note:

This function assumes 32-bit access to the flash array.

```
int  
XFlashIntel_UnlockAddr ( Xuint32  
                        BaseAddress,  
                        Xuint32 Offset  
                        )
```

Unlock the block beginning at the specified address. The user is assumed to know the block boundaries and pass in an address/offset that is block aligned.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to unlock. This offset is assumed to be a block boundary.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```
int  
XFlashIntel_WaitReady ( Xuint32  
                       BaseAddress,  
                       Xuint32 Offset  
                       )
```

Wait for the flash array to be in the ready state (ready for a command).

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```
int  
XFlashIntel_WriteAddr ( Xuint32  
                        Xuint32  
                        Xuint8 *  
                        unsigned int  
                        )  
                        BaseAddress,  
                        Offset,  
                        BufferPtr,  
                        Length
```

Write the specified address with some number of bytes.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to begin writing.

BufferPtr is the buffer that will be written to flash.

Length is the number of bytes in *BufferPtr* that will be written to flash.

Returns:

The number of bytes actually written.

Note:

This function assumes 32-bit access to the flash array.

flash/v1_00_a/src/xflash_intel_l.c File Reference

Detailed Description

Contains low-level functions for the XFlashIntel driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes

1.00a	rpm	05/06/02	First release

```
#include "xflash_intel_l.h"
```

Functions

int **XFlashIntel_WriteAddr** (**Xuint32** BaseAddress, **Xuint32** Offset, **Xuint8** *BufferPtr, unsigned int Length)

int **XFlashIntel_ReadAddr** (**Xuint32** BaseAddress, **Xuint32** Offset, **Xuint8** *BufferPtr, unsigned int Length)

int **XFlashIntel_EraseAddr** (**Xuint32** BaseAddress, **Xuint32** Offset)

int **XFlashIntel_LockAddr** (**Xuint32** BaseAddress, **Xuint32** Offset)

int **XFlashIntel_UnlockAddr** (**Xuint32** BaseAddress, **Xuint32** Offset)

int **XFlashIntel_WaitReady** (**Xuint32** BaseAddress, **Xuint32** Offset)

Function Documentation

```
int  
XFlashIntel_EraseAddr ( Xuint32  
                        BaseAddress,  
                        Xuint32 Offset  
                        )
```

Erase the block beginning at the specified address. The user is assumed to know the block boundaries and pass in an address/offset that is block aligned.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to begin erasing. This offset is assumed to be a block boundary.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```
int  
XFlashIntel_LockAddr ( Xuint32  
                      BaseAddress,  
                      Xuint32 Offset  
                      )
```

Lock the block beginning at the specified address. The user is assumed to know the block boundaries and pass in an address/offset that is block aligned.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to lock. This offset is assumed to be a block boundary.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```

int
XFlashIntel_ReadAddr ( Xuint32
                        BaseAddress,
                        Xuint32 Offset,
                        Xuint8 * BufferPtr,
                        unsigned int Length
                        )

```

Read some number of bytes from the specified address.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to begin reading.

BufferPtr is the buffer used to store the bytes that are read.

Length is the number of bytes to read from flash.

Returns:

The number of bytes actually read.

Note:

This function assumes 32-bit access to the flash array.

```

int
XFlashIntel_UnlockAddr ( Xuint32
                          BaseAddress,
                          Xuint32 Offset
                          )

```

Unlock the block beginning at the specified address. The user is assumed to know the block boundaries and pass in an address/offset that is block aligned.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to unlock. This offset is assumed to be a block boundary.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```

int
XFlashIntel_WaitReady ( Xuint32
                        BaseAddress,
                        Xuint32 Offset
                        )

```

Wait for the flash array to be in the ready state (ready for a command).

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```

int
XFlashIntel_WriteAddr ( Xuint32
                       BaseAddress,
                       Xuint32 Offset,
                       Xuint8 * BufferPtr,
                       unsigned int Length
                       )

```

Write the specified address with some number of bytes.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to begin writing.

BufferPtr is the buffer that will be written to flash.

Length is the number of bytes in *BufferPtr* that will be written to flash.

Returns:

The number of bytes actually written.

Note:

This function assumes 32-bit access to the flash array.

gemac/v1_00_f/src/xgemac.h File Reference

Detailed Description

The Xilinx 1 gigabit Ethernet driver component (GEMAC).

The Xilinx Ethernet 1Gbit MAC supports the following features:

- Scatter-gather & simple DMA operations, as well as simple memory mapped direct I/O interface (FIFOs)
- Gigabit Media Independent Interface (GMII) for connection to external 1Gbit Mbps PHY transceivers. Supports 125Mhz 10 bit interface (TBI) to external PHY and SerDes to external transceiver
- GMII management control reads and writes with GMII PHYs
- Independent internal transmit and receive FIFOs
- CSMA/CD compliant operations for half-duplex modes
- Internal loopback
- Automatic source address insertion or overwrite (programmable)
- Automatic FCS insertion and stripping (programmable)
- Automatic pad insertion and stripping (programmable)
- Pause frame (flow control) detection in full-duplex mode
- Programmable interframe gap
- VLAN frame support
- Jumbo frame support
- Pause frame support

Hardware limitations in this version

- Always in promiscuous mode
- Hardware statistic counters not implemented
- Unicast, multicast, broadcast, and promiscuous address filtering not implemented
- Half-duplex mode not implemented
- Auto source address insertion not implemented

The device driver does not support the features listed below

- Programmable PHY reset signal

Device driver limitations in this version

- Simple DMA untested

Driver Description

The device driver enables higher layer software (e.g., an application) to communicate to the GEMAC. The driver handles transmission and reception of Ethernet frames, as well as configuration of the controller. It does not handle protocol stack

functionality such as Link Layer Control (LLC) or the Address Resolution Protocol (ARP). The protocol stack that makes use of the driver handles this functionality.

Since the driver is a simple pass-through mechanism between a protocol stack and the GEMAC, no assembly or disassembly of Ethernet frames is done at the driver-level. This assumes that the protocol stack passes a correctly formatted Ethernet frame to the driver for transmission, and that the driver does not validate the contents of an incoming frame

A single device driver can support multiple GEMACs.

The driver is designed for a zero-copy buffer scheme when used with DMA. Direct FIFO modes requires a buffer copy to/from data FIFOs.

PHY Communication

The driver provides rudimentary read and write functions to allow the higher layer software to access the PHY. The GEMAC provides MII registers for the driver to access. This management interface can be parameterized away in the FPGA implementation process.

Asynchronous Callbacks

The driver services interrupts and passes Ethernet frames to the higher layer software through asynchronous callback functions. When using the driver directly (i.e., not with the RTOS protocol stack), the higher layer software must register its callback functions during initialization. The driver requires callback functions for received frames, for confirmation of transmitted frames, and for asynchronous errors.

Interrupts

The driver has no dependencies on the interrupt controller. The driver provides two interrupt handlers.

[XGemac_IntrHandlerDma\(\)](#) handles interrupts when the GEMAC is configured with scatter-gather DMA.

[XGemac_IntrHandlerFifo\(\)](#) handles interrupts when the GEMAC is configured for direct FIFO I/O or simple DMA. Either of these routines can be connected to the system interrupt controller by the user.

Device Reset

Some errors that can occur in the device require a device reset. These errors are listed in the [XGemac_SetErrorHandler\(\)](#) function header. The user's error handler is responsible for resetting the device and re-configuring it based on its needs (the driver does not save the current configuration). When integrating into an RTOS, these reset and re-configure obligations are taken care of by the Xilinx adapter software.

Polled Mode Operation

We hope you didn't purchase 1GB/sec GEMAC only to use it in polled mode, but in case you did, the driver supports this mode. See [XGemac_SetOptions\(\)](#), [XGemac_PollSend\(\)](#), and [XGemac_PollRecv\(\)](#).

Buffer data is copied to and from the FIFOs under processor control. The calling function is blocked during this copy.

Interrupt Driven FIFO Mode Operation

Buffer data is copied to and from the FIFOs under processor control. Interrupts occur when a new frame has arrived or a frame queued to transmit has been sent. The user must register callback functions with the driver to service frame reception and transmission. See [XGemac_FifoSend\(\)](#), [XGemac_FifoRecv\(\)](#), [XGemac_SetFifoRecvHandler\(\)](#),

[XGmac_SetFifoSendHandler\(\)](#).

Interrupt Driven DMA Mode Operation

TBD

Interrupt Driven Scatter Gather DMA Mode Operation

This is the fastest mode of operation. Buffer data is copied to and from the FIFOs under DMA control. Multiple frames either partial or whole can be transferred with no processor intervention using the scatter gather buffer descriptor list. The user must register callback functions with the driver to service frame reception and transmission. See [XGmac_SgSend\(\)](#), [XGmac_SgRecv\(\)](#), [XGmac_SetSgRecvHandler\(\)](#), [XGmac_SetSgSendHandler\(\)](#).

The frequency of interrupts can be controlled with the interrupt coalescing features of the scatter-gather DMA engine. Instead of interrupting after each packet has been processed, the scatter-gather DMA engine will interrupt when the packet count threshold is reached OR when the packet waitbound timer has expired. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in this implementation. See [XGmac_SetPktThreshold\(\)](#), and [XGmac_SetPktWaitBound\(\)](#).

The user must setup a block of memory for transmit and receive buffer descriptor storage. Prior to using scatter gather. See [XGmac_SetSgRecvSpace\(\)](#) and [XGmac_SetSgSendSpace\(\)](#).

PLB Alignment Considerations

Scatter gather buffer descriptors must be aligned on 8 byte boundaries. Frame buffers can be on any alignment. Failure to follow alignment restrictions will result in asserts from the driver or bad/corrupted data being transferred.

Cache Considerations

Do not cache buffers or scatter-gather buffer descriptor space when using DMA mode. Doing so will cause cache coherency problems resulting in bad/corrupted data being transferred.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are stored in the [xgemac_g.c](#) files. A table is defined where each entry contains configuration information for a GEMAC device. This information includes such things as the base address of the memory-mapped device, and whether the device has DMA, counter registers, or GMII support.

The driver tries to use the features built into the device. So if, for example, the hardware is configured with scatter-gather DMA, the driver expects to start the scatter-gather channels. If circumstances exist when the hardware must be used in a mode that differs from its default configuration, the user may modify the device config table prior to invoking [XGmac_Initialize\(\)](#):

```
XGmac_Config *ConfigPtr;  
  
ConfigPtr = XGmac_LookupConfig(DeviceId);  
ConfigPtr->IpIfDmaConfig = XGE_CFG_NO_DMA;
```

The user should understand that changing the config table is not without risk. For example, if the hardware is not configured without DMA and the config table is changed to include it, then system errors will occur when the driver is initialized.

Asserts

Asserts are used within all Xilinx drivers to enforce constraints on argument values. Asserts can be turned off on a system-wide basis by defining, at compile time, the NDEBUG identifier. By default, asserts are turned on and it is recommended that application developers leave asserts on during development. Substantial performance improvements can be seen when asserts are disabled.

Building the driver

The **XGemac** driver is composed of several source files. Why so many? This allows the user to build and link only those parts of the driver that are necessary. Since the GEMAC hardware can be configured in various ways (e.g., with or without DMA), the driver too can be built with varying features. For the most part, this means that besides always linking in **xgemac.c**, you link in only the driver functionality you want. Some of the choices you have are polled vs. interrupt, interrupt with FIFOs only vs. interrupt with DMA, self-test diagnostics, and driver statistics. Note that currently the DMA code must be linked in, even if you don't have DMA in the device.

Note:

Xilinx drivers are typically composed of two components, one is the driver and the other is the adapter. The driver is independent of OS and processor and is intended to be highly portable. The adapter is OS-specific and facilitates communication between the driver and the OS.

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/13/03	First release
1.00b	ecm	03/25/03	Revision update
1.00c	rmm	05/28/03	Dma added, interframe gap interface change, added auto-negotiate option, removed phy function prototypes, added constant to default to no hw counters
1.00d	rmm	06/11/03	Added Jumbo frame capabilities, removed no hw counters constant
1.00e	rmm	11/14/03	Added XGE_NO_SGEND_INT_OPTION. SG DMA callbacks are invoked once for all packets received instead of once for each packet.
1.00f	rmm	12/22/03	Added XGemac_GetSgRecvFreeDesc() and GetSgSendFreeDesc() functions. Switched to DRE enabled write Packet FIFOs. Fixed SG DMA bug that could lead to a partially queued packet being transferred. Redesigned XGemac_Start() and XGemac_Stop() functions. Moved where interrupt status is cleared after transmission in XGemac_PollSend() .

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xparameters.h"
#include "xpacket_fifo_v2_00_a.h"
#include "xdma_channel.h"
```

Data Structures

```
struct XGmac
struct XGmac_Config
struct XGmac_HardStats
struct XGmac_SoftStats
```

Configuration options

See **XGmac_SetOptions()**

```
#define XGE_UNICAST_OPTION
#define XGE_BROADCAST_OPTION
#define XGE_PROMISC_OPTION
#define XGE_FDUPLEX_OPTION
#define XGE_POLLED_OPTION
#define XGE_LOOPBACK_OPTION
#define XGE_FLOW_CONTROL_OPTION
#define XGE_INSERT_PAD_OPTION
#define XGE_INSERT_FCS_OPTION
#define XGE_STRIP_PAD_FCS_OPTION
#define XGE_AUTO_NEGOTIATE_OPTION
#define XGE_VLAN_OPTION
#define XGE_JUMBO_OPTION
#define XGE_NO_SGEND_INT_OPTION
```

Configuration options not yet supported

These options to be supported in later versions of HW and this driver.

```
#define XGE_MULTICAST_OPTION
#define XGE_INSERT_ADDR_OPTION
#define XGE_OVWRT_ADDR_OPTION
```

Macro functions

```
#define XGmac_mIsSgDma(InstancePtr)
#define XGmac_mIsSimpleDma(InstancePtr)
#define XGmac_mIsDma(InstancePtr)
```

Callbacks

```
typedef void(* XGmac_SgHandler )(void *CallBackRef, XBufDescriptor *BdPtr, Xuint32 NumBds)
typedef void(* XGmac_FifoHandler )(void *CallBackRef)
typedef void(* XGmac_ErrorHandler )(void *CallBackRef, XStatus ErrorCode)
```

Functions

XStatus **XGmac_Initialize** (**XGmac** *InstancePtr, **Xuint16** DeviceId)

XStatus **XGmac_Start** (**XGmac** *InstancePtr)

XStatus **XGmac_Stop** (**XGmac** *InstancePtr)

void **XGmac_Reset** (**XGmac** *InstancePtr)

XGmac_Config * **XGmac_LookupConfig** (**Xuint16** DeviceId)

XStatus **XGmac_SetMacAddress** (**XGmac** *InstancePtr, **Xuint8** *AddressPtr)

void **XGmac_GetMacAddress** (**XGmac** *InstancePtr, **Xuint8** *BufferPtr)

XStatus **XGmac_SelfTest** (**XGmac** *InstancePtr)

XStatus **XGmac_PollSend** (**XGmac** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** ByteCount)

XStatus **XGmac_PollRecv** (**XGmac** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** *ByteCountPtr)

XStatus **XGmac_SgSend** (**XGmac** *InstancePtr, XBufDescriptor *BdPtr, int Delay)

XStatus **XGmac_SgRecv** (**XGmac** *InstancePtr, XBufDescriptor *BdPtr)

XStatus **XGmac_SetPktThreshold** (**XGmac** *InstancePtr, **Xuint32** Direction, **Xuint8** Threshold)

XStatus **XGmac_GetPktThreshold** (**XGmac** *InstancePtr, **Xuint32** Direction, **Xuint8** *ThreshPtr)

XStatus **XGmac_SetPktWaitBound** (**XGmac** *InstancePtr, **Xuint32** Direction, **Xuint32** TimerValue)

XStatus **XGmac_GetPktWaitBound** (**XGmac** *InstancePtr, **Xuint32** Direction, **Xuint32** *WaitPtr)

XStatus **XGmac_SetSgRecvSpace** (**XGmac** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)

XStatus **XGmac_SetSgSendSpace** (**XGmac** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)

void **XGmac_SetSgRecvHandler** (**XGmac** *InstancePtr, void *CallBackRef, **XGmac_SgHandler** FuncPtr)

void **XGmac_SetSgSendHandler** (**XGmac** *InstancePtr, void *CallBackRef, **XGmac_SgHandler** FuncPtr)

unsigned **XGmac_GetSgSendFreeDesc** (**XGmac** *InstancePtr)

unsigned **XGmac_GetSgRecvFreeDesc** (**XGmac** *InstancePtr)

void **XGmac_IntrHandlerDma** (void *InstancePtr)

XStatus **XGmac_FifoSend** (**XGmac** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** ByteCount)

XStatus **XGmac_FifoRecv** (**XGmac** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** *ByteCountPtr)

void **XGmac_SetFifoRecvHandler** (**XGmac** *InstancePtr, void *CallBackRef, **XGmac_FifoHandler** FuncPtr)

void **XGmac_SetFifoSendHandler** (**XGmac** *InstancePtr, void *CallBackRef, **XGmac_FifoHandler** FuncPtr)

void **XGmac_IntrHandlerFifo** (void *InstancePtr)

void **XGmac_SetErrorHandler** (**XGmac** *InstancePtr, void *CallBackRef, **XGmac_ErrorHandler** FuncPtr)

XStatus **XGmac_SetOptions** (**XGmac** *InstancePtr, **Xuint32** OptionFlag)

Xuint32 **XGmac_GetOptions** (**XGmac** *InstancePtr)

XStatus **XGmac_MulticastAdd** (**XGmac** *InstancePtr, **Xuint8** Location, **Xuint8** *AddressPtr)

XStatus **XGmac_MulticastClear** (**XGmac** *InstancePtr, **Xuint8** Location)

void **XGmac_GetSoftStats** (**XGmac** *InstancePtr, **XGmac_SoftStats** *StatsPtr)

void **XGmac_ClearSoftStats** (**XGmac** *InstancePtr)

XStatus **XGmac_GetHardStats** (**XGmac** *InstancePtr, **XGmac_HardStats** *StatsPtr)

XStatus **XGmac_SetInterframeGap** (**XGmac** *InstancePtr, **Xuint8** Ifg)

void **XGmac_GetInterframeGap** (**XGmac** *InstancePtr, **Xuint8** *IfgPtr)

XStatus **XGmac_SendPause** (**XGmac** *InstancePtr, **Xuint16** PausePeriod)

XStatus **XGmac_MgtRead** (**XGmac** *InstancePtr, int PhyAddress, int Register, **Xuint16** *DataPtr)

XStatus **XGmac_MgtWrite** (**XGmac** *InstancePtr, int PhyAddress, int Register, **Xuint16** Data)

Define Documentation

#define

XGE_AUTO_NEGOTIATE_OPTION

Turn on PHY auto-negotiation (default is on)

#define

XGE_BROADCAST_OPTION

Broadcast addressing on or off (default is on)

#define

XGE_FDUPLEX_OPTION

Full duplex on or off (default is off)

#define

XGE_FLOW_CONTROL_OPTION

Interpret pause frames in full duplex mode (default is off)

#define

XGE_INSERT_ADDR_OPTION

Insert source address on transmit (default is on)

#define

XGE_INSERT_FCS_OPTION

Insert FCS (CRC) on transmit (default is on)

#define

XGE_INSERT_PAD_OPTION

Pad short frames on transmit (default is on)

#define

XGE_JUMBO_OPTION

Allow reception and transmission of Jumbo frames (default is off)

#define

XGE_LOOPBACK_OPTION

Internal loopback on or off (default is off)

#define
XGE_MULTICAST_OPTION

Multicast addressing on or off (default is off)

#define
XGE_NO_SGEND_INT_OPTION

Disables the SGEND interrupt with SG DMA. Setting this option to ON may help bulk data transfer performance when utilizing higher packet threshold counts on slower systems (default is off)

#define
XGE_OVWRT_ADDR_OPTION

Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)

#define
XGE_POLLED_OPTION

Polled mode on or off (default is off)

#define
XGE_PROMISC_OPTION

Promiscuous addressing on or off (default is off)

#define
XGE_STRIP_PAD_FCS_OPTION

Strip padding and FCS from received frames (default is off)

#define
XGE_UNICAST_OPTION

Unicast addressing on or off (default is on)

#define
XGE_VLAN_OPTION

Allow reception and transmission of VLAN frames (default is off)

#define XGemac_mIsDma(InstancePtr)

This macro determines if the device is currently configured with DMA (either simple DMA or scatter-gather DMA)

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Returns:

Boolean XTRUE if the device is configured with DMA, or XFALSE otherwise

Note:

Signature: Xboolean **XGmac_mIsDma**(XGmac *InstancePtr)

```
#define XGmac_mIsSgDma( InstancePtr )
```

This macro determines if the device is currently configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Returns:

Boolean XTRUE if the device is configured for scatter-gather DMA, or XFALSE if it is not.

Note:

Signature: Xboolean **XGmac_mIsSgDma**(XGmac *InstancePtr)

```
#define XGmac_mIsSimpleDma( InstancePtr )
```

This macro determines if the device is currently configured for simple DMA.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Returns:

Boolean XTRUE if the device is configured for simple DMA, or XFALSE otherwise

Note:

Signature: Xboolean **XGmac_mIsSimpleDma**(XGmac *InstancePtr)

Typedef Documentation

```
typedef void(* XGmac_ErrorHandler)(void *CallbackRef, XStatus  
ErrorCode)
```

Callback when an asynchronous error occurs.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions (see [XGmac_SetFifoRecvHandler\(\)](#), [XGmac_SetFifoSendHandler\(\)](#), [XGmac_SetSgRecvHandler\(\)](#), and [XGmac_SetSgSendHandler\(\)](#)).

ErrorCode is the Xilinx error code that was detected. (see [xstatus.h](#)).

```
typedef void(* XGmac_FifoHandler)(void  
*CallbackRef)
```

Callback when data is sent or received with direct FIFO communication.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions (see [XGmac_SetFifoRecvHandler\(\)](#) and [XGmac_SetFifoSendHandler\(\)](#)).

```
typedef void(* XGmac_SgHandler)(void *CallbackRef, XBufDescriptor *BdPtr, Xuint32  
NumBds)
```

Callback when an Ethernet frame is sent or received with scatter-gather DMA.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions (see [XGmac_SetSgRecvHandler\(\)](#) and [XGmac_SetSgSendHandler\(\)](#)).

BdPtr is a pointer to the first buffer descriptor in a list of buffer descriptors that describe a single frame.

NumBds is the number of buffer descriptors in the list pointed to by *BdPtr*.

Function Documentation

```
void  
XGmac_ClearSoftStats ( XGmac *  
InstancePtr )
```

Clear the [XGmac_SoftStats](#) structure for this driver.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

Returns:

None.

Note:

None.

```

XStatus
XGmac_FifoRecv
    ( XGmac *
      InstancePtr,
      Xuint8 * BufPtr,
      Xuint32 * ByteCountPtr
    )

```

Receive an Ethernet frame into the given buffer if a frame has been received by the hardware. This function is typically called by the user in response to an interrupt invoking the receive callback function as set by **XGmac_SetFifoRecvHandler()**.

The supplied buffer should be properly aligned (see **xgemac.h**) and large enough to contain the biggest frame for the current operating mode of the GEMAC device (approx 1518 bytes for normal frames and 9000 bytes for jumbo frames).

If the device is configured with DMA, simple DMA will be used to transfer the buffer from the GEMAC to memory. In this case, the receive buffer must not be cached (see **xgemac.h**).

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BufPtr is a pointer to a memory buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- o XST_SUCCESS if the frame was sent successfully
- o XST_DEVICE_IS_STOPPED if the device has not yet been started
- o XST_NOT_INTERRUPT if the device is not in interrupt mode
- o XST_NO_DATA if there is no frame to be received from the FIFO
- o XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.
- o XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- o XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

The input buffer must be big enough to hold the largest Ethernet frame.

```

XStatus
XGmac_FifoSend
    ( XGmac *
      InstancePtr,
      Xuint8 * BufPtr,
      Xuint32 ByteCount
    )

```

Send an Ethernet frame using packet FIFO with interrupts. The caller provides a contiguous-memory buffer and its length.

The callback function set by using **XGmac_SetFifoSendHandler()** is invoked when the transmission is complete.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field.

If the device is configured with simple DMA, simple DMA will be used to transfer the buffer from memory to the GEMAC. This means that this buffer should not be cached.

Parameters:

- InstancePtr* is a pointer to the **XGmac** instance to be worked on.
- BufPtr* is a pointer to a buffer containing the Ethernet frame to be sent.
- ByteCount* is the size of the Ethernet frame.

Returns:

- o XST_SUCCESS if the frame was successfully sent. An interrupt is generated when the GEMAC transmits the frame and the driver calls the callback set with **XGmac_SetFifoSendHandler()**
- o XST_DEVICE_IS_STOPPED if the device has not yet been started
- o XST_NOT_INTERRUPT if the device is not in interrupt mode
- o XST_FIFO_NO_ROOM if there is no room in the FIFO for this frame
- o XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- o XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

```
XStatus XGmac_GetHardStats(XGmac *InstancePtr,  
                          XGmac_HardStats *StatsPtr)
```

Get a snapshot of the current hardware statistics counters.

Parameters:

- InstancePtr* is a pointer to the **XGmac** instance to be worked on.
- StatsPtr* is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

XST_SUCCESS if counters were read and copied to user space XST_NO_FEATURE if counters are not part of the gemac hw

Note:

None.

```
void
XGmac_GetInterframeGap ( XGmac *
                        InstancePtr,
                        Xuint8 * IfgPtr
                        )
```

Get the interframe gap. See the description of interframe gap above in [XGmac_SetInterframeGap\(\)](#).

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

IfgPtr is a pointer to an 8-bit buffer into which the interframe gap value will be copied. The LSB value is 8 bit times.

Returns:

None. The values of the interframe gap parts are copied into the output parameters.

```
void
XGmac_GetMacAddress ( XGmac *
                    InstancePtr,
                    Xuint8 * BufferPtr
                    )
```

Get the MAC address for this driver/device.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

BufferPtr is an output parameter, and is a pointer to a buffer into which the current MAC address will be copied. The buffer must be at least 6 bytes.

Returns:

None.

Note:

None.

```
Xuint32
XGmac_GetOptions ( XGmac *
                 InstancePtr
                 )
```

Get Ethernet driver/device options. The 32-bit value returned is a bit-mask representing the options (XGE_*_OPTION). A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

Returns:

The 32-bit value of the Ethernet options. The value is a bit-mask representing all options that are currently enabled. See [xgmac.h](#) for a description of the available options.

Note:

None.

```

XStatus ( XGemac *
XGemac_GetPktThreshold InstancePtr,
Xuint32 Direction,
Xuint8 * ThreshPtr
)

```

Get the value of the packet count threshold for the scatter-gather DMA engine. See [xgemac.h](#) for more discussion of interrupt coalescing features.

Parameters:

- InstancePtr* is a pointer to the **XGemac** instance to be worked on.
- Direction* indicates the channel, XGE_SEND or XGE_RECV, to get.
- ThreshPtr* is a pointer to the byte into which the current value of the packet threshold register will be copied.

Returns:

- o XST_SUCCESS if the packet threshold was retrieved successfully
- o XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- o XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None

```

XStatus ( XGemac *
XGemac_GetPktWaitBound InstancePtr,
Xuint32 Direction,
Xuint32 * WaitPtr
)

```

Get the packet wait bound timer for this driver/device. See [xgemac.h](#) for more discussion of interrupt coalescing features.

Parameters:

- InstancePtr* is a pointer to the **XGemac** instance to be worked on.
- Direction* indicates the channel, XGE_SEND or XGE_RECV, to read.
- WaitPtr* is a pointer to the byte into which the current value of the packet wait bound register will be copied. Units are in milliseconds. Range is 0 - 1023. A value of 0 disables the timer.

Returns:

- o XST_SUCCESS if the packet wait bound was retrieved successfully
- o XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- o XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```

unsigned ( XGemac *
XGemac_GetSgRecvFreeDesc InstancePtr
)

```

Return the number of free buffer descriptor slots that can be added to the receive descriptor ring with **XGemac_SgRecv()** before filling it up.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

- The number of descriptors that can be given to the HW with **XGemac_SgRecv()**
- 0 if no room is left or the device is not configured for SG DMA

Note:

None.

```
unsigned ( XGemac * )  
XGemac_GetSgSendFreeDesc InstancePtr
```

Return the number of free buffer descriptor slots that can be added to the send descriptor ring with **XGemac_SgSend()** before filling it up.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

- The number of descriptors that can be given to the HW with **XGemac_SgSend()**
- 0 if no room is left or the device is not configured for SG DMA

Note:

None.

```
void ( XGemac *  
XGemac_GetSoftStats InstancePtr,  
XGemac_SoftStats * StatsPtr  
)
```

Get a copy of the **XGemac_SoftStats** structure, which contains the current statistics for this driver as maintained by software counters. The statistics are cleared at initialization or on demand using the **XGemac_ClearSoftStats()** function.

The **DmaErrors** and **FifoErrors** counts indicate that the device has been or needs to be reset. Reset of the device is the responsibility of the upper layer software.

Use **XGemac_GetHardStats()** to retrieve hardware maintained statistics (if so configured).

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None.

Note:

None.

```
XStatus ( XGmac *
XGmac_Initialize InstancePtr,
Xuint16 DeviceId
)
```

Initialize a specific **XGmac** instance/driver. The initialization entails:

- Clearing memory occupied by the **XGmac** structure
- Initialize fields of the **XGmac** structure
- Clear the Ethernet statistics for this device
- Initialize the IPIF component with its register base address
- Configure the FIFO components with their register base addresses.
- If the device is configured with DMA, configure the DMA channel components with their register base addresses. At some later time, memory pools for the scatter-gather descriptor lists are to be passed to the driver.
- Reset the Ethernet MAC

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XGmac** instance. Passing in a device id associates the generic **XGmac** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None.

```
void ( void *
XGmac_IntrHandlerDma InstancePtr
)
```

The interrupt handler for the Ethernet driver when configured with scatter- gather DMA.

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, Send Packet FIFO, Recv DMA channel, or Send DMA channel. The packet FIFOs only interrupt during "deadlock" conditions.

Parameters:

InstancePtr is a pointer to the **XGmac** instance that just interrupted.

Returns:

None.

Note:

None.

```
void  
XGmac_IntrHandlerFifo ( void *  
InstancePtr )
```

The interrupt handler for the Ethernet driver when configured for direct FIFO communication (as opposed to DMA).

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, or Send Packet FIFO. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the MAC.

Parameters:

InstancePtr is a pointer to the **XGmac** instance that just interrupted.

Returns:

None.

Note:

None.

```
XGmac_Config* ( Xuint16  
XGmac_LookupConfig DeviceId )
```

Lookup the device configuration based on the unique device ID. The table EmacConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

```
XStatus ( XGmac *  
XGmac_MgtRead InstancePtr,  
int PhyAddress,  
int Register,  
Xuint16 * DataPtr  
)
```

Read a PHY register through the GMII Management Control mechanism.

Parameters:

- InstancePtr* is a pointer to the **XGmac** instance to be worked on.
- PhyAddress* is the address of the PHY to be accessed. Valid range is 0 to 31.
- Register* is the register in the PHY to be accessed. Valid range is 0 to 31.
- DataPtr* is an output parameter that will contain the contents of the register.

Returns:

- o XST_SUCCESS if the PHY register was successfully read and its contents were placed in DataPtr.
- o XST_NO_FEATURE if GMII is not present with this GEMAC instance.
- o XST_DEVICE_BUSY if another GMII read/write operation is already in progress.
- o XST_FAILURE if an GMII read error is detected

Note:

This function blocks until the read operation has completed. If there is a HW problem then this function may not return.

```
XStatus ( XGmac *
XGmac_MgtWrite InstancePtr,
int PhyAddress,
int Register,
Xuint16 Data
)
```

Write to a PHY register through the GMII Management Control mechanism.

Parameters:

- InstancePtr* is a pointer to the **XGmac** instance to be worked on.
- PhyAddress* is the address of the PHY to be accessed.
- Register* is the register in the PHY to be accessed.
- Data* is the what will be written to the register.

Returns:

- o XST_SUCCESS if the PHY register was successfully read and its contents are placed in DataPtr.
- o XST_DEVICE_BUSY if another GMII read/write operation is already in progress.
- o XST_NO_FEATURE if GMII is not present with this GEMAC instance.

Note:

This function blocks until the write operation has completed. If there is a HW problem then this function may not return.

```

XStatus                ( XGemac *
XGemac_MulticastAdd    InstancePtr,
                          Xuint8   Location,
                          Xuint8 * AddressPtr
                          )

```

Set a discrete multicast address entry in the CAM lookup table. There are up to XGE_CAM_MAX_ADDRESSES in this table. The GEMAC must be stopped (see [XGemac_Stop\(\)](#)) before multicast addresses can be modified.

Once set, the multicast address cannot be retrieved. It can be disabled by clearing it using [XGemac_MulticastClear\(\)](#).

Parameters:

InstancePtr is a pointer to the [XGemac](#) instance to be worked on.

Location indicates which of the entries is to be updated. Valid range is 0 to XGE_CAM_MAX_ADDRESSES-1.

AddressPtr is the multicast address to set.

Returns:

- o XST_SUCCESS if the multicast address table was updated successfully
- o XST_NO_FEATURE if this feature is not included in HW
- o XST_DEVICE_IS_STARTED if the device has not yet been stopped
- o XST_INVALID_PARAM if the Location parameter is greater than XGE_CAM_MAX_ADDRESSES

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

```

XStatus                ( XGemac *
XGemac_MulticastClear InstancePtr,
                          Xuint8   Location
                          )

```

Clear a discrete multicast address entry in the CAM lookup table. There are up to XGE_CAM_MAX_ADDRESSES in this table. The GEMAC must be stopped (see [XGemac_Stop\(\)](#)) before multicast addresses can be cleared.

The entry is cleared by writing an address of 00:00:00:00:00:00 to its location.

Parameters:

InstancePtr is a pointer to the [XGemac](#) instance to be worked on.

Location indicates which of the entries is to be cleared. Valid range is 0 to XGE_CAM_MAX_ADDRESSES-1.

Returns:

- o XST_SUCCESS if the multicast address table was updated successfully
- o XST_NO_FEATURE if this feature is not included in HW
- o XST_DEVICE_IS_STARTED if the device has not yet been stopped
- o XST_INVALID_PARAM if the Location parameter is greater than XGE_CAM_MAX_ADDRESSES

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and

SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

```
XStatus ( XGmac * InstancePtr,  
XGmac_PollRecv Xuint8 * BufPtr,  
 Xuint32 * ByteCountPtr  
 )
```

Receive an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver receives the frame directly from the MAC's packet FIFO. This is a non-blocking receive, in that if there is no frame ready to be received at the device, the function returns with an error. The MAC's error status is not checked, so statistics are not updated for polled receive. The buffer into which the frame will be received must be properly aligned (see [xgemac.h](#)).

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BufPtr is a pointer to an aligned buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_NO_DATA if there is no frame to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.

Note:

Input buffer must be big enough to hold the largest Ethernet frame. Buffer must also be 32-bit aligned.

```
XStatus ( XGmac * InstancePtr,  
XGmac_PollSend Xuint8 * BufPtr,  
 Xuint32 ByteCount  
 )
```

Send an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver writes the frame directly to the MAC's packet FIFO, then enters a loop checking the device status for completion or error. Statistics are updated if an error occurs. The buffer to be sent must be properly aligned (see [xgemac.h](#)).

It is assumed that the upper layer software supplies a correctly formatted and aligned Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BufPtr is a pointer to a buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_FIFO_NO_ROOM if there is no room in the GEMAC's length FIFO for this frame
- XST_FIFO_ERROR if the FIFO was overrun or underrun. This error is critical and requires the caller to reset the device.
- XST_EMAC_COLLISION if the send failed due to excess deferral or late collision

Note:

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PollSend thread. On a 1 Gbit (1000Mbps) MAC, it takes about 12.1 usecs to transmit a maximum size Ethernet frame.

```
void ( XGmac * )  
XGmac_Reset InstancePtr
```

Reset the Ethernet MAC. This is a graceful reset in that the device is stopped first. Resets the DMA channels, the FIFOs, the transmitter, and the receiver. All options are placed in their default state. Any frames in the scatter- gather descriptor lists will remain in the lists. The side effect of doing this is that after a reset and following a restart of the device, frames that were in the list before the reset may be transmitted or received.

The upper layer software is responsible for re-configuring (if necessary) and restarting the MAC after the reset. Note also that driver statistics are not cleared on reset. It is up to the upper layer software to clear the statistics if needed.

When a reset is required due to an internal error, the driver notifies the upper layer software of this need through the ErrorHandler callback and specific status codes. The upper layer software is responsible for calling this Reset function and then re-configuring the device.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( XGmac * )  
XGmac_SelfTest InstancePtr
```

Performs a self-test on the Ethernet device. The test includes:

- Run self-test on DMA channel, FIFO, and IPIF components
- Reset the Ethernet device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run, and starting it when ready to send and receive frames.

It should be noted that data caching must be disabled when this function is called because the DMA self-test uses two local buffers (on the stack) for the transfer test.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

XST_SUCCESS	Self-test was successful
XST_PFIPO_BAD_REG_VALUE	FIFO failed register self-test
XST_DMA_TRANSFER_ERROR	DMA failed data transfer self-test
XST_DMA_RESET_REGISTER_ERROR	DMA control register value was incorrect after a reset
XST_REGISTER_ERROR	Ethernet failed register reset test
XST_LOOPBACK_ERROR	Internal loopback failed
XST_IPIF_REG_WIDTH_ERROR	An invalid register width was passed into the function
XST_IPIF_RESET_REGISTER_ERROR	The value of a register at reset was invalid
XST_IPIF_DEVICE_STATUS_ERROR	A write to the device status register did not read back correctly
XST_IPIF_DEVICE_ACK_ERROR	A bit in the device status register did not reset when acked
XST_IPIF_DEVICE_ENABLE_ERROR	The device interrupt enable register was not updated correctly by the hardware when other registers were written to
XST_IPIF_IP_STATUS_ERROR	A write to the IP interrupt status register did not read back correctly
XST_IPIF_IP_ACK_ERROR	One or more bits in the IP status register did not reset when acked
XST_IPIF_IP_ENABLE_ERROR	The IP interrupt enable register was not updated correctly when other registers were written to

Note:

This function makes use of options-related functions, and the **XGemac_PollSend()** and **XGemac_PollRecv()** functions.

Because this test uses the PollSend function for its loopback testing, there is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the self-test thread.

```

XStatus ( XGmac *
XGmac_SendPause InstancePtr,
Xuint16 PausePeriod
)

```

Send a pause packet. When called, the GEMAC hardware will initiate transmission of an automatically formed pause packet. This action will not disrupt any frame transmission in progress but will take priority over any pending frame transmission. The pause frame will be sent even if the transmitter is in the paused state.

For this function to have any effect, the XGE_FLOW_CONTROL option must be set (see XGmac_SetOptions)).

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

PausePeriod is the amount of time to pause. The LSB is 512 bit times.

Returns:

- o XST_SUCCESS if the pause frame transmission mechanism was successfully started.
- o XST_DEVICE_IS_STARTED if the device has not been stopped

```

void ( XGmac * InstancePtr,
XGmac_SetErrorHandler
void * CallbackRef,
XGmac_ErrorHandler FuncPtr
)

```

Set the callback function for handling asynchronous errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- XST_DMA_ERROR indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- XST_FIFO_ERROR indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- XST_RESET_ERROR indicates an unrecoverable MAC error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- XST_DMA_SG_NO_LIST indicates an attempt was made to access a scatter-gather DMA list that has not yet been created.
- XST_DMA_SG_LIST_EMPTY indicates the driver tried to get a descriptor from the receive descriptor list, but the list was empty.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

CallbackRef is reference data to be passed back to the callback function. Its value is arbitrary and not used by the driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void
XGemac_SetFifoRecvHandler ( XGemac * InstancePtr,
                             void * CallbackRef,
                             XGemac_FifoHandler FuncPtr
                             )
```

Set the callback function for handling confirmation of transmitted frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called by the driver once per frame sent. The callback is responsible for freeing the transmitted buffer if necessary.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

CallbackRef is reference data to be passed back to the callback function. Its value is arbitrary and not used by the driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void
XGemac_SetFifoSendHandler ( XGemac * InstancePtr,
                             void * CallbackRef,
                             XGemac_FifoHandler FuncPtr
                             )
```

Set the callback function for handling received frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called once per frame received. During the callback, the upper layer software should call **XGmac_FifoRecv()** to retrieve the received frame.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

CallBackRef is reference data to be passed back to the callback function. Its value is arbitrary and not used by the driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus ( XGmac *  
XGmac_SetInterframeGap InstancePtr,  
 Xuint8 Ifg  
 )
```

Set the Interframe Gap (IFG), which is the time the MAC delays between transmitting frames.

The device must be stopped before setting the interframe gap.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Ifg is the interframe gap to set, the LSB is 8 bit times.

Returns:

- o XST_SUCCESS if the interframe gap was set successfully
- o XST_DEVICE_IS_STARTED if the device has not been stopped

Note:

None.

```
XStatus ( XGmac *  
XGmac_SetMacAddress InstancePtr,  
 Xuint8 * AddressPtr  
 )
```

Set the MAC address for this driver/device. The address is a 48-bit value. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

AddressPtr is a pointer to a 6-byte MAC address.

Returns:

- o XST_SUCCESS if the MAC address was set successfully
- o XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```
XStatus ( XGemac *  
XGemac_SetOptions InstancePtr,  
 Xuint32 OptionsFlag  
 )
```

Set Ethernet driver/device options. The options (XGE_*_OPTION) constants can be OR'd together to set/clear multiple options. A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off.

The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

OptionsFlag is a bit-mask representing the Ethernet options to turn on or off. See [xgemac.h](#) for a description of the available options.

Returns:

- o XST_SUCCESS if the options were set successfully
- o XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

```
XStatus ( XGemac *  
XGemac_SetPktThreshold InstancePtr,  
 Xuint32 Direction,  
 Xuint8 Threshold  
 )
```

Set the scatter-gather DMA packet count threshold for this device. See [xgemac.h](#) for more discussion of interrupt coalescing features.

The device must be stopped before setting the threshold.

Parameters:

- InstancePtr* is a pointer to the **XGemac** instance to be worked on.
- Direction* indicates the channel, XGE_SEND or XGE_RECV, to set.
- Threshold* is the value of the packet threshold count used during interrupt coalescing. Valid range is 0 - 255. A value of 0 disables the use of packet threshold by the hardware.

Returns:

- o XST_SUCCESS if the threshold was successfully set
- o XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- o XST_DEVICE_IS_STARTED if the device has not been stopped
- o XST_DMA_SG_COUNT_EXCEEDED if the threshold must be equal to or less than the number of descriptors in the list
- o XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None

```
XStatus ( XGemac *  
XGemac_SetPktWaitBound InstancePtr,  
                        Xuint32 Direction,  
                        Xuint32 TimerValue  
                        )
```

Set the scatter-gather DMA packet wait bound timer for this device. See [xgemac.h](#) for more discussion of interrupt coalescing features.

Parameters:

- InstancePtr* is a pointer to the **XGemac** instance to be worked on.
- Direction* indicates the channel, XGE_SEND or XGE_RECV, from which the threshold register is read.
- TimerValue* is the value of the packet wait bound timer to set. Units are in milliseconds. A value of 0 means the timer is disabled.

Returns:

- o XST_SUCCESS if the packet wait bound was set successfully
- o XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- o XST_DEVICE_IS_STARTED if the device has not been stopped
- o XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```

void
XGemac_SetSgRecvHandler
    ( XGemac *
      InstancePtr,
      void *
      CallbackRef,
      XGemac_SgHandler FuncPtr
    )

```

Set the callback function for handling received frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame received. The head of a descriptor list is passed in along with the number of descriptors in the list. Before leaving the callback, the upper layer software should attach a new buffer to each descriptor in the list.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

CallbackRef is reference data to be passed back to the callback function. Its value is arbitrary and not used by the driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```

XStatus
XGemac_SetSgRecvSpace
    ( XGemac *
      InstancePtr,
      Xuint32 *
      MemoryPtr,
      Xuint32
      ByteCount
    )

```

Give the driver memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The **xgemac.h** file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be properly aligned (see **xgemac.h**).

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

MemoryPtr is a pointer to the beginning of the memory space.

ByteCount is the length, in bytes, of the memory space.

Returns:

- o XST_SUCCESS if the space was initialized successfully
- o XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA

- o XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XGmac_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```
void XGmac_SetSgSendHandler ( XGmac * InstancePtr, void * CallbackRef, XGmac_SgHandler FuncPtr )
```

Set the callback function for handling confirmation of transmitted frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame sent. The head of a descriptor list is passed in along with the number of descriptors in the list. The callback is responsible for freeing buffers attached to these descriptors.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

CallbackRef is reference data to be passed back to the callback function. Its value is arbitrary and not used by the driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XGmac_SetSgSendSpace ( XGmac * InstancePtr, Xuint32 * MemoryPtr, Xuint32 ByteCount )
```

Give the driver memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The [xgemac.h](#) file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be properly aligned (see [xgemac.h](#)).

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

MemoryPtr is a pointer to the beginning of the memory space.

ByteCount is the length, in bytes, of the memory space.

Returns:

- o XST_SUCCESS if the space was initialized successfully
- o XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- o XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the [XGmac_Initialize\(\)](#) function because the DMA channel components must be initialized before the memory space is set.

```
XStatus ( XGmac *  
XGmac_SgRecv InstancePtr,  
 XBufDescriptor * BdPtr  
 )
```

Add a descriptor, with an attached empty buffer, into the receive descriptor list. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of frames to replace filled buffers with empty buffers. This function can be called when the device is started or stopped. Note that it does start the scatter-gather DMA engine. Although this is not necessary during initialization, it is not a problem during initialization because the MAC receiver is not yet started.

The buffer attached to the descriptor and the descriptor itself must be properly aligned (see [xgemac.h](#)).

Notification of received frames are done asynchronously through the receive callback function.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- o XST_SUCCESS if a descriptor was successfully returned to the driver
- o XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- o XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- o XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- o XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

```

XStatus          ( XGemac *
XGemac_SgSend   InstancePtr,
                  XBufDescriptor * BdPtr,
                  int                Delay
                  )

```

Send an Ethernet frame using scatter-gather DMA. The caller attaches the frame to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire Ethernet frame may or may not be contained within one descriptor. This function simply inserts the descriptor into the scatter-gather engine's transmit list. The caller is responsible for providing mutual exclusion to guarantee that a frame is contiguous in the transmit list. The descriptor must be properly aligned (see [xgemac.h](#)).

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the frame, the inserts are committed, which means the descriptors for this frame are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

This call is non-blocking. Notification of error or successful transmission is done asynchronously through the send or error callback function.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Delay indicates whether to start the scatter-gather DMA channel immediately, or whether to wait. This allows the user to queue up a list of more than one descriptor before starting the transmission of the packets. Use XEM_SGDMA_NODELAY or XEM_SGDMA_DELAY, defined in [xgemac.h](#), as the value of this argument. If the user chooses to delay and build a list, the user must call this function with the XEM_SGDMA_NODELAY option or call **XGemac_Start()** to kick off the transmissions.

Returns:

- o XST_SUCCESS if the buffer was successfull sent
- o XST_DEVICE_IS_STOPPED if the Ethernet MAC has not been started yet
- o XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- o XST_DMA_SG_LIST_FULL if the descriptor list for the DMA channel is full
- o XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- o XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

```

XStatus          ( XGemac *
XGemac_Start    InstancePtr
                  )

```

Start the Ethernet controller as follows:

- Disable global device interrupt
- If polled mode, then enable transmitter/receiver and return
- If SG DMA interrupt driven mode
 - Start the send and receive DMA channels
 - Set DMA specific interrupt enable registers appropriately
- If FIFO interrupt driven mode
 - Set FIFO specific interrupt enable registers appropriately
- Enable transmitter/receiver and return

The PHY is enabled after driver initialization. We assume the upper layer software has configured it and the GEMAC appropriately before this function is called.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully
- XST_NO_CALLBACK if a callback function has not yet been registered using the SetxxxHandler function. This is required if in interrupt mode.
- XST_DEVICE_IS_STARTED if the device is already started
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.
- XST_DMA_SG_LIST_EMPTY if configured for scatter-gather DMA but no receive buffer descriptors have been initialized.

Note:

The driver tries to match the hardware configuration. So if the hardware is configured with scatter-gather DMA, the driver expects to start the scatter-gather channels and expects that the user has previously set up the buffer descriptor lists. If the user expects to use the driver in a mode different than how the hardware is configured, the user should modify the configuration table to reflect the mode to be used.

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

```
XStatus ( XGemac * )  
XGemac_Stop InstancePtr
```

Stop the Ethernet MAC as follows:

- Disable all interrupts from this device
- If the device is configured with scatter-gather DMA, stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver

Device options currently in effect are not changed.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

gemac/v1_00_f/src/xgemac_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of GEMAC devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	01/13/03	First release

```
#include "xgemac.h"  
#include "xparameters.h"
```

Variables

[XGmac_Config](#) [XGmac_ConfigTable](#) [XPAR_XGEMAC_NUM_INSTANCES]

Variable Documentation

[XGmac_Config](#) [XGmac_ConfigTable](#)
[XPAR_XGEMAC_NUM_INSTANCES]

This table contains configuration information for each GEMAC device in the system.

XGmac_Config Struct Reference

```
#include <xgemac.h>
```

Detailed Description

This typedef contains configuration information for a device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xuint8 IpIfDmaConfig](#)

[Xboolean HasGmii](#)

[Xboolean HasCounters](#)

Field Documentation

[Xuint32 XGmac_Config::](#)

BaseAddress

Register base
address

[Xuint16 XGmac_Config::](#)

DeviceId

Unique ID of
device

Xboolean XGmac_Config::

HasCounters

Does device have HW statistic counters

Xboolean XGmac_Config::

HasGmii

Does device support GMII?

Xuint8 XGmac_Config::

IpIfDmaConfig

IPIF/DMA hardware configuration

The documentation for this struct was generated from the following file:

- [gemac/v1_00_f/src/xgemac.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

gemac/v1_00_f/src/xgemac.c File Reference

Detailed Description

The **XGemac** driver. Functions in this file are the minimum required functions for this driver. See [xgemac.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Who	Date	Changes
ecm	01/13/03	First release
ecm	03/25/03	Revision update
rmm	05/28/03	DMA mods
rmm	09/12/03	Cleanup
rmm	11/14/03	Implemented XGE_NO_SGEND_INT_OPTION
rmm	03/03/04	Restructured Start() and Stop() functions.

```
#include "xbasic_types.h"  
#include "xgemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"  
#include "xenv.h"
```

Functions

XStatus XGemac_Initialize (**XGemac** *InstancePtr, **Xuint16** DeviceId)
XStatus XGemac_Start (**XGemac** *InstancePtr)
XStatus XGemac_Stop (**XGemac** *InstancePtr)
void **XGemac_Reset** (**XGemac** *InstancePtr)
XStatus XGemac_SetMacAddress (**XGemac** *InstancePtr, **Xuint8** *AddressPtr)

void **XGmac_GetMacAddress** (**XGmac** *InstancePtr, **Xuint8** *BufferPtr)

XGmac_Config * **XGmac_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
void  
XGmac_GetMacAddress ( XGmac *  
                        InstancePtr,  
                        Xuint8 * BufferPtr  
                      )
```

Get the MAC address for this driver/device.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BufferPtr is an output parameter, and is a pointer to a buffer into which the current MAC address will be copied. The buffer must be at least 6 bytes.

Returns:

None.

Note:

None.

```
XStatus  
XGmac_Initialize ( XGmac *  
                    InstancePtr,  
                    Xuint16 DeviceId  
                  )
```

Initialize a specific **XGmac** instance/driver. The initialization entails:

- Clearing memory occupied by the **XGmac** structure
- Initialize fields of the **XGmac** structure
- Clear the Ethernet statistics for this device
- Initialize the IPIF component with its register base address
- Configure the FIFO components with their register base addresses.
- If the device is configured with DMA, configure the DMA channel components with their register base addresses. At some later time, memory pools for the scatter-gather descriptor lists are to be passed to the driver.
- Reset the Ethernet MAC

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XGemac** instance. Passing in a device id associates the generic **XGemac** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None.

```
XGemac_Config* ( Xuint16 )  
XGemac_LookupConfig DeviceId
```

Lookup the device configuration based on the unique device ID. The table EmacConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

```
void ( XGemac * )  
XGemac_Reset InstancePtr
```

Reset the Ethernet MAC. This is a graceful reset in that the device is stopped first. Resets the DMA channels, the FIFOs, the transmitter, and the receiver. All options are placed in their default state. Any frames in the scatter- gather descriptor lists will remain in the lists. The side effect of doing this is that after a reset and following a restart of the device, frames that were in the list before the reset may be transmitted or received.

The upper layer software is responsible for re-configuring (if necessary) and restarting the MAC after the reset. Note also that driver statistics are not cleared on reset. It is up to the upper layer software to clear the statistics if needed.

When a reset is required due to an internal error, the driver notifies the upper layer software of this need through the ErrorHandler callback and specific status codes. The upper layer software is responsible for calling this Reset function and then re-configuring the device.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus XGemac_SetMacAddress ( XGemac *  
                                InstancePtr,  
                                Xuint8 * AddressPtr  
                                )
```

Set the MAC address for this driver/device. The address is a 48-bit value. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

AddressPtr is a pointer to a 6-byte MAC address.

Returns:

- XST_SUCCESS if the MAC address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

XStatus (**XGmac ***)
XGmac_Start *InstancePtr*

Start the Ethernet controller as follows:

- Disable global device interrupt
- If polled mode, then enable transmitter/receiver and return
- If SG DMA interrupt driven mode
 - Start the send and receive DMA channels
 - Set DMA specific interrupt enable registers appropriately
- If FIFO interrupt driven mode
 - Set FIFO specific interrupt enable registers appropriately
- Enable transmitter/receiver and return

The PHY is enabled after driver initialization. We assume the upper layer software has configured it and the GEMAC appropriately before this function is called.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully
- XST_NO_CALLBACK if a callback function has not yet been registered using the SetxxxHandler function. This is required if in interrupt mode.
- XST_DEVICE_IS_STARTED if the device is already started
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.
- XST_DMA_SG_LIST_EMPTY if configured for scatter-gather DMA but no receive buffer descriptors have been initialized.

Note:

The driver tries to match the hardware configuration. So if the hardware is configured with scatter-gather DMA, the driver expects to start the scatter-gather channels and expects that the user has previously set up the buffer descriptor lists. If the user expects to use the driver in a mode different than how the hardware is configured, the user should modify the configuration table to reflect the mode to be used.

This function makes use of internal resources that are shared between the Start, Stop, and

SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

```
XStatus ( XGmac * )  
XGmac_Stop InstancePtr
```

Stop the Ethernet MAC as follows:

- Disable all interrupts from this device
- If the device is configured with scatter-gather DMA, stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver

Device options currently in effect are not changed.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XGmac Struct Reference

```
#include <xgmac.h>
```

Detailed Description

The XGmac driver instance data. The user is required to allocate a variable of this type for every GEMAC device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [gemac/v1_00_f/src/xgmac.h](#)
-

gemac/v1_00_f/src/xgemac_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between **XGemac** components. The identifiers in this file are not intended for use external to the driver.

MODIFICATION HISTORY:

Who	Date	Changes
-----	------	---------

ecm	01/13/03	First release
-----	----------	---------------

ecm	03/25/03	Revision update
-----	----------	-----------------

rmm	05/28/03	Revision update
-----	----------	-----------------

rmm	09/23/03	Removed use of XGE_EIR_RECV_DFIFO_OVER_MASK, XGE_EIR_RECV_MISSED_FRAME_MASK, XGE_EIR_RECV_COLLISION_MASK, XGE_EIR_RECV_SHORT_ERROR_MASK,
-----	----------	--

XGE_EIR_XMIT_ERROR_MASK.

Redefined default interrupt masks.

```
#include "xgemac.h"
```

```
#include "xgemac_1.h"
```

Variables

[XGemac_Config](#) [XGemac_ConfigTable](#) []

Variable Documentation

XGmac_Config XGmac_ConfigTable ()

[]

This table contains configuration information for each GEMAC device in the system.

Generated on 24 Jun 2004 for Xilinx Device Drivers

gemac/v1_00_f/src/xgemac_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xgemac.h](#).

MODIFICATION HISTORY:

Who	Date	Changes
ecm	01/13/03	First release
ecm	03/25/03	Revision update
rmm	05/28/03	Removed tabs characters in file, added auto negotiate bit to ECR constants.
rmm	09/15/03	Moved XGE_MAX_IFG definition here from xgemac_options.c and changed its value to 255. Added XMIT jumbo frame bitmask to ECR register bitmasks. Removed unused and added new XGE_EIR bitmasks. Removed XGE_TEDC_OFFSET and XGE_SLEC_OFFSET register definitions. Added XGE_RSR_OFFSET register definition.

```
#include "xbasic_types.h"
```

```
#include "xio.h"
```

GEMAC Core Register Offsets

Each register is 32 bits wide

```
#define XGE_EMIR_OFFSET
#define XGE_ECR_OFFSET
#define XGE_IFGP_OFFSET
#define XGE_SAH_OFFSET
#define XGE_SAL_OFFSET
#define XGE_MGTCR_OFFSET
#define XGE_MGTDR_OFFSET
#define XGE_RPLR_OFFSET
#define XGE_TPLR_OFFSET
#define XGE_TSR_OFFSET
#define XGE_TPPR_OFFSET
#define XGE_CEAH_OFFSET
#define XGE_CEAL_OFFSET
#define XGE_RSR_OFFSET
```

GEMAC IPIF Register Offsets

Each register is 32 bits wide

```
#define XGE_ISR_OFFSET
#define XGE_DMA_OFFSET
#define XGE_DMA_SEND_OFFSET
#define XGE_DMA_RECV_OFFSET
#define XGE_PFIFO_OFFSET
#define XGE_PFIFO_TXREG_OFFSET
#define XGE_PFIFO_RXREG_OFFSET
#define XGE_PFIFO_TXDATA_OFFSET
#define XGE_PFIFO_RXDATA_OFFSET
```

GEMAC Counter Register Offsets

Each register is 64 bits wide

```
#define XGE_STAT_REG_OFFSET
#define XGE_STAT_RXOK_OFFSET
#define XGE_STAT_FCSERR_OFFSET
#define XGE_STAT_BFRXOK_OFFSET
#define XGE_STAT_MCRXOK_OFFSET
#define XGE_STAT_64RXOK_OFFSET
```

```
#define XGE_STAT_127RXOK_OFFSET
#define XGE_STAT_255RXOK_OFFSET
#define XGE_STAT_511RXOK_OFFSET
#define XGE_STAT_1023RXOK_OFFSET
#define XGE_STAT_MAXRXOK_OFFSET
#define XGE_STAT_CFRXOK_OFFSET
#define XGE_STAT_LTERROR_OFFSET
#define XGE_STAT_VLANRXOK_OFFSET
#define XGE_STAT_PFRXOK_OFFSET
#define XGE_STAT_CFUNSUP_OFFSET
#define XGE_STAT_OFRXOK_OFFSET
#define XGE_STAT_UFRX_OFFSET
#define XGE_STAT_FRAGRX_OFFSET
#define XGE_STAT_RXBYTES_OFFSET
#define XGE_STAT_TXBYTES_OFFSET
#define XGE_STAT_TXOK_OFFSET
#define XGE_STAT_BFTXOK_OFFSET
#define XGE_STAT_MFTXOK_OFFSET
#define XGE_STAT_TXURUNERR_OFFSET
#define XGE_STAT_CFTXOK_OFFSET
#define XGE_STAT_64TXOK_OFFSET
#define XGE_STAT_127TXOK_OFFSET
#define XGE_STAT_255TXOK_OFFSET
#define XGE_STAT_511TXOK_OFFSET
#define XGE_STAT_1023TXOK_OFFSET
#define XGE_STAT_MAXTXOK_OFFSET
#define XGE_STAT_VLANTXOK_OFFSET
#define XGE_STAT_PFTXOK_OFFSET
#define XGE_STAT_OFTXOK_OFFSET
#define XGE_STAT_SCOLL_OFFSET
#define XGE_STAT_MCOLL_OFFSET
#define XGE_STAT_DEFERRED_OFFSET
#define XGE_STAT_LATECOLL_OFFSET
#define XGE_STAT_TXABORTED_OFFSET
#define XGE_STAT_CARRIERERR_OFFSET
#define XGE_STAT_EXCESSDEF_OFFSET
```

Macro functions

```
#define XGemac_mReadReg(BaseAddress, RegOffset)
#define XGemac_mWriteReg(BaseAddress, RegOffset, Data)
#define XGemac_mSetControlReg(BaseAddress, Mask)
#define XGemac_mSetMacAddress(BaseAddress, AddressPtr)
#define XGemac_mEnable(BaseAddress)
#define XGemac_mDisable(BaseAddress)
#define XGemac_mIsTxDone(BaseAddress)
#define XGemac_mIsRxEmpty(BaseAddress)
#define XGemac_mPhyReset(BaseAddress)
```

Define Documentation

```
#define  
XGE_CEAH_OFFSET
```

CAM Entry Address
High

```
#define  
XGE_CEAL_OFFSET
```

CAM Entry Address
Low

```
#define  
XGE_DMA_OFFSET
```

DMA base
offset

```
#define  
XGE_DMA_RECV_OFFSET
```

DMA recv
channel

```
#define  
XGE_DMA_SEND_OFFSET
```

DMA send
channel

```
#define  
XGE_ECR_OFFSET
```

```
MAC  
Control
```

```
#define  
XGE_EMIR_OFFSET
```

```
GEMAC Module  
ID
```

```
#define  
XGE_IFGP_OFFSET
```

```
Interframe  
Gap
```

```
#define  
XGE_ISR_OFFSET
```

```
Interrupt  
status
```

```
#define  
XGE_MGTCR_OFFSET
```

```
MII mgmt  
control
```

```
#define  
XGE_MGTDR_OFFSET
```

```
MII mgmt  
data
```

```
#define  
XGE_PFIFO_OFFSET
```

```
FIFO base  
offset
```

```
#define  
XGE_PFIPO_RXDATA_OFFSET
```

```
Rx  
keyhole
```

```
#define  
XGE_PFIPO_RXREG_OFFSET
```

```
Rx  
registers
```

```
#define  
XGE_PFIPO_TXDATA_OFFSET
```

```
Tx  
keyhole
```

```
#define  
XGE_PFIPO_TXREG_OFFSET
```

```
Tx  
registers
```

```
#define  
XGE_RPLR_OFFSET
```

```
Rx packet  
length
```

```
#define  
XGE_RSR_OFFSET
```

```
Receive  
status
```

```
#define  
XGE_SAH_OFFSET
```

```
Station addr,  
high
```

```
#define  
XGE_SAL_OFFSET
```

Station addr,
low

```
#define  
XGE_STAT_1023RXOK_OFFSET
```

512-1023 byte frames RX'd
ok

```
#define  
XGE_STAT_1023TXOK_OFFSET
```

512-1023 byte frames TX'd
ok

```
#define  
XGE_STAT_127RXOK_OFFSET
```

65-127 byte frames RX'd
ok

```
#define  
XGE_STAT_127TXOK_OFFSET
```

65-127 byte frames TX'd
ok

```
#define  
XGE_STAT_255RXOK_OFFSET
```

128-255 byte frames RX'd
ok

```
#define  
XGE_STAT_255TXOK_OFFSET
```

128-255 byte frames TX'd
ok

```
#define  
XGE_STAT_511RXOK_OFFSET
```

256-511 byte frames RX'd
ok

```
#define  
XGE_STAT_511TXOK_OFFSET
```

```
256-511 byte frames TX'd  
ok
```

```
#define  
XGE_STAT_64RXOK_OFFSET
```

```
64 byte frames RX'd  
ok
```

```
#define  
XGE_STAT_64TXOK_OFFSET
```

```
64 byte frames TX'd  
ok
```

```
#define  
XGE_STAT_BFRXOK_OFFSET
```

```
Broadcast Frames RX'd  
ok
```

```
#define  
XGE_STAT_BFTXOK_OFFSET
```

```
Broadcast Frames TX'd  
ok
```

```
#define  
XGE_STAT_CARRIERERR_OFFSET
```

```
Carrier sense  
errors
```

```
#define  
XGE_STAT_CFRXOK_OFFSET
```

```
Control Frames RX'd  
ok
```

#define

XGE_STAT_CFTXOK_OFFSET

Control Frames TX'd
ok

#define

XGE_STAT_CFUNSUP_OFFSET

Control Frames with unsupported opcode
RX's

#define

XGE_STAT_DEFERRED_OFFSET

Deferred
Frames

#define

XGE_STAT_EXCESSDEF_OFFSET

Excess Deferral
error

#define

XGE_STAT_FCSERR_OFFSET

RX FCS
error

#define

XGE_STAT_FRAGRX_OFFSET

Fragment Frames
RX'd

#define

XGE_STAT_LATECOLL_OFFSET

Late Collision
Frames

#define

XGE_STAT_LTERROR_OFFSET

length/type out of
range

```
#define  
XGE_STAT_MAXRXOK_OFFSET
```

1024-max byte frames RX'd
ok

```
#define  
XGE_STAT_MAXTXOK_OFFSET
```

1024-Max byte frames TX'd
ok

```
#define  
XGE_STAT_MCOLL_OFFSET
```

Multiple Collision
Frames

```
#define  
XGE_STAT_MCRXOK_OFFSET
```

Multicast Frames RX'd
ok

```
#define  
XGE_STAT_MFTXOK_OFFSET
```

Multicast Frames TX'd
ok

```
#define  
XGE_STAT_OFRXOK_OFFSET
```

Oversize Frames RX'd
ok

```
#define  
XGE_STAT_OFTXOK_OFFSET
```

Oversize Frames TX'd
ok

#define
XGE_STAT_PFRXOK_OFFSET

Pause Frames RX'd
ok

#define
XGE_STAT_PFTXOK_OFFSET

Pause Frames TX'd
ok

#define
XGE_STAT_REG_OFFSET

Offset of the MAC Statistics registers from the IPIF base
address

#define
XGE_STAT_RXBYTES_OFFSET

RX Byte
Count

#define
XGE_STAT_RXOK_OFFSET

Frames RX'd
ok

#define
XGE_STAT_SCOLL_OFFSET

Single Collision
Frames

#define
XGE_STAT_TXABORTED_OFFSET

Frames aborted due to excess
collisions

```
#define  
XGE_STAT_TXBYTES_OFFSET
```

```
TX Byte  
Count
```

```
#define  
XGE_STAT_TXOK_OFFSET
```

```
Frames TX'd  
ok
```

```
#define  
XGE_STAT_TXURUNERR_OFFSET
```

```
TX Underrun  
error
```

```
#define  
XGE_STAT_UFRX_OFFSET
```

```
Undersize Frames  
RX'd
```

```
#define  
XGE_STAT_VLANRXOK_OFFSET
```

```
VLAN Frames RX'd  
ok
```

```
#define  
XGE_STAT_VLANTXOK_OFFSET
```

```
VLAN Frames TX'd  
ok
```

```
#define  
XGE_TPLR_OFFSET
```

```
Tx packet  
length
```

```
#define  
XGE_TPPR_OFFSET
```

Tx Pause
Pkt

```
#define  
XGE_TSR_OFFSET
```

Tx
status

```
#define XGemac_mDisable( BaseAddress )
```

Disable the transmitter and receiver. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XGemac_mEnable( BaseAddress )
```

Enable the transmitter and receiver. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XGemac_mIsRxEmpty( BaseAddress )
```

Check to see if the receive FIFO is empty.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is empty, or XFALSE if it is not.

Note:

None.

```
#define XGmac_mIsTxDone( BaseAddress )
```

Check to see if the transmission is complete.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is done, or XFALSE if it is not.

Note:

None.

```
#define XGmac_mPhyReset( BaseAddress )
```

Reset MII compliant PHY

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XGemac_mReadReg( BaseAddress,  
                        RegOffset  )
```

Read the given register.

Parameters:

BaseAddress is the base address of the device

RegOffset is the register offset to be read

Returns:

The 32-bit value of the register

Note:

None.

```
#define XGemac_mSetControlReg( BaseAddress,  
                              Mask      )
```

Set the contents of the control register. Use the XGE_ECR_* constants defined above to create the bit-mask to be written to the register.

Parameters:

BaseAddress is the base address of the device

Mask is the 16-bit value to write to the control register

Returns:

None.

Note:

None.

```
#define XGemac_mSetMacAddress( BaseAddress,  
                              AddressPtr )
```

Set the station address of the GEMAC device.

Parameters:

BaseAddress is the base address of the device

AddressPtr is a pointer to a 6-byte MAC address

Returns:

None.

Note:

None.

```
#define XGemac_mWriteReg( BaseAddress,  
                          RegOffset,  
                          Data      )
```

Write the given register.

Parameters:

BaseAddress is the base address of the device

RegOffset is the register offset to be written

Data is the 32-bit value to write to the register

Returns:

None.

Note:

None.

gemac/v1_00_f/src/xgemac_options.c File Reference

Detailed Description

Functions in this file handle configuration of the **XGemac** driver.

MODIFICATION HISTORY:

Who	Date	Changes
-----	-----	-----
ecm	01/13/03	First release
ecm	03/25/03	Revision update
rmm	05/28/03	Changed interframe gap API, process auto negotiate option
rmm	06/04/03	Moved XGE_MAX_IFG definition to xgemac_1.h . Fixed assert on MAX_IFG value in XGemac_SetInterframeGap() . Added support for XGE_VLAN_OPTION and XGE_JUMBO_OPTION.
Updated		description of XGemac_SetInterframeGap() .
rmm	11/14/03	Added XGE_NO_SGEND_INT_OPTION processing.

```
#include "xbasic\_types.h"  
#include "xgemac\_i.h"  
#include "xio.h"
```

Data Structures

```
struct OptionMap
```

Functions

XStatus **XGemac_SetOptions** (**XGemac** *InstancePtr, **Xuint32** OptionsFlag)
Xuint32 **XGemac_GetOptions** (**XGemac** *InstancePtr)

Function Documentation

Xuint32 (**XGemac** *)
XGemac_GetOptions *InstancePtr*

Get Ethernet driver/device options. The 32-bit value returned is a bit-mask representing the options (XGE_*_OPTION). A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

The 32-bit value of the Ethernet options. The value is a bit-mask representing all options that are currently enabled. See [xgemac.h](#) for a description of the available options.

Note:

None.

XStatus (**XGemac** *)
XGemac_SetOptions *InstancePtr,*
Xuint32 *OptionsFlag*
)

Set Ethernet driver/device options. The options (XGE_*_OPTION) constants can be OR'd together to set/clear multiple options. A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off.

The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

OptionsFlag is a bit-mask representing the Ethernet options to turn on or off. See **xgemac.h** for a description of the available options.

Returns:

- XST_SUCCESS if the options were set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

XGemac_HardStats Struct Reference

```
#include <xgemac.h>
```

Detailed Description

Statistics maintained by HW

Data Fields

Xuint64 [RecvFrames](#)

Xuint64 [RecvFcs](#)

Xuint64 [RecvBroadcast](#)

Xuint64 [RecvMulticast](#)

Xuint64 [Recv64Byte](#)

Xuint64 [Recv65_127Byte](#)

Xuint64 [Recv128_255Byte](#)

Xuint64 [Recv256_511Byte](#)

Xuint64 [Recv512_1023Byte](#)

Xuint64 [Recv1024_MaxByte](#)

Xuint64 [RecvControl](#)

Xuint64 [RecvLengthRange](#)

Xuint64 [RecvVlan](#)

Xuint64 [RecvPause](#)

Xuint64 [RecvBadOpcode](#)

Xuint64 [RecvLong](#)

Xuint64 [RecvShort](#)

Xuint64 [RecvFragment](#)

Xuint64 [RecvBytes](#)

Xuint64 [XmitBytes](#)

Xuint64 [XmitFrames](#)
Xuint64 [XmitBroadcast](#)
Xuint64 [XmitMulticast](#)
Xuint64 [XmitUnderrun](#)
Xuint64 [XmitControl](#)
Xuint64 [Xmit64Byte](#)
Xuint64 [Xmit65_127Byte](#)
Xuint64 [Xmit128_255Byte](#)
Xuint64 [Xmit256_511Byte](#)
Xuint64 [Xmit512_1023Byte](#)
Xuint64 [Xmit1024_MaxByte](#)
Xuint64 [XmitVlan](#)
Xuint64 [XmitPause](#)
Xuint64 [XmitLong](#)
Xuint64 [Xmit1stCollision](#)
Xuint64 [XmitMultiCollision](#)
Xuint64 [XmitDeferred](#)
Xuint64 [XmitLateCollision](#)
Xuint64 [XmitExcessCollision](#)
Xuint64 [XmitCarrierSense](#)
Xuint64 [XmitExcessDeferred](#)

Field Documentation

**Xuint64 XGemac_HardStats::
Recv1024_MaxByte**

Number of 1024 and larger byte frames received

**Xuint64 XGemac_HardStats::
Recv128_255Byte**

Number of 128-255 byte frames received

**Xuint64 XGemac_HardStats::
Recv256_511Byte**

Number of 256-511 byte frames
received

**Xuint64 XGemac_HardStats::
Recv512_1023Byte**

Number of 512-1023 byte frames
received

**Xuint64 XGemac_HardStats::
Recv64Byte**

Number of 64 byte frames
received

**Xuint64 XGemac_HardStats::
Recv65_127Byte**

Number of 65-127 byte frames
received

**Xuint64 XGemac_HardStats::
RecvBadOpcode**

Number of control frames received with an invalid
opcode

**Xuint64 XGemac_HardStats::
RecvBroadcast**

Number of broadcast frames
received

**Xuint64 XGemac_HardStats::
RecvBytes**

Number of bytes
received

**Xuint64 XGemac_HardStats::
RecvControl**

Number of control frames
received

Xuint64 XGemac_HardStats::**RecvFcs**

Number of received frames discarded due to FCS errors

Xuint64 XGemac_HardStats::**RecvFragment**

Number of received frames less than 64 bytes discarded due to FCS errors

Xuint64 XGemac_HardStats::**RecvFrames**

Number of frames received

Xuint64 XGemac_HardStats::**RecvLengthRange**

Number of received frames with length or type that didn't match number of bytes actually received

Xuint64 XGemac_HardStats::**RecvLong**

Number of oversized frames received

Xuint64 XGemac_HardStats::**RecvMulticast**

Number of multicast frames received

Xuint64 XGemac_HardStats::**RecvPause**

Number of pause frames received

**Xuint64 XGemac_HardStats::
RecvShort**

Number of undersized frames
received

**Xuint64 XGemac_HardStats::
RecvVlan**

Number of VLAN frames
received

**Xuint64 XGemac_HardStats::
Xmit1024_MaxByte**

Number of 1024 and larger byte frames
transmitted

**Xuint64 XGemac_HardStats::
Xmit128_255Byte**

Number of 128-255 byte frames
transmitted

**Xuint64 XGemac_HardStats::
Xmit1stCollision**

Number of frames involved in a single collision but sent
successfully

**Xuint64 XGemac_HardStats::
Xmit256_511Byte**

Number of 256-511 byte frames
transmitted

**Xuint64 XGemac_HardStats::
Xmit512_1023Byte**

Number of 512-1023 byte frames
transmitted

**Xuint64 XGemac_HardStats::
Xmit64Byte**

Number of 64 byte frames
transmitted

**Xuint64 XGemac_HardStats::
Xmit65_127Byte**

Number of 65-127 byte frames
transmitted

**Xuint64 XGemac_HardStats::
XmitBroadcast**

Number of broadcast frames
transmitted

**Xuint64 XGemac_HardStats::
XmitBytes**

Number of bytes
transmitted

**Xuint64 XGemac_HardStats::
XmitCarrierSense**

Number of frames not sent due to the GMII_CRS signal being
negated

**Xuint64 XGemac_HardStats::
XmitControl**

Number of control frames
transmitted

**Xuint64 XGemac_HardStats::
XmitDeferred**

Number of frames delayed because the medium was
busy

**Xuint64 XGemac_HardStats::
XmitExcessCollision**

Number of frames discarded due to excess
collisions

Xuint64 XGemac_HardStats::**XmitExcessDeferred**

Number of frames not sent due to excess deferral times

Xuint64 XGemac_HardStats::**XmitFrames**

Number of frames transmitted

Xuint64 XGemac_HardStats::**XmitLateColision**

Number of frames involved in a late collision but sent successfully

Xuint64 XGemac_HardStats::**XmitLong**

Number of oversized frames transmitted

Xuint64 XGemac_HardStats::**XmitMulticast**

Number of multicast frames transmitted

Xuint64 XGemac_HardStats::**XmitMultiCollision**

Number of frames involved in a multiple collision but sent successfully

Xuint64 XGemac_HardStats::**XmitPause**

Number of pause frames transmitted

Xuint64 XGmac_HardStats::**XmitUnderrun**

Number of frames not sent due to underrun

Xuint64 XGmac_HardStats::**XmitVlan**

Number of VLAN frames transmitted

The documentation for this struct was generated from the following file:

- [gemac/v1_00_f/src/xgemac.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

XGemac_SoftStats Struct Reference

```
#include <xgemac.h>
```

Detailed Description

Statistics maintained by SW

Data Fields

[Xuint32 XmitOverrunErrors](#)
[Xuint32 XmitUnderrunErrors](#)
[Xuint32 XmitExcessDeferralErrors](#)
[Xuint32 XmitPFifoUnderrunErrors](#)
[Xuint32 XmitLateCollErrors](#)
[Xuint32 RecvSlotLengthErrors](#)
[Xuint32 RecvOverrunErrors](#)
[Xuint32 RecvUnderrunErrors](#)
[Xuint32 RecvLengthFieldErrors](#)
[Xuint32 RecvLongErrors](#)
[Xuint32 RecvFcsErrors](#)
[Xuint32 DmaErrors](#)
[Xuint32 FifoErrors](#)
[Xuint32 RecvInterrupts](#)
[Xuint32 XmitInterrupts](#)
[Xuint32 EmacInterrupts](#)
[Xuint32 TotalInterrupts](#)

Field Documentation

Xuint32 XGemac_SoftStats::

DmaErrors

Number of DMA errors since
init

Xuint32 XGemac_SoftStats::

EmacInterrupts

Number of MAC (device)
interrupts

Xuint32 XGemac_SoftStats::

FifoErrors

Number of FIFO errors since
init

Xuint32 XGemac_SoftStats::

RecvFcsErrors

Number of recv FCS
errors

Xuint32 XGemac_SoftStats::

RecvInterrupts

Number of receive
interrupts

Xuint32 XGemac_SoftStats::

RecvLengthFieldErrors

Number of recv frames discarded with invalid length
field

Xuint32 XGemac_SoftStats::

RecvLongErrors

Number of recv long frames
discarded

Xuint32 XGemac_SoftStats::

RecvOverrunErrors

Number of recv frames discarded due to overrun errors

Xuint32 XGemac_SoftStats::

RecvSlotLengthErrors

Number of recv frames received with slot length errors

Xuint32 XGemac_SoftStats::

RecvUnderrunErrors

Number of recv underrun errors

Xuint32 XGemac_SoftStats::

TotalInterrupts

Total interrupts

Xuint32 XGemac_SoftStats::

XmitExcessDeferralErrors

Number of transmit deferral errors

Xuint32 XGemac_SoftStats::

XmitInterrupts

Number of transmit interrupts

Xuint32 XGemac_SoftStats::

XmitLateCollErrors

Number of late collision errors

Xuint32 XGmac_SoftStats::**XmitOverrunErrors**

Number of transmit overrun errors

Xuint32 XGmac_SoftStats::**XmitPFifoUnderrunErrors**

Number of transmit packet fifo underrun errors

Xuint32 XGmac_SoftStats::**XmitUnderrunErrors**

Number of transmit underrun errors

The documentation for this struct was generated from the following file:

- [gemac/v1_00_f/src/xgemac.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

gemac/v1_00_f/src/xgemac_stats.c File Reference

Detailed Description

Contains functions to get and clear the **XGemac** driver statistics.

MODIFICATION HISTORY:

Who	Date	Changes
ecm	01/13/03	First release
ecm	03/25/03	Revision update
rmm	05/28/03	Revision update
rmm	09/12/03	Added (commented out) HW counter code

```
#include "xbasic_types.h"  
#include "xenv.h"  
#include "xgemac_i.h"
```

Functions

```
void XGemac_GetSoftStats (XGemac *InstancePtr, XGemac_SoftStats *StatsPtr)
```

```
void XGemac_ClearSoftStats (XGemac *InstancePtr)
```

```
XStatus XGemac_GetHardStats (XGemac *InstancePtr, XGemac_HardStats *StatsPtr)
```

Function Documentation

```
void ( XGemac * )
XGemac_ClearSoftStats InstancePtr
```

Clear the **XGemac_SoftStats** structure for this driver.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( XGemac *
XGemac_GetHardStats InstancePtr,
XGemac_HardStats * StatsPtr
)
```

Get a snapshot of the current hardware statistics counters.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

XST_SUCCESS if counters were read and copied to user space XST_NO_FEATURE if counters are not part of the gemac hw

Note:

None.

```
void ( XGemac *
XGemac_GetSoftStats InstancePtr,
XGemac_SoftStats * StatsPtr
)
```

Get a copy of the **XGmac_SoftStats** structure, which contains the current statistics for this driver as maintained by software counters. The statistics are cleared at initialization or on demand using the **XGmac_ClearSoftStats()** function.

The DmaErrors and FifoErrors counts indicate that the device has been or needs to be reset. Reset of the device is the responsibility of the upper layer software.

Use **XGmac_GetHardStats()** to retrieve hardware maintained statistics (if so configured).

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None.

Note:

None.

gemac/v1_00_f/src/xgemac_intr_fifo.c File Reference

Detailed Description

Contains functions related to interrupt mode using direct FIFO communication.

The interrupt handler, [XGemac_IntrHandlerFifo\(\)](#), must be connected by the user to the interrupt controller.

MODIFICATION HISTORY:

Who	Date	Changes
ecm	01/13/03	First release
ecm	02/05/03	includes support for simple DMA
ecm	03/25/03	Revision update
rmm	05/28/03	Revision update
rmm	02/24/04	Changed packet FIFO write to use DRE.

```
#include "xbasic_types.h"
#include "xgemac_i.h"
#include "xio.h"
#include "xipif_v1_23_b.h"
```

Functions

[XStatus XGemac_FifoSend](#) ([XGemac](#) *InstancePtr, [Xuint8](#) *BufPtr, [Xuint32](#) ByteCount)

[XStatus XGemac_FifoRecv](#) ([XGemac](#) *InstancePtr, [Xuint8](#) *BufPtr, [Xuint32](#) *ByteCountPtr)

void [XGemac_IntrHandlerFifo](#) (void *InstancePtr)

```
void XGemac_SetFifoRecvHandler (XGemac *InstancePtr, void *CallbackRef,  
    XGemac_FifoHandler FuncPtr)  
void XGemac_SetFifoSendHandler (XGemac *InstancePtr, void *CallbackRef,  
    XGemac_FifoHandler FuncPtr)
```

Function Documentation

```
XStatus (XGemac *InstancePtr,  
XGemac_FifoRecv BufPtr,  
    Xuint8 *ByteCountPtr)
```

Receive an Ethernet frame into the given buffer if a frame has been received by the hardware. This function is typically called by the user in response to an interrupt invoking the receive callback function as set by **XGemac_SetFifoRecvHandler**().

The supplied buffer should be properly aligned (see **xgemac.h**) and large enough to contain the biggest frame for the current operating mode of the GEMAC device (approx 1518 bytes for normal frames and 9000 bytes for jumbo frames).

If the device is configured with DMA, simple DMA will be used to transfer the buffer from the GEMAC to memory. In this case, the receive buffer must not be cached (see **xgemac.h**).

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

BufPtr is a pointer to a memory buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_INTERRUPT if the device is not in interrupt mode
- XST_NO_DATA if there is no frame to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.
- XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy

- XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

The input buffer must be big enough to hold the largest Ethernet frame.

```

XStatus          ( XGemac *
XGemac_FifoSend      InstancePtr,
                        Xuint8 *   BufPtr,
                        Xuint32   ByteCount
                        )

```

Send an Ethernet frame using packet FIFO with interrupts. The caller provides a contiguous-memory buffer and its length.

The callback function set by using **XGemac_SetFifoSendHandler()** is invoked when the transmission is complete.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field.

If the device is configured with simple DMA, simple DMA will be used to transfer the buffer from memory to the GEMAC. This means that this buffer should not be cached.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

BufPtr is a pointer to a buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was successfully sent. An interrupt is generated when the GEMAC transmits the frame and the driver calls the callback set with **XGemac_SetFifoSendHandler()**
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_INTERRUPT if the device is not in interrupt mode
- XST_FIFO_NO_ROOM if there is no room in the FIFO for this frame
- XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

```
void  
XGemac_IntrHandlerFifo ( void *  
                        InstancePtr )
```

The interrupt handler for the Ethernet driver when configured for direct FIFO communication (as opposed to DMA).

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, or Send Packet FIFO. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the MAC.

Parameters:

InstancePtr is a pointer to the **XGemac** instance that just interrupted.

Returns:

None.

Note:

None.

```
void  
XGemac_SetFifoRecvHandler ( XGemac *  
                            InstancePtr,  
                            void *  
                            XGemac_FifoHandler FuncPtr  
                            )
```

Set the callback function for handling confirmation of transmitted frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called by the driver once per frame sent. The callback is responsible for freeing the transmitted buffer if necessary.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

CallbackRef is reference data to be passed back to the callback function. Its value is arbitrary and not used by the driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void XGemac_SetFifoSendHandler ( XGemac * InstancePtr, void * CallbackRef, XGemac_FifoHandler FuncPtr )
```

Set the callback function for handling received frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called once per frame received. During the callback, the upper layer software should call **XGemac_FifoRecv()** to retrieve the received frame.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

CallbackRef is reference data to be passed back to the callback function. Its value is arbitrary and not used by the driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

Generated on 24 Jun 2004 for Xilinx Device Drivers

gemac/v1_00_f/src/xgemac_control.c File Reference

Detailed Description

Functions in this file handle various control functions of the **XGmac** driver.

MODIFICATION HISTORY:

Who	Date	Changes
-----	------	---------

---	-----	-----
rmm	06/04/03	First release

```
#include "xbasic_types.h"  
#include "xgemac.h"  
#include "xgemac_1.h"  
#include "xio.h"
```

Functions

XStatus XGmac_SetInterframeGap (**XGmac** *InstancePtr, **Xuint8** Ifg)

void **XGmac_GetInterframeGap** (**XGmac** *InstancePtr, **Xuint8** *IfgPtr)

XStatus XGmac_SendPause (**XGmac** *InstancePtr, **Xuint16** PausePeriod)

XStatus XGmac_MgtRead (**XGmac** *InstancePtr, int PhyAddress, int Register, **Xuint16** *DataPtr)

XStatus XGmac_MgtWrite (**XGmac** *InstancePtr, int PhyAddress, int Register, **Xuint16** Data)

Function Documentation

```

void
XGmac_GetInterframeGap ( XGmac *
                        InstancePtr,
                        Xuint8 * IfgPtr
                        )

```

Get the interframe gap. See the description of interframe gap above in [XGmac_SetInterframeGap\(\)](#).

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

IfgPtr is a pointer to an 8-bit buffer into which the interframe gap value will be copied. The LSB value is 8 bit times.

Returns:

None. The values of the interframe gap parts are copied into the output parameters.

```

XStatus
XGmac_MgtRead ( XGmac *
               InstancePtr,
               int PhyAddress,
               int Register,
               Xuint16 * DataPtr
               )

```

Read a PHY register through the GMII Management Control mechanism.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

PhyAddress is the address of the PHY to be accessed. Valid range is 0 to 31.

Register is the register in the PHY to be accessed. Valid range is 0 to 31.

DataPtr is an output parameter that will contain the contents of the register.

Returns:

- XST_SUCCESS if the PHY register was successfully read and its contents were placed in *DataPtr*.
- XST_NO_FEATURE if GMII is not present with this GEMAC instance.
- XST_DEVICE_BUSY if another GMII read/write operation is already in progress.
- XST_FAILURE if an GMII read error is detected

Note:

This function blocks until the read operation has completed. If there is a HW problem then

this function may not return.

```
XStatus ( XGemac *  
XGemac_MgtWrite InstancePtr,  
  
    int PhyAddress,  
    int Register,  
    Xuint16 Data  
    )
```

Write to a PHY register through the GMII Management Control mechanism.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

PhyAddress is the address of the PHY to be accessed.

Register is the register in the PHY to be accessed.

Data is the what will be written to the register.

Returns:

- XST_SUCCESS if the PHY register was successfully read and its contents are placed in DataPtr.
- XST_DEVICE_BUSY if another GMII read/write operation is already in progress.
- XST_NO_FEATURE if GMII is not present with this GEMAC instance.

Note:

This function blocks until the write operation has completed. If there is a HW problem then this function may not return.

```
XStatus ( XGemac *  
XGemac_SendPause InstancePtr,  
  
    Xuint16 PausePeriod  
    )
```

Send a pause packet. When called, the GEMAC hardware will initiate transmission of an automatically formed pause packet. This action will not disrupt any frame transmission in progress but will take priority over any pending frame transmission. The pause frame will be sent even if the transmitter is in the paused state.

For this function to have any effect, the `XGE_FLOW_CONTROL` option must be set (see `XGmac_SetOptions`)).

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

PausePeriod is the amount of time to pause. The LSB is 512 bit times.

Returns:

- `XST_SUCCESS` if the pause frame transmission mechanism was successfully started.
- `XST_DEVICE_IS_STARTED` if the device has not been stopped

```
XStatus XGmac_SetInterframeGap ( XGmac *  
InstancePtr,  
Xuint8 Ifg  
)
```

Set the Interframe Gap (IFG), which is the time the MAC delays between transmitting frames.

The device must be stopped before setting the interframe gap.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Ifg is the interframe gap to set, the LSB is 8 bit times.

Returns:

- `XST_SUCCESS` if the interframe gap was set successfully
- `XST_DEVICE_IS_STARTED` if the device has not been stopped

Note:

None.

gemac/v1_00_f/src/xgemac_intr_dma.c File Reference

Detailed Description

Contains functions used in interrupt mode when configured with scatter-gather DMA.

The interrupt handler, [XGemac_IntrHandlerDma\(\)](#), must be connected by the user to the interrupt controller.

MODIFICATION HISTORY:

Who	Date	Changes
-----	-----	-----
rmm	05/28/03	New capability for driver
rmm	06/06/03	Used XGemac_mIsSgDma() instead of XGemac_mIsDma()
rmm	11/14/03	Instead of invoking once for each packet received, send/recv callbacks are invoked once for all packets.
rmm	12/22/03	Added functions XGemac_GetSgRecvFreeDesc() and XGemac_GetSgSendFreeDesc() .
rmm	02/24/04	Removed comments regarding buffer alignment requirements. Removed SgStart call from HandleDmaSendIntr() and HandleDmaRecvIntr() .

```
#include "xbasic_types.h"
#include "xgemac_i.h"
#include "xio.h"
#include "xbuf_descriptor.h"
#include "xdma_channel.h"
```

```
#include "xipif_v1_23_b.h"
```

Functions

XStatus XGemac_SgSend (**XGemac** *InstancePtr, XBufDescriptor *BdPtr, int Delay)

XStatus XGemac_SgRecv (**XGemac** *InstancePtr, XBufDescriptor *BdPtr)

void **XGemac_IntrHandlerDma** (void *InstancePtr)

XStatus XGemac_SetPktThreshold (**XGemac** *InstancePtr, **Xuint32** Direction, **Xuint8** Threshold)

XStatus XGemac_GetPktThreshold (**XGemac** *InstancePtr, **Xuint32** Direction, **Xuint8** *ThreshPtr)

XStatus XGemac_SetPktWaitBound (**XGemac** *InstancePtr, **Xuint32** Direction, **Xuint32** TimerValue)

XStatus XGemac_GetPktWaitBound (**XGemac** *InstancePtr, **Xuint32** Direction, **Xuint32** *WaitPtr)

XStatus XGemac_SetSgRecvSpace (**XGemac** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)

XStatus XGemac_SetSgSendSpace (**XGemac** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)

unsigned **XGemac_GetSgSendFreeDesc** (**XGemac** *InstancePtr)

unsigned **XGemac_GetSgRecvFreeDesc** (**XGemac** *InstancePtr)

void **XGemac_SetSgRecvHandler** (**XGemac** *InstancePtr, void *CallBackRef, **XGemac_SgHandler** FuncPtr)

void **XGemac_SetSgSendHandler** (**XGemac** *InstancePtr, void *CallBackRef, **XGemac_SgHandler** FuncPtr)

Function Documentation

XStatus	(XGemac *
XGemac_GetPktThreshold	<i>InstancePtr,</i>
	Xuint32 <i>Direction,</i>
	Xuint8 * <i>ThreshPtr</i>
)

Get the value of the packet count threshold for the scatter-gather DMA engine. See [xgemac.h](#) for more discussion of interrupt coalescing features.

Parameters:

InstancePtr is a pointer to the **XGEmac** instance to be worked on.

Direction indicates the channel, XGE_SEND or XGE_RECV, to get.

ThreshPtr is a pointer to the byte into which the current value of the packet threshold register will be copied.

Returns:

- XST_SUCCESS if the packet threshold was retrieved successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None

```
XStatus XGEmac_GetPktWaitBound(XGEmac *InstancePtr,  
                            Xuint32 Direction,  
                            Xuint32 *WaitPtr)
```

Get the packet wait bound timer for this driver/device. See [xgemac.h](#) for more discussion of interrupt coalescing features.

Parameters:

InstancePtr is a pointer to the **XGEmac** instance to be worked on.

Direction indicates the channel, XGE_SEND or XGE_RECV, to read.

WaitPtr is a pointer to the byte into which the current value of the packet wait bound register will be copied. Units are in milliseconds. Range is 0 - 1023. A value of 0 disables the timer.

Returns:

- XST_SUCCESS if the packet wait bound was retrieved successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
unsigned ( XGmac * )  
XGmac_GetSgRecvFreeDesc InstancePtr
```

Return the number of free buffer descriptor slots that can be added to the receive descriptor ring with [XGmac_SgRecv\(\)](#) before filling it up.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

Returns:

- The number of descriptors that can be given to the HW with [XGmac_SgRecv\(\)](#)
- 0 if no room is left or the device is not configured for SG DMA

Note:

None.

```
unsigned ( XGmac * )  
XGmac_GetSgSendFreeDesc InstancePtr
```

Return the number of free buffer descriptor slots that can be added to the send descriptor ring with [XGmac_SgSend\(\)](#) before filling it up.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

Returns:

- The number of descriptors that can be given to the HW with [XGmac_SgSend\(\)](#)
- 0 if no room is left or the device is not configured for SG DMA

Note:

None.

```
void ( void * )  
XGmac_IntrHandlerDma InstancePtr
```

The interrupt handler for the Ethernet driver when configured with scatter-gather DMA.

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, Send Packet FIFO, Recv DMA channel, or Send DMA channel. The packet FIFOs only interrupt during "deadlock" conditions.

Parameters:

InstancePtr is a pointer to the **XGEmac** instance that just interrupted.

Returns:

None.

Note:

None.

```
XStatus ( XGEmac *  
XGEmac_SetPktThreshold InstancePtr,  
                        Xuint32 Direction,  
                        Xuint8 Threshold  
                        )
```

Set the scatter-gather DMA packet count threshold for this device. See [xgemac.h](#) for more discussion of interrupt coalescing features.

The device must be stopped before setting the threshold.

Parameters:

InstancePtr is a pointer to the **XGEmac** instance to be worked on.

Direction indicates the channel, XGE_SEND or XGE_RECV, to set.

Threshold is the value of the packet threshold count used during interrupt coalescing. Valid range is 0 - 255. A value of 0 disables the use of packet threshold by the hardware.

Returns:

- XST_SUCCESS if the threshold was successfully set
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_DMA_SG_COUNT_EXCEEDED if the threshold must be equal to or less than the number of descriptors in the list
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts

would also catch this error.

Note:

None

```
XStatus ( XGemac *  
XGemac_SetPktWaitBound InstancePtr,  
                        Xuint32 Direction,  
                        Xuint32 TimerValue  
                        )
```

Set the scatter-gather DMA packet wait bound timer for this device. See [xgemac.h](#) for more discussion of interrupt coalescing features.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Direction indicates the channel, XGE_SEND or XGE_RECV, from which the threshold register is read.

TimerValue is the value of the packet wait bound timer to set. Units are in milliseconds. A value of 0 means the timer is disabled.

Returns:

- XST_SUCCESS if the packet wait bound was set successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
void ( XGemac *  
XGemac_SetSgRecvHandler InstancePtr,  
                        void *  
                        XGemac_SgHandler CallbackRef,  
                        FuncPtr  
                        )
```

Set the callback function for handling received frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame received. The head of a descriptor list is passed in along with the number of descriptors in the list. Before leaving the callback, the upper layer software should attach a new buffer to each descriptor in the list.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

CallbackRef is reference data to be passed back to the callback function. Its value is arbitrary and not used by the driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus ( XGemac *  
XGemac_SetSgRecvSpace InstancePtr,  
 Xuint32 * MemoryPtr,  
 Xuint32 ByteCount  
 )
```

Give the driver memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The **xgemac.h** file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be properly aligned (see **xgemac.h**).

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

MemoryPtr is a pointer to the beginning of the memory space.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the [XGmac_Initialize\(\)](#) function because the DMA channel components must be initialized before the memory space is set.

```
void XGmac_SetSgSendHandler ( XGmac * InstancePtr,  
                             void * CallbackRef,  
                             XGmac_SgHandler FuncPtr  
                             )
```

Set the callback function for handling confirmation of transmitted frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame sent. The head of a descriptor list is passed in along with the number of descriptors in the list. The callback is responsible for freeing buffers attached to these descriptors.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

CallbackRef is reference data to be passed back to the callback function. Its value is arbitrary and not used by the driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```

XStatus ( XGmac *
XGmac_SetSgSendSpace InstancePtr,
                        Xuint32 * MemoryPtr,
                        Xuint32 ByteCount
                        )

```

Give the driver memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The [xgemac.h](#) file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be properly aligned (see [xgemac.h](#)).

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

MemoryPtr is a pointer to the beginning of the memory space.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XGmac_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```

XStatus ( XGmac *
XGmac_SgRecv InstancePtr,
              XBufDescriptor * BdPtr
              )

```

Add a descriptor, with an attached empty buffer, into the receive descriptor list. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of frames to replace filled buffers with empty buffers. This function can be called when the device is started or stopped. Note that it does start the scatter-gather DMA engine. Although this is not necessary during initialization, it is not a problem during initialization because the MAC receiver is not yet started.

The buffer attached to the descriptor and the descriptor itself must be properly aligned (see [xgemac.h](#)).

Notification of received frames are done asynchronously through the receive callback function.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

```
XStatus ( XGmac *  
XGmac_SgSend InstancePtr,  
 XBufDescriptor * BdPtr,  
 int Delay  
 )
```

Send an Ethernet frame using scatter-gather DMA. The caller attaches the frame to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire Ethernet frame may or may not be contained within one descriptor. This function simply inserts the descriptor into the scatter-gather engine's transmit list. The caller is responsible for providing mutual exclusion to guarantee that a frame is contiguous in the transmit list. The descriptor must be properly aligned (see [xgemac.h](#)).

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the frame, the inserts are committed, which means the descriptors for this frame are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

This call is non-blocking. Notification of error or successful transmission is done asynchronously through the send or error callback function.

Parameters:

InstancePtr is a pointer to the [XGemac](#) instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Delay indicates whether to start the scatter-gather DMA channel immediately, or whether to wait. This allows the user to queue up a list of more than one descriptor before starting the transmission of the packets. Use `XEM_SGDMA_NODELAY` or `XEM_SGDMA_DELAY`, defined in [xgemac.h](#), as the value of this argument. If the user chooses to delay and build a list, the user must call this function with the `XEM_SGDMA_NODELAY` option or call [XGemac_Start\(\)](#) to kick off the transmissions.

Returns:

- `XST_SUCCESS` if the buffer was successful sent
- `XST_DEVICE_IS_STOPPED` if the Ethernet MAC has not been started yet
- `XST_NOT_SGDMA` if the device is not in scatter-gather DMA mode
- `XST_DMA_SG_LIST_FULL` if the descriptor list for the DMA channel is full
- `XST_DMA_SG_BD_LOCKED` if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- `XST_DMA_SG_NOTHING_TO_COMMIT` if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this

function if there are to be multiple threads that can call it.

Generated on 24 Jun 2004 for Xilinx Device Drivers

gemac/v1_00_f/src/xgemac_multicast.c File Reference

Detailed Description

Functions in this file handle multicast addressing capabilities

MODIFICATION HISTORY:

Who	Date	Changes
-----	------	---------

rmm	06/04/03	First release
-----	----------	---------------

```
#include "xbasic_types.h"
```

```
#include "xgemac_i.h"
```

```
#include "xio.h"
```

Functions

XStatus XGemac_MulticastAdd (**XGemac** *InstancePtr, **Xuint8** Location, **Xuint8** *AddressPtr)

XStatus XGemac_MulticastClear (**XGemac** *InstancePtr, **Xuint8** Location)

Function Documentation

```

XStatus                ( XGemac *
XGemac_MulticastAdd    InstancePtr,
                          Xuint8   Location,
                          Xuint8 * AddressPtr
                          )

```

Set a discrete multicast address entry in the CAM lookup table. There are up to XGE_CAM_MAX_ADDRESSES in this table. The GEMAC must be stopped (see [XGemac_Stop\(\)](#)) before multicast addresses can be modified.

Once set, the multicast address cannot be retrieved. It can be disabled by clearing it using [XGemac_MulticastClear\(\)](#).

Parameters:

InstancePtr is a pointer to the [XGemac](#) instance to be worked on.

Location indicates which of the entries is to be updated. Valid range is 0 to XGE_CAM_MAX_ADDRESSES-1.

AddressPtr is the multicast address to set.

Returns:

- XST_SUCCESS if the multicast address table was updated successfully
- XST_NO_FEATURE if this feature is not included in HW
- XST_DEVICE_IS_STARTED if the device has not yet been stopped
- XST_INVALID_PARAM if the Location parameter is greater than XGE_CAM_MAX_ADDRESSES

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

```

XStatus                ( XGemac *
XGemac_MulticastClear InstancePtr,
                          Xuint8   Location
                          )

```

Clear a discrete multicast address entry in the CAM lookup table. There are up to XGE_CAM_MAX_ADDRESSES in this table. The GEMAC must be stopped (see [XGmac_Stop\(\)](#)) before multicast addresses can be cleared.

The entry is cleared by writing an address of 00:00:00:00:00:00 to its location.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

Location indicates which of the entries is to be cleared. Valid range is 0 to XGE_CAM_MAX_ADDRESSES-1.

Returns:

- XST_SUCCESS if the multicast address table was updated successfully
- XST_NO_FEATURE if this feature is not included in HW
- XST_DEVICE_IS_STARTED if the device has not yet been stopped
- XST_INVALID_PARAM if the Location parameter is greater than XGE_CAM_MAX_ADDRESSES

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

gemac/v1_00_f/src/xgemac_polled.c File Reference

Detailed Description

Contains functions used when the driver is in polled mode. Use the [XGemac_SetOptions\(\)](#) function to put the driver into polled mode.

MODIFICATION HISTORY:

Who	Date	Changes
-----	-----	-----
ecm	01/13/03	First release
ecm	03/25/03	Revision update
rmm	05/28/03	Revision update
rmm	09/23/03	Removed use of XGE_EIR_RECV_DFIFO_OVER_MASK, XGE_EIR_XMIT _PAUSE_MASK, XGE_EIR_RECV_MISSED_FRAME_MASK, XGE_EIR_RECV _COLLISION_MASK, XGE_EIR_RECV_SHORT_ERROR_MASK
rmm	02/24/04	Changed Packet FIFO write to use the DRE version. Changed where Xmit interrupt status is cleared in PollSend().

```
#include "xbasic_types.h"  
#include "xgemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

XStatus **XGemac_PollSend** (**XGemac** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** ByteCount)

XStatus **XGemac_PollRecv** (**XGemac** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** *ByteCountPtr)

Function Documentation

```
XStatus
XGemac_PollRecv
    ( XGemac *
      InstancePtr,
      Xuint8 *
      BufPtr,
      Xuint32 *
      ByteCountPtr
    )
```

Receive an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver receives the frame directly from the MAC's packet FIFO. This is a non-blocking receive, in that if there is no frame ready to be received at the device, the function returns with an error. The MAC's error status is not checked, so statistics are not updated for polled receive. The buffer into which the frame will be received must be properly aligned (see [xgemac.h](#)).

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

BufPtr is a pointer to an aligned buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_NO_DATA if there is no frame to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.

Note:

Input buffer must be big enough to hold the largest Ethernet frame. Buffer must also be 32-bit aligned.

```
XStatus ( XGmac *
XGmac_PollSend InstancePtr,
Xuint8 * BufPtr,
Xuint32 ByteCount
)
```

Send an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver writes the frame directly to the MAC's packet FIFO, then enters a loop checking the device status for completion or error. Statistics are updated if an error occurs. The buffer to be sent must be properly aligned (see [xgmac.h](#)).

It is assumed that the upper layer software supplies a correctly formatted and aligned Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BufPtr is a pointer to a buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_FIFO_NO_ROOM if there is no room in the GEMAC's length FIFO for this frame
- XST_FIFO_ERROR if the FIFO was overrun or underrun. This error is critical and requires the caller to reset the device.
- XST_EMAC_COLLISION if the send failed due to excess deferral or late collision

Note:

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PollSend thread. On a 1 Gbit (1000Mbps) MAC, it takes about 12.1 usecs to transmit a maximum size Ethernet frame.

gemac/v1_00_f/src/xgemac_selftest.c File Reference

Detailed Description

Self-test and diagnostic functions of the **XGemac** driver.

MODIFICATION HISTORY:

Who	Date	Changes
-----	-----	-----
ecm	01/13/03	First release
ecm	03/25/03	Revision update
rmm	05/28/03	Revision update
rmm	06/12/03	Check for GMII before reading registers in SelfTest Auto negotiate is on by default now in XGE_ECR_RESET_STATE Redesign LoopbackTest().

```
#include "xbasic_types.h"  
#include "xgemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

XStatus XGemac_SelfTest (**XGemac** *InstancePtr)

Function Documentation

```
XStatus                    ( XGemac *  
XGemac_SelfTest                    InstancePtr )
```

Performs a self-test on the Ethernet device. The test includes:

- Run self-test on DMA channel, FIFO, and IPIF components
- Reset the Ethernet device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run, and starting it when ready to send and receive frames.

It should be noted that data caching must be disabled when this function is called because the DMA self-test uses two local buffers (on the stack) for the transfer test.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

XST_SUCCESS	Self-test was successful
XST_PFIPO_BAD_REG_VALUE	FIFO failed register self-test
XST_DMA_TRANSFER_ERROR	DMA failed data transfer self-test
XST_DMA_RESET_REGISTER_ERROR	DMA control register value was incorrect after a reset
XST_REGISTER_ERROR	Ethernet failed register reset test
XST_LOOPBACK_ERROR	Internal loopback failed
XST_IPIF_REG_WIDTH_ERROR	An invalid register width was passed into the function
XST_IPIF_RESET_REGISTER_ERROR	The value of a register at reset was invalid
XST_IPIF_DEVICE_STATUS_ERROR	A write to the device status register did not read back correctly
XST_IPIF_DEVICE_ACK_ERROR	A bit in the device status register did not reset when acked
XST_IPIF_DEVICE_ENABLE_ERROR	The device interrupt enable register was not updated correctly by the hardware when other registers were written to
XST_IPIF_IP_STATUS_ERROR	A write to the IP interrupt status register did not read back correctly
XST_IPIF_IP_ACK_ERROR	One or more bits in the IP status register did not reset when acked
XST_IPIF_IP_ENABLE_ERROR	The IP interrupt enable register was not updated correctly when other registers were written to

Note:

This function makes use of options-related functions, and the **XGemac_PollSend()** and **XGemac_PollRecv()** functions.

Because this test uses the PollSend function for its loopback testing, there is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the self-test thread.

gemac/v1_00_f/src/xgemac_intr.c File Reference

Detailed Description

This file contains general interrupt-related functions of the **XGemac** driver.

MODIFICATION HISTORY:

Who	Date	Changes
ecm	01/13/03	First release
ecm	03/25/03	Revision update
rmm	05/28/03	Revision update
rmm	09/23/03	Removed use of XGE_EIR_RECV_DFIFO_OVER_MASK, XGE_EIR_RECV_MISSED_FRAME_MASK, XGE_EIR_RECV_COLLISION_MASK, XGE_EIR_RECV_SHORT_ERROR_MASK

```
#include "xbasic_types.h"  
#include "xgemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

```
void XGemac_SetErrorHandler (XGemac *InstancePtr, void *CallBackRef,  
          XGemac_ErrorHandler FuncPtr)
```

Function Documentation

```

void
XGmac_SetErrorHandler
( XGmac *
InstancePtr,
void *
XGmac_ErrorHandler FuncPtr
)

```

Set the callback function for handling asynchronous errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- XST_DMA_ERROR indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- XST_FIFO_ERROR indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- XST_RESET_ERROR indicates an unrecoverable MAC error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- XST_DMA_SG_NO_LIST indicates an attempt was made to access a scatter-gather DMA list that has not yet been created.
- XST_DMA_SG_LIST_EMPTY indicates the driver tried to get a descriptor from the receive descriptor list, but the list was empty.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

CallbackRef is reference data to be passed back to the callback function. Its value is arbitrary and not used by the driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

gpio/v2_00_a/src/xgpio_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between the files of the driver. It is intended for internal use only.

NOTES:

None.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	03/13/02	First release

```
#include "xgpio.h"
```

Variables

[XGpio_Config](#) [XGpio_ConfigTable](#) []

Variable Documentation

[XGpio_Config](#) [XGpio_ConfigTable](#) ()

[]

This table contains configuration information for each GPIO device in the system.

gpio/v2_00_a/src/xgpio.h File Reference

Detailed Description

This file contains the software API definition of the Xilinx General Purpose I/O (**XGpio**) device driver component.

The Xilinx GPIO controller is a soft IP core designed for Xilinx FPGAs on the OPB or PLB bus and contains the following general features:

- Support for up to 32 I/O discretes for each channel (64 bits total).
- Each of the discretes can be configured for input or output.
- Configurable support for dual channels and interrupt generation.

The driver provides interrupt management functions. Implementation of interrupt handlers is left to the user. Refer to the provided interrupt example in the examples directory for details.

This driver is intended to be RTOS and processor independent. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

Note:

This API utilizes 32 bit I/O to the GPIO registers. With less than 32 bits, the unused bits from registers are read as zero and written as don't cares.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	03/13/02	First release
2.00a	jhl	11/26/03	Added support for dual channels and interrupts

```
#include "xbasic_types.h"
```

```
#include "xstatus.h"
#include "xgpio_1.h"
```

Data Structures

```
struct XGpio
struct XGpio_Config
```

Functions

XStatus XGpio_Initialize (**XGpio** *InstancePtr, **Xuint16** DeviceId)

void **XGpio_SetDataDirection** (**XGpio** *InstancePtr, unsigned Channel, **Xuint32** DirectionMask)

Xuint32 XGpio_DiscreteRead (**XGpio** *InstancePtr, unsigned Channel)

void **XGpio_DiscreteWrite** (**XGpio** *InstancePtr, unsigned Channel, **Xuint32** Mask)

XGpio_Config * XGpio_LookupConfig (**Xuint16** DeviceId)

void **XGpio_DiscreteSet** (**XGpio** *InstancePtr, unsigned Channel, **Xuint32** Mask)

void **XGpio_DiscreteClear** (**XGpio** *InstancePtr, unsigned Channel, **Xuint32** Mask)

XStatus XGpio_SelfTest (**XGpio** *InstancePtr)

void **XGpio_InterruptGlobalEnable** (**XGpio** *InstancePtr)

void **XGpio_InterruptGlobalDisable** (**XGpio** *InstancePtr)

void **XGpio_InterruptEnable** (**XGpio** *InstancePtr, **Xuint32** Mask)

void **XGpio_InterruptDisable** (**XGpio** *InstancePtr, **Xuint32** Mask)

void **XGpio_InterruptClear** (**XGpio** *InstancePtr, **Xuint32** Mask)

Xuint32 XGpio_InterruptGetEnabled (**XGpio** *InstancePtr)

Xuint32 XGpio_InterruptGetStatus (**XGpio** *InstancePtr)

Function Documentation

```
void
XGpio_DiscreteClear      ( XGpio *
                           InstancePtr,
                           unsigned Channel,
                           Xuint32 Mask
                           )
```

Set output discrete(s) to logic 0 for the specified GPIO channel.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Channel contains the channel of the GPIO (1 or 2) to operate on.

Mask is the set of bits that will be set to 0 in the discrete data register. All other bits in the data register are unaffected.

Returns:

None.

Note:

The hardware must be built for dual channels if this function is used with any channel other than 1. If it is not, this function will assert.

```
Xuint32 ( XGpio *  
XGpio_DiscreteRead InstancePtr,  
                unsigned Channel  
                )
```

Read state of discretess for the specified GPIO channel.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Channel contains the channel of the GPIO (1 or 2) to operate on.

Returns:

Current copy of the discretess register.

Note:

The hardware must be built for dual channels if this function is used with any channel other than 1. If it is not, this function will assert.

```
void ( XGpio *  
XGpio_DiscreteSet InstancePtr,  
                unsigned Channel,  
                Xuint32 Mask  
                )
```

Set output discrete(s) to logic 1 for the specified GPIO channel.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Channel contains the channel of the GPIO (1 or 2) to operate on.

Mask is the set of bits that will be set to 1 in the discrete data register. All other bits in the data register are unaffected.

Returns:

None.

Note:

The hardware must be built for dual channels if this function is used with any channel other than 1. If it is not, this function will assert.

```
void XGpio_DiscreteWrite ( XGpio * InstancePtr,
                          unsigned Channel,
                          Xuint32 Data
                          )
```

Write to discretess register for the specified GPIO channel.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Channel contains the channel of the GPIO (1 or 2) to operate on.

Data is the value to be written to the discretess register.

Returns:

None.

Note:

The hardware must be built for dual channels if this function is used with any channel other than 1. If it is not, this function will assert. See also **XGpio_DiscreteSet()** and **XGpio_DiscreteClear()**.

```
XStatus ( XGpio *  
XGpio_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize the **XGpio** instance provided by the caller based on the given DeviceID.

Nothing is done except to initialize the InstancePtr.

Parameters:

InstancePtr is a pointer to an **XGpio** instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the **XGpio** API must be made with this pointer.

DeviceId is the unique id of the device controlled by this **XGpio** component. Passing in a device id associates the generic **XGpio** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS Initialization was successful.
- XST_DEVICE_NOT_FOUND Device configuration data was not found for a device with the supplied device ID.

Note:

None.

```
void ( XGpio *  
XGpio_InterruptClear InstancePtr,  
 Xuint32 Mask  
 )
```

Clear pending interrupts with the provided mask. This function should be called after the software has serviced the interrupts that are pending. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XGPIO_IR* bits which are contained in [xgpio_1.h](#).

Returns:

None.

Note:

None.

```
void XGpio_InterruptDisable ( XGpio * InstancePtr, Xuint32 Mask )
```

Disable interrupts. This function allows specific interrupts for each channel to be disabled. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Mask is the mask to disable. Bits set to 1 are disabled. This mask is formed by OR'ing bits from XGPIO_IR* bits which are contained in [xgpio_1.h](#).

Returns:

None.

Note:

None.

```
void XGpio_InterruptEnable ( XGpio * InstancePtr, Xuint32 Mask )
```

Enable interrupts. The global interrupt must also be enabled by calling [XGpio_InterruptGlobalEnable\(\)](#) for interrupts to occur. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Mask is the mask to enable. Bit positions of 1 are enabled. This mask is formed by OR'ing bits from XGPIO_IR* bits which are contained in [xgpio_l.h](#).

Returns:

None.

Note:

None.

Xuint32 (**XGpio ***)
XGpio_InterruptGetEnabled *InstancePtr*

Returns the interrupt enable mask. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Returns:

A mask of bits made from XGPIO_IR* bits which are contained in [xgpio_l.h](#).

Returns:

None.

Note:

None.

Xuint32 (**XGpio ***)
XGpio_InterruptGetStatus *InstancePtr*

Returns the status of interrupt signals. Any bit in the mask set to 1 indicates that the channel associated with the bit has asserted an interrupt condition. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Returns:

A pointer to a mask of bits made from XGPIO_IR* bits which are contained in [xgpio_1.h](#).

Note:

The interrupt status indicates the status of the device irregardless if the interrupts from the devices have been enabled or not through [XGpio_InterruptEnable\(\)](#).

```
void XGpio_InterruptGlobalDisable ( XGpio * InstancePtr )
```

Disable the interrupt output signal. Interrupts enabled through [XGpio_InterruptEnable\(\)](#) will no longer be passed through until the global enable bit is set by [XGpio_InterruptGlobalEnable\(\)](#). This function is designed to allow all interrupts (both channels) to be disabled easily for entering a critical section. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Returns:

None.

Note:

None.

```
void XGpio_InterruptGlobalEnable ( XGpio * InstancePtr )
```

Enable the interrupt output signal. Interrupts enabled through **XGpio_InterruptEnable()** will not be passed through until the global enable bit is set by this function. This function is designed to allow all interrupts (both channels) to be enabled easily for exiting a critical section. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Returns:

None.

Note:

None.

XGpio_Config* (**Xuint16**)
XGpio_LookupConfig *DeviceId*

Lookup the device configuration based on the unique device ID. The table ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the device identifier to lookup.

Returns:

- A pointer of data type **XGpio_Config** which points to the device configuration if DeviceID is found.
- XNULL if DeviceID is not found.

Note:

None.

XStatus (**XGpio ***)
XGpio_SelfTest *InstancePtr*

Run a self-test on the driver/device. This function does a minimal test in which the data register is read. It only does a read without any kind of test because the hardware has been parameterized such that it may be only an input such that the state of the inputs won't be known.

All other hardware features of the device are not guaranteed to be in the hardware since they are parameterizable.

ARGUMENTS:

Parameters:

InstancePtr is a pointer to the **XGpio** instance to be worked on. This parameter must have been previously initialized with **XGpio_Initialize()**.

Returns:

XST_SUCCESS always. If the GPIO device was not present in the hardware a bus error could be generated. Other indicators of a bus error, such as registers in bridges or buses, may be necessary to determine if this function caused a bus error.

Note:

None.

```
void  
XGpio_SetDataDirection    ( XGpio * InstancePtr,  
                           unsigned Channel,  
                           Xuint32 DirectionMask  
                           )
```

Set the input/output direction of all discrete signals for the specified GPIO channel.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Channel contains the channel of the GPIO (1 or 2) to operate on.

DirectionMask is a bitmask specifying which discretions are input and which are output. Bits set to 0 are output and bits set to 1 are input.

Returns:

None.

Note:

The hardware must be built for dual channels if this function is used with any channel other

than 1. If it is not, this function will assert.

Generated on 24 Jun 2004 for Xilinx Device Drivers

XGpio Struct Reference

```
#include <xgpio.h>
```

Detailed Description

The XGpio driver instance data. The user is required to allocate a variable of this type for every GPIO device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [gpio/v2_00_a/src/xgpio.h](#)
-

gpio/v2_00_a/src/xgpio_1.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. High-level driver functions are defined in [xgpio.h](#).

The macros that are available in this file use a multiply to calculate the addresses of registers. The user can control whether that multiply is done at run time or at compile time. A constant passed as the channel parameter will cause the multiply to be done at compile time. A variable passed as the channel parameter will cause it to occur at run time.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	jhl	04/24/02	First release of low level driver
2.00a	jhl	11/26/03	Added support for dual channels and interrupts. This change required the functions to be changed such that the interface is not compatible with previous versions. See the examples in the example directory for macros designed for registers accessible using the ReadReg and WriteReg macros and a channel parameter was added to the other macros.

```
#include "xbasic_types.h"
#include "xio.h"
```

Registers

Register offsets for this device. This device utilizes IPIF interrupt registers.

```
#define XGPIO_DATA_OFFSET
#define XGPIO_TRI_OFFSET
#define XGPIO_DATA2_OFFSET
#define XGPIO_TRI2_OFFSET
#define XGPIO_GIER_OFFSET
#define XGPIO_ISR_OFFSET
#define XGPIO_IER_OFFSET
```

Interrupt Status and Enable Register bitmaps and masks

Bit definitions for the interrupt status register and interrupt enable registers.

```
#define XGPIO_IR_MASK
#define XGPIO_IR_CH1_MASK
#define XGPIO_IR_CH2_MASK
```

Defines

```
#define XGpio_mWriteReg(BaseAddress, RegOffset, Data)
#define XGpio_mReadReg(BaseAddress, RegOffset)
#define XGpio_mGetDataReg(BaseAddress, Channel)
#define XGpio_mSetDataReg(BaseAddress, Channel, Data)
```

Define Documentation

```
#define
XGPIO_DATA2_OFFSET
```

Data register for 2nd
channel

```
#define  
XGPIO_DATA_OFFSET
```

Data register for 1st
channel

```
#define  
XGPIO_GIER_OFFSET
```

Global interrupt enable
register

```
#define  
XGPIO_IER_OFFSET
```

Interrupt enable
register

```
#define  
XGPIO_IR_CH1_MASK
```

Mask for the 1st
channel

```
#define  
XGPIO_IR_CH2_MASK
```

Mask for the 2nd
channel

```
#define  
XGPIO_IR_MASK
```

Mask of all
bits

```
#define  
XGPIO_ISR_OFFSET
```

Interrupt status
register

```
#define XGpio_mGetDataReg( BaseAddress,  
                          Channel      )
```

Get the data register of the specified GPIO channel.

Parameters:

BaseAddress contains the base address of the GPIO device.

Channel contains the channel (1 or 2) to operate on.

Returns:

The contents of the data register.

Note:

None.

C-style signature: Xuint32 **XGpio_mGetDataReg**(Xuint32 BaseAddress, unsigned Channel)

```
#define XGpio_mReadReg( BaseAddress,  
                      RegOffset   )
```

Read a value from a GPIO register. A 32 bit read is performed. If the GPIO component is implemented in a smaller width, only the least significant data is read from the register. The most significant data will be read as 0.

Parameters:

BaseAddress is the base address of the GPIO device.

Register is the register offset from the base to read from.

Data is the data from the register.

Returns:

None.

Note:

None.

C-style signature: Xuint32 **XGpio_mReadReg**(Xuint32 BaseAddress, unsigned RegOffset)

```
#define XGpio_mSetDataReg( BaseAddress,  
                          Channel,  
                          Data      )
```

Set the data register of the specified GPIO channel.

Parameters:

BaseAddress contains the base address of the GPIO device.
Channel contains the channel (1 or 2) to operate on.
Data is the value to be written to the data register.

Returns:

None.

Note:

None.

C-style signature: void XGpio_mSetDataReg(Xuint32 BaseAddress, unsigned Channel, Xuint32 Data)

```
#define XGpio_mWriteReg( BaseAddress,  
                       RegOffset,  
                       Data      )
```

Write a value to a GPIO register. A 32 bit write is performed. If the GPIO component is implemented in a smaller width, only the least significant data is written.

Parameters:

BaseAddress is the base address of the GPIO device.
RegOffset is the register offset from the base to write to.
Data is the data written to the register.

Returns:

None.

Note:

None.

C-style signature: void XGpio_mWriteReg(Xuint32 BaseAddress, unsigned RegOffset, Xuint32 Data)

#define

XGPIO_TRI2_OFFSET

I/O direction register for 2nd
channel

#define

XGPIO_TRI_OFFSET

I/O direction register for 1st
channel

Generated on 24 Jun 2004 for Xilinx Device Drivers

XGpio_Config Struct Reference

```
#include <xgpio.h>
```

Detailed Description

This typedef contains configuration information for the device.

The documentation for this struct was generated from the following file:

- [gpio/v2_00_a/src/xgpio.h](#)
-

gpio/v2_00_a/src/xgpio_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of GPIO devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	02/04/02	First release
2.00a	jhl	12/16/02	Update for dual channel and interrupt support

```
#include "xgpio.h"  
#include "xparameters.h"
```

Variables

[XGpio_Config](#) [XGpio_ConfigTable](#) []

Variable Documentation

[XGpio_Config](#) [XGpio_ConfigTable](#)

[]

This table contains configuration information for each GPIO device in the system.

gpio/v2_00_a/src/xgpio_extra.c File Reference

Detailed Description

The implementation of the **XGpio** component's advanced discrete functions. See [xgpio.h](#) for more information about the component.

Note:

None

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	02/04/02	First release
2.00a	jhl	12/16/02	Update for dual channel and interrupt support

```
#include "xgpio.h"  
#include "xgpio_i.h"
```

Functions

```
void XGpio_DiscreteSet (XGpio *InstancePtr, unsigned Channel, Xuint32 Mask)  
void XGpio_DiscreteClear (XGpio *InstancePtr, unsigned Channel, Xuint32 Mask)
```

Function Documentation

```
void  
XGpio_DiscreteClear ( XGpio *  
                    InstancePtr,  
                    unsigned Channel,  
                    Xuint32 Mask  
                    )
```

Set output discrete(s) to logic 0 for the specified GPIO channel.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Channel contains the channel of the GPIO (1 or 2) to operate on.

Mask is the set of bits that will be set to 0 in the discrete data register. All other bits in the data register are unaffected.

Returns:

None.

Note:

The hardware must be built for dual channels if this function is used with any channel other than 1. If it is not, this function will assert.

```
void  
XGpio_DiscreteSet ( XGpio *  
                  InstancePtr,  
                  unsigned Channel,  
                  Xuint32 Mask  
                  )
```

Set output discrete(s) to logic 1 for the specified GPIO channel.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Channel contains the channel of the GPIO (1 or 2) to operate on.

Mask is the set of bits that will be set to 1 in the discrete data register. All other bits in the data register are unaffected.

Returns:

None.

Note:

The hardware must be built for dual channels if this function is used with any channel other than 1. If it is not, this function will assert.

Generated on 24 Jun 2004 for Xilinx Device Drivers

gpio/v2_00_a/src/xgpio.c File Reference

Detailed Description

The implementation of the **XGpio** component's basic functionality. See [xgpio.h](#) for more information about the component.

Note:

None

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	02/04/02	First release
2.00a	jhl	12/16/02	Update for dual channel and interrupt support

```
#include "xparameters.h"  
#include "xgpio.h"  
#include "xgpio_i.h"  
#include "xstatus.h"
```

Functions

XStatus **XGpio_Initialize** (**XGpio** *InstancePtr, **Xuint16** DeviceId)

XGpio_Config * **XGpio_LookupConfig** (**Xuint16** DeviceId)

void **XGpio_SetDataDirection** (**XGpio** *InstancePtr, unsigned Channel, **Xuint32** DirectionMask)

Xuint32 **XGpio_DiscreteRead** (**XGpio** *InstancePtr, unsigned Channel)

void **XGpio_DiscreteWrite** (**XGpio** *InstancePtr, unsigned Channel, **Xuint32** Data)

Function Documentation

```
Xuint32 ( XGpio *  
XGpio_DiscreteRead InstancePtr,  
                unsigned Channel  
                )
```

Read state of discretes for the specified GPIO channel.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Channel contains the channel of the GPIO (1 or 2) to operate on.

Returns:

Current copy of the discretes register.

Note:

The hardware must be built for dual channels if this function is used with any channel other than 1. If it is not, this function will assert.

```
void ( XGpio *  
XGpio_DiscreteWrite InstancePtr,  
                   unsigned Channel,  
                   Xuint32 Data  
                   )
```

Write to discretes register for the specified GPIO channel.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Channel contains the channel of the GPIO (1 or 2) to operate on.

Data is the value to be written to the discretes register.

Returns:

None.

Note:

The hardware must be built for dual channels if this function is used with any channel other than 1. If it is not, this function will assert. See also **XGpio_DiscreteSet()** and

XGpio_DiscreteClear().

```
XStatus ( XGpio *  
XGpio_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize the **XGpio** instance provided by the caller based on the given DeviceID.

Nothing is done except to initialize the InstancePtr.

Parameters:

InstancePtr is a pointer to an **XGpio** instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the **XGpio** API must be made with this pointer.

DeviceId is the unique id of the device controlled by this **XGpio** component. Passing in a device id associates the generic **XGpio** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS Initialization was successful.
- XST_DEVICE_NOT_FOUND Device configuration data was not found for a device with the supplied device ID.

Note:

None.

```
XGpio_Config* ( Xuint16 )  
XGpio_LookupConfig DeviceId
```

Lookup the device configuration based on the unique device ID. The table ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the device identifier to lookup.

Returns:

- A pointer of data type **XGpio_Config** which points to the device configuration if DeviceID is found.
- XNULL if DeviceID is not found.

Note:

None.

```
void  
XGpio_SetDataDirection ( XGpio * InstancePtr,  
                          unsigned Channel,  
                          Xuint32 DirectionMask  
                          )
```

Set the input/output direction of all discrete signals for the specified GPIO channel.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Channel contains the channel of the GPIO (1 or 2) to operate on.

DirectionMask is a bitmask specifying which discretés are input and which are output. Bits set to 0 are output and bits set to 1 are input.

Returns:

None.

Note:

The hardware must be built for dual channels if this function is used with any channel other than 1. If it is not, this function will assert.

gpio/v2_00_a/src/xgpio_intr.c File Reference

Detailed Description

Implements GPIO interrupt processing functions for the **XGpio** component. See [xgpio.h](#) for more information about the component.

The functions in this file require the hardware device to be built with interrupt capabilities. The functions will assert if called using hardware that does not have interrupt capabilities.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
2.00a	jhl	11/26/03	Initial release

```
#include "xgpio.h"  
#include "xipif_v1_23_b.h"
```

Functions

```
void XGpio_InterruptGlobalEnable (XGpio *InstancePtr)  
void XGpio_InterruptGlobalDisable (XGpio *InstancePtr)  
void XGpio_InterruptEnable (XGpio *InstancePtr, Xuint32 Mask)  
void XGpio_InterruptDisable (XGpio *InstancePtr, Xuint32 Mask)  
void XGpio_InterruptClear (XGpio *InstancePtr, Xuint32 Mask)
```

```
Xuint32 XGpio_InterruptGetEnabled (XGpio *InstancePtr)
```

```
Xuint32 XGpio_InterruptGetStatus (XGpio *InstancePtr)
```

Function Documentation

```
void          ( XGpio *
XGpio_InterruptClear      InstancePtr,
                          Xuint32 Mask
                          )
```

Clear pending interrupts with the provided mask. This function should be called after the software has serviced the interrupts that are pending. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XGPIO_IR* bits which are contained in [xgpio_l.h](#).

Returns:

None.

Note:

None.

```
void          ( XGpio *
XGpio_InterruptDisable    InstancePtr,
                          Xuint32 Mask
                          )
```

Disable interrupts. This function allows specific interrupts for each channel to be disabled. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Mask is the mask to disable. Bits set to 1 are disabled. This mask is formed by OR'ing bits from XGPIO_IR* bits which are contained in [xgpio_l.h](#).

Returns:

None.

Note:

None.

```
void  
XGpio_InterruptEnable ( XGpio *  
                       InstancePtr,  
                       Xuint32 Mask  
                       )
```

Enable interrupts. The global interrupt must also be enabled by calling [XGpio_InterruptGlobalEnable\(\)](#) for interrupts to occur. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Mask is the mask to enable. Bit positions of 1 are enabled. This mask is formed by OR'ing bits from XGPIO_IR* bits which are contained in [xgpio_l.h](#).

Returns:

None.

Note:

None.

```
Xuint32 ( XGpio *  
XGpio_InterruptGetEnabled InstancePtr
```

Returns the interrupt enable mask. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Returns:

A mask of bits made from XGPIO_IR* bits which are contained in [xgpio_l.h](#).

Returns:

None.

Note:

None.

Xuint32 (**XGpio ***)
XGpio_InterruptGetStatus *InstancePtr*

Returns the status of interrupt signals. Any bit in the mask set to 1 indicates that the channel associated with the bit has asserted an interrupt condition. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Returns:

A pointer to a mask of bits made from XGPIO_IR* bits which are contained in [xgpio_1.h](#).

Note:

The interrupt status indicates the status of the device irregardless if the interrupts from the devices have been enabled or not through [XGpio_InterruptEnable\(\)](#).

void (**XGpio ***)
XGpio_InterruptGlobalDisable *InstancePtr*

Disable the interrupt output signal. Interrupts enabled through [XGpio_InterruptEnable\(\)](#) will no longer be passed through until the global enable bit is set by [XGpio_InterruptGlobalEnable\(\)](#). This function is designed to allow all interrupts (both channels) to be disabled easily for entering a critical section. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Returns:

None.

Note:

None.

void (**XGpio ***)
XGpio_InterruptGlobalEnable *InstancePtr*

Enable the interrupt output signal. Interrupts enabled through **XGpio_InterruptEnable()** will not be passed through until the global enable bit is set by this function. This function is designed to allow all interrupts (both channels) to be enabled easily for exiting a critical section. This function will assert if the hardware device has not been built with interrupt capabilities.

Parameters:

InstancePtr is the GPIO component to operate on.

Returns:

None.

Note:

None.

gpio/v2_00_a/src/xgpio_selftest.c File Reference

Detailed Description

The implementation of the **XGpio** component's self test function. See [xgpio.h](#) for more information about the component.

Note:

None

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	02/04/02	First release
2.00a	jhl	01/13/04	Addition of dual channels and interrupts.

```
#include "xgpio.h"
```

Functions

[XStatus XGpio_SelfTest](#) ([XGpio](#) *InstancePtr)

Function Documentation

```
XStatus (XGpio *InstancePtr)  
XGpio\_SelfTest
```

Run a self-test on the driver/device. This function does a minimal test in which the data register is read. It only does a read without any kind of test because the hardware has been parameterized such that it may be only an input such that the state of the inputs won't be known.

All other hardware features of the device are not guaranteed to be in the hardware since they are parameterizable.

ARGUMENTS:

Parameters:

InstancePtr is a pointer to the **XGpio** instance to be worked on. This parameter must have been previously initialized with **XGpio_Initialize()**.

Returns:

XST_SUCCESS always. If the GPIO device was not present in the hardware a bus error could be generated. Other indicators of a bus error, such as registers in bridges or buses, may be necessary to determine if this function caused a bus error.

Note:

None.

hdlc/v2_01_a/src/xhdlc_stats.c File Reference

Detailed Description

Contains functions to get and clear the [XHdlc](#) driver statistics.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
2.00a	ecm	09/16/03	First release
2.01a	ecm	02/16/04	Add multichannel scatter-gather DMA support

```
#include "xbasic_types.h"  
#include "xhdlc_i.h"
```

Functions

```
void XHdlc\_GetStats (XHdlc *InstancePtr, XHdlc\_ChannelID ChannelID, XHdlc\_Stats *StatsPtr)  
void XHdlc\_ClearStats (XHdlc *InstancePtr, XHdlc\_ChannelID ChannelID)
```

Function Documentation

```
void  
XHdlc\_ClearStats ( XHdlc *  
                  InstancePtr,  
                  XHdlc\_ChannelID ChannelID  
                  )
```

Clear the statistics for the specified HDLC driver instance and physical channel/ sub channel pair.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel

Returns:

None.

Note:

None.

```
void XHdlc_GetStats ( XHdlc * InstancePtr, XHdlc_ChannelID ChannelID, XHdlc_Stats * StatsPtr )
```

Get a copy of the statistics structure, which contains the current statistics for this instance and physical channel/ sub channel pair. The statistics are only cleared at initialization or on demand using the **XHdlc_ClearStats()** function.

The FifoErrors counts indicate that the device has been or needs to be reset. Reset of the device is the responsibility of the caller.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None.

Note:

None.

hdlc/v2_01_a/src/xhdlc_l.c File Reference

Detailed Description

This file contains low-level polled functions to send and receive HDLC frames.

This file allocates a block of memory needed to access the various registers within the HDLC device. The offsets to these registers are generated in arrays to assist with access.

There are three arrays initialized here to speed up register access. They are:

[XHdlc_RegisterOffsets](#) XHdlc_Registers[]; Xuint16 XHdlc_CtrlRegSubOffset[]; Xuint16 XHdlc_FifoSubOffset[];

All told these arrays use 176 bytes of memory on the 8 physical channel 32 sub channel per physical channel configuration. This memory allocation can be reduced by adjusting the following constants to better reflect the hardware configuration. These constants are found in [xhdlc_l.h](#)

```
define XHD_MAX_NUM_PHY_CHANNELS 8 define XHD_MAX_NUM_SUB_CHANNELS 32
```

MODIFICATION HISTORY:

Ver	Who	Date	Changes
2.00a	ecm	04/17/03	First release
2.01a	ecm	02/16/04	Add multichannel scatter-gather DMA support

```
#include "xhdlc\_l.h"  
#include "xchannel_fifo_v1_00_a.h"  
#include "xio.h"
```

Functions

void **XHdlc_InitializeRegisterAccess** (**Xuint32** BaseAddress)

Xuint32 XHdlc_PollSend (**Xuint32** BaseAddress, **XHdlc_ChannelID** ChannelID, **Xuint32** *FramePtr, **Xuint32** ByteCount, **Xuint32** *ChannelStatusPtr)

Xuint32 XHdlc_PollRecv (**Xuint32** BaseAddress, **XHdlc_ChannelID** *ChannelIDPtr, **Xuint32** *FramePtr, **Xuint32** ByteCount, **Xuint32** *ChannelStatusPtr, **Xuint32** *RxErrorStatus)

void **XHdlc_DeviceEnable** (**Xuint32** BaseAddress)

void **XHdlc_DeviceDisable** (**Xuint32** BaseAddress)

void **XHdlc_PhysEnable** (**Xuint32** BaseAddress, **XHdlc_ChannelID** ChannelID)

void **XHdlc_PhysDisable** (**Xuint32** BaseAddress, **XHdlc_ChannelID** ChannelID)

void **XHdlc_PolledSubEnable** (**Xuint32** BaseAddress, **XHdlc_ChannelID** ChannelID)

void **XHdlc_SubEnable** (**Xuint32** BaseAddress, **XHdlc_ChannelID** ChannelID)

void **XHdlc_SubDisable** (**Xuint32** BaseAddress, **XHdlc_ChannelID** ChannelID)

void **XHdlc_PolledSubTxEnable** (**Xuint32** BaseAddress, **XHdlc_ChannelID** ChannelID)

void **XHdlc_SubTxEnable** (**Xuint32** BaseAddress, **XHdlc_ChannelID** ChannelID)

void **XHdlc_SubTxDisable** (**Xuint32** BaseAddress, **XHdlc_ChannelID** ChannelID)

Function Documentation

void (**Xuint32**)
XHdlc_DeviceDisable *BaseAddress*

Disable the HDLC Device. This prevents transmit and receive for the entire device. If there are channel(s) in use when the device is disabled, there will likely be garbage in the FIFO of the receive that needs to be cleaned out prior to using the channel(s) again.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
void ( Xuint32 )
XHdlc_DeviceEnable BaseAddress
```

Enable the HDLC Device. This allows the channels to transmit and receive data if available and the physical channel(s) and sub channel(s) are enabled.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
void ( Xuint32 )
XHdlc_InitializeRegisterAccess BaseAddress
```

Level 0 driver initialization. This function initializes the register access arrays allowing faster access to the hardware registers. This function also initializes the sub channel registers in the device itself since they are actually FPGA Block RAM (BRAM) which is not initialized in the FPGA configuration process.

Parameters:

param BaseAddress is the base address of the device

Returns:

none

Note:

None.

```
void ( Xuint32
XHdlc_PhysDisable BaseAddress,
XHdlc_ChannelID ChannelID
)
```

Disable the physical channel, both TX and RX. The sub channel of the channelID is ignored. If there are channel(s) in use when the device is disabled, there will likely be garbage in the FIFO of the receive that needs to be cleaned out prior to using the channel(s) again.

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```
void XHdlc_PhysEnable ( Xuint32 BaseAddress, XHdlc_ChannelID ChannelID )
```

Enable the physical channel, both TX and RX. The sub channel of the channelID is ignored.

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```
void XHdlc_PolledSubEnable ( Xuint32 BaseAddress, XHdlc_ChannelID ChannelID )
```

Enable the sub channel, both TX and RX for Polled Operation.

This configuration is intended for the polling operation in that the TX/RX complete event is written to the event FIFOs to allow the send/ receive function to just poll on that event FIFOs and not have to sort through the status FIFO events waiting for the complete. If the XHD_SCCR_*XCR_EVT0_MASK bit is not set, the TX/RX complete events are not sent to the FIFO's and the routines will not know when to write/read to/from the FIFO's

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```
void XHdlc_PolledSubTxEnable ( Xuint32 BaseAddress, XHdlc_ChannelID ChannelID )
```

Enable the transmitter only on the sub channel for Polled Operation.

This configuration is intended for the polling operation in that the TX complete event is written to the event FIFOs to allow the send function to just poll on that event FIFOs and not have to sort through the status FIFO events waiting for the complete. If the XHD_SCCR_*XCR_EVT0_MASK bit is not set, the TX complete events are not sent to the FIFO's and the routines will not know when to write to the FIFO's

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```
Xuint32
XHdlc_PollRecv
( Xuint32
    BaseAddress,
    XHdlc_ChannelID * ChannelIDPtr,
    Xuint32 * FramePtr,
    Xuint32 ByteCount,
    Xuint32 * ChannelStatusPtr,
    Xuint32 * RxErrorStatus
)
```

Receive a frame. Wait for a frame to arrive.

Parameters:

BaseAddress is the base address of the device

ChannelID is the channel data was received on.

FramePtr is a pointer to a 32 bit word-aligned buffer where the frame

ByteCount contains the number of bytes available in the buffer.

ChannelStatusPtr is a pointer to a channel status that will be valid after this function returns. The definitions of the bit positions are in [xhdlc_1.h](#) in the XHD_STAT_FIFO_*_MASK group of defines

RxErrorStatus is a pointer to a receive event that will be valid after this function returns. The definitions of the bit positions are in [xhdlc_1.h](#) in the XHD_RXEF_*_MASK group of defines

Returns:

number of Bytes read from the FIFO

Note:

This function assumes that the channel has been enabled at the sub channel and physical channel and that the entire device has been enabled. The function will hang if all of the enables are not correct.

This function assumes packet size is such that the entire received packet/ frame fits within the FIFO including CRC.

```

Xuint32
XHdlc_PollSend
    ( Xuint32
        BaseAddress,
        XHdlc_ChannelID ChannelID,
        Xuint32 * FramePtr,
        Xuint32 ByteCount,
        Xuint32 * ChannelStatusPtr
    )

```

Send a HDLC frame. This size is the total frame size, including header. This function blocks waiting for the frame to be transmitted.

Parameters:

BaseAddress is the base address of the device
ChannelID is the channel data is to be transmitted on.
FramePtr is a pointer to 32 bit word-aligned frame
ByteCount is the number of bytes in the frame to be sent
ChannelStatusPtr is a pointer to a channel status that will be valid after this function returns. The definitions of the bit positions are in [xhdlc_1.h](#) in the XHD_TXEF_*_MASK group of defines

Returns:

number of Bytes that still need to be sent, i.e. Byte Count requested - Actual Bytes sent

Note:

This function assumes that the channel has been enabled at the sub channel and physical channel and that the entire device has been enabled. The function will hang if all of the enables are not correct. This function also assumes that the FIFO will hold the entire packet to be sent.

```

void
XHdlc_SubDisable
    ( Xuint32
        BaseAddress,
        XHdlc_ChannelID ChannelID
    )

```

Disable the sub channel, both TX and RX.

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```
void XHdlc_SubEnable ( Xuint32 BaseAddress, XHdlc_ChannelID ChannelID )
```

Enable the sub channel, both TX and RX.

This function sets the TX and RX enable bits in the sub channel control register. It can be used for interrupt driven with FIFOs and SGDMA modes of operation.

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```
void XHdlc_SubTxDisable ( Xuint32 BaseAddress, XHdlc_ChannelID ChannelID )
```

Disable the transmitter only on sub channel.

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

```
void XHdlc_SubTxEnable ( Xuint32 BaseAddress, XHdlc_ChannelID ChannelID )
```

Enable the transmitter only on the sub channel.

This function sets the TX enable bit in the sub channel control register. It can be used for interrupt driven with FIFOs and SGDMA modes of operation.

Parameters:

BaseAddress is the base address of the device

ChannelID is specific channel

Returns:

None.

Note:

None.

hdlc/v2_01_a/src/xhdlc_options.c File Reference

Detailed Description

Functions in this file allows options for the **XHdlc** driver. These options are only sub channel specific. The physical channel configurations are set and cleared through the **XHdlc_PhysEnable()** / **XHdlc_PhysDisable()** and the **XHdlc_EnableLoopback()** / **XHdlc_DisableLoopback()** functions.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
2.00a	ecm	09/16/03	First release
2.01a	ecm	02/16/04	Add multichannel scatter-gather DMA support

```
#include "xstatus.h"  
#include "xhdlc_i.h"  
#include "xipif_v1_23_b.h"  
#include "xio.h"
```

Data Structures

```
struct OptionMap
```

Functions

```
XStatus XHdlc_SetOptions (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, Xuint16 Options)  
Xuint16 XHdlc_GetOptions (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)  
void XHdlc_SetAddress (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, Xuint16 Address)  
Xuint16 XHdlc_GetAddress (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)  
void XHdlc_EnableLoopback (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)  
void XHdlc_DisableLoopback (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)
```

Function Documentation

```
void  
XHdlc_DisableLoopback ( XHdlc *  
                        InstancePtr,  
                        XHdlc_ChannelID ChannelID  
                        )
```

Disable the loopback for the Physical channel provided. The sub channel in the ChannelID is ignored.

This function clears the loopback functionality for the physical channel. This takes all of the associated sub channels out of loopback also.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel, the sub channel is ignored

Returns:

None.

Note:

None.

```
void  
XHdlc_EnableLoopback ( XHdlc *  
                       InstancePtr,  
                       XHdlc_ChannelID ChannelID  
                       )
```

Enable the loopback for the Physical channel provided. The sub channel in the ChannelID is ignored.

This function set the loopback functionality for the physical channel. This puts all of the associated sub channels into loopback also.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel, the sub channel is ignored

Returns:

None.

Note:

None

```
Xuint16  
XHdlc_GetAddress ( XHdlc *  
                  InstancePtr,  
                  XHdlc_ChannelID ChannelID  
                  )
```

Get the receive address for this Sub Channel. The address is a 8 or 16 bit value within an HDLC frame. The Address applies only to the specified physical channel/ sub channel pair provided in the arguments.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel

Returns:

The receive address of the HDLC device.

Note:

None.

```
Xuint16 ( XHdlc *  
XHdlc_GetOptions InstancePtr,  
 XHdlc_ChannelID ChannelID  
 )
```

Get HDLC Sub Channel Options. A value is returned which is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel

Returns:

The value of the HDLC options. The value is a bit-mask representing all options that are currently enabled. See [xhdlc.h](#) for a description of the available options.

Note:

None.

```
void ( XHdlc *  
XHdlc_SetAddress InstancePtr,  
 XHdlc_ChannelID ChannelID,  
 Xuint16 Address  
 )
```

Set the receive address for this Sub Channel. The address is a 8 or 16 bit value within an HDLC frame. The Address applies only to the specified physical channel/ sub channel pair provided in the arguments.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel.

Address is the address to be set.

Returns:

None.

Note:

None.

```
XStatus ( XHdlc *  
XHdlc_SetOptions InstancePtr,  
 XHdlc_ChannelID ChannelID,  
 Xuint16 Options  
 )
```

Set HDLC Sub Channel options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option (i.e., you can OR the options together). A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel

Options is a bit-mask representing the HDLC options to turn on or off. See **xhdlc.h** for a description of the available options.

Returns:

- o XST_SUCCESS if the options were set successfully
- o XST_IS_STARTED if the channel has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

hdlc/v2_01_a/src/xhdlc.c File Reference

Detailed Description

Functions in this file are the minimum required functions for the HDLC driver. See [xhdlc.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

2.00a	ecm	09/16/03	First release
-------	-----	----------	---------------

2.01a	ecm	02/16/04	Add multichannel scatter-gather DMA support
-------	-----	----------	---

```
#include "xstatus.h"  
#include "xhdlc_1.h"  
#include "xhdlc_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"  
#include "xhdlc.h"
```

Functions

[XStatus XHdlc_Initialize](#) ([XHdlc](#) *InstancePtr, [Xuint32](#) *MemoryPtr, [Xuint16](#) DeviceId)

[XStatus XHdlc_Start](#) ([XHdlc](#) *InstancePtr)

[XStatus XHdlc_Stop](#) ([XHdlc](#) *InstancePtr)

void [XHdlc_EnableSubChannel](#) ([XHdlc](#) *InstancePtr, [XHdlc_ChannelID](#) ChannelID)

void [XHdlc_DisableSubChannel](#) ([XHdlc](#) *InstancePtr, [XHdlc_ChannelID](#) ChannelID)

void [XHdlc_Reset](#) ([XHdlc](#) *InstancePtr)

void **XHdlc_EventFlush** (**XHdlc** *InstancePtr)

void **XHdlc_SetChannelUserData** (**XHdlc** *InstancePtr, **XHdlc_ChannelID** ChannelID,
Xuint32 UserData)

Xuint32 **XHdlc_GetChannelUserData** (**XHdlc** *InstancePtr, **XHdlc_ChannelID**
ChannelID)

XHdlc_Config * **XHdlc_LookupConfig** (**Xuint16** DeviceId)

void **XHdlc_SetErrorHandler** (**XHdlc** *InstancePtr, void *CallBackRef,
XHdlc_ErrorHandler FuncPtr)

Function Documentation

```
void  
XHdlc_DisableSubChannel ( XHdlc *  
  
InstancePtr,  
XHdlc_ChannelID ChannelID  
)
```

This function disables the sub channel associated with the provided ChannelID. This function leaves the physical channel and the device enabled.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel.

Returns:

None.

Note:

None.

```
void  
XHdlc_EnableSubChannel ( XHdlc *  
  
InstancePtr,  
XHdlc_ChannelID ChannelID  
)
```

Enable the HDLC sub Channel. This function enables the sub channel associated with the provided ChannelID. This allows the channel to be used for data transfer.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.
ChannelID is the intended channel.

Returns:

None

Note:

```
void ( XHdlc * )  
XHdlc_EventFlush InstancePtr
```

Empty the HDLC Event and Data FIFO's. This empties the Status, Receive Event, and Transmit Event FIFO's. Then the Receive Channel FIFO's are emptied to prevent initial overruns and garbled data. This data is usually left over after either a power up reset or a user directed reset occurs. EventFlush must only be called after the driver has been initialized.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint32 ( XHdlc *  
XHdlc_GetChannelUserData InstancePtr,  
XHdlc_ChannelID ChannelID  
)
```

Gets the user data for the channel. This uses the channel specific data area provided by the application at initialization. The data itself can be anything that fits into an Xuint32 memory location.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the channel the data was stored in.

Returns:

- UserData is available for application use.

Note:

XHdlc_Initialize must be run prior to running this function

```
XStatus ( XHdlc *  
XHdlc_Initialize InstancePtr,  
                Xuint32 * MemoryPtr,  
                Xuint16 DeviceId  
                )
```

Initialize a specific **XHdlc** instance/driver. The initialization entails:

- Initialize fields of the **XHdlc** instance structure and access arrays.
- Clear the HDLC statistics for each channel configured in this device.
- Configure the FIFO components and DMA channels if available.
- Reset the HDLC device.

The driver defaults to interrupt mode operation.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

MemoryPtr is a pointer to the block of memory for the channel specific data such as the statistics (see notes). 32-bit aligned.

DeviceId is the unique id of the device controlled by this **XHdlc** instance. Passing in a device id associates the generic **XHdlc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- 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.
- XST_NO_FEATURE if the device configuration information indicates a feature that is not supported by this driver (no IPIF or DMA).

Note:

MemoryPtr points to a block of memory for the management of the individual channels. The size of this block of data is dependent on the number of channels available in the hardware. The **XHdlc_mSizeNeeded**(NumChannels) macro can be used to determine the number of bytes needed for this block of memory. This memory is divided up into two regions, an array of pointers which allow direct access into the larger block of memory which is the channel specific data itself. This block of memory is 32-bit aligned to prevent alignment errors in various processors.

XHdlc_Config* (**Xuint16**)
XHdlc_LookupConfig *DeviceId*

Lookup the device configuration based on the unique device ID. The table XHdlc_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

void (**XHdlc ***)
XHdlc_Reset *InstancePtr*

Reset the HDLC instance. This is a graceful reset in that the device is stopped first then it resets the FIFOs. Reset must only be called after the driver has been initialized.

The configuration after this reset is as follows:

- Disable all of the valid sub channels.
- Disable all of the valid physical channels.
- Disable the device itself.
- Device interrupts are disabled.

The upper layer software is responsible for re-configuring (if necessary) and restarting the HDLC device after the reset. Note also that driver statistics are not cleared on reset. It is up to the upper layer software to clear the statistics if needed.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

None.

Note:

None.

```
void XHdlc_SetChannelUserData ( XHdlc * InstancePtr,
                               XHdlc_ChannelID ChannelID,
                               Xuint32 UserData
                               )
```

Sets the user data for the channel. This uses the channel specific data area provided by the application at initialization. The data itself can be anything that fits into an Xuint32 memory location.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended channel to store the data in.

UserData is available for application use.

Returns:

None

Note:

XHdlc_Initialize must be run prior to running this function.

```
void XHdlc_SetErrorHandler ( XHdlc * InstancePtr, void * CallbackRef, XHdlc_ErrorHandler FuncPtr )
```

Sets the callback function for handling errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback which should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- XST_RESET_ERROR indicates an unrecoverable HDLC device error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

XStatus (**XHdlc** *)
XHdlc_Start *InstancePtr*

Start the HDLC Channel and driver by enabling the hardware device, enabling the DMA functionality if configured for it, Enables all of the available physical channels, and enables the interrupts from the device.

This function must be called before other functions to send or receive data. It supports interrupt driven modes with FIFOs and scatter gather DMA. The user should have connected the interrupt handler of the driver to an interrupt source such as an interrupt controller or the processor interrupt prior to this function being called.

This function starts all of the configured Physical Channels and enables the interrupt outputs from the device and finally enables the device itself. The application must then enable any/all sub channels using **XHdlc_EnableSubChannel()** before data will be transmitted and/or received across that channel.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully.
- XST_DEVICE_IS_STARTED if the device is already started.
- XST_NO_CALLBACK if a callback function has not yet been registered using the SetxxxHandler function. This is required if in interrupt mode.
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XStatus (**XHdlc ***)
XHdlc_Stop *InstancePtr*

Stop the HDLC device as follows:

- If the device is configured with DMA, stop the DMA channels (wait for acknowledgment of stop).
- Disables the available sub channels.
- Disables the available physical channels.
- Disables the device.
- Disables the interrupts from the device(the higher layer software is responsible for disabling interrupts at the interrupt controller).

If the device is configured for scatter-gather DMA, the DMA engine stops at the next buffer descriptor in its list. The remaining descriptors in the list are not removed, so anything in the list will be transmitted or received when the device is restarted. The side effect of doing this is that the last buffer descriptor processed by the DMA engine before stopping may not be the last descriptor in the HDLC frame. So when the device is restarted, a partial frame (i.e., a bad frame) may be transmitted/received. This is only a concern if a frame can span multiple buffer descriptors, which is dependent on the size of the network buffers.

To start using the device again, `_Start` must be called and any sub channels that are to be used must be enabled prior to use. Calling `_Reset` is recommended since there is the possibility that there will be garbage in the FIFO's.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully.
- XST_IS_STOPPED if the device is already stopped.

Note:

None.

hdlc/v2_01_a/src/xhdlc_intr_fifo.c File Reference

Detailed Description

This file contains the HDLC FIFO processing. This file contains send and receive functions as well as interrupt service routines.

Note:

None.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
2.00a	ecm	09/16/03	First release
2.01a	ecm	02/16/04	Add multichannel scatter-gather DMA support

```
#include "xhdlc.h"  
#include "xhdlc_1.h"  
#include "xhdlc_i.h"  
#include "xchannel_fifo_v1_00_a.h"  
#include "xipif_v1_23_b.h"  
#include "xio.h"  
#include "xstatus.h"
```

Functions

Xuint32 XHdlc_Send (**XHdlc** *InstancePtr, **XHdlc_ChannelID** ChannelID, **Xuint32** *FramePtr, **Xuint32** ByteCount, **Xboolean** EndOfPacket)

Xuint32 XHdlc_Recv (**XHdlc** *InstancePtr, **XHdlc_ChannelID** ChannelID, **Xuint32** *FramePtr, **Xuint32** ByteCount)

```
void XHdlc_InterruptHandlerFifo (void *InstancePtr)
void XHdlc_SetFifoRecvHandler (XHdlc *InstancePtr, void *CallbackRef,
XHdlc_FifoHandler FuncPtr)
void XHdlc_SetFifoSendHandler (XHdlc *InstancePtr, void *CallbackRef,
XHdlc_FifoHandler FuncPtr)
```

Function Documentation

```
void XHdlc_InterruptHandlerFifo ( void * InstancePtr )
```

Interrupt handler for the HDLC driver. It performs the following processing:

- Get the interrupt status from the IpIf to determine the source of the interrupt.
- Call the appropriate handler based on the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the HDLC device instance for the interrupt.

Returns:

None.

Note:

This handler reads all three FIFO's, Status, TX Event, and RX Event, if there are events in them and processes the event(s).

```
Xuint32 XHdlc_Recv ( XHdlc * InstancePtr,
XHdlc_ChannelID ChannelID,
Xuint32 * FramePtr,
Xuint32 ByteCount
)
```

Gets the received frame.

This function is called to initiate the reception of the frame, usually from within the callback. and, in the case of frames longer than the FIFO size, called repeatedly from within the callback to continue receiving the frame.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended receive channel.

FramePtr is the address of a buffer for received data, 32-bit aligned.

ByteCount is the size of the available buffer.

Returns:

The number of bytes received.

Note:

None.

```
Xuint32 ( XHdlc * InstancePtr,  
XHdlc_Send ChannelID,  
Xuint32 * FramePtr,  
Xuint32 ByteCount,  
Xboolean EndOfPacket  
)
```

Sends a HDLC frame using interrupt mode.

It is assumed that the upper layer software supplies a correctly formatted HDLC frame based upon the configuration of the HDLC device. The HDLC device must be started before calling this function.

This function is called to initiate a transmission and, in the case of frames longer than the FIFO size, called from within the callback to continue the transmission.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the intended transmit channel.
FramePtr is the address of a buffer to transmit, 32-bit aligned.
ByteCount is the number of bytes to be sent.
EndOfPacket indicates that this send is the end of packet.

Returns:

The number of bytes actually sent.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

The EndOfPacket flag allows the calling function to control whether the EndOfPacket is indicated to the hardware when the last of the data is written to the FIFO. This allows for multiple calls to the _Send function without having the EOP inserted at the wrong time. This flag is not the final decision on whether the EOP is indicated. In the situation where the data to be sent is larger than the the space available in the FIFO, the function will not indicate an EOP to the hardware and return to the caller with the actual number of bytes sent which will be less than the number of bytes requested to be sent. At this point, the calling function can call the _Send function again, after updating the buffer pointer to reflect the amount of data sent, with the EndOfPacket = XTRUE and the function will again try to put all of the requested data into the FIFO. Looping on the _Send until the entire buffer is sent effectively turns the non-blocking call to _Send into a blocking call until the buffer is completely sent.

Warning: EndOfPacket = XFALSE and not getting back to the _Send before the FIFO is empty causes underruns which can force the entire packet to be resent from the beginning.

```
void  
XHdlc_SetFifoRecvHandler ( XHdlc * InstancePtr,  
void * CallbackRef,  
XHdlc_FifoHandler FuncPtr  
)
```

Sets the callback function for handling received frames in FIFO mode. The upper layer software should call this function during initialization. The callback is called when a frame is received. The callback function should communicate the data to a thread such that the processing is not performed in an interrupt context.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void  
XHdlc_SetFifoSendHandler ( XHdlc * InstancePtr,  
void * CallbackRef,  
XHdlc_FifoHandler FuncPtr  
)
```

Sets the callback function for handling confirmation of transmitted frame in FIFO mode. The upper layer software should call this function during initialization. The callback is called when a frames is sent. The callback function should communicate the data to a thread such that processing is not performed in an interrupt context.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Note that a callback may occur with a Status of 0 and a ByteCount of 0. This represents a half-empty interrupt. The callback is still invoked here so the user has a chance to refill the FIFO before an underrun if not the end of packet.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

hdlc/v2_01_a/src/xhdlc_dmasg.c File Reference

Detailed Description

This file contains the HDLC scatter gather processing. This file contains send and receive functions as well as interrupt service routines.

Note:

None.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
2.00a	ecm	06/20/03	First release
2.01a	ecm	02/16/04	Add multichannel scatter-gather DMA support

```
#include "xhdlc.h"  
#include "xhdlc_l.h"  
#include "xhdlc_i.h"  
#include "xchannel_fifo_v1_00_a.h"  
#include "xipif_v1_23_b.h"  
#include "xdma_multi.h"  
#include "xio.h"  
#include "xstatus.h"
```

Functions

XStatus XHdlc_SgSend (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, XBufDescriptor *BdPtr)

XStatus XHdlc_SgRecv (**XHdlc** *InstancePtr, **XHdlc_ChannelID** ChannelID, XBufDescriptor *BdPtr)

XStatus XHdlc_SgGetSendFrame (**XHdlc** *InstancePtr, **XHdlc_ChannelID** ChannelID, XBufDescriptor **PtrToBdPtr, unsigned *BdCountPtr)

XStatus XHdlc_SgGetRecvFrame (**XHdlc** *InstancePtr, **XHdlc_ChannelID** ChannelID, XBufDescriptor **PtrToBdPtr, int *BdCountPtr)

void **XHdlc_InterruptHandlerSg** (void *InstancePtr)

XStatus XHdlc_SetSgRecvSpace (**XHdlc** *InstancePtr, **XHdlc_ChannelID** ChannelID, **Xuint32** *MemoryPtr, unsigned ByteCount)

XStatus XHdlc_SetSgSendSpace (**XHdlc** *InstancePtr, **XHdlc_ChannelID** ChannelID, **Xuint32** *MemoryPtr, unsigned ByteCount)

void **XHdlc_SetSgRecvHandler** (**XHdlc** *InstancePtr, void *CallBackRef, **XHdlc_SgHandler** FuncPtr)

void **XHdlc_SetSgSendHandler** (**XHdlc** *InstancePtr, void *CallBackRef, **XHdlc_SgHandler** FuncPtr)

Function Documentation

```
void XHdlc_InterruptHandlerSg ( void * InstancePtr )
```

Interrupt handler for the HDLC driver. It performs the following processing:

- Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: the HDLC device, the send packet FIFO, the receive packet FIFO, the send DMA channel, or the receive DMA channel. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the HDLC device.
- Call the appropriate handler based on the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the HDLC device instance for the interrupt.

Returns:

None.

Note:

None.

```

void ( XHdlc * InstancePtr,
XHdlc_SetSgRecvHandler
void * CallbackRef,
XHdlc_SgHandler FuncPtr
)

```

Sets the callback function for handling received cells in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of cells, determined by the DMA scatter-gather packet threshold, are received. The number of received cells is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each received cell from the list and should attach a new buffer to each descriptor. It is important that the specified number of cells passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```

XStatus ( XHdlc * InstancePtr,
XHdlc_SetSgRecvSpace
XHdlc_ChannelID ChannelID,
Xuint32 * MemoryPtr,
unsigned ByteCount
)

```

Gives the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the HDLC driver. The memory space must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the specific channel associated with this data.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the HDLC device is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the XHdlc_Initialize function because the DMA channel components must be initialized before the memory space is set.

```
void XHdlc_SetSgSendHandler ( XHdlc * InstancePtr, void * CallbackRef, XHdlc_SgHandler FuncPtr )
```

Sets the callback function for handling confirmation of transmitted cells in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of cells, determined by the DMA scatter-gather packet threshold, are sent. The number of sent cells is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each sent cell from the list and should also free the buffers attached to the descriptors if necessary. It is important that the specified number of cells passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If

there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallBackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus ( XHdlc *  
XHdlc_SetSgSendSpace InstancePtr,  
 XHdlc_ChannelID ChannelID,  
 Xuint32 * MemoryPtr,  
 unsigned ByteCount  
 )
```

Gives the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the HDLC driver. The memory space must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the specific channel associated with this data.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- o XST_SUCCESS if the space was initialized successfully
- o XST_NOT_SGDMA if the HDLC device is not configured for scatter-gather DMA
- o XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the

XHdlc_Initialize function because the DMA channel components must be initialized before the memory space is set.

```
XStatus ( XHdlc * InstancePtr,  
XHdlc_SgGetRecvFrame ChannelID,  
 XHdlc_ChannelID ChannelID,  
 XBufDescriptor ** PtrToBdPtr,  
 int * BdCountPtr  
 )
```

Gets the first buffer descriptor of the oldest frame which was received by the scatter-gather DMA channel of the HDLC device. This function is provided to be called from a callback function such that the buffer descriptors for received frames can be processed. The function should be called by the application repetitively for the number of frames indicated as an argument in the callback function.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the specific channel associated with this data.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the frame. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the frame. This input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the frame pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- XST_SUCCESS if a descriptor was successfully returned to the driver.
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode.
- XST_DMA_SG_NO_LIST if the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```

XStatus ( XHdlc *
XHdlc_SgGetSendFrame InstancePtr,
XHdlc_ChannelID ChannelID,
XBufDescriptor ** PtrToBdPtr,
unsigned * BdCountPtr
)

```

Gets the first buffer descriptor of the oldest frame which was sent by the scatter-gather DMA channel of the HDLC device. This function is provided to be called from a callback function such that the buffer descriptors for sent frames can be processed. The function should be called by the application repetitively for the number of frames indicated as an argument in the callback function.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the specific channel associated with this data.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the frame. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the frame. this input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the frame pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- XST_SUCCESS if a descriptor was successfully returned to the driver.
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode.
- XST_DMA_SG_NO_LIST if the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```

XStatus          ( XHdlc *
XHdlc_SgRecv      InstancePtr,
                    XHdlc_ChannelID ChannelID,
                    XBufDescriptor * BdPtr
                    )

```

Adds this descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be word-aligned. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of frames to replace filled buffers with empty buffers. The contents of the specified buffer descriptor are copied into the scatter-gather transmit list. This function can be called when the device is started or stopped.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the specific channel associated with this data.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

Note:

None.

```

XStatus          ( XHdlc *
XHdlc_SgSend      InstancePtr,
                    XHdlc_ChannelID ChannelID,
                    XBufDescriptor * BdPtr
                    )

```

Sends a HDLC frame using scatter-gather DMA. The caller attaches the frame to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire frame may or may not be contained within one descriptor. The contents of the buffer descriptor are copied into the scatter-gather transmit list. The caller is responsible for providing mutual exclusion to guarantee that a frame is contiguous in the transmit list. The buffer attached to the descriptor must be word-aligned.

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the frame, the inserts are committed, which means the descriptors for this frame are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted HDLC frame based upon the configuration of the HDLC device. The HDLC device must be started before calling this function.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

ChannelID is the specific channel associated with this data.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Returns:

- XST_SUCCESS if the buffer was successfully sent
- XST_IS_STOPPED if the HDLC device has not been started yet
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the descriptor list for the DMA channel is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

hdlc/v2_01_a/src/xhdlc_i.c File Reference

Detailed Description

Functions in this file are to the driver. They are not intended for use by applications.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
2.00a	ecm	09/16/03	First release
2.01a	ecm	02/16/04	Add multichannel scatter-gather DMA support

```
#include "xstatus.h"  
#include "xhdlc_1.h"  
#include "xhdlc_i.h"  
#include "xio.h"  
#include "xhdlc.h"
```

Functions

```
void XHdlc_LogicalToPhysicalSub (XHdlc *InstancePtr, Xuint32 LogicalChannel, XHdlc_ChannelID  
*ChannelIDPtr)
```

```
void XHdlc_PhysicalSubToLogical (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID, Xuint32  
*LogicalChannelPtr)
```

```
Xboolean XHdlc_IsPhysicalSubEnabled (XHdlc *InstancePtr, XHdlc_ChannelID ChannelID)
```

Function Documentation

```
Xboolean  
XHdlc_IsPhysicalSubEnabled  
( XHdlc *  
InstancePtr,  
XHdlc_ChannelID ChannelID  
)
```

Determine if the Physical channel/ Sub channel pair is enabled at the hardware level. If the hardware itself is disabled, by default from a functional perspective, the channel pair is also disabled regardless of the status of the bits in the respective registers.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.
ChannelID contains the PhysicalChannel/Subchannel pair to be checked.

Returns:

- o XTRUE if the physical channel, sub channel, and device are enabled.
- o XFALSE otherwise.

Note:

XHdlc_Initialize must be run prior to running this function.

```
void XHdlc_LogicalToPhysicalSub ( XHdlc * InstancePtr,
                                Xuint32 LogicalChannel,
                                XHdlc_ChannelID * ChannelIDPtr
                                )
```

Convert the logical ID to a ChannelID which is a physical channel number/ subchannel number pair. This functions converts the provided logical number to the physical/sub channel pair by stepping through the PhysChanSubCount array in the instance which contains the available channels associated with a given physical channel.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.
LogicalChannel to be converted from.
ChannelIDPtr pointer for the resulting data pair to be written to.

Returns:

the ChannelID pair corresponding to the Logical ID provided in ChannelID.

Note:

XHdlc_Initialize must be run prior to running this function.

```
void XHdlc_PhysicalSubToLogical ( XHdlc * InstancePtr,
                                  XHdlc_ChannelID ChannelID,
                                  Xuint32 * LogicalChannelPtr
                                  )
```

Convert the ChannelID which is a physical channel number/ subchannel number pair into an incrementing Logical Channel Number.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.
ChannelID data pair to be converted from.
LogicalChannelPtr pointer for the resulting channel to be written to.

Returns:

the Logical ID corresponding to the ChannelID pair provided.

Note:

XHdlc_Initialize must be run prior to running this function.

hwicap/v1_00_a/src/xhwicap_clb_ff.h File Reference

Detailed Description

This header file contains bit information about the CLB FF resource. This header file can be used with the [XHwIcap_GetClbBits\(\)](#) and [XHwIcap_SetClbBits\(\)](#) functions.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	bjb	11/14/03	First release

Data Structures

```
struct XHwIcap_ClbFf
```

Defines

```
#define XHI_CLB_XQ  
#define XHI_CLB_YQ
```

Variables

```
const XHwIcap_ClbFf XHI_CLB_FF
```

Define Documentation

```
#define  
XHI_CLB_XQ
```

Index into the CONTENTS and SRMODE for XQ Register.

```
#define  
XHI_CLB_YQ
```

Index into the CONTENTS and SRMODE for YQ Register.

Variable Documentation

```
const XHwIcap_ClbFf  
XHI_CLB_FF
```

This structure defines the bits associated with a Flip Flop in a CLB tile. Note that there are 8 FFs, the XQ and YQ Registers in Slice 0, 1, 2 and 3.

Generated on 24 Jun 2004 for Xilinx Device Drivers

hwicap/v1_00_a/src/xhwicap.h File Reference

Detailed Description

This file contains the software API definition of the Xilinx Hardware ICAP (hwicap) component.

The Xilinx Hardware ICAP controller is designed to allow reconfiguration of select FPGA resources as well as loading partial bitstreams from system memory through the Internal Configuration Access Port (ICAP).

The source code for the XHwIcap_SetClbBits and XHwIcap_GetClbBits functions are not included. These functions are delivered as .o files. Libgen uses the appropriate .o files for the target processor. This is specified by the hwicap_v2_1_0.tcl file in the data directory.

Note:

There are a few items to be aware of when using this driver. 1) Only Virtex 2 and Virtex 2 Pro devices are supported as they are the only devices that contain the ICAP_VIRTEX2 component. 2) The ICAP port is disabled when the configuration mode, via the MODE pins, is set to Boundary Scan/JTAG. The ICAP is enabled in all other configuration modes and it is possible to configure the device via JTAG in all configuration modes. 3) Reading or writing to columns containing SRL16's or LUT RAM's can cause corruption of data in those elements. Avoid reading or writing to columns containing SRL16's or LUT RAM's.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	bjb	11/17/03	First release

```
#include "xhwicap_1.h"  
#include <xstatus.h>
```

Data Structures

```
struct XHwIcap  
struct XHwIcap_Config
```

Defines

```
#define XHwIcap_mSliceX2Col(X)  
#define XHwIcap_mSliceY2Row(InstancePtr, Y)  
#define XHwIcap_mSliceXY2Slice(X, Y)
```

Functions

```
XStatus XHwIcap_Initialize (XHwIcap *InstancePtr, Xuint16 DeviceId, Xuint32 DeviceIdCode)  
XStatus XHwIcap_DeviceRead (XHwIcap *InstancePtr, Xuint32 Offset, Xuint32 NumInts)  
XStatus XHwIcap_DeviceWrite (XHwIcap *InstancePtr, Xuint32 Offset, Xuint32 NumInts)  
void XHwIcap_StorageBufferWrite (XHwIcap *InstancePtr, Xuint32 Address, Xuint32 Data)  
Xuint32 XHwIcap_StorageBufferRead (XHwIcap *InstancePtr, Xuint32 Address)  
XStatus XHwIcap_DeviceReadFrame (XHwIcap *InstancePtr, Xint32 Block, Xint32 MajorFrame,  
    Xint32 MinorFrame)  
XStatus XHwIcap_DeviceWriteFrame (XHwIcap *InstancePtr, Xint32 Block, Xint32 MajorFrame,  
    Xint32 MinorFrame)  
XStatus XHwIcap_SetConfiguration (XHwIcap *InstancePtr, Xuint32 *Data, Xuint32 Size)  
XStatus XHwIcap_CommandDesync (XHwIcap *InstancePtr)  
XStatus XHwIcap_CommandCapture (XHwIcap *InstancePtr)  
Xuint32 XHwIcap_GetConfigReg (XHwIcap *InstancePtr, Xuint32 ConfigReg)  
XStatus XHwIcap_SetClbBits (XHwIcap *InstancePtr, Xint32 Row, Xint32 Col, const Xuint8  
    Resource[][2], const Xuint8 Value[], Xint32 NumBits)  
XStatus XHwIcap_GetClbBits (XHwIcap *InstancePtr, Xint32 Row, Xint32 Col, const Xuint8  
    Resource[][2], Xuint8 Value[], Xint32 NumBits)
```

Define Documentation

```
#define XHwIcap_mSliceX2Col( X )
```

Converts a CLB SliceX coordinate to a column coordinate used by the XHwIcap_GetClibBits and XHwIcap_SetClibBits functions.

Parameters:

X - the SliceX coordinate to be converted

Returns:

Column

Note:

Xuint32 [XHwIcap_mSliceX2Col](#)(Xuint32 X);

```
#define XHwIcap_mSliceXY2Slice( X,  
                               Y )
```

Figures out which slice in a CLB is targeted by a given (SliceX,SliceY) pair. This slice value is used for indexing in resource arrays.

Parameters:

X - the SliceX coordinate to be converted

Y - the SliceY coordinate to be converted

Returns:

Slice index

Note:

Xuint32 [XHwIcap_mSliceXY2Slice](#)(Xuint32 X, Xuint32 Y);

```
#define XHwIcap_mSliceY2Row( InstancePtr,  
                             Y )
```

Converts a CLB SliceY coordinate to a row coordinate used by the XHwIcap_GetClbBits and XHwIcap_SetClbBits functions.

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Y - the SliceY coordinate to be converted

Returns:

Row

Note:

Xuint32 **XHwIcap_mSliceY2Row**(XHwIcap *InstancePtr, Xuint32 Y);

Function Documentation

XStatus (**XHwIcap** *)
XHwIcap_CommandCapture *InstancePtr*

Sends a CAPTURE command to the ICAP port. This command captures all of the flip flop states so they will be available during readback. One can use this command instead of enabling the CAPTURE block in the design.

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Returns:

XStatus - XST_SUCCESS or XST_DEVICE_BUSY or XST_INVALID_PARAM

Note:

None.

XStatus (**XHwIcap** *)
XHwIcap_CommandDesync *InstancePtr*

Sends a DESYNC command to the ICAP port.

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Returns:

XStatus - XST_SUCCESS or XST_DEVICE_BUSY or XST_INVALID_PARAM

Note:

None.

```
XStatus ( XHwIcap *  
XHwIcap_DeviceRead InstancePtr,  
 Xuint32 Offset,  
 Xuint32 NumInts  
 )
```

Reads bytes from the device (ICAP) and puts it in the storage buffer.

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Offset - The storage buffer start address.

NumInts - The number of words (32 bit) to read from the device (ICAP).

Returns:

XStatus - XST_SUCCESS or XST_DEVICE_BUSY or XST_INVALID_PARAM

Note:

None.

```
XStatus ( XHwIcap *  
XHwIcap_DeviceReadFrame InstancePtr,  
 Xint32 Block,  
 Xint32 MajorFrame,  
 Xint32 MinorFrame  
 )
```

Reads one frame from the device and puts it in the storage buffer.

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Block - Block Address (XHI_FAR_CLB_BLOCK, XHI_FAR_BRAM_BLOCK, XHI_FAR_BRAM_INT_BLOCK)

MajorFrame - selects the column

MinorFrame - selects frame inside column

Returns:

XST_SUCCESS, XST_BUFFER_TOO_SMALL or XST_INVALID_PARAM.

Note:

None.

```
XStatus ( XHwIcap *  
XHwIcap_DeviceWrite InstancePtr,  
 Xuint32 Offset,  
 Xuint32 NumInts  
 )
```

Writes bytes from the storage buffer and puts it in the device (ICAP).

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Offset - The storage buffer start address.

NumInts - The number of words (32 bit) to read from the device (ICAP).

Returns:

XStatus - XST_SUCCESS or XST_DEVICE_BUSY or XST_INVALID_PARAM

Note:

None.

```

XStatus
XHwIcap_DeviceWriteFrame      ( XHwIcap *
                                InstancePtr,
                                Xint32    Block,
                                Xint32    MajorFrame,
                                Xint32    MinorFrame
                                )

```

Writes one frame from the storage buffer and puts it in the device (ICAP).

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Block - Block Address (XHI_FAR_CLB_BLOCK, XHI_FAR_BRAM_BLOCK, XHI_FAR_BRAM_INT_BLOCK)

MajorFrame - selects the column

MinorFrame - selects frame inside column

Returns:

XST_SUCCESS, XST_BUFFER_TOO_SMALL or XST_INVALID_PARAM.

Note:

None.

```

XStatus
XHwIcap_GetClbBits            ( XHwIcap *
                                InstancePtr,
                                Xint32    Row,
                                Xint32    Col,
                                const Xuint8 Resource[
                                    [2],
                                Xuint8    Value[],
                                Xint32    NumBits
                                )

```

Gets bits contained in a Center tile specified by the CLB row and col coordinates. The coordinate system labels the upper left CLB as (1,1). There are four slices per CLB.

Parameters:

- InstancePtr* - a pointer to the **XHwIcap** instance to be worked on.
- Row* - CLB row. (1,1) is the upper left CLB.
- Col* - CLB col. (1,1) is the upper left CLB.
- Resource* - Target bits (first dimension length will be the number of bits to set and must match the numBits parameter) (second dimension contains two value -- one for minor row and one for col information from within the Center tile targetted by the above row and col coords).
- Value* - Values to store each of the target bits into. The size of this array must be equal to numBits.
- NumBits* - The number of Bits to change in this method.

Returns:

XST_SUCCESS, XST_BUFFER_TOO_SMALL or XST_INVALID_PARAM.

Note:

None.

```
Xuint32 ( XHwIcap *  
XHwIcap_GetConfigReg InstancePtr,  
 Xuint32 ConfigReg  
 )
```

This function returns the value of the specified configuration register.

Parameters:

- InstancePtr* - a pointer to the **XHwIcap** instance to be worked on.
- ConfigReg* - A constant which represents the configuration register value to be returned. Constants specified in **xhwicap_i.h**. Examples: XHI_IDCODE, XHI_FLR.

Returns:

The value of the specified configuration register.

```

XStatus
XHwIcap_Initialize
    ( XHwIcap *
      InstancePtr,
      Xuint16   DeviceId,
      Xuint32   DeviceIdCode
    )

```

Initialize a **XHwIcap** instance..

Parameters:

- InstancePtr* - a pointer to the **XHwIcap** instance to be worked on.
- BaseAddress* - Base Address of the instance of this component.
- DeviceId* - User defined ID for the instance of this component.
- DeviceIdCode* - IDCODE of the FPGA device. Examples of constants that can be used: XHI_XC2V40, XHI_XC2VP7 etc. The constant XHI_READ_DEVICEID_FROM_ICAP can be used instead of specifying the IDCODE directly.

Returns:

XST_SUCCESS, XST_INVALID_PARAM, or the invalid status from CommandDesync when using XHI_READ_DEVICEID_FROM_ICAP.

Note:

Virtex2/Pro devices only have one ICAP port so there should only be one opb_hwicap instantiated (per FPGA) in a system.

```

XStatus
XHwIcap_SetClbBits
    ( XHwIcap *
      InstancePtr,
      Xint32   Row,
      Xint32   Col,
      const Xuint8 Resource[]
      [2],
      const Xuint8 Value[],
      Xint32   NumBits
    )

```

Sets bits contained in a Center tile specified by the CLB row and col coordinates. The coordinate system labels the upper left CLB as (1,1). There are four slices per CLB.

Parameters:

- InstancePtr* - a pointer to the **XHwIcap** instance to be worked on.
- Row* - CLB row. (1,1) is the upper left CLB.
- Col* - CLB col. (1,1) is the upper left CLB.
- Resource* - Target bits (first dimension length will be the number of bits to set and must match the numBits parameter) (second dimension contains two value -- one for minor row and one for col information from within the Center tile targetted by the above row and col coords).
- Value* - Values to set each of the targets bits to. The size of this array must be equal to NumBits.
- NumBits* - The number of Bits to change in this method.

Returns:

XST_SUCCESS, XST_BUFFER_TOO_SMALL or XST_INVALID_PARAM.

Note:

None.

```
XStatus ( XHwIcap *  
XHwIcap_SetConfiguration InstancePtr,  
 Xuint32 * Data,  
 Xuint32 Size  
 )
```

Loads a partial bitstream from system memory.

Parameters:

- InstancePtr* - a pointer to the **XHwIcap** instance to be worked on.
- Data* - Address of the data representing the partial bitstream
- Size* - the size of the partial bitstream in 32 bit words.

Returns:

XST_SUCCESS, XST_BUFFER_TOO_SMALL or XST_INVALID_PARAM.

Note:

None.

```
Xuint32 ( XHwIcap *  
XHwIcap_StorageBufferRead InstancePtr,  
 Xuint32 Address  
 )
```

Read data from the specified address in the storage buffer..

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Address - bram word address

Returns:

Data.

Note:

None.

```
void ( XHwIcap *  
XHwIcap_StorageBufferWrite InstancePtr,  
 Xuint32 Address,  
 Xuint32 Data  
 )
```

Stores data in the storage buffer at the specified address.

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Address - bram word address

Data - data to be stored at address

Returns:

None.

Note:

None.

hwicap/v1_00_a/src/xhwicap_i.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xhwicap.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a	bjb	11/14/03	First release
	bjs	03/05/04	Updated done to status register

```
#include "xbasic_types.h"
```

```
#include "xio.h"
```

Defines

```
#define XHwIcap_mGetSizeReg(BaseAddress)  
#define XHwIcap_mGetOffsetReg(BaseAddress)  
#define XHwIcap_mGetDoneReg(BaseAddress)  
#define XHwIcap_mGetStatusReg(BaseAddress)  
#define XHwIcap_mGetBram(BaseAddress, Offset)  
#define XHwIcap_mSetSizeReg(BaseAddress, Data)  
#define XHwIcap_mSetOffsetReg(BaseAddress, Data)  
#define XHwIcap_mSetRncReg(BaseAddress, Data)  
#define XHwIcap_mSetBram(BaseAddress, Offset, Data)
```

Define Documentation

```
#define XHwIcap_mGetBram( BaseAddress,  
                          Offset      )
```

Reads data from the storage buffer bram.

A bram is used as a configuration memory cache. One frame of data can be stored in this "storage buffer".

Parameters:

BaseAddress - contains the base address of the component.

Offset - The offset into which the data should be read.

Returns:

The value of the specified offset in the bram.

Note:

Xuint32 **XHwIcap_mGetBram**(Xuint32 BaseAddress, Xuint32 Offset);

```
#define XHwIcap_mGetDoneReg( BaseAddress )
```

Get the contents of the done register.

The done register is set to zero during configuration or readback. When the current configuration or readback completes the done register is set to one.

Parameters:

BaseAddress is the base address of the device

Returns:

A 32-bit value with bit 1 representing done or not

Note:

Xuint32 **XHwIcap_mGetDoneReg**(Xuint32 BaseAddress);

```
#define XHwIcap_mGetOffsetReg( BaseAddress )
```

Get the contents of the bram offset register.

The bram offset register holds the starting bram address to transfer data from during configuration or write data to during readback.

Parameters:

BaseAddress is the base address of the device

Returns:

A 32-bit value representing the contents of the bram offset register.

Note:

Xuint32 **XHwIcap_mGetOffsetReg**(Xuint32 BaseAddress);

```
#define XHwIcap_mGetSizeReg( BaseAddress )
```

Get the contents of the size register.

The size register holds the number of 32 bit words to transfer between bram and the icap (or icap to bram).

Parameters:

BaseAddress is the base address of the device

Returns:

A 32-bit value representing the contents of the size register.

Note:

Xuint32 **XHwIcap_mGetSizeReg**(Xuint32 BaseAddress);

```
#define XHwIcap_mGetStatusReg( BaseAddress )
```

Get the contents of the status register.

The status register contains the ICAP status and the done bit.

D8 - cfgerr D7 - dalign D6 - rip D5 - in_abort_1 D4 - Always 1 D3 - Always 1 D2 - Always 1 D1 - Always 1 D0 - Done bit

Parameters:

BaseAddress is the base address of the device

Returns:

A 32-bit value representing the contents of the status register

Note:

Xuint32 [XHwIcap_mGetStatusReg](#)(Xuint32 BaseAddress);

```
#define XHwIcap_mSetBram( BaseAddress,  
                          Offset,  
                          Data      )
```

Write data to the storage buffer bram.

A bram is used as a configuration memory cache. One frame of data can be stored in this "storage buffer".

Parameters:

BaseAddress - contains the base address of the component.

Offset - The offset into which the data should be written.

Data - The value to be written to the bram offset.

Returns:

None.

Note:

void [XHwIcap_mSetBram](#)(Xuint32 BaseAddress, Xuint32 Offset, Xuint32 Data);

```
#define XHwIcap_mSetOffsetReg( BaseAddress,  
                               Data      )
```

Set the bram offset register.

The bram offset register holds the starting bram address to transfer data from during configuration or write data to during readback.

Parameters:

BaseAddress contains the base address of the device.

Data is the value to be written to the data register.

Returns:

None.

Note:

void [XHwIcap_mSetOffsetReg](#)(Xuint32 BaseAddress, Xuint32 Data);

```
#define XHwIcap_mSetRncReg( BaseAddress,  
                           Data          )
```

Set the RNC (Readback not Configure) register.

The RNC register determines the direction of the data transfer. It controls whether a configuration or readback take place. Writing to this register initiates the transfer. A value of 1 initiates a readback while writing a value of 0 initiates a configuration.

Parameters:

BaseAddress contains the base address of the device.

Data is the value to be written to the data register.

Returns:

None.

Note:

void [XHwIcap_mSetRncReg](#)(Xuint32 BaseAddress, Xuint32 Data);

```
#define XHwIcap_mSetSizeReg( BaseAddress,  
                             Data          )
```

Set the size register.

The size register holds the number of 8 bit bytes to transfer between bram and the icap (or icap to bram).

Parameters:

BaseAddress - contains the base address of the device.

Data - The size in bytes.

Returns:

None.

Note:

void [XHwIcap_mSetSizeReg](#)(Xuint32 BaseAddress, Xuint32 Data);

Generated on 24 Jun 2004 for Xilinx Device Drivers

XHwIcap Struct Reference

```
#include <xhwicap.h>
```

Detailed Description

The XHwIcap driver instance data. The user is required to allocated a variable of this type for every opb_hwicap device in the system. A pointer to a variable of this type is then passed to the driver API functions.

Note - Virtex2/Pro devices only have one ICAP port so there should be at most only one opb_hwicap instantiated (per FPGA) in a system.

The documentation for this struct was generated from the following file:

- hwicap/v1_00_a/src/[xhwicap.h](#)

XHwIcap_Config Struct Reference

```
#include <xhwicap.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

Field Documentation

Xuint32 XHwIcap_Config::

BaseAddress

Register base
address

Xuint16 XHwIcap_Config::

DeviceId

Unique ID of
device

The documentation for this struct was generated from the following file:

- [hwicap/v1_00_a/src/xhwicap.h](#)

hwicap/v1_00_a/src/xhwicap_i.h File Reference

Detailed Description

This head file contains internal identifiers, which are those shared between the files of the driver. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----

1.00a	bjb	11/14/03	First release

```
#include "xhwicap.h"
```

Defines

```
#define XHwIcap_Type1Read(Register)  
#define XHwIcap_Type1Write(Register)  
#define XHwIcap_SetupFar(Block, MajorAddress, MinorAddress)
```

Variables

```
XHwIcap_Config XHwIcap_ConfigTable []
```

Define Documentation

```
#define XHwIcap_SetupFar( Block,  
                        MajorAddress,  
                        MinorAddress )
```

Generates a Type 1 packet header that writes to the FAR (Frame Address Register).

Parameters:

Block - Address Block Type (CLB or BRAM address space)

MajorAddress - CLB or BRAM column

MinorAddress - Frame within column

Returns:

Type 1 packet header to write the FAR

Note:

None.

```
#define XHwIcap_Type1Read( Register )
```

Generates a Type 1 packet header that reads back the requested configuration register.

Parameters:

Register is the address of the register to be read back. Register constants are defined in this file.

Returns:

Type 1 packet header to read the specified register

Note:

None.

```
#define XHwIcap_Type1Write( Register )
```

Generates a Type 1 packet header that writes to the requested configuration register.

Parameters:

Register is the address of the register to be written to. Register constants are defined in this file.

Returns:

Type 1 packet header to write the specified register

Note:

None.

Variable Documentation

[XHwIcap_Config](#) XHwIcap_ConfigTable ()

[]

The configuration table for opb_hwicap devices

Generated on 24 Jun 2004 for Xilinx Device Drivers

hwicap/v1_00_a/src/xhwicap_clb_lut.h File Reference

Detailed Description

This header file contains bit information about the CLB LUT resource. This header file can be used with the [XHwIcap_GetClbBits\(\)](#) and [XHwIcap_SetClbBits\(\)](#) functions.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	bjb	11/14/03	First release

Data Structures

```
struct XHwIcap_ClbLut
```

Defines

```
#define XHI_CLB_LUT_F  
#define XHI_CLB_LUT_G
```

Variables

```
const XHwIcap_ClbLut XHI_CLB_LUT
```

Define Documentation

```
#define  
XHI_CLB_LUT_F
```

Index into SLICE and MODE for F LUT.

```
#define  
XHI_CLB_LUT_G
```

Index into SLICE and MODE for G LUT.

Variable Documentation

```
const XHwIcap_ClbLut  
XHI_CLB_LUT
```

This structure defines the Look Up Tables, or *LUTs*, in the Virtex2/Pro CLB. Note that there are 8 16-bit LUTs, the F and G LUTs in Slice 0, 1, 2 and 3. These LUTs can take any arbitrary bit pattern.

Note, that DUAL_PORT mode cannot be configured here. That's because it is essentially always in effect. But, it can only be used in the top two slices (2 and 3) using the address lines from the bottom two slices (0 and 1) for the write address. Although you can technically put the bottom two slice LUTs in dual port mode in the fpga_editor, the read and write addresses will always be the same. This is different from the Virtex where the two LUTs in a slice were combined to make a dual port RAM. In Virtex 2, every LUT is dual ported, but only the top two have different read/write addresses.

Generated on 24 Jun 2004 for Xilinx Device Drivers

hwicap/v1_00_a/src/xhwicap_clb_srinv.h File Reference

Detailed Description

This header file contains bit information about the CLB SRINV resource. This header file can be used with the [XHwIcap_GetClbBits\(\)](#) and [XHwIcap_SetClbBits\(\)](#) functions.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	bjb	11/14/03	First release

Data Structures

struct [XHwIcap_ClbSrinv](#)

Variables

const XHwIcap_ClbSrinv [XHI_CLB_SRINV](#)

Variable Documentation

const XHwIcap_ClbSrinv
XHI_CLB_SRINV

This structure defines the SRINV
mux

Generated on 24 Jun 2004 for Xilinx Device Drivers

hwicap/v1_00_a/src/xhwicap_srp.c File Reference

Detailed Description

This file contains SRP (self reconfigurable platform) driver functions.

The SRP contains functions that allow low level access to configuration memory through the ICAP port. This API provide methods for reading and writing data, frames, and partial bitstreams to the ICAP port.

Note:

Only Virtex 2 and Virtex 2 Pro devices are supported as they are the only devices that contain the VIRTEX2_ICAP component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a	bjb	11/17/03	First release
-------	-----	----------	---------------

	bjs	03/10/04	Fixed bug with XHI_READ_DEVICEID_FROM_ICAP
--	-----	----------	--

```
#include <xparameters.h>
#include <xbasic_types.h>
#include <xstatus.h>
#include "xhwicap_i.h"
#include "xhwicap.h"
```

Functions

XStatus **XHwIcap_Initialize** (**XHwIcap** *InstancePtr, **Xuint16** DeviceId, **Xuint32** DeviceIdCode)
void **XHwIcap_StorageBufferWrite** (**XHwIcap** *InstancePtr, **Xuint32** Address, **Xuint32** Data)
Xuint32 **XHwIcap_StorageBufferRead** (**XHwIcap** *InstancePtr, **Xuint32** Address)
XStatus **XHwIcap_DeviceRead** (**XHwIcap** *InstancePtr, **Xuint32** Offset, **Xuint32** NumInts)
XStatus **XHwIcap_DeviceWrite** (**XHwIcap** *InstancePtr, **Xuint32** Offset, **Xuint32** NumInts)
XStatus **XHwIcap_CommandDesync** (**XHwIcap** *InstancePtr)
XStatus **XHwIcap_CommandCapture** (**XHwIcap** *InstancePtr)
Xuint32 **XHwIcap_GetConfigReg** (**XHwIcap** *InstancePtr, **Xuint32** ConfigReg)

Function Documentation

XStatus (**XHwIcap** **InstancePtr*)
XHwIcap_CommandCapture

Sends a CAPTURE command to the ICAP port. This command captures all of the flip flop states so they will be available during readback. One can use this command instead of enabling the CAPTURE block in the design.

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Returns:

XStatus - XST_SUCCESS or XST_DEVICE_BUSY or XST_INVALID_PARAM

Note:

None.

XStatus (**XHwIcap** **InstancePtr*)
XHwIcap_CommandDesync

Sends a DESYNC command to the ICAP port.

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Returns:

XStatus - XST_SUCCESS or XST_DEVICE_BUSY or XST_INVALID_PARAM

Note:

None.

```
XStatus ( XHwIcap *  
XHwIcap_DeviceRead InstancePtr,  
 Xuint32 Offset,  
 Xuint32 NumInts  
 )
```

Reads bytes from the device (ICAP) and puts it in the storage buffer.

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Offset - The storage buffer start address.

NumInts - The number of words (32 bit) to read from the device (ICAP).

Returns:

XStatus - XST_SUCCESS or XST_DEVICE_BUSY or XST_INVALID_PARAM

Note:

None.

```
XStatus ( XHwIcap *  
XHwIcap_DeviceWrite InstancePtr,  
 Xuint32 Offset,  
 Xuint32 NumInts  
 )
```

Writes bytes from the storage buffer and puts it in the device (ICAP).

Parameters:

- InstancePtr* - a pointer to the **XHwIcap** instance to be worked on.
- Offset* - The storage buffer start address.
- NumInts* - The number of words (32 bit) to read from the device (ICAP).

Returns:

XStatus - XST_SUCCESS or XST_DEVICE_BUSY or XST_INVALID_PARAM

Note:

None.

```
Xuint32 ( XHwIcap *  
XHwIcap_GetConfigReg InstancePtr,  
 Xuint32 ConfigReg  
 )
```

This function returns the value of the specified configuration register.

Parameters:

- InstancePtr* - a pointer to the **XHwIcap** instance to be worked on.
- ConfigReg* - A constant which represents the configuration register value to be returned. Constants specified in **xhwicap_i.h**. Examples: XHI_IDCODE, XHI_FLR.

Returns:

The value of the specified configuration register.

```
XStatus ( XHwIcap *  
XHwIcap_Initialize InstancePtr,  
 Xuint16 DeviceId,  
 Xuint32 DeviceIdCode  
 )
```

Initialize a **XHwIcap** instance..

Parameters:

- InstancePtr* - a pointer to the **XHwIcap** instance to be worked on.
- BaseAddress* - Base Address of the instance of this component.
- DeviceId* - User defined ID for the instance of this component.
- DeviceIdCode* - IDCODE of the FPGA device. Examples of constants that can be used: XHI_XC2V40, XHI_XC2VP7 etc. The constant XHI_READ_DEVICEID_FROM_ICAP can be used instead of specifying the IDCODE directly.

Returns:

XST_SUCCESS, XST_INVALID_PARAM, or the invalid status from CommandDesync when using XHI_READ_DEVICEID_FROM_ICAP.

Note:

Virtex2/Pro devices only have one ICAP port so there should only be one opb_hwicap instantiated (per FPGA) in a system.

```
Xuint32 ( XHwIcap *  
XHwIcap_StorageBufferRead InstancePtr,  
 Xuint32 Address  
 )
```

Read data from the specified address in the storage buffer..

Parameters:

- InstancePtr* - a pointer to the **XHwIcap** instance to be worked on.
- Address* - bram word address

Returns:

Data.

Note:

None.

```
void  
XHwIcap_StorageBufferWrite ( XHwIcap *  
                               InstancePtr,  
                               Xuint32   Address,  
                               Xuint32   Data  
                               )
```

Stores data in the storage buffer at the specified address.

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Address - bram word address

Data - data to be stored at address

Returns:

None.

Note:

None.

hwicap/v1_00_a/src/xhwicap_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of opb_hwicap devices in the system. Each device in the system should have an entry in the table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	bjb	12/08/03	First release

```
#include "xhwicap.h"  
#include "xparameters.h"
```

Variables

[XHwIcap_Config](#) [XHwIcap_ConfigTable](#) [XPAR_XHWICAP_NUM_INSTANCES]

Variable Documentation

[XHwIcap_Config](#) [XHwIcap_ConfigTable](#)
[XPAR_XHWICAP_NUM_INSTANCES]

The configuration table for opb_hwicap devices

hwicap/v1_00_a/src/xhwicap_device_read_frame. c File Reference

Detailed Description

This file contains the function that reads a specified frame from the device (ICAP) and stores it in the bram storage buffer.

Note:

none.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	
1.00a	bjb	11/20/03	First release

```
#include "xhwicap.h"  
#include "xhwicap_i.h"  
#include <xbasic_types.h>  
#include <xstatus.h>
```

Functions

XStatus **XHwIcap_DeviceReadFrame** (**XHwIcap** *InstancePtr, **Xint32** Block, **Xint32** MajorFrame, **Xint32** MinorFrame)

Function Documentation

```
XStatus ( XHwIcap * InstancePtr,  
XHwIcap_DeviceReadFrame Block,  
                Xint32 MajorFrame,  
                Xint32 MinorFrame  
        )
```

Reads one frame from the device and puts it in the storage buffer.

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Block - Block Address (XHI_FAR_CLB_BLOCK, XHI_FAR_BRAM_BLOCK, XHI_FAR_BRAM_INT_BLOCK)

MajorFrame - selects the column

MinorFrame - selects frame inside column

Returns:

XST_SUCCESS, XST_BUFFER_TOO_SMALL or XST_INVALID_PARAM.

Note:

None.

hwicap/v1_00_a/src/ xhwicap_device_write_frame.c File Reference

Detailed Description

This file contains the function that writes the frame stored in the bram storage buffer to the device (ICAP).

Note:

none.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a	bjb	11/20/03	First release
-------	-----	----------	---------------

```
#include "xhwicap.h"  
#include "xhwicap_i.h"  
#include <xbasic_types.h>  
#include <xstatus.h>
```

Functions

XStatus XHwIcap_DeviceWriteFrame (**XHwIcap** *InstancePtr, **Xint32** Block, **Xint32** MajorFrame, **Xint32** MinorFrame)

Function Documentation

```
XStatus ( XHwIcap * InstancePtr,  
XHwIcap_DeviceWriteFrame Block,  
 Xint32 MajorFrame,  
 Xint32 MinorFrame  
 )
```

Writes one frame from the storage buffer and puts it in the device (ICAP).

Parameters:

InstancePtr - a pointer to the **XHwIcap** instance to be worked on.

Block - Block Address (XHI_FAR_CLB_BLOCK, XHI_FAR_BRAM_BLOCK, XHI_FAR_BRAM_INT_BLOCK)

MajorFrame - selects the column

MinorFrame - selects frame inside column

Returns:

XST_SUCCESS, XST_BUFFER_TOO_SMALL or XST_INVALID_PARAM.

Note:

None.

hwicap/v1_00_a/src/xhwicap_set_configuration. c File Reference

Detailed Description

This file contains the function that loads a partial bitstream located in system memory into the device (ICAP).

Note:

none.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a bjb 11/20/03 First release

```
#include "xhwicap.h"  
#include "xhwicap_i.h"  
#include <xbasic_types.h>  
#include <xstatus.h>
```

Functions

[XStatus XHwIcap_SetConfiguration](#) ([XHwIcap](#) *InstancePtr, [Xuint32](#) *Data, [Xuint32](#) Size)

Function Documentation

```
XStatus ( XHwIcap *  
XHwIcap_SetConfiguration InstancePtr,  
                Xuint32 * Data,  
                Xuint32 Size  
                )
```

Loads a partial bitstream from system memory.

Parameters:

- InstancePtr* - a pointer to the **XHwIcap** instance to be worked on.
- Data* - Address of the data representing the partial bitstream
- Size* - the size of the partial bitstream in 32 bit words.

Returns:

XST_SUCCESS, XST_BUFFER_TOO_SMALL or XST_INVALID_PARAM.

Note:

None.

iic/v1_01_d/src/xiic.h File Reference

Detailed Description

XIic is the driver for an IIC master or slave device.

In order to reduce the memory requirements of the driver it is partitioned such that there are optional parts of the driver. Slave, master, and multimaster features are optional such that these files are not required. In order to use the slave and multimaster features of the driver, the user must call functions (XIic_SlaveInclude and XIic_MultiMasterInclude) to dynamically include the code . These functions may be called at any time.

General Purpose Output The IIC hardware provides a General Purpose Output Register that allows the user to connect general purpose outputs to devices, such as a write protect, for an EEPROM. This register is parameterizable in the hardware such that there could be zero bits in this register and in this case it will cause a bus error if read or written.

Bus Throttling

The IIC hardware provides bus throttling which allows either the device, as either a master or a slave, to stop the clock on the IIC bus. This feature allows the software to perform the appropriate processing for each interrupt without an unreasonable response restriction. With this design, it is important for the user to understand the implications of bus throttling.

Repeated Start

An application can send multiple messages, as a master, to a slave device and re-acquire the IIC bus each time a message is sent. The repeated start option allows the application to send multiple messages without re-acquiring the IIC bus for each message. This feature also could cause the application to lock up, or monopolize the IIC bus, should repeated start option be enabled and sequences of messages never end (periodic data collection). Also when repeated start is not disabled before the last master message is sent or received, will leave the bus captive to the master, but unused.

Addressing

The IIC hardware is parameterized such that it can be built for 7 or 10 bit addresses. The driver provides the ability to control which address size is sent in messages as a master to a slave device. The address size which the hardware responds to as a slave is parameterized as 7 or 10 bits but fixed by the hardware build.

Addresses are represented as hex values with no adjustment for the data direction bit as the software manages address bit placement. This is especially important as the bit placement is not handled the same depending on which options are used such as repeated start and 7 vs 10 bit addressing.

Data Rates

The IIC hardware is parameterized such that it can be built to support data rates from DC to 400KBit. The frequency of the interrupts which occur is proportional to the data rate.

Polled Mode Operation

This driver does not provide a polled mode of operation primarily because polled mode which is non-blocking is difficult with the

amount of interaction with the hardware that is necessary.

Interrupts

The device has many interrupts which allow IIC data transactions as well as bus status processing to occur.

The interrupts are divided into two types, data and status. Data interrupts indicate data has been received or transmitted while the status interrupts indicate the status of the IIC bus. Some of the interrupts, such as Not Addressed As Slave and Bus Not Busy, are only used when these specific events must be recognized as opposed to being enabled at all times.

Many of the interrupts are not a single event in that they are continuously present such that they must be disabled after recognition or when undesired. Some of these interrupts, which are data related, may be acknowledged by the software by reading or writing data to the appropriate register, or must be disabled. The following interrupts can be continuous rather than single events.

- Data Transmit Register Empty/Transmit FIFO Empty
- Data Receive Register Full/Receive FIFO
- Transmit FIFO Half Empty
- Bus Not Busy
- Addressed As Slave
- Not Addressed As Slave

The following interrupts are not passed directly to the application thru the status callback. These are only used internally for the driver processing and may result in the receive and send handlers being called to indicate completion of an operation. The following interrupts are data related rather than status.

- Data Transmit Register Empty/Transmit FIFO Empty
- Data Receive Register Full/Receive FIFO
- Transmit FIFO Half Empty
- Slave Transmit Complete

Interrupt To Event Mapping

The following table provides a mapping of the interrupts to the events which are passed to the status handler and the intended role (master or slave) for the event. Some interrupts can cause multiple events which are combined together into a single status event such as XII_MASTER_WRITE_EVENT and XII_GENERAL_CALL_EVENT

Interrupt	Event(s)	Role
Arbitration Lost Interrupt	XII_ARB_LOST_EVENT	Master
Transmit Error	XII_SLAVE_NO_ACK_EVENT	Master
IIC Bus Not Busy	XII_BUS_NOT_BUSY_EVENT	Master
Addressed As Slave	XII_MASTER_READ_EVENT,	Slave
	XII_MASTER_WRITE_EVENT,	Slave
	XII_GENERAL_CALL_EVENT	Slave

Not Addressed As Slave Interrupt

The Not Addressed As Slave interrupt is not passed directly to the application thru the status callback. It is used to determine the end of a message being received by a slave when there was no stop condition (repeated start). It will cause the receive handler to be called to indicate completion of the operation.

RTOS Independence

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01a	rfp	10/19/01	release
1.01c	ecm	12/05/02	new rev
1.01d	jhl	10/08/03	Added general purpose output feature

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xipif_v1_23_b.h"  
#include "xiic_1.h"
```

Data Structures

```
struct XIic  
struct XIic_Config  
struct XIicStats
```

Configuration options

The following options may be specified or retrieved for the device and enable/disable additional features of the IIC bus. Each of the options are bit fields such that more than one may be specified.

```
#define XII_GENERAL_CALL_OPTION  
#define XII_REPEATED_START_OPTION  
#define XII_SEND_10_BIT_OPTION
```

Status events

The following status events occur during IIC bus processing and are passed to the status callback. Each event is only valid during the appropriate processing of the IIC bus. Each of these events are bit fields such that more than one may be specified.

```
#define XII_BUS_NOT_BUSY_EVENT  
#define XII_ARB_LOST_EVENT  
#define XII_SLAVE_NO_ACK_EVENT  
#define XII_MASTER_READ_EVENT  
#define XII_MASTER_WRITE_EVENT  
#define XII_GENERAL_CALL_EVENT
```

Defines

```
#define XII_ADDR_TO_SEND_TYPE  
#define XII_ADDR_TO_RESPOND_TYPE
```

Typedefs

```
typedef void(* XIic_Handler)(void *CallBackRef, int ByteCount)  
typedef void(* XIic_StatusHandler)(void *CallBackRef, XStatus StatusEvent)
```

Functions

```
XStatus XIic_Initialize (XIic *InstancePtr, Xuint16 DeviceId)
XStatus XIic_Start (XIic *InstancePtr)
XStatus XIic_Stop (XIic *InstancePtr)
    void XIic_Reset (XIic *InstancePtr)
XStatus XIic_SetAddress (XIic *InstancePtr, int AddressType, int Address)
Xuint16 XIic_GetAddress (XIic *InstancePtr, int AddressType)
XStatus XIic_SetGpOutput (XIic *InstancePtr, Xuint8 OutputValue)
XStatus XIic_GetGpOutput (XIic *InstancePtr, Xuint8 *OutputValuePtr)
Xboolean XIic_IsSlave (XIic *InstancePtr)
    void XIic_SetRecvHandler (XIic *InstancePtr, void *CallBackRef, XIic_Handler FuncPtr)
    void XIic_SetSendHandler (XIic *InstancePtr, void *CallBackRef, XIic_Handler FuncPtr)
    void XIic_SetStatusHandler (XIic *InstancePtr, void *CallBackRef, XIic_StatusHandler FuncPtr)
XIic_Config * XIic_LookupConfig (Xuint16 DeviceId)
    void XIic_InterruptHandler (void *InstancePtr)
XStatus XIic_MasterRecv (XIic *InstancePtr, Xuint8 *RxMsgPtr, int ByteCount)
XStatus XIic_MasterSend (XIic *InstancePtr, Xuint8 *TxMsgPtr, int ByteCount)
    void XIic_GetStats (XIic *InstancePtr, XIicStats *StatsPtr)
    void XIic_ClearStats (XIic *InstancePtr)
XStatus XIic_SelfTest (XIic *InstancePtr)
    void XIic_SetOptions (XIic *InstancePtr, Xuint32 Options)
Xuint32 XIic_GetOptions (XIic *InstancePtr)
    void XIic_MultiMasterInclude (void)
```

Define Documentation

#define

XII_ADDR_TO_RESPOND_TYPE

this device's bus address when
slave

#define

XII_ADDR_TO_SEND_TYPE

bus address of slave
device

#define

XII_ARB_LOST_EVENT

XII_BUS_NOT_BUSY_EVENT	bus transitioned to not busy
XII_ARB_LOST_EVENT	arbitration was lost
XII_SLAVE_NO_ACK_EVENT	slave did not acknowledge data (had error)
XII_MASTER_READ_EVENT	master reading from slave
XII_MASTER_WRITE_EVENT	master writing to slave
XII_GENERAL_CALL_EVENT	general call to all slaves

#define**XII_BUS_NOT_BUSY_EVENT**

XII_BUS_NOT_BUSY_EVENT	bus transitioned to not busy
XII_ARB_LOST_EVENT	arbitration was lost
XII_SLAVE_NO_ACK_EVENT	slave did not acknowledge data (had error)
XII_MASTER_READ_EVENT	master reading from slave
XII_MASTER_WRITE_EVENT	master writing to slave
XII_GENERAL_CALL_EVENT	general call to all slaves

#define**XII_GENERAL_CALL_EVENT**

XII_BUS_NOT_BUSY_EVENT	bus transitioned to not busy
XII_ARB_LOST_EVENT	arbitration was lost
XII_SLAVE_NO_ACK_EVENT	slave did not acknowledge data (had error)
XII_MASTER_READ_EVENT	master reading from slave
XII_MASTER_WRITE_EVENT	master writing to slave
XII_GENERAL_CALL_EVENT	general call to all slaves

#define**XII_GENERAL_CALL_OPTION**

XII_GENERAL_CALL_OPTION	The general call option allows an IIC slave to recognized the general call address. The status handler is called as usual indicating the device has been addressed as a slave with a general call. It is the application's responsibility to perform any special processing for the general call.
XII_REPEATED_START_OPTION	The repeated start option allows multiple messages to be sent/received on the IIC bus without rearbitrating for the bus. The messages are sent as a series of messages such that the option must be enabled before the 1st message of the series, to prevent an stop condition from being generated on the bus, and disabled before the last message of the series, to allow the stop condition to be generated.
XII_SEND_10_BIT_OPTION	The send 10 bit option allows 10 bit addresses to be sent on the bus when the device is a master. The device can be configured to respond as to 7 bit addresses even though it may be communicating with other devices that support 10 bit addresses. When this option is not enabled, only 7 bit addresses are sent on the bus.

#define**XII_MASTER_READ_EVENT**

XII_BUS_NOT_BUSY_EVENT	bus transitioned to not busy
XII_ARB_LOST_EVENT	arbitration was lost
XII_SLAVE_NO_ACK_EVENT	slave did not acknowledge data (had error)
XII_MASTER_READ_EVENT	master reading from slave
XII_MASTER_WRITE_EVENT	master writing to slave
XII_GENERAL_CALL_EVENT	general call to all slaves

#define

XII_MASTER_WRITE_EVENT

XII_BUS_NOT_BUSY_EVENT	bus transitioned to not busy
XII_ARB_LOST_EVENT	arbitration was lost
XII_SLAVE_NO_ACK_EVENT	slave did not acknowledge data (had error)
XII_MASTER_READ_EVENT	master reading from slave
XII_MASTER_WRITE_EVENT	master writing to slave
XII_GENERAL_CALL_EVENT	general call to all slaves

#define

XII_REPEATED_START_OPTION

XII_GENERAL_CALL_OPTION	The general call option allows an IIC slave to recognized the general call address. The status handler is called as usual indicating the device has been addressed as a slave with a general call. It is the application's responsibility to perform any special processing for the general call.
-------------------------	---

XII_REPEATED_START_OPTION	The repeated start option allows multiple messages to be sent/received on the IIC bus without rearbitrating for the bus. The messages are sent as a series of messages such that the option must be enabled before the 1st message of the series, to prevent an stop condition from being generated on the bus, and disabled before the last message of the series, to allow the stop condition to be generated.
---------------------------	--

XII_SEND_10_BIT_OPTION	The send 10 bit option allows 10 bit addresses to be sent on the bus when the device is a master. The device can be configured to respond as to 7 bit addresses even though it may be communicating with other devices that support 10 bit addresses. When this option is not enabled, only 7 bit addresses are sent on the bus.
------------------------	--

#define

XII_SEND_10_BIT_OPTION

XII_GENERAL_CALL_OPTION	The general call option allows an IIC slave to recognized the general call address. The status handler is called as usual indicating the device has been addressed as a slave with a general call. It is the application's responsibility to perform any special processing for the general call.
XII_REPEATED_START_OPTION	The repeated start option allows multiple messages to be sent/received on the IIC bus without rearbitrating for the bus. The messages are sent as a series of messages such that the option must be enabled before the 1st message of the series, to prevent an stop condition from being generated on the bus, and disabled before the last message of the series, to allow the stop condition to be generated.
XII_SEND_10_BIT_OPTION	The send 10 bit option allows 10 bit addresses to be sent on the bus when the device is a master. The device can be configured to respond as to 7 bit addresses even though it may be communicating with other devices that support 10 bit addresses. When this option is not enabled, only 7 bit addresses are sent on the bus.

#define

XII_SLAVE_NO_ACK_EVENT

XII_BUS_NOT_BUSY_EVENT	bus transitioned to not busy
XII_ARB_LOST_EVENT	arbitration was lost
XII_SLAVE_NO_ACK_EVENT	slave did not acknowledge data (had error)
XII_MASTER_READ_EVENT	master reading from slave
XII_MASTER_WRITE_EVENT	master writing to slave
XII_GENERAL_CALL_EVENT	general call to all slaves

Typedef Documentation

```
typedef void(* XIic_Handler)(void *CallbackRef, int
ByteCount)
```

This callback function data type is defined to handle the asynchronous processing of sent and received data of the IIC driver. The application using this driver is expected to define a handler of this type to support interrupt driven mode. The handlers are called in an interrupt context such that minimal processing should be performed. The handler data type is utilized for both send and receive handlers.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked. Its type is unimportant to the driver component, so it is a void pointer.

ByteCount indicates the number of bytes remaining to be sent or received. A value of zero indicates that the requested number of bytes were sent or received.

```
typedef void(* XIic_StatusHandler)(void *CallbackRef, XStatus  
StatusEvent)
```

This callback function data type is defined to handle the asynchronous processing of status events of the IIC driver. The application using this driver is expected to define a handler of this type to support interrupt driven mode. The handler is called in an interrupt context such that minimal processing should be performed.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked. Its type is unimportant to the driver component, so it is a void pointer.

StatusEvent indicates one or more status events that occurred. See the definition of the status events above.

Function Documentation

```
void  
XIic_ClearStats ( XIic *  
InstancePtr )
```

Clears the statistics for the IIC device by zeroing all counts.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint16  
XIic_GetAddress ( XIic *  
InstancePtr,  
int AddressType  
)
```

This function gets the addresses for the IIC device driver. The addresses include the device address that the device responds to as a slave, or the slave address to communicate with on the bus. The address returned has the same format whether 7 or 10 bits.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

AddressType indicates which address, the address which this responds to on the IIC bus as a slave, or the slave address to communicate with when this device is a master. One of the following values must be contained in this argument.

XII_ADDRESS_TO_SEND_TYPE	slave being addressed as a master
XII_ADDRESS_TO_RESPOND_TYPE	slave address to respond to as a slave

If neither of the two valid arguments are used, the function returns the address of the slave device

Returns:

The address retrieved.

Note:

None.

```
XStatus XIic_GetGpOutput ( XIic * InstancePtr,  
                        Xuint8 * OutputValuePtr  
                        )
```

This function gets the contents of the General Purpose Output register for the IIC device driver. Note that the number of bits in this register is parameterizable in the hardware such that it may not exist. This function checks to ensure that it does exist to prevent bus errors.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

OutputValuePtr contains the value which was read from the register.

Returns:

A value indicating success, XST_SUCCESS, or XST_NO_FEATURE if the hardware is configured such that this register does not contain any bits to read or write.

The OutputValuePtr is also an output as it contains the value read.

Note:

None.

```
Xuint32 XIic_GetOptions ( XIic *  
                          InstancePtr  
                          )
```

This function gets the current options for the IIC device. Options control the how the device behaves on the IIC bus. See [SetOptions](#) for more information on options.

Parameters:

InstancePtr is a pointer to the [XIic](#) instance to be worked on.

Returns:

The options of the IIC device. See [xiic.h](#) for a list of available options.

Note:

Options enabled will have a 1 in its appropriate bit position.

```
void  
XIic_GetStats ( XIic *  
               InstancePtr,  
               XIicStats * StatsPtr  
             )
```

Gets a copy of the statistics for an IIC device.

Parameters:

InstancePtr is a pointer to the [XIic](#) instance to be worked on.

StatsPtr is a pointer to a [XIicStats](#) structure which will get a copy of current statistics.

Returns:

None.

Note:

None.

```
XStatus  
XIic_Initialize ( XIic *  
                 InstancePtr,  
                 Xuint16 DeviceId  
               )
```

Initializes a specific [XIic](#) instance. The initialization entails:

- Check the device has an entry in the configuration table.
- Initialize the driver to allow access to the device registers and initialize other subcomponents necessary for the operation of the device.
- Default options to:
 - 7-bit slave addressing
 - Send messages as a slave device
 - Repeated start off
 - General call recognition disabled
- Clear messageing and error statistics

The [XIic_Start\(\)](#) function must be called after this function before the device is ready to send and receive data on the IIC bus.

Before [XIic_Start\(\)](#) is called, the interrupt control must connect the ISR routine to the interrupt handler. This is done by the user, and not [XIic_Start\(\)](#) to allow the user to use an interrupt controller of their choice.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XIic** instance. Passing in a device id associates the generic **XIic** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS when successful
- XST_DEVICE_IS_STARTED indicates the device is started (i.e. interrupts enabled and messaging is possible). Must stop before re-initialization is allowed.

Note:

None.

```
void ( void * )  
XIic_InterruptHandler InstancePtr
```

This function is the interrupt handler for the **XIic** driver. This function should be connected to the interrupt system.

Only one interrupt source is handled for each interrupt allowing higher priority system interrupts quicker response time.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

None.

```
Xboolean ( XIic * )  
XIic_IsSlave InstancePtr
```

A function to determine if the device is currently addressed as a slave

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

XTRUE if the device is addressed as slave, and XFALSE otherwise.

Note:

None.

```
XIic_Config* ( Xuint16 )  
XIic_LookupConfig DeviceId
```

Looks up the device configuration based on the unique device ID. The table `IicConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for

Returns:

A pointer to the configuration data of the device, or XNULL if no match is found.

Note:

None.

```
XStatus          ( XIic *
XIic_MasterRecv  InstancePtr,
                  Xuint8 * RxMsgPtr,
                  int      ByteCount
                  )
```

This function receives data as a master from a slave device on the IIC bus. If the bus is busy, it will indicate so and then enable an interrupt such that the status handler will be called when the bus is no longer busy. The slave address which has been set with the [XIic_SetAddress\(\)](#) function is the address from which data is received. Receiving data on the bus performs a read operation.

Parameters:

InstancePtr is a pointer to the Iic instance to be worked on.

RxMsgPtr is a pointer to the data to be transmitted

ByteCount is the number of message bytes to be sent

Returns:

- o XST_SUCCESS indicates the message reception processes has been initiated.
- o XST_IIC_BUS_BUSY indicates the bus was in use and that the BusNotBusy interrupt is enabled which will update the EventStatus when the bus is no longer busy.
- o XST_IIC_GENERAL_CALL_ADDRESS indicates the slave address is set to the the general call address. This is not allowed for Master receive mode.

```
XStatus          ( XIic *
XIic_MasterSend  InstancePtr,
                  Xuint8 * TxMsgPtr,
                  int      ByteCount
                  )
```

This function sends data as a master on the IIC bus. If the bus is busy, it will indicate so and then enable an interrupt such that the status handler will be called when the bus is no longer busy. The slave address which has been set with the [XIic_SetAddress\(\)](#) function is the address to which the specific data is sent. Sending data on the bus performs a write operation.

Parameters:

InstancePtr points to the Iic instance to be worked on.

TxMsgPtr points to the data to be transmitted

ByteCount is the number of message bytes to be sent

Returns:

- XST_SUCCESS indicates the message transmission has been initiated.
- XST_IIC_BUS_BUSY indicates the bus was in use and that the BusNotBusy interrupt is enabled which will update the EventStatus when the bus is no longer busy.

Note:

None

```
void ( void )
XIic_MultiMasterInclude
```

This function includes multi-master code such that multi-master events are handled properly. Multi-master events include a loss of arbitration and the bus transitioning from busy to not busy. This function allows the multi-master processing to be optional. This function must be called prior to allowing any multi-master events to occur, such as after the driver is initialized.

Note:

None

```
void ( XIic * )
XIic_Reset InstancePtr
```

Resets the IIC device. Reset must only be called after the driver has been initialized. The configuration after this reset is as follows:

- Repeated start is disabled
- General call is disabled

The upper layer software is responsible for initializing and re-configuring (if necessary) and restarting the IIC device after the reset.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( XIic * )
XIic_SelfTest InstancePtr
```

Runs a limited self-test on the driver/device. The self-test is destructive in that a reset of the device is performed in order to check the reset values of the registers and to get the device into a known state. There is no loopback capabilities for the device such that this test does not send or receive data.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

XST_SUCCESS	No errors found
XST_IIC_STAND_REG_ERROR	One or more IIC regular registers did not zero on reset or read back correctly based on what was written

XST_IIC_TX_FIFO_REG_ERROR	to it One or more IIC parametrizable TX FIFO registers did not zero on reset or read back correctly based on what was written to it
XST_IIC_RX_FIFO_REG_ERROR	One or more IIC parametrizable RX FIFO registers did not zero on reset or read back correctly based on what was written to it
XST_IIC_STAND_REG_RESET_ERROR	A non parameterizable reg value after reset not valid
XST_IIC_TX_FIFO_REG_RESET_ERROR	Tx fifo, included in design, value after reset not valid
XST_IIC_RX_FIFO_REG_RESET_ERROR	Rx fifo, included in design, value after reset not valid
XST_IIC_TBA_REG_RESET_ERROR	10 bit addr, incl in design, value after reset not valid
XST_IIC_CR_READBACK_ERROR	Read of the control register didn't return value written
XST_IIC_DTR_READBACK_ERROR	Read of the data Tx reg didn't return value written
XST_IIC_DRR_READBACK_ERROR	Read of the data Receive reg didn't return value written
XST_IIC_ADR_READBACK_ERROR	Read of the data Tx reg didn't return value written
XST_IIC_TBA_READBACK_ERROR	Read of the 10 bit addr reg didn't return written value

Note:

Only the registers that have be included into the hardware design are tested, such as, 10-bit vs 7-bit addressing.

```

XStatus
XIic_SetAddress
    ( XIic *
      InstancePtr,
      int
      AddressType,
      int
      Address
    )

```

This function sets the bus addresses. The addresses include the device address that the device responds to as a slave, or the slave address to communicate with on the bus. The IIC device hardware is built to allow either 7 or 10 bit slave addressing only at build time rather than at run time. When this device is a master, slave addressing can be selected at run time to match addressing modes for other bus devices.

Addresses are represented as hex values with no adjustment for the data direction bit as the software manages address bit placement. Example: For a 7 address written to the device of 1010 011X where X is the transfer direction (send/recv), the address parameter for this function needs to be 01010011 or 0x53 where the correct bit allignment will be handled for 7 as well as 10 bit devices. This is especially important as the bit placement is not handled the same depending on which options are used such as repeated start.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

AddressType indicates which address is being modified; the address which this device responds to on the IIC bus as a slave, or the slave address to communicate with when this device is a master. One of the following values must be contained in this argument.

XII_ADDRESS_TO_SEND	Slave being addressed by a this master
XII_ADDRESS_TO_RESPOND	Address to respond to as a slave device

Address contains the address to be set; 7 bit or 10 bit address. A ten bit address must be within the range: 0 - 1023 and a 7 bit address must be within the range 0 - 127.

Returns:

XST_SUCCESS is returned if the address was successfully set, otherwise one of the following errors is returned.

- o XST_IIC_NO_10_BIT_ADDRESSING indicates only 7 bit addressing supported.
- o XST_INVALID_PARAM indicates an invalid parameter was specified.

Note:

Upper bits of 10-bit address is written only when current device is built as a ten bit device.

```
XStatus
XIic_SetGpOutput ( XIic *
                    InstancePtr,
                    Xuint8 OutputValue
                  )
```

This function sets the contents of the General Purpose Output register for the IIC device driver. Note that the number of bits in this register is parameterizable in the hardware such that it may not exist. This function checks to ensure that it does exist to prevent bus errors, but does not ensure that the number of bits in the register are sufficient for the value being written (won't cause a bus error).

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

OutputValue contains the value to be written to the register.

Returns:

A value indicating success, XST_SUCCESS, or XST_NO_FEATURE if the hardware is configured such that this register does not contain any bits to read or write.

Note:

None.

```
void
XIic_SetOptions ( XIic *
                  InstancePtr,
                  Xuint32 NewOptions
                )
```

This function sets the options for the IIC device driver. The options control how the device behaves relative to the IIC bus. If an option applies to how messages are sent or received on the IIC bus, it must be set prior to calling functions which send or receive data.

To set multiple options, the values must be ORed together. To not change existing options, read/modify/write with the current options using [XIic_GetOptions\(\)](#).

USAGE EXAMPLE:

Read/modify/write to enable repeated start:

```
Xuint8 Options;  
Options = XIic_GetOptions(&Iic);  
XIic_SetOptions(&Iic, Options | XII_REPEATED_START_OPTION);
```

Disabling General Call:

```
Options = XIic_GetOptions(&Iic);  
XIic_SetOptions(&Iic, Options &= ~XII_GENERAL_CALL_OPTION);
```

Parameters:

InstancePtr is a pointer to the [XIic](#) instance to be worked on.

NewOptions are the options to be set. See [xiic.h](#) for a list of the available options.

Returns:

None.

Note:

Sending or receiving messages with repeated start enabled, and then disabling repeated start, will not take effect until another master transaction is completed. i.e. After using repeated start, the bus will continue to be throttled after repeated start is disabled until a master transaction occurs allowing the IIC to release the bus.

Options enabled will have a 1 in its appropriate bit position.

```
void  
XIic_SetRecvHandler ( XIic *      InstancePtr,  
  
                    void *      CallbackRef,  
                    XIic_Handler FuncPtr  
                    )
```

Sets the receive callback function, the receive handler, which the driver calls when it finishes receiving data. The number of bytes used to signal when the receive is complete is the number of bytes set in the [XIic_Recv](#) function.

The handler executes in an interrupt context such that it must minimize the amount of processing performed such as transferring data to a thread context.

The number of bytes received is passed to the handler as an argument.

Parameters:

InstancePtr is a pointer to the [XIic](#) instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context ...

```
void  
XIic_SetSendHandler ( XIic * InstancePtr,  
                    void * CallbackRef,  
                    XIic_Handler FuncPtr  
                    )
```

Sets the send callback function, the send handler, which the driver calls when it receives confirmation of sent data. The handler executes in an interrupt context such that it must minimize the amount of processing performed such as transferring data to a thread context.

Parameters:

InstancePtr the pointer to the **XIic** instance to be worked on.

CallbackRef the upper layer callback reference passed back when the callback function is invoked.

FuncPtr the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context ...

```
void  
XIic_SetStatusHandler ( XIic * InstancePtr,  
                      void * CallbackRef,  
                      XIic_StatusHandler FuncPtr  
                      )
```

Sets the status callback function, the status handler, which the driver calls when it encounters conditions which are not data related. The handler executes in an interrupt context such that it must minimize the amount of processing performed such as transferring data to a thread context. The status events that can be returned are described in [xiic.h](#).

Parameters:

InstancePtr points to the **XIic** instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context ...

XStatus (**XIic ***)
XIic_Start *InstancePtr*

This function starts the IIC device and driver by enabling the proper interrupts such that data may be sent and received on the IIC bus. This function must be called before the functions to send and receive data.

Before **XIic_Start()** is called, the interrupt control must connect the ISR routine to the interrupt handler. This is done by the user, and not **XIic_Start()** to allow the user to use an interrupt controller of their choice.

Start enables:

- IIC device
- Interrupts:
 - Addressed as slave to allow messages from another master
 - Arbitration Lost to detect Tx arbitration errors
 - Global IIC interrupt within the IPIF interface

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

XST_SUCCESS always

Note:

The device interrupt is connected to the interrupt controller, but no "messaging" interrupts are enabled. Addressed as Slave is enabled to reception of messages when this devices address is written to the bus. The correct messaging interrupts are enabled when sending or receiving via the **IicSend()** and **IicRecv()** functions. No action is required by the user to control any IIC interrupts as the driver completely manages all 8 interrupts. Start and Stop control the ability to use the device. Stopping the device completely stops all device interrupts from the processor.

XStatus (**XIic ***)
XIic_Stop *InstancePtr*

This function stops the IIC device and driver such that data is no longer sent or received on the IIC bus. This function stops the device by disabling interrupts. This function only disables interrupts within the device such that the caller is responsible for disconnecting the interrupt handler of the device from the interrupt source and disabling interrupts at other levels.

Due to bus throttling that could hold the bus between messages when using repeated start option, stop will not occur when the device is actively sending or receiving data from the IIC bus or the bus is being throttled by this device, but instead return XST_IIC_BUS_BUSY.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

- XST_SUCCESS indicates all IIC interrupts are disabled. No messages can be received or transmitted until **XIic_Start()** is called.
- XST_IIC_BUS_BUSY indicates this device is currently engaged in message traffic and cannot be stopped.

Note:

None.

Generated on 24 Jun 2004 for Xilinx Device Drivers

Xlic Struct Reference

```
#include <xiic.h>
```

Detailed Description

The XIic driver instance data. The user is required to allocate a variable of this type for every IIC device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [iic/v1_01_d/src/xiic.h](#)
-

iic/v1_01_d/src/xiic_I.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xiic.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	05/07/02	First release
1.01c	ecm	12/05/02	new rev
1.01d	jhl	10/08/03	Added general purpose output feature

```
#include "xbasic\_types.h"
```

Functions

unsigned [XIic_Recv](#) ([Xuint32](#) BaseAddress, [Xuint8](#) Address, [Xuint8](#) *BufferPtr, unsigned ByteCount)

unsigned [XIic_Send](#) ([Xuint32](#) BaseAddress, [Xuint8](#) Address, [Xuint8](#) *BufferPtr, unsigned ByteCount)

Function Documentation

```
unsigned      ( Xuint32  
Xlic_Recv    BaseAddress,  
             Xuint8 Address,  
             Xuint8 * BufferPtr,  
             unsigned ByteCount  
             )
```

Receive data as a master on the IIC bus. This function receives the data using polled I/O and blocks until the data has been received. It only supports 7 bit addressing and non-repeated start modes of operation. The user is responsible for ensuring the bus is not busy if multiple masters are present on the bus.

Parameters:

BaseAddress contains the base address of the IIC device.

Address contains the 7 bit IIC address of the device to send the specified data to.

BufferPtr points to the data to be sent.

ByteCount is the number of bytes to be sent.

Returns:

The number of bytes received.

Note:

None

```
unsigned      ( Xuint32  
Xlic_Send    BaseAddress,  
             Xuint8 Address,  
             Xuint8 * BufferPtr,  
             unsigned ByteCount  
             )
```

Send data as a master on the IIC bus. This function sends the data using polled I/O and blocks until the data has been sent. It only supports 7 bit addressing and non-repeated start modes of operation. The user is responsible for ensuring the bus is not busy if multiple masters are present on the bus.

Parameters:

BaseAddress contains the base address of the IIC device.

Address contains the 7 bit IIC address of the device to send the specified data to.

BufferPtr points to the data to be sent.

ByteCount is the number of bytes to be sent.

Returns:

The number of bytes sent.

Note:

None

Xlic_Config Struct Reference

```
#include <xiic.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xboolean Has10BitAddr](#)

[Xuint8 GpOutWidth](#)

Field Documentation

[Xuint32 XIic_Config::](#)

BaseAddress

Device base
address

[Xuint16 XIic_Config::](#)

DeviceId

Unique ID of
device

Xuint8 XIic_Config::

GpOutWidth

number of bits in general purpose
output

Xboolean XIic_Config::

Has10BitAddr

does device have 10 bit address
decoding

The documentation for this struct was generated from the following file:

- [iic/v1_01_d/src/xiic.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

XlicStats Struct Reference

```
#include <xiic.h>
```

Detailed Description

XIic statistics

Data Fields

Xuint8 ArbitrationLost

Xuint8 RepeatedStarts

Xuint8 BusBusy

Xuint8 RecvBytes

Xuint8 RecvInterrupts

Xuint8 SendBytes

Xuint8 SendInterrupts

Xuint8 TxErrors

Xuint8 IicInterrupts

Field Documentation

Xuint8 XIicStats::

ArbitrationLost

Number of times arbitration was lost

Xuint8 XIicStats::

BusBusy

Number of times bus busy status returned

Xuint8 XIicStats::

IicInterrupts

Number of IIC (device) interrupts

Xuint8 XIicStats::

RecvBytes

Number of bytes received

Xuint8 XIicStats::

RecvInterrupts

Number of receive interrupts

Xuint8 XIicStats::

RepeatedStarts

Number of repeated starts

Xuint8 XIicStats::

SendBytes

Number of transmit bytes received

Xuint8 XIicStats::

SendInterrupts

Number of transmit interrupts

Xuint8 XIicStats::

TxErrors

Number of transmit errors (no
ack)

The documentation for this struct was generated from the following file:

- [iic/v1_01_d/src/xiic.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

iic/v1_01_d/src/xiic_stats.c File Reference

Detailed Description

Contains statistics functions for the **XIic** component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01b	jhl	3/26/02	repartitioned the driver
1.01c	ecm	12/05/02	new rev

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

Functions

```
void XIic_GetStats (XIic *InstancePtr, XIicStats *StatsPtr)  
void XIic_ClearStats (XIic *InstancePtr)
```

Function Documentation

```
void  
XIic_ClearStats ( XIic *  
                 InstancePtr )
```

Clears the statistics for the IIC device by zeroing all counts.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

None.

Note:

None.

```
void  
XIic_GetStats ( XIic *  
               InstancePtr,  
               XIicStats * StatsPtr  
             )
```

Gets a copy of the statistics for an IIC device.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

StatsPtr is a pointer to a **XIicStats** structure which will get a copy of current statistics.

Returns:

None.

Note:

None.

iic/v1_01_d/src/xiic_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between **XIic** components. The identifiers in this file are not intended for use external to the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01a	rfp	10/19/01	release
1.01c	ecm	12/05/02	new rev

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

Variables

XIic_Config XIic_ConfigTable []

Variable Documentation

XIic_Config XIic_ConfigTable ()
[]

The IIC configuration table, sized by the number of instances defined in **xparameters.h**.

iic/v1_01_d/src/xiic_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of IIC devices in the system. Each IIC device should have an entry in this table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01a	rfp	10/19/01	release
1.01c	ecm	12/05/02	new rev
1.01d	jhl	10/08/03	Added general purpose output feature

```
#include "xiic.h"  
#include "xparameters.h"
```

Variables

[XIic_Config XIic_ConfigTable](#) [XPAR_XIIC_NUM_INSTANCES]

Variable Documentation

[XIic_Config XIic_ConfigTable](#)
[XPAR_XIIC_NUM_INSTANCES]

The IIC configuration table, sized by the number of instances defined in [xparameters.h](#).

iic/v1_01_d/src/xiic.c File Reference

Detailed Description

Contains required functions for the **XIic** component. See [xiic.h](#) for more information on the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01a	rfp	10/19/01	release
1.01c	ecm	12/05/02	new rev
1.01c	rmm	05/14/03	Fixed diab compiler warnings relating to asserts.
1.01d	jhl	10/08/03	Added general purpose output feature

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"  
#include "xparameters.h"
```

Functions

```
XStatus XIic_Initialize (XIic *InstancePtr, Xuint16 DeviceId)  
XStatus XIic_Start (XIic *InstancePtr)  
XStatus XIic_Stop (XIic *InstancePtr)  
void XIic_Reset (XIic *InstancePtr)  
XStatus XIic_SetAddress (XIic *InstancePtr, int AddressType, int Address)  
Xuint16 XIic_GetAddress (XIic *InstancePtr, int AddressType)  
XStatus XIic_SetGpOutput (XIic *InstancePtr, Xuint8 OutputValue)  
XStatus XIic_GetGpOutput (XIic *InstancePtr, Xuint8 *OutputValuePtr)  
Xboolean XIic_IsSlave (XIic *InstancePtr)  
void XIic_SetRecvHandler (XIic *InstancePtr, void *CallBackRef, XIic_Handler FuncPtr)  
void XIic_SetSendHandler (XIic *InstancePtr, void *CallBackRef, XIic_Handler FuncPtr)  
void XIic_SetStatusHandler (XIic *InstancePtr, void *CallBackRef, XIic_StatusHandler FuncPtr)  
XIic_Config * XIic_LookupConfig (Xuint16 DeviceId)
```

Function Documentation

```
Xuint16  
XIic_GetAddress (XIic *  
                  InstancePtr,  
                  int     AddressType  
                  )
```

This function gets the addresses for the IIC device driver. The addresses include the device address that the device responds to as a slave, or the slave address to communicate with on the bus. The address returned has the same format whether 7 or 10 bits.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

AddressType indicates which address, the address which this responds to on the IIC bus as a slave, or the slave address to communicate with when this device is a master. One of the following values must be contained in this argument.

XII_ADDRESS_TO_SEND_TYPE	slave being addressed as a master
XII_ADDRESS_TO_RESPOND_TYPE	slave address to respond to as a slave

If neither of the two valid arguments are used, the function returns the address of the slave device

Returns:

The address retrieved.

Note:

None.

```
XStatus
XIic_GetGpOutput ( XIic * InstancePtr,
                    Xuint8 * OutputValuePtr
                  )
```

This function gets the contents of the General Purpose Output register for the IIC device driver. Note that the number of bits in this register is parameterizable in the hardware such that it may not exist. This function checks to ensure that it does exist to prevent bus errors.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

OutputValuePtr contains the value which was read from the register.

Returns:

A value indicating success, XST_SUCCESS, or XST_NO_FEATURE if the hardware is configured such that this register does not contain any bits to read or write.

The OutputValuePtr is also an output as it contains the value read.

Note:

None.

```
XStatus
XIic_Initialize ( XIic * InstancePtr,
                  Xuint16 DeviceId
                  )
```

Initializes a specific **XIic** instance. The initialization entails:

- Check the device has an entry in the configuration table.
- Initialize the driver to allow access to the device registers and initialize other subcomponents necessary for the operation of the device.
- Default options to:
 - 7-bit slave addressing
 - Send messages as a slave device
 - Repeated start off
 - General call recognition disabled
- Clear messageing and error statistics

The **XIic_Start()** function must be called after this function before the device is ready to send and receive data on the IIC bus.

Before **XIic_Start()** is called, the interrupt control must connect the ISR routine to the interrupt handler. This is done by the user, and not **XIic_Start()** to allow the user to use an interrupt controller of their choice.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XIic** instance. Passing in a device id associates the generic **XIic** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS when successful
- XST_DEVICE_IS_STARTED indicates the device is started (i.e. interrupts enabled and messaging is possible). Must stop before re-initialization is allowed.

Note:

None.

Xboolean (**XIic** *)
XIic_IsSlave *InstancePtr*

A function to determine if the device is currently addressed as a slave

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

XTRUE if the device is addressed as slave, and XFALSE otherwise.

Note:

None.

XIic_Config* (**Xuint16**)
XIic_LookupConfig *DeviceId*

Looks up the device configuration based on the unique device ID. The table `IicConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for

Returns:

A pointer to the configuration data of the device, or XNULL if no match is found.

Note:

None.

```
void ( Xlic * )  
Xlic_Reset InstancePtr
```

Resets the IIC device. Reset must only be called after the driver has been initialized. The configuration after this reset is as follows:

- Repeated start is disabled
- General call is disabled

The upper layer software is responsible for initializing and re-configuring (if necessary) and restarting the IIC device after the reset.

Parameters:

InstancePtr is a pointer to the [Xlic](#) instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( Xlic * InstancePtr,  
Xlic_SetAddress int AddressType,  
int Address )
```

This function sets the bus addresses. The addresses include the device address that the device responds to as a slave, or the slave address to communicate with on the bus. The IIC device hardware is built to allow either 7 or 10 bit slave addressing only at build time rather than at run time. When this device is a master, slave addressing can be selected at run time to match addressing modes for other bus devices.

Addresses are represented as hex values with no adjustment for the data direction bit as the software manages address bit placement. Example: For a 7 address written to the device of 1010 011X where X is the transfer direction (send/recv), the address parameter for this function needs to be 01010011 or 0x53 where the correct bit allignment will be handled for 7 as well as 10 bit devices. This is especially important as the bit placement is not handled the same depending on which options are used such as repeated start.

Parameters:

InstancePtr is a pointer to the [Xlic](#) instance to be worked on.

AddressType indicates which address is being modified; the address which this device responds to on the IIC bus as a slave, or the slave address to communicate with when this device is a master. One of the following values must be contained in this argument.

XII_ADDRESS_TO_SEND	Slave being addressed by a this master
XII_ADDRESS_TO_RESPOND	Address to respond to as a slave device

Address contains the address to be set; 7 bit or 10 bit address. A ten bit address must be within the range: 0 - 1023 and a 7 bit address must be within the range 0 - 127.

Returns:

XST_SUCCESS is returned if the address was successfully set, otherwise one of the following errors is returned.

- o XST_IIC_NO_10_BIT_ADDRESSING indicates only 7 bit addressing supported.
- o XST_INVALID_PARAM indicates an invalid parameter was specified.

Note:

Upper bits of 10-bit address is written only when current device is built as a ten bit device.

```
XStatus
XIic_SetGpOutput ( XIic * InstancePtr,
                    Xuint8 OutputValue
                  )
```

This function sets the contents of the General Purpose Output register for the IIC device driver. Note that the number of bits in this register is parameterizable in the hardware such that it may not exist. This function checks to ensure that it does exist to prevent bus errors, but does not ensure that the number of bits in the register are sufficient for the value being written (won't cause a bus error).

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

OutputValue contains the value to be written to the register.

Returns:

A value indicating success, XST_SUCCESS, or XST_NO_FEATURE if the hardware is configured such that this register does not contain any bits to read or write.

Note:

None.

```
void
XIic_SetRecvHandler ( XIic * InstancePtr,
                       void * CallbackRef,
                       XIic_Handler FuncPtr
                     )
```

Sets the receive callback function, the receive handler, which the driver calls when it finishes receiving data. The number of bytes used to signal when the receive is complete is the number of bytes set in the `XIic_Recv` function.

The handler executes in an interrupt context such that it must minimize the amount of processing performed such as transferring data to a thread context.

The number of bytes received is passed to the handler as an argument.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context ...

```
void  
XIic_SetSendHandler ( XIic * InstancePtr,  
  
void * CallbackRef,  
XIic_Handler FuncPtr  
)
```

Sets the send callback function, the send handler, which the driver calls when it receives confirmation of sent data. The handler executes in an interrupt context such that it must minimize the amount of processing performed such as transferring data to a thread context.

Parameters:

InstancePtr the pointer to the **XIic** instance to be worked on.

CallbackRef the upper layer callback reference passed back when the callback function is invoked.

FuncPtr the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context ...

```
void  
XIic_SetStatusHandler ( XIic * InstancePtr,  
  
void * CallbackRef,  
XIic_StatusHandler FuncPtr  
)
```

Sets the status callback function, the status handler, which the driver calls when it encounters conditions which are not data related. The handler executes in an interrupt context such that it must minimize the amount of processing performed such as transferring data to a thread context. The status events that can be returned are described in [xiic.h](#).

Parameters:

- InstancePtr* points to the [XIic](#) instance to be worked on.
- CallbackRef* is the upper layer callback reference passed back when the callback function is invoked.
- FuncPtr* is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context ...

```
XStatus ( XIic * )  
XIic_Start InstancePtr
```

This function starts the IIC device and driver by enabling the proper interrupts such that data may be sent and received on the IIC bus. This function must be called before the functions to send and receive data.

Before [XIic_Start\(\)](#) is called, the interrupt control must connect the ISR routine to the interrupt handler. This is done by the user, and not [XIic_Start\(\)](#) to allow the user to use an interrupt controller of their choice.

Start enables:

- IIC device
- Interrupts:
 - Addressed as slave to allow messages from another master
 - Arbitration Lost to detect Tx arbitration errors
 - Global IIC interrupt within the IPIF interface

Parameters:

InstancePtr is a pointer to the [XIic](#) instance to be worked on.

Returns:

XST_SUCCESS always

Note:

The device interrupt is connected to the interrupt controller, but no "messaging" interrupts are enabled. Addressed as Slave is enabled to reception of messages when this devices address is written to the bus. The correct messaging interrupts are enabled when sending or receiving via the [IicSend\(\)](#) and [IicRecv\(\)](#) functions. No action is required by the user to control any IIC interrupts as the driver completely manages all 8 interrupts. Start and Stop control the ability to use the device. Stopping the device completely stops all device interrupts from the processor.

```
XStatus ( XIic * )  
XIic_Stop InstancePtr
```

This function stops the IIC device and driver such that data is no longer sent or received on the IIC bus. This function stops the device by disabling interrupts. This function only disables interrupts within the device such that the caller is responsible for disconnecting the interrupt handler of the device from the interrupt source and disabling interrupts at other levels.

Due to bus throttling that could hold the bus between messages when using repeated start option, stop will not occur when the device is actively sending or receiving data from the IIC bus or the bus is being throttled by this device, but instead return XST_IIC_BUS_BUSY.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

- o XST_SUCCESS indicates all IIC interrupts are disabled. No messages can be received or transmitted until **XIic_Start** () is called.
- o XST_IIC_BUS_BUSY indicates this device is currently engaged in message traffic and cannot be stopped.

Note:

None.

iic/v1_01_d/src/xiic_options.c File Reference

Detailed Description

Contains options functions for the **XIic** component. This file is not required unless the functions in this file are called.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01b	jhl	3/26/02	repartitioned the driver
1.01c	ecm	12/05/02	new rev

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

Functions

```
void XIic_SetOptions (XIic *InstancePtr, Xuint32 NewOptions)  
Xuint32 XIic_GetOptions (XIic *InstancePtr)
```

Function Documentation

```
Xuint32  
XIic_GetOptions (XIic *  
InstancePtr)
```

This function gets the current options for the IIC device. Options control the how the device behaves on the IIC bus. See `SetOptions` for more information on options.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

The options of the IIC device. See **xiic.h** for a list of available options.

Note:

Options enabled will have a 1 in its appropriate bit position.

```
void  
XIic_SetOptions (XIic *  
                InstancePtr,  
                Xuint32 NewOptions  
                )
```

This function sets the options for the IIC device driver. The options control how the device behaves relative to the IIC bus. If an option applies to how messages are sent or received on the IIC bus, it must be set prior to calling functions which send or receive data.

To set multiple options, the values must be ORed together. To not change existing options, read/modify/write with the current options using **XIic_GetOptions()**.

USAGE EXAMPLE:

Read/modify/write to enable repeated start:

```
Xuint8 Options;  
Options = XIic_GetOptions(&Iic);  
XIic_SetOptions(&Iic, Options | XII_REPEATED_START_OPTION);
```

Disabling General Call:

```
Options = XIic_GetOptions(&Iic);  
XIic_SetOptions(&Iic, Options &= ~XII_GENERAL_CALL_OPTION);
```

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

NewOptions are the options to be set. See **xiic.h** for a list of the available options.

Returns:

None.

Note:

Sending or receiving messages with repeated start enabled, and then disabling repeated start, will not take effect until another master transaction is completed. i.e. After using repeated start, the bus will continue to be throttled after repeated start is disabled until a master transaction occurs allowing the IIC to release the bus.

Options enabled will have a 1 in its appropriate bit position.

iic/v1_01_d/src/xiic_intr.c File Reference

Detailed Description

Contains interrupt functions of the **XIic** driver. This file is required for the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01a	rfp	10/19/01	release
1.01c	ecm	12/05/02	new rev
1.01c	rmm	05/14/03	Fixed diab compiler warnings relating to asserts.

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

Functions

void **XIic_InterruptHandler** (void *InstancePtr)

Function Documentation

```
void  
XIic_InterruptHandler ( void *  
                       InstancePtr )
```

This function is the interrupt handler for the **Xlic** driver. This function should be connected to the interrupt system.

Only one interrupt source is handled for each interrupt allowing higher priority system interrupts quicker response time.

Parameters:

InstancePtr is a pointer to the **Xlic** instance to be worked on.

Returns:

None.

iic/v1_01_d/src/xiic_master.c File Reference

Detailed Description

Contains master functions for the **XIic** component. This file is necessary to send or receive as a master on the IIC bus.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.01b	jhl	3/27/02	Repartitioned the driver
-------	-----	---------	--------------------------

1.01c	ecm	12/05/02	new rev
-------	-----	----------	---------

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

Functions

XStatus XIic_MasterSend (**XIic** *InstancePtr, **Xuint8** *TxMsgPtr, int ByteCount)

XStatus XIic_MasterRecv (**XIic** *InstancePtr, **Xuint8** *RxMsgPtr, int ByteCount)

Function Documentation

```

XStatus ( XIic *
XIic_MasterRecv InstancePtr,
Xuint8 * RxMsgPtr,
int ByteCount
)

```

This function receives data as a master from a slave device on the IIC bus. If the bus is busy, it will indicate so and then enable an interrupt such that the status handler will be called when the bus is no longer busy. The slave address which has been set with the [XIic_SetAddress\(\)](#) function is the address from which data is received. Receiving data on the bus performs a read operation.

Parameters:

InstancePtr is a pointer to the Iic instance to be worked on.

RxMsgPtr is a pointer to the data to be transmitted

ByteCount is the number of message bytes to be sent

Returns:

- XST_SUCCESS indicates the message reception processes has been initiated.
- XST_IIC_BUS_BUSY indicates the bus was in use and that the BusNotBusy interrupt is enabled which will update the EventStatus when the bus is no longer busy.
- XST_IIC_GENERAL_CALL_ADDRESS indicates the slave address is set to the the general call address. This is not allowed for Master receive mode.

```

XStatus ( XIic *
XIic_MasterSend InstancePtr,
Xuint8 * TxMsgPtr,
int ByteCount
)

```

This function sends data as a master on the IIC bus. If the bus is busy, it will indicate so and then enable an interrupt such that the status handler will be called when the bus is no longer busy. The slave address which has been set with the [XIic_SetAddress\(\)](#) function is the address to which the specific data is sent. Sending data on the bus performs a write operation.

Parameters:

- InstancePtr* points to the Iic instance to be worked on.
- TxMsgPtr* points to the data to be transmitted
- ByteCount* is the number of message bytes to be sent

Returns:

- XST_SUCCESS indicates the message transmission has been initiated.
- XST_IIC_BUS_BUSY indicates the bus was in use and that the BusNotBusy interrupt is enabled which will update the EventStatus when the bus is no longer busy.

Note:

None

iic/v1_01_d/src/xiic_multi_master.c File Reference

Detailed Description

Contains multi-master functions for the **XIic** component. This file is necessary if multiple masters are on the IIC bus such that arbitration can be lost or the bus can be busy.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.01b	jhl	3/27/02	Repartitioned the driver
-------	-----	---------	--------------------------

1.01c	ecm	12/05/02	new rev
-------	-----	----------	---------

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

Functions

void **XIic_MultiMasterInclude** ()

Function Documentation

```
void  
XIic_MultiMasterInclude ( void )
```

This function includes multi-master code such that multi-master events are handled properly. Multi-master events include a loss of arbitration and the bus transitioning from busy to not busy. This function allows the multi-master processing to be optional. This function must be called prior to allowing any multi-master events to occur, such as after the driver is initialized.

Note:

None

Generated on 24 Jun 2004 for Xilinx Device Drivers

iic/v1_01_d/src/xiic_l.c File Reference

Detailed Description

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.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01b	jhl	5/13/02	First release
1.01b	jhl	10/14/02	Corrected bug in the receive function, the setup of the interrupt status mask was not being done in the loop such that a read would sometimes fail on the last byte because the transmit error being used. This and the bus would leave an extra byte in the FIFO and the bus throttled such that the next operation would also fail. Also updated the receive function to not disable the device after the last byte until after the bus transitions to not busy which is more consistent with the expected behavior.
1.01c	ecm	12/05/02	new rev

```
#include "xbasic_types.h"
#include "xio.h"
#include "xipif_v1_23_b.h"
#include "xiic_1.h"
```

Functions

unsigned **XIic_Recv** (**Xuint32** BaseAddress, **Xuint8** Address, **Xuint8** *BufferPtr, unsigned ByteCount)

unsigned **XIic_Send** (**Xuint32** BaseAddress, **Xuint8** Address, **Xuint8** *BufferPtr, unsigned ByteCount)

Function Documentation

```
unsigned          ( Xuint32
XIic_Recv          BaseAddress,
                   Xuint8  Address,
                   Xuint8 * BufferPtr,
                   unsigned ByteCount
                   )
```

Receive data as a master on the IIC bus. This function receives the data using polled I/O and blocks until the data has been received. It only supports 7 bit addressing and non-repeated start modes of operation. The user is responsible for ensuring the bus is not busy if multiple masters are present on the bus.

Parameters:

BaseAddress contains the base address of the IIC device.

Address contains the 7 bit IIC address of the device to send the specified data to.

BufferPtr points to the data to be sent.

ByteCount is the number of bytes to be sent.

Returns:

The number of bytes received.

Note:

None

```
unsigned
Xlic_Send      ( Xuint32
                  BaseAddress,
                  Xuint8  Address,
                  Xuint8 * BufferPtr,
                  unsigned ByteCount
                )
```

Send data as a master on the IIC bus. This function sends the data using polled I/O and blocks until the data has been sent. It only supports 7 bit addressing and non-repeated start modes of operation. The user is responsible for ensuring the bus is not busy if multiple masters are present on the bus.

Parameters:

BaseAddress contains the base address of the IIC device.

Address contains the 7 bit IIC address of the device to send the specified data to.

BufferPtr points to the data to be sent.

ByteCount is the number of bytes to be sent.

Returns:

The number of bytes sent.

Note:

None

iic/v1_01_d/src/xiic_selftest.c File Reference

Detailed Description

Contains selftest functions for the **XIic** component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	---	-----	-----
1.01b	jhl	3/26/02	repartitioned the driver
1.01c	ecm	12/05/02	new rev

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

Functions

XStatus XIic_SelfTest (**XIic** *InstancePtr)

Function Documentation

XStatus (**XIic** *
XIic_SelfTest *InstancePtr*)

Runs a limited self-test on the driver/device. The self-test is destructive in that a reset of the device is performed in order to check the reset values of the registers and to get the device into a known state. There is no loopback capabilities for the device such that this test does not send or receive data.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

XST_SUCCESS	No errors found
XST_IIC_STAND_REG_ERROR	One or more IIC regular registers did not zero on reset or read back correctly based on what was written

XST_IIC_TX_FIFO_REG_ERROR	to it One or more IIC parametrizable TX FIFO registers did not zero on reset or read back correctly based on what was written to it
XST_IIC_RX_FIFO_REG_ERROR	One or more IIC parametrizable RX FIFO registers did not zero on reset or read back correctly based on what was written to it
XST_IIC_STAND_REG_RESET_ERROR	A non parameterizable reg value after reset not valid
XST_IIC_TX_FIFO_REG_RESET_ERROR	Tx fifo, included in design, value after reset not valid
XST_IIC_RX_FIFO_REG_RESET_ERROR	Rx fifo, included in design, value after reset not valid
XST_IIC_TBA_REG_RESET_ERROR	10 bit addr, incl in design, value after reset not valid
XST_IIC_CR_READBACK_ERROR	Read of the control register didn't return value written
XST_IIC_DTR_READBACK_ERROR	Read of the data Tx reg didn't return value written
XST_IIC_DRR_READBACK_ERROR	Read of the data Receive reg didn't return value written
XST_IIC_ADR_READBACK_ERROR	Read of the data Tx reg didn't return value written
XST_IIC_TBA_READBACK_ERROR	Read of the 10 bit addr reg didn't return written value

Note:

Only the registers that have be included into the hardware design are tested, such as, 10-bit vs 7-bit addressing.

intc/v1_00_c/src/xintc_options.c File Reference

Detailed Description

Contains option functions for the [XIntc](#) driver. These functions allow the user to configure an instance of the [XIntc](#) driver. This file requires other files of the component to be linked in also.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00b	jhl	02/21/02	First release
-------	-----	----------	---------------

1.00c	rpm	10/17/03	New release. Support the relocation of the options flag
-------	-----	----------	---

from the instance structure to the [xintc_g.c](#) configuration table.

```
#include "xbasic_types.h"
```

```
#include "xintc.h"
```

Functions

[XStatus XIntc_SetOptions](#) ([XIntc](#) *InstancePtr, [Xuint32](#) Options)

[Xuint32 XIntc_GetOptions](#) ([XIntc](#) *InstancePtr)

Function Documentation

```
Xuint32 ( XIntc * )  
XIntc_GetOptions InstancePtr
```

Return the currently set options.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

The currently set options. The options are described in **xintc.h**.

Note:

None.

```
XStatus ( XIntc *  
XIntc_SetOptions InstancePtr,  
 Xuint32 Options  
 )
```

Set the options for the interrupt controller driver.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Options to be set. The available options are described in **xintc.h**.

Returns:

- XST_SUCCESS if the options were set successfully
- XST_INVALID_PARAM if the specified option was not valid

Note:

None.

intc/v1_00_c/src/xintc_selftest.c File Reference

Detailed Description

Contains diagnostic self-test functions for the **XIntc** component. This file requires other files of the component to be linked in also.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	02/21/02	First release

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

Functions

XStatus **XIntc_SelfTest** (**XIntc** *InstancePtr)

XStatus **XIntc_SimulateIntr** (**XIntc** *InstancePtr, **Xuint8** Id)

Function Documentation

XStatus	(XIntc * <i>InstancePtr</i>)
XIntc_SelfTest	

Run a self-test on the driver/device. This is a destructive test.

This involves forcing interrupts into the controller and verifying that they are recognized and can be acknowledged. This test will not succeed if the interrupt controller has been started in real mode such that interrupts cannot be forced.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

- XST_SUCCESS if self-test is successful.
- XST_INTC_FAIL_SELFTEST if the Interrupt controller fails the self-test. It will fail the self test if the device has previously been started in real mode.

Note:

None.

```
XStatus ( XIntc *  
XIntc_SimulateIntr InstancePtr,  
 Xuint8 Id  
 )
```

Allows software to simulate an interrupt in the interrupt controller. This function will only be successful when the interrupt controller has been started in simulation mode. Once it has been started in real mode, interrupts cannot be simulated. A simulated interrupt allows the interrupt controller to be tested without any device to drive an interrupt input signal into it.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id is the interrupt ID for which to simulate an interrupt.

Returns:

XST_SUCCESS if successful, or XST_FAILURE if the interrupt could not be simulated because the interrupt controller is or has previously been in real mode.

Note:

None.

cpu_ppc405/v1_00_a/src/xio_dcr.h File Reference

Detailed Description

The DCR I/O access functions.

Note:

These access functions are specific to the PPC405 CPU. Changes might be necessary for other members of the IBM PPC Family.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	10/18/01	First release

```
#include "xbasic_types.h"
```

Functions

```
void XIo_DcrOut (Xuint32 DcrRegister, Xuint32 Data)
```

```
Xuint32 XIo_DcrIn (Xuint32 DcrRegister)
```

Function Documentation

```
Xuint32  
XIo_DcrIn ( Xuint32  
           DcrRegister )
```

Reads value from specified register.

Parameters:

DcrRegister is the intended source DCR register

Returns:

Contents of the specified DCR register.

Note:

None.

```
void      ( Xuint32
XIo_DcrOut      DcrRegister,
              Xuint32 Data
            )
```

Outputs value provided to specified register defined in the header file.

Parameters:

DcrRegister is the intended destination DCR register

Data is the value to be placed into the specified DCR register

Returns:

None.

Note:

None.

cpu_ppc405/v1_00_a/src/xio_dcr.c File Reference

Detailed Description

The implementation of the XDcrIo interface. See [xio_dcr.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	10/18/01	First release
	ecm	01/29/03	Cleaned up the table and made it more readable. The DCR_REG_512 was 17 and the table used DCR_REG_32+15 which is actually the same as what should have been,16. I removed the blocks using DCR_REG_32+15 and added blocks using DCR_REG_512+14 to make the table correct when DCR_REG_512 = 16. This is functionally exactly the same, just more readable hence no rev bump.

```
#include "xstatus.h"  
#include "xbasic_types.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

Data Structures

struct **DcrFunctionTableEntryTag**

Functions

void **XIo_DcrOut** (**Xuint32** DcrRegister, **Xuint32** Data)

Xuint32 **XIo_DcrIn** (**Xuint32** DcrRegister)

Function Documentation

Xuint32 (**Xuint32**)
XIo_DcrIn *DcrRegister*

Reads value from specified register.

Parameters:

DcrRegister is the intended source DCR register

Returns:

Contents of the specified DCR register.

Note:

None.

void (**Xuint32**
XIo_DcrOut *DcrRegister*,
Xuint32 *Data*
)

Outputs value provided to specified register defined in the header file.

Parameters:

DcrRegister is the intended destination DCR register

Data is the value to be placed into the specified DCR register

Returns:

None.

Note:

None.

Generated on 24 Jun 2004 for Xilinx Device Drivers

cpu/v1_00_a/src/xio.h File Reference

Detailed Description

This file contains the interface for the XIo component, which encapsulates the Input/Output functions for processors that do not require any special I/O handling.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a	rpm	11/07/03	Added InSwap/OutSwap routines for endian conversion
-------	-----	----------	---

Note:

This file may contain architecture-dependent items (memory-mapped or non-memory-mapped I/O).

```
#include "xbasic\_types.h"
```

Defines

```
#define XIo\_In8(InputPtr)  
#define XIo\_In16(InputPtr)  
#define XIo\_In32(InputPtr)  
#define XIo\_Out8(OutputPtr, Value)  
#define XIo\_Out16(OutputPtr, Value)  
#define XIo\_Out32(OutputPtr, Value)
```

Typedefs

typedef [Xuint32](#) [XIo_Address](#)

Functions

void [XIo_EndianSwap16](#) ([Xuint16](#) Source, [Xuint16](#) *DestPtr)

void [XIo_EndianSwap32](#) ([Xuint32](#) Source, [Xuint32](#) *DestPtr)

[Xuint16](#) [XIo_InSwap16](#) ([XIo_Address](#) InAddress)

[Xuint32](#) [XIo_InSwap32](#) ([XIo_Address](#) InAddress)

void [XIo_OutSwap16](#) ([XIo_Address](#) OutAddress, [Xuint16](#) Value)

void [XIo_OutSwap32](#) ([XIo_Address](#) OutAddress, [Xuint32](#) Value)

Define Documentation

```
#define XIo\_In16( InputPtr )
```

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InputPtr contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

```
#define XIo\_In32( InputPtr )
```

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InputPtr contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

#define XIo_In8(InputPtr)

Performs an input operation for an 8-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InputPtr contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

#define XIo_Out16(OutputPtr, Value)

Performs an output operation for a 16-bit memory location by writing the specified value to the the specified address.

Parameters:

OutputPtr contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

#define XIo_Out32(OutputPtr, Value)

Performs an output operation for a 32-bit memory location by writing the specified value to the the specified address.

Parameters:

OutputPtr contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

```
#define XIo_Out8( OutputPtr,  
                Value      )
```

Performs an output operation for an 8-bit memory location by writing the specified value to the the specified address.

Parameters:

OutputPtr contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Typedef Documentation

```
typedef Xuint32
```

```
XIo_Address
```

Typedef for an I/O address. Typically correlates to the width of the address bus.

Function Documentation

```
void  
XIo_EndianSwap16 ( Xuint16  
                  Source,  
                  Xuint16 * DestPtr  
                  )
```

Performs a 16-bit endian conversion.

Parameters:

Source contains the value to be converted.

DestPtr contains a pointer to the location to put the converted value.

Returns:

None.

Note:

None.

```
void  
XIo_EndianSwap32 ( Xuint32  
                  Source,  
                  Xuint32 * DestPtr  
                  )
```

Performs a 32-bit endian conversion.

Parameters:

Source contains the value to be converted.

DestPtr contains a pointer to the location to put the converted value.

Returns:

None.

Note:

None.

```
Xuint16  
XIo_InSwap16 ( XIo_Address  
               InAddress )
```

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the byte-swapped value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The byte-swapped value read from the specified input address.

Note:

None.

```
Xuint32 ( XIo_Address )  
XIo_InSwap32 InAddress
```

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the byte-swapped value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The byte-swapped value read from the specified input address.

Note:

None.

```
void ( XIo_Address  
XIo_OutSwap16 OutAddress,  
 Xuint16 Value  
 )
```

Performs an output operation for a 16-bit memory location by writing the specified value to the the specified address. The value is byte-swapped before being written.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void      ( XIo_Address
XIo_OutSwap32      OutAddress,
                Xuint32      Value
                )
```

Performs an output operation for a 32-bit memory location by writing the specified value to the the specified address. The value is byte-swapped before being written.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

cpu/v1_00_a/src/xio.c File Reference

Detailed Description

Contains I/O functions for memory-mapped or non-memory-mapped I/O architectures. These functions encapsulate generic CPU I/O requirements.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a	rpm	11/07/03	Added InSwap/OutSwap routines for endian conversion
-------	-----	----------	---

Note:

This file may contain architecture-dependent code.

```
#include "xio.h"  
#include "xbasic_types.h"
```

Functions

void **XIo_EndianSwap16** (**Xuint16** Source, **Xuint16** *DestPtr)

void **XIo_EndianSwap32** (**Xuint32** Source, **Xuint32** *DestPtr)

Xuint16 **XIo_InSwap16** (**XIo_Address** InAddress)

Xuint32 **XIo_InSwap32** (**XIo_Address** InAddress)

void **XIo_OutSwap16** (**XIo_Address** OutAddress, **Xuint16** Value)

void **XIo_OutSwap32** (**XIo_Address** OutAddress, **Xuint32** Value)

Function Documentation

```
void  
XIo_EndianSwap16 ( Xuint16  
                  Source,  
                  Xuint16 * DestPtr  
                  )
```

Performs a 16-bit endian conversion.

Parameters:

Source contains the value to be converted.

DestPtr contains a pointer to the location to put the converted value.

Returns:

None.

Note:

None.

```
void  
XIo_EndianSwap32 ( Xuint32  
                  Source,  
                  Xuint32 * DestPtr  
                  )
```

Performs a 32-bit endian conversion.

Parameters:

Source contains the value to be converted.

DestPtr contains a pointer to the location to put the converted value.

Returns:

None.

Note:

None.

```
Xuint16  
XIo_InSwap16 ( XIo_Address  
              InAddress )
```

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the byte-swapped value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The byte-swapped value read from the specified input address.

Note:

None.

```
Xuint32 ( XIo_Address )  
XIo_InSwap32 InAddress
```

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the byte-swapped value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The byte-swapped value read from the specified input address.

Note:

None.

```
void ( XIo_Address  
XIo_OutSwap16 OutAddress,  
 Xuint16 Value  
 )
```

Performs an output operation for a 16-bit memory location by writing the specified value to the the specified address. The value is byte-swapped before being written.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void      ( XIo_Address
XIo_OutSwap32      OutAddress,
                Xuint32      Value
                )
```

Performs an output operation for a 32-bit memory location by writing the specified value to the the specified address. The value is byte-swapped before being written.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

cpu_ppc405/v1_00_a/src/xio.c File Reference

Detailed Description

Contains I/O functions for memory-mapped or non-memory-mapped I/O architectures. These functions encapsulate PowerPC architecture-specific I/O requirements.

Note:

This file contains architecture-dependent code.

The order of the SYNCHRONIZE_IO and the read or write operation is important. For the Read operation, all I/O needs to complete prior to the desired read to insure valid data from the address. The PPC is a weakly ordered I/O model and reads can and will occur prior to writes and the SYNCHRONIZE_IO ensures that any writes occur prior to the read. For the Write operation the SYNCHRONIZE_IO occurs after the desired write to ensure that the address is updated with the new value prior to any subsequent read.

```
#include "xio.h"  
#include "xbasic_types.h"
```

Functions

[Xuint8 XIo_In8 \(XIo_Address InAddress\)](#)
[Xuint16 XIo_In16 \(XIo_Address InAddress\)](#)
[Xuint32 XIo_In32 \(XIo_Address InAddress\)](#)
[Xuint16 XIo_InSwap16 \(XIo_Address InAddress\)](#)
[Xuint32 XIo_InSwap32 \(XIo_Address InAddress\)](#)
void [XIo_Out8 \(XIo_Address OutAddress, Xuint8 Value\)](#)
void [XIo_Out16 \(XIo_Address OutAddress, Xuint16 Value\)](#)
void [XIo_Out32 \(XIo_Address OutAddress, Xuint32 Value\)](#)
void [XIo_EndianSwap16OLD \(Xuint16 Source, Xuint16 *DestPtr\)](#)

void **XIo_EndianSwap32OLD** (**Xuint32** Source, **Xuint32** *DestPtr)

void **XIo_OutSwap16** (**XIo_Address** OutAddress, **Xuint16** Value)

void **XIo_OutSwap32** (**XIo_Address** OutAddress, **Xuint32** Value)

Function Documentation

```
void  
XIo_EndianSwap16OLD ( Xuint16  
                        Source,  
                        Xuint16 * DestPtr  
                      )
```

Performs a 16-bit endian conversion.

Parameters:

Source contains the value to be converted.

DestPtr contains a pointer to the location to put the converted value.

Returns:

None.

Note:

None.

```
void  
XIo_EndianSwap32OLD ( Xuint32  
                        Source,  
                        Xuint32 * DestPtr  
                      )
```

Performs a 32-bit endian conversion.

Parameters:

Source contains the value to be converted.

DestPtr contains a pointer to the location to put the converted value.

Returns:

None.

Note:

None.

Xuint16 (**XIo_Address**)

XIo_In16 *InAddress*

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

Note:

None.

Xuint32 (**XIo_Address**)

XIo_In32 *InAddress*

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

Note:

None.

Xuint8 (**XIo_Address**)

XIo_In8 *InAddress*

Performs an input operation for an 8-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

Note:

None.

Xuint16 (**XIo_Address**)
XIo_InSwap16 *InAddress*

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the byte-swapped value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The byte-swapped value read from the specified input address.

Note:

None.

Xuint32 (**XIo_Address**)
XIo_InSwap32 *InAddress*

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the byte-swapped value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The byte-swapped value read from the specified input address.

Note:

None.

```
void      ( XIo_Address  
XIo_Out16      OutAddress,  
           Xuint16      Value  
           )
```

Performs an output operation for a 16-bit memory location by writing the specified value to the the specified address.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void      ( XIo_Address  
XIo_Out32      OutAddress,  
           Xuint32      Value  
           )
```

Performs an output operation for a 32-bit memory location by writing the specified value to the the specified address.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void      ( XIo_Address
XIo_Out8  OutAddress,
          Xuint8     Value
          )
```

Performs an output operation for an 8-bit memory location by writing the specified value to the the specified address.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void      ( XIo_Address
XIo_OutSwap16  OutAddress,
              Xuint16     Value
              )
```

Performs an output operation for a 16-bit memory location by writing the specified value to the the specified address. The value is byte-swapped before being written.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void      ( XIo_Address
XIo_OutSwap32      OutAddress,
                Xuint32      Value
                )
```

Performs an output operation for a 32-bit memory location by writing the specified value to the the specified address. The value is byte-swapped before being written.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

opb2plb/v1_00_a/src/xopb2plb.h File Reference

Detailed Description

This component contains the implementation of the **XOpb2Plb** component. It is the driver for the OPB to PLB Bridge. The bridge converts OPB bus transactions to PLB bus transactions. The hardware acts as a slave on the OPB side and as a master on the PLB side. This interface is necessary for the peripherals to access PLB based memory.

This driver allows the user to access the Bridge registers to support the handling of bus errors and other access errors and determine an appropriate solution if possible.

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

The Xilinx OPB to PLB Bridge is a soft IP core designed for Xilinx FPGAs and contains the following features:

- 64-bit PLB Master interface.
- Communicates with 32- or 64-bit PLB slaves.
- Non-burst transfers of 1-8 bytes.
- Burst transfers, including word and double-word bursts of fixed lengths, up to the depth of the burst buffer. Buffer depth configurable via a design parameter.
- Cacheline transactions of 4, 8, and 16 words.
- Programmable address boundaries that encompass the inverse of the PLB to OPB Bridge address space.
- Translates OPB data bursts to either cacheline or fixed length PLB burst transfers.
- Performing only CPU subset transactions (cachelines) can reduce system logic utilization and improve timing through the simplification of PLB slave IP.
- Using PLB burst transfers yields better bus cycle efficiency but may increase logic utilization and degrade timing in the system OPB Slave interface.
- 32-bit OPB Slave interface with byte enable transfers. *Note:* Does not support dynamic bus

sizing or non-byte enable transactions. PLB and OPB clocks can have an asynchronous relationship (OPB clock frequency must be less than or equal to the PLB clock frequency).

- Bus Error Status Register (BESR) and Bus Error Address Register (BEAR) to report errors.
- DCR Slave interface provides access to BESR and BEAR.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. The current configuration data is contained in the `xopb2plb_g.c`. A table is defined where each entry contains configuration information for a device. This information includes such things as the base address of the DCR mapped device.

Register Access

The bridge registers reside on the DCR address bus which is a parameter that can be selected in the hardware. If the DCR is not used, the registers reside in the OPB address space. A restriction of this driver is that if more than one bridge exists in the system, all must be configured the same way. That is, all must use DCR or all must use OPB.

Note:

This driver is not thread-safe. Thread safety must be guaranteed by the layer above this driver if there is a need to access the device from multiple threads.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/22/02	First release
1.00a	rpm	05/14/02	Made configuration table/data public

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xopb2plb_1.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

Data Structures

```
struct XOpb2Plb
```

struct **XOpb2Plb_Config**

OPB-PLB bridge error status values

```
#define XO2P_READ_ERROR  
#define XO2P_WRITE_ERROR  
#define XO2P_NO_ERROR
```

Functions

XStatus **XOpb2Plb_Initialize** (**XOpb2Plb** *InstancePtr, **Xuint16** DeviceId)

void **XOpb2Plb_Reset** (**XOpb2Plb** *InstancePtr)

XOpb2Plb_Config * **XOpb2Plb_LookupConfig** (**Xuint16** DeviceId)

Xboolean **XOpb2Plb_IsError** (**XOpb2Plb** *InstancePtr)

void **XOpb2Plb_ClearErrors** (**XOpb2Plb** *InstancePtr)

Xuint32 **XOpb2Plb_GetErrorStatus** (**XOpb2Plb** *InstancePtr)

Xuint32 **XOpb2Plb_GetErrorAddress** (**XOpb2Plb** *InstancePtr)

void **XOpb2Plb_EnableInterrupt** (**XOpb2Plb** *InstancePtr)

void **XOpb2Plb_DisableInterrupt** (**XOpb2Plb** *InstancePtr)

void **XOpb2Plb_EnableLock** (**XOpb2Plb** *InstancePtr)

void **XOpb2Plb_DisableLock** (**XOpb2Plb** *InstancePtr)

XStatus **XOpb2Plb_SelfTest** (**XOpb2Plb** *InstancePtr, **Xuint32** TestAddress)

Define Documentation

#define

XO2P_NO_ERROR

XO2P_READ_ERROR	A read error occurred
XO2P_WRITE_ERROR	A write error occurred
XO2P_NO_ERROR	There is no error

#define

XO2P_READ_ERROR

XO2P_READ_ERROR	A read error occurred
XO2P_WRITE_ERROR	A write error occurred
XO2P_NO_ERROR	There is no error

#define

XO2P_WRITE_ERROR

XO2P_READ_ERROR	A read error occurred
XO2P_WRITE_ERROR	A write error occurred
XO2P_NO_ERROR	There is no error

Function Documentation

```
void (XOpb2Plb * InstancePtr)  
XOpb2Plb_ClearErrors
```

Clears the errors. If the lock bit is set, this allows subsequent errors to be recognized.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

None.

```
void (XOpb2Plb * InstancePtr)  
XOpb2Plb_DisableInterrupt
```

Disables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void ( XOpb2Plb * )  
XOpb2Plb_DisableLock InstancePtr
```

Disables the locking of the status on error for the bridge. This 'unlocks' the status and address registers allowing subsequent errors to overwrite the current values when an error occurs.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XOpb2Plb * )  
XOpb2Plb_EnableInterrupt InstancePtr
```

Enables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void ( XOpb2Plb * )  
XOpb2Plb_EnableLock InstancePtr
```

Enables the locking of the status on error for the bridge. This 'locks' the status and address register values when an error occurs, preventing subsequent errors from overwriting the values. Clearing the error allows the status and address registers to update with the next error that occurs

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint32 ( XOpb2Plb * )  
XOpb2Plb_GetErrorAddress InstancePtr
```

Returns the PLB Address where the most recent error occurred. If there isn't an outstanding error, the last address in error is returned. 0x00000000 is the initial value coming out of reset.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

Address where error causing access occurred

Note:

Calling **XOpb2Plb_IsError()** is recommended to confirm that an error has occurred prior to calling **XOpb2Plb_GetErrorAddress()** to ensure that the data in the error address register is relevant.

```
Xuint32 ( XOpb2Plb * )  
XOpb2Plb_GetErrorStatus InstancePtr
```

Returns the error status indicating the type of error that has occurred.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

The current error status for the OPB to PLB bridge. The possible return values are described in **xopb2plb.h**.

Note:

None.

```
XStatus ( XOpb2Plb *  
XOpb2Plb_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific **XOpb2Plb** instance. Looks up the configuration data for the given device ID and then initializes instance data.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XOpb2Plb** component. Passing in a device id associates the generic **XOpb2Plb** component to a specific device, as chosen by the caller or application developer.

Returns:

- o XST_SUCCESS if everything starts up as expected.
- o XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

Xboolean (**XOpb2Plb ***)
XOpb2Plb_IsError *InstancePtr*

Returns XTRUE is there is an error outstanding

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

Boolean XTRUE if there is an error, XFALSE if there is no current error.

Note:

None.

XOpb2Plb_Config* (**Xuint16**)
XOpb2Plb_LookupConfig *DeviceId*

Looks up the device configuration based on the unique device ID. The table `OpbPlbConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device to look for.

Returns:

A pointer to the configuration data of the given device, or `XNULL` if no match is found.

Note:

None.

```
void ( XOpb2Plb * )  
XOpb2Plb_Reset InstancePtr
```

Forces a software-induced reset to occur in the bridge and disables interrupts and the locking functionality in the process.

Parameters:

InstancePtr is a pointer to the `XOpb2Plb` instance to be worked on.

Returns:

None.

Note:

Disables interrupts and the locking functionality in the process.

```
XStatus ( XOpb2Plb *  
XOpb2Plb_SelfTest InstancePtr,  
Xuint32 TestAddress  
)
```

Runs a self-test on the driver/device.

This tests reads the provided `TestAddress` which is intended to cause an error Then the `XOpb2Plb_IsError()` routine is called and if there is an error, the address is checked against the provided location and if they match `XST_SUCCESS` is returned and all errors are then cleared.

If the `XOpb2Plb_IsError()` is called and no error is indicated, `XST_FAILURE` is returned.

Parameters:

InstancePtr is a pointer to the `XOpb2Plb` instance to be worked on.

TestAddress is a location that should cause an error on read.

Returns:

`XST_SUCCESS` if successful, or `XST_FAILURE` if the driver fails the self-test.

Note:

This test assumes that the bus error interrupts to the processor are not enabled.

XOpb2Plb Struct Reference

```
#include <xopb2plb.h>
```

Detailed Description

The XOpb2Plb driver instance data. The user is required to allocate a variable of this type for every OPB-to-PLB bridge device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [opb2plb/v1_00_a/src/xopb2plb.h](#)
-

opb2plb/v1_00_a/src/xopb2plb_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of OPB-to-PLB bridge devices in the system. Each bridge device should have an entry in this table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/22/02	First release
1.00a	rpm	05/14/02	Made configuration table/data public

```
#include "xopb2plb.h"  
#include "xparameters.h"
```

Variables

[XOpb2Plb_Config](#) [XOpb2Plb_ConfigTable](#) [XPAR_XOPB2PLB_NUM_INSTANCES]

Variable Documentation

[XOpb2Plb_Config](#) [XOpb2Plb_ConfigTable](#)
[XPAR_XOPB2PLB_NUM_INSTANCES]

The OPB-to-PLB bridge configuration table, sized by the number of instances defined in [xparameters.h](#).

XOpb2Plb_Config Struct Reference

```
#include <xopb2plb.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

Field Documentation

[Xuint32 XOpb2Plb_Config::](#)

BaseAddress

Register base
address

[Xuint16 XOpb2Plb_Config::](#)

DeviceId

Unique ID of
device

The documentation for this struct was generated from the following file:

- `opb2plb/v1_00_a/src/xopb2plb.h`

opb2plb/v1_00_a/src/xopb2plb_l.h File Reference

Detailed Description

This file contains identifiers and low-level macros that can be used to access the device directly.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	05/14/02	First release

```
#include "xparameters.h"  
#include "xbasic_types.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

Defines

```
#define XOpb2Plb_mGetBesrReg(BaseAddress)  
#define XOpb2Plb_mGetBearReg(BaseAddress)  
#define XOpb2Plb_mSetControlReg(BaseAddress, Mask)  
#define XOpb2Plb_mGetControlReg(BaseAddress)
```

Define Documentation

```
#define XOpb2Plb_mGetBearReg( BaseAddress )
```

Get the bus error address register (BEAR).

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the error address register.

Note:

None.

```
#define XOpb2Plb_mGetBesrReg( BaseAddress )
```

Get the bus error status register (BESR).

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the error status register.

Note:

None.

```
#define XOpb2Plb_mGetControlReg( BaseAddress )
```

Get the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the control register.

Note:

None.

```
#define XOpb2Plb_mSetControlReg( BaseAddress,  
                                Mask          )
```

Set the control register to the given value.

Parameters:

BaseAddress is the base address of the device

Mask is the value to write to the control register.

Returns:

None.

Note:

None.

Generated on 24 Jun 2004 for Xilinx Device Drivers

opbarb/v1_02_a/src/xopbarb.h File Reference

Detailed Description

This component contains the implementation of the **XOpbArb** component. It is the driver for the On-chip Peripheral Bus (OPB) Arbiter. The arbiter performs bus arbitration for devices on the OPB

Priority Arbitration

By default, the arbiter prioritizes masters on the bus based on their master IDs. A master's ID corresponds to the signals with which it connects to the arbiter. Master 0 is the highest priority, master 1 the next highest, and so on. The device driver allows an application to modify this default behavior.

There are two modes of priority arbitration, dynamic and fixed. The device can be parameterized in either of these modes. Fixed priority arbitration makes use of priority levels that can be configured by software. There is one level for each master on the bus. Priority level 0 is assigned the master ID of the highest priority master, priority level 1 is assigned the master ID of the next highest priority master, and so on.

Dynamic priority arbitration utilizes a Least Recently Used(LRU) algorithm to determine the priorities of masters on the bus. Once a master is granted the bus, it falls to the lowest priority master. A master that is not granted the bus moves up in priority until it becomes the highest priority master based on the fact that it has been the least recently used master. The arbiter uses the currently assigned priority levels as its starting configuration for the LRU algorithm. Software can modify this starting configuration by assigning master IDs to the priority levels.

When configuring priority levels (i.e., assigning master IDs to priority levels), the application must suspend use of the priority levels by the device. Every master must be represented by one and only one priority level. The device driver enforces this by making the application suspend use of the priority levels by the device during the time it takes to correctly configure the levels. Once the levels are configured, the application must explicitly resume use of the priority levels by the device. During the time priority levels are suspended, the device reverts to its default behavior of assigning priorities based on master IDs.

Bus Parking

By default, bus parking is disabled. The device driver allows an application to enable bus parking, which forces the arbiter to always assert the grant signal for a specific master when no other masters are requesting the bus. The master chosen for parking is either the master that was last granted the bus, or the master configured by the SetParkId function. When bus parking is enabled but no park ID has been set, bus parking defaults to the master that was last granted the bus.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are stored in `xopbarb_g.c`. A table is defined where each entry contains configuration information for a device. This information includes such things as the base address of the memory-mapped device, the number of masters on the bus, and the priority arbitration scheme.

When the device is parameterized with only 1 master, or if the device is parameterized without a slave interface, there are no registers accessible to software and no configuration entry is available. In these configurations it is likely that the driver will not be loaded or used by the application. But in the off-chance that it is, it is assumed that no configuration information for the arbiter is entered in the `xopbarb_g.c` table. If config information were entered for the device, there will be nothing to prevent the driver's use, and any use of the driver under these circumstances will result in undefined behavior.

Note:

This driver is not thread-safe. Thread safety must be guaranteed by the layer above this driver if there is a need to access the device from multiple threads.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.02a	rpm	08/13/01	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xopbarb_1.h"
```

Data Structures

```
struct XOpbArb
```

struct **XOpbArb_Config**

Configuration options

```
#define XOA_DYNAMIC_PRIORITY_OPTION  
#define XOA_PARK_ENABLE_OPTION  
#define XOA_PARK_BY_ID_OPTION
```

Functions

```
XStatus XOpbArb_Initialize (XOpbArb *InstancePtr, Xuint16 DeviceId)  
XOpbArb_Config * XOpbArb_LookupConfig (Xuint16 DeviceId)  
XStatus XOpbArb_SelfTest (XOpbArb *InstancePtr)  
XStatus XOpbArb_SetOptions (XOpbArb *InstancePtr, Xuint32 Options)  
Xuint32 XOpbArb_GetOptions (XOpbArb *InstancePtr)  
XStatus XOpbArb_SetPriorityLevel (XOpbArb *InstancePtr, Xuint8 Level, Xuint8  
    MasterId)  
Xuint8 XOpbArb_GetPriorityLevel (XOpbArb *InstancePtr, Xuint8 Level)  
void XOpbArb_SuspendPriorityLevels (XOpbArb *InstancePtr)  
XStatus XOpbArb_ResumePriorityLevels (XOpbArb *InstancePtr)  
XStatus XOpbArb_SetParkId (XOpbArb *InstancePtr, Xuint8 MasterId)  
XStatus XOpbArb_GetParkId (XOpbArb *InstancePtr, Xuint8 *MasterIdPtr)
```

Define Documentation

```
#define  
XOA_DYNAMIC_PRIORITY_OPTION
```

The options enable or disable additional features of the OPB Arbiter. Each of the options are bit fields such that more than one may be specified.

- **XOA_DYNAMIC_PRIORITY_OPTION**
The Dynamic Priority option configures the device for dynamic priority arbitration, which uses a Least Recently Used (LRU) algorithm to determine the priorities of OPB masters. This option is not applicable if the device is parameterized for fixed priority arbitration. When the device is parameterized for dynamic priority arbitration, it can still use a fixed priority arbitration by turning this option off. Fixed priority arbitration uses the priority levels as written by software to determine the priorities of OPB masters. The default is fixed priority arbitration.

- **XOA_PARK_ENABLE_OPTION**
The Park Enable option enables bus parking, which forces the arbiter to always assert the grant signal for a specific master when no other masters are requesting the bus. The master chosen for parking is either the master that was last granted the bus, or the master configured by the SetParkId function.
- **XOA_PARK_BY_ID_OPTION**
The Park By ID option enables bus parking based on the specific master ID. The master ID defaults to master 0, and can be changed using [XOpbArb_SetParkId\(\)](#). When this option is disabled, bus parking defaults to the master that was last granted the bus. The park enable option must be set for this option to take effect.

#define

XOA_PARK_BY_ID_OPTION

The options enable or disable additional features of the OPB Arbiter. Each of the options are bit fields such that more than one may be specified.

- **XOA_DYNAMIC_PRIORITY_OPTION**
The Dynamic Priority option configures the device for dynamic priority arbitration, which uses a Least Recently Used (LRU) algorithm to determine the priorities of OPB masters. This option is not applicable if the device is parameterized for fixed priority arbitration. When the device is parameterized for dynamic priority arbitration, it can still use a fixed priority arbitration by turning this option off. Fixed priority arbitration uses the priority levels as written by software to determine the priorities of OPB masters. The default is fixed priority arbitration.
- **XOA_PARK_ENABLE_OPTION**
The Park Enable option enables bus parking, which forces the arbiter to always assert the grant signal for a specific master when no other masters are requesting the bus. The master chosen for parking is either the master that was last granted the bus, or the master configured by the SetParkId function.
- **XOA_PARK_BY_ID_OPTION**
The Park By ID option enables bus parking based on the specific master ID. The master ID defaults to master 0, and can be changed using [XOpbArb_SetParkId\(\)](#). When this option is disabled, bus parking defaults to the master that was last granted the bus. The park enable option must be set for this option to take effect.

#define

XOA_PARK_ENABLE_OPTION

The options enable or disable additional features of the OPB Arbiter. Each of the options are bit fields such that more than one may be specified.

- **XOA_DYNAMIC_PRIORITY_OPTION**
The Dynamic Priority option configures the device for dynamic priority arbitration, which uses a Least Recently Used (LRU) algorithm to determine the priorities of OPB masters. This option is not applicable if the device is parameterized for fixed priority arbitration. When the device is parameterized for dynamic priority arbitration, it can still use a fixed priority arbitration by turning this option off. Fixed priority arbitration uses the priority levels as written by software to determine the priorities of OPB masters. The default is fixed priority arbitration.
- **XOA_PARK_ENABLE_OPTION**
The Park Enable option enables bus parking, which forces the arbiter to always assert the grant signal for a specific master when no other masters are requesting the bus. The master chosen for parking is either the master that was last granted the bus, or the master configured by the SetParkId function.
- **XOA_PARK_BY_ID_OPTION**
The Park By ID option enables bus parking based on the specific master ID. The master ID defaults to master 0, and can be changed using [XOpbArb_SetParkId\(\)](#). When this option is disabled, bus parking defaults to the master that was last granted the bus. The park enable option must be set for this option to take effect.

Function Documentation

[Xuint32](#) ([XOpbArb *](#))
[XOpbArb_GetOptions](#) *InstancePtr*

Gets the options for the arbiter. The options control how the device grants the bus to requesting masters.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

The options of the device. This is a bit mask where a 1 means the option is on, and a 0 means the option is off. See **xopbarb.h** for a description of the options.

Note:

None.

```
XStatus ( XOpbArb *  
XOpbArb_GetParkId InstancePtr,  
 Xuint8 * MasterIdPtr  
 )
```

Gets the master ID currently used for bus parking.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

MasterIdPtr is a pointer to a byte that will hold the master ID currently used for bus parking. This is an output parameter. The ID can range from 0 to N, where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

XST_SUCCESS if the park ID is successfully retrieved, or XST_NO_FEATURE if bus parking is not supported by the device.

Note:

None.

```
Xuint8 ( XOpbArb *  
XOpbArb_GetPriorityLevel InstancePtr,  
 Xuint8 Level  
 )
```

Get the master ID at the given priority level.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Level is the priority level being retrieved. The level can range from 0 (highest) to N (lowest), where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

The master ID assigned to the given priority level. The ID can range from 0 to N, where N is the number of masters minus one.

Note:

If the arbiter is operating in dynamic priority mode, the value returned from this function may not be predictable because the arbiter changes the values on the fly.

```
XStatus ( XOpbArb *  
XOpbArb_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific **XOpbArb** instance. The driver is initialized to allow access to the device registers. In addition, the configuration information is retrieved for the device. Currently, configuration information is stored in **xopbarb_g.c**.

The state of the device after initialization is:

- Fixed or dynamic priority arbitration based on hardware parameter
- Bus parking is disabled

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XOpbArb** component. Passing in a device id associates the generic **XOpbArb** component to a specific device, as chosen by the caller or application developer.

Returns:

The return value is XST_SUCCESS if successful or XST_DEVICE_NOT_FOUND if no configuration data was found for this device.

Note:

None.

XOpbArb_Config* (**Xuint16**)
XOpbArb_LookupConfig *DeviceId*

Looks up the device configuration based on the unique device ID. The table OpbArbConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to match on.

Returns:

A pointer to the configuration information for the matching device instance, or XNULL if no match is found.

Note:

None.

XStatus (**XOpbArb ***)
XOpbArb_ResumePriorityLevels *InstancePtr*

Resumes use of the priority levels by the device. This function is typically called sometime after a call to SuspendPriorityLevels. The application must resume use of priority levels by the device when all modifications are done. If no call is made to this function after use of the priority levels has been suspended, the device will remain in its default priority arbitration mode of assigning priorities based on master IDs. A call to this function has no effect if no prior call was made to suspend the use of priority levels.

Every master must be represented by one and only one fixed priority level before the use of priority levels can be resumed.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

- XST_SUCCESS if the slave is selected successfully.
- XST_OPBARB_INVALID_PRIORITY if there is either a master that is not assigned a priority level, or a master that is assigned two or more priority levels.

Note:

None.

```
XStatus ( XOpbArb * )  
XOpbArb_SelfTest InstancePtr
```

Runs a self-test on the driver/device. The self-test simply verifies that the arbiter's registers can be read and written. This is an intrusive test in that the arbiter will not be using the priority registers while the test is being performed.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

XST_SUCCESS if successful, or XST_REGISTER_ERROR if a register did not read or write correctly

Note:

The priority level registers are restored after testing them in order to prevent problems with the registers being the same value after the test.

If the arbiter is in dynamic priority mode, this test changes the mode to fixed to ensure that the priority registers aren't changed by the arbiter during this test. The mode is restored to its entry value on exit.

```
XStatus ( XOpbArb *  
XOpbArb_SetOptions InstancePtr,  
 Xuint32 Options  
 )
```

Sets the options for the OPB arbiter. The options control how the device grants the bus to requesting masters.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Options contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. See **xopbarb.h** for a description of the options.

Returns:

- XST_SUCCESS if options are successfully set.
- XST_NO_FEATURE if an attempt was made to enable dynamic priority arbitration when the device is configured only for fixed priority arbitration, or an attempt was made to enable parking when bus parking is not supported by the device.
- XST_OPBARB_PARK_NOT_ENABLED if bus parking by park ID was enabled but bus parking itself was not enabled.

Note:

None.

```
XStatus ( XOpbArb *  
XOpbArb_SetParkId InstancePtr,  
 Xuint8 MasterId  
 )
```

Sets the master ID used for bus parking. Bus parking must be enabled and the option to use bus parking by park ID must be set for this park ID to take effect (see the SetOptions function). If the option to use bus parking by park ID is set but this function is not called, bus parking defaults to master 0.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

MasterId is the ID of the master that will be parked if bus parking is enabled. This master's grant signal remains asserted as long as no other master requests the bus. The ID can range from 0 to N, where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

XST_SUCCESS if the park ID is successfully set, or XST_NO_FEATURE if bus parking is not supported by the device.

Note:

None.

```

XStatus
XOpbArb_SetPriorityLevel ( XOpbArb *
                           InstancePtr,
                           Xuint8
                           Level,
                           Xuint8
                           MasterId
                           )

```

Assigns a master ID to the given priority level. The use of priority levels by the device must be suspended before calling this function. Every master ID must be assigned to one and only one priority level. The driver enforces this before allowing use of priority levels by the device to be resumed.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Level is the priority level being set. The level can range from 0 (highest) to N (lowest), where N is the number of masters minus one. The device currently supports up to 16 masters.

MasterId is the ID of the master being assigned to the priority level. The ID can range from 0 to N, where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

- XST_SUCCESS if the slave is selected successfully.
- XST_OPBARB_NOT_SUSPENDED if priority levels have not been suspended. Before modifying the priority levels, use of priority levels by the device must be suspended.
- XST_OPBARB_NOT_FIXED_PRIORITY if the arbiter is in dynamic mode. It must be in fixed mode to modify the priority levels.

Note:

None.

```

void
XOpbArb_SuspendPriorityLevels ( XOpbArb *
                               InstancePtr
                               )

```

Suspends use of the priority levels by the device. Before modifying priority levels, the application must first suspend use of the levels by the device. This is to prevent possible OPB problems if no master is assigned a priority during the modification of priority levels. The application must resume use of priority levels by the device when all modifications are done. During the time priority levels are suspended, the device reverts to its default behavior of assigning priorities based on master IDs.

This function can be used when the device is configured for either fixed priority arbitration or dynamic priority arbitration. When used during dynamic priority arbitration, the application can configure the priority levels as a starting point for the LRU algorithm.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

None.

Note:

None.

XOpbArb Struct Reference

```
#include <xopbarb.h>
```

Detailed Description

The XOpbArb driver instance data. The user is required to allocate a variable of this type for every OPB arbiter device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [opbarb/v1_02_a/src/xopbarb.h](#)
-

opbarb/v1_02_a/src/xopbarb_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of OPB arbiter devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.02a	rpm	08/13/01	First release

```
#include "xopbarb.h"  
#include "xparameters.h"
```

Variables

[XOpbArb_Config](#) [XOpbArb_ConfigTable](#) [XPAR_XOPBARB_NUM_INSTANCES]

Variable Documentation

[XOpbArb_Config](#) [XOpbArb_ConfigTable](#)
[XPAR_XOPBARB_NUM_INSTANCES]

The OPB arbiter configuration table, sized by the number of instances defined in [xparameters.h](#).

XOpbArb_Config Struct Reference

```
#include <xopbarb.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xuint8 NumMasters](#)

Field Documentation

[Xuint32 XOpbArb_Config::](#)

BaseAddress

Register base
address

[Xuint16 XOpbArb_Config::](#)

DeviceId

Unique ID of
device

[Xuint8 XOpbArb_Config::](#)

NumMasters

Number of masters on the
bus

The documentation for this struct was generated from the following file:

- `opbarb/v1_02_a/src/xopbarb.h`
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

opbarb/v1_02_a/src/xopbarb_i.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. High-level driver functions are defined in [xopbarb.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	04/24/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

Defines

```
#define XOpbArb_mSetControlReg(BaseAddress, RegisterValue)  
#define XOpbArb_mGetControlReg(BaseAddress)  
#define XOpbArb_mEnableDynamic(BaseAddress)  
#define XOpbArb_mDisableDynamic(BaseAddress)  
#define XOpbArb_mEnableParking(BaseAddress)  
#define XOpbArb_mDisableParking(BaseAddress)  
#define XOpbArb_mSetParkMasterNot(BaseAddress)  
#define XOpbArb_mClearParkMasterNot(BaseAddress)  
#define XOpbArb_mSetPriorityRegsValid(BaseAddress)  
#define XOpbArb_mClearPriorityRegsValid(BaseAddress)  
#define XOpbArb_mSetParkedMasterId(BaseAddress, ParkedMasterId)
```

```
#define XOpbArb_mSetPriorityReg(BaseAddress, Level, MasterId)
```

```
#define XOpbArb_mGetPriorityReg(BaseAddress, Level)
```

Define Documentation

```
#define XOpbArb_mClearParkMasterNot( BaseAddress )
```

Clear park on master not last (park on a specific master ID) in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XOpbArb_mClearPriorityRegsValid( BaseAddress )
```

Clear the priority registers valid in the Control Register of the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XOpbArb_mDisableDynamic( BaseAddress )
```

Disable dynamic priority arbitration in the Control Register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XOpbArb_mDisableParking( BaseAddress )
```

Disable parking in the Control Register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XOpbArb_mEnableDynamic( BaseAddress )
```

Enable dynamic priority arbitration in the Control Register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XOpbArb_mEnableParking( BaseAddress )
```

Enable parking in the Control Register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XOpbArb_mGetControlReg( BaseAddress )
```

Get the Control Register of the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

The value read from the register.

Note:

None.

```
#define XOpbArb_mGetPriorityReg( BaseAddress,  
                                Level      )
```

Get the priority register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Level contain the priority level of the register to get (0 - 15).

Returns:

The contents of the specified priority register, a master ID (0 - 15).

Note:

None.

```
#define XOpbArb_mSetControlReg( BaseAddress,  
                                RegisterValue )
```

Set the Control Register of the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

RegisterValue contains the value to be written to the register.

Returns:

None.

Note:

None.

```
#define XOpbArb_mSetParkedMasterId( BaseAddress,  
                                    ParkedMasterId )
```

Set the parked master ID in the Control Register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

ParkedMasterId contains the ID of the master to park on (0 - 15).

Returns:

None.

Note:

None.

```
#define XOpbArb_mSetParkMasterNot( BaseAddress )
```

Set park on master not last (park on a specific master ID) in the Control Register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XOpbArb_mSetPriorityReg( BaseAddress,  
                                Level,  
                                MasterId    )
```

Set the priority register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Level contain the priority level of the register to set (0 - 15).

MasterId contains the value to be written to the register (0 - 15).

Returns:

None.

Note:

None.

```
#define XOpbArb_mSetPriorityRegsValid( BaseAddress )
```

Set the priority registers valid in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

Generated on 24 Jun 2004 for Xilinx Device Drivers

opb2plb/v1_00_a/src/xopb2plb.c File Reference

Detailed Description

Contains required functions for the [XOpb2Plb](#) component. See [xopb2plb.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	01/22/02	First release

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xopb2plb.h"  
#include "xopb2plb_i.h"
```

Functions

```
XStatus XOpb2Plb\_Initialize (XOpb2Plb *InstancePtr, Xuint16 DeviceId)  
void XOpb2Plb\_Reset (XOpb2Plb *InstancePtr)  
Xboolean XOpb2Plb\_IsError (XOpb2Plb *InstancePtr)  
void XOpb2Plb\_ClearErrors (XOpb2Plb *InstancePtr)  
Xuint32 XOpb2Plb\_GetErrorStatus (XOpb2Plb *InstancePtr)  
Xuint32 XOpb2Plb\_GetErrorAddress (XOpb2Plb *InstancePtr)  
void XOpb2Plb\_EnableInterrupt (XOpb2Plb *InstancePtr)  
void XOpb2Plb\_DisableInterrupt (XOpb2Plb *InstancePtr)  
void XOpb2Plb\_EnableLock (XOpb2Plb *InstancePtr)  
void XOpb2Plb\_DisableLock (XOpb2Plb *InstancePtr)
```

Function Documentation

```
void ( XOpb2Plb * )  
XOpb2Plb_ClearErrors InstancePtr
```

Clears the errors. If the lock bit is set, this allows subsequent errors to be recognized.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XOpb2Plb * )  
XOpb2Plb_DisableInterrupt InstancePtr
```

Disables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void ( XOpb2Plb * )  
XOpb2Plb_DisableLock InstancePtr
```

Disables the locking of the status on error for the bridge. This 'unlocks' the status and address registers allowing subsequent errors to overwrite the current values when an error occurs.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XOpb2Plb * )  
XOpb2Plb_EnableInterrupt InstancePtr
```

Enables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void ( XOpb2Plb * )  
XOpb2Plb_EnableLock InstancePtr
```

Enables the locking of the status on error for the bridge. This 'locks' the status and address register values when an error occurs, preventing subsequent errors from overwriting the values. Clearing the error allows the status and address registers to update with the next error that occurs

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

None.

Xuint32 (**XOpb2Plb** *)
XOpb2Plb_GetErrorAddress *InstancePtr*

Returns the PLB Address where the most recent error occurred. If there isn't an outstanding error, the last address in error is returned. 0x00000000 is the initial value coming out of reset.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

Address where error causing access occurred

Note:

Calling **XOpb2Plb_IsError()** is recommended to confirm that an error has occurred prior to calling **XOpb2Plb_GetErrorAddress()** to ensure that the data in the error address register is relevant.

Xuint32 (**XOpb2Plb** *)
XOpb2Plb_GetErrorStatus *InstancePtr*

Returns the error status indicating the type of error that has occurred.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

The current error status for the OPB to PLB bridge. The possible return values are described in **xopb2plb.h**.

Note:

None.

```
XStatus ( XOpb2Plb *  
XOpb2Plb_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific **XOpb2Plb** instance. Looks up the configuration data for the given device ID and then initializes instance data.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XOpb2Plb** component. Passing in a device id associates the generic **XOpb2Plb** component to a specific device, as chosen by the caller or application developer.

Returns:

- o XST_SUCCESS if everything starts up as expected.
- o XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

```
Xboolean ( XOpb2Plb *  
XOpb2Plb_IsError InstancePtr  
 )
```

Returns XTRUE if there is an error outstanding

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

Boolean XTRUE if there is an error, XFALSE if there is no current error.

Note:

None.

```
XOpb2Plb_Config* ( Xuint16 )  
XOpb2Plb_LookupConfig DeviceId
```

Looks up the device configuration based on the unique device ID. The table OpbPlbConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device to look for.

Returns:

A pointer to the configuration data of the given device, or XNULL if no match is found.

Note:

None.

```
void ( XOpb2Plb * )  
XOpb2Plb_Reset InstancePtr
```

Forces a software-induced reset to occur in the bridge and disables interrupts and the locking functionality in the process.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

Disables interrupts and the locking functionality in the process.

opb2plb/v1_00_a/src/xopb2plb_i.h File Reference

Detailed Description

This file contains data which is shared between files and internal to the **XOpb2Plb** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a	ecm	02/28/02	First release
-------	-----	----------	---------------

1.00a	rpm	05/14/02	Moved identifiers to xopb2plb_1.h
-------	-----	----------	---

```
#include "xopb2plb\_1.h"
```

Variables

[XOpb2Plb_Config](#) [XOpb2Plb_ConfigTable](#) []

Variable Documentation

[XOpb2Plb_Config](#) [XOpb2Plb_ConfigTable](#) ()

[]

The OPB-to-PLB bridge configuration table, sized by the number of instances defined in [xparameters.h](#).

opb2plb/v1_00_a/src/xopb2plb_selftest.c File Reference

Detailed Description

Contains diagnostic self-test functions for the **XOpb2Plb** component. See [xopb2plb.h](#) for more information about the component.

This functionality assumes that the initialize function has been called prior to calling the self-test function.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	11/16/01	First release

```
#include "xstatus.h"  
#include "xopb2plb.h"  
#include "xopb2plb_i.h"
```

Functions

XStatus XOpb2Plb_SelfTest (**XOpb2Plb** *InstancePtr, **Xuint32** TestAddress)

Function Documentation

```
XStatus ( XOpb2Plb *  
XOpb2Plb_SelfTest InstancePtr,  
Xuint32 TestAddress  
)
```

Runs a self-test on the driver/device.

This tests reads the provided *TestAddress* which is intended to cause an error Then the **XOpb2Plb_IsError**() routine is called and if there is an error, the address is checked against the provided location and if they match **XST_SUCCESS** is returned and all errors are then cleared.

If the **XOpb2Plb_IsError**() is called and no error is indicated, **XST_FAILURE** is returned.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

TestAddress is a location that should cause an error on read.

Returns:

XST_SUCCESS if successful, or **XST_FAILURE** if the driver fails the self-test.

Note:

This test assumes that the bus error interrupts to the processor are not enabled.

opbarb/v1_02_a/src/xopbarb.c File Reference

Detailed Description

This component contains the implementation of the **XOpbArb** driver component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.02a	rpm	08/13/01	First release

```
#include "xparameters.h"  
#include "xopbarb.h"  
#include "xio.h"
```

Data Structures

```
struct OptionsMap
```

Functions

```
XStatus XOpbArb_Initialize (XOpbArb *InstancePtr, Xuint16 DeviceId)  
XStatus XOpbArb_SelfTest (XOpbArb *InstancePtr)  
XStatus XOpbArb_SetOptions (XOpbArb *InstancePtr, Xuint32 Options)  
Xuint32 XOpbArb_GetOptions (XOpbArb *InstancePtr)  
XStatus XOpbArb_SetPriorityLevel (XOpbArb *InstancePtr, Xuint8 Level, Xuint8  
MasterId)  
Xuint8 XOpbArb_GetPriorityLevel (XOpbArb *InstancePtr, Xuint8 Level)  
void XOpbArb_SuspendPriorityLevels (XOpbArb *InstancePtr)
```

XStatus XOpbArb_ResumePriorityLevels (**XOpbArb** *InstancePtr)

XStatus XOpbArb_SetParkId (**XOpbArb** *InstancePtr, **Xuint8** MasterId)

XStatus XOpbArb_GetParkId (**XOpbArb** *InstancePtr, **Xuint8** *MasterIdPtr)

XOpbArb_Config * **XOpbArb_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

Xuint32 (**XOpbArb** *
XOpbArb_GetOptions *InstancePtr*)

Gets the options for the arbiter. The options control how the device grants the bus to requesting masters.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

The options of the device. This is a bit mask where a 1 means the option is on, and a 0 means the option is off. See [xopbarb.h](#) for a description of the options.

Note:

None.

XStatus (**XOpbArb** *
XOpbArb_GetParkId *InstancePtr*,
Xuint8 **MasterIdPtr*)

Gets the master ID currently used for bus parking.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

MasterIdPtr is a pointer to a byte that will hold the master ID currently used for bus parking. This is an output parameter. The ID can range from 0 to N, where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

XST_SUCCESS if the park ID is successfully retrieved, or XST_NO_FEATURE if bus

parking is not supported by the device.

Note:

None.

```
Xuint8 ( XOpbArb *  
XOpbArb_GetPriorityLevel InstancePtr,  
 Xuint8 Level  
 )
```

Get the master ID at the given priority level.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Level is the priority level being retrieved. The level can range from 0 (highest) to N (lowest), where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

The master ID assigned to the given priority level. The ID can range from 0 to N, where N is the number of masters minus one.

Note:

If the arbiter is operating in dynamic priority mode, the value returned from this function may not be predictable because the arbiter changes the values on the fly.

```
XStatus ( XOpbArb *  
XOpbArb_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific **XOpbArb** instance. The driver is initialized to allow access to the device registers. In addition, the configuration information is retrieved for the device. Currently, configuration information is stored in **xopbarb_g.c**.

The state of the device after initialization is:

- Fixed or dynamic priority arbitration based on hardware parameter
- Bus parking is disabled

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XOpbArb** component. Passing in a device id associates the generic **XOpbArb** component to a specific device, as chosen by the caller or application developer.

Returns:

The return value is XST_SUCCESS if successful or XST_DEVICE_NOT_FOUND if no configuration data was found for this device.

Note:

None.

XOpbArb_Config* (**Xuint16**)
XOpbArb_LookupConfig *DeviceId*

Looks up the device configuration based on the unique device ID. The table OpbArbConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to match on.

Returns:

A pointer to the configuration information for the matching device instance, or XNULL if no match is found.

Note:

None.

XStatus (**XOpbArb ***)
XOpbArb_ResumePriorityLevels *InstancePtr*

Resumes use of the priority levels by the device. This function is typically called sometime after a call to SuspendPriorityLevels. The application must resume use of priority levels by the device when all modifications are done. If no call is made to this function after use of the priority levels has been suspended, the device will remain in its default priority arbitration mode of assigning priorities based on master IDs. A call to this function has no effect if no prior call was made to suspend the use of priority levels.

Every master must be represented by one and only one fixed priority level before the use of priority levels can be resumed.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

- XST_SUCCESS if the slave is selected successfully.
- XST_OPBARB_INVALID_PRIORITY if there is either a master that is not assigned a priority level, or a master that is assigned two or more priority levels.

Note:

None.

XStatus (**XOpbArb ***)
XOpbArb_SelfTest *InstancePtr*

Runs a self-test on the driver/device. The self-test simply verifies that the arbiter's registers can be read and written. This is an intrusive test in that the arbiter will not be using the priority registers while the test is being performed.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

XST_SUCCESS if successful, or XST_REGISTER_ERROR if a register did not read or write correctly

Note:

The priority level registers are restored after testing them in order to prevent problems with the registers being the same value after the test.

If the arbiter is in dynamic priority mode, this test changes the mode to fixed to ensure that the priority registers aren't changed by the arbiter during this test. The mode is restored to its entry value on exit.

```
XStatus ( XOpbArb *  
XOpbArb_SetOptions InstancePtr,  
 Xuint32 Options  
 )
```

Sets the options for the OPB arbiter. The options control how the device grants the bus to requesting masters.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Options contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. See [xopbarb.h](#) for a description of the options.

Returns:

- XST_SUCCESS if options are successfully set.
- XST_NO_FEATURE if an attempt was made to enable dynamic priority arbitration when the device is configured only for fixed priority arbitration, or an attempt was made to enable parking when bus parking is not supported by the device.
- XST_OPBARB_PARK_NOT_ENABLED if bus parking by park ID was enabled but bus parking itself was not enabled.

Note:

None.

```
XStatus ( XOpbArb *  
XOpbArb_SetParkId InstancePtr,  
 Xuint8 MasterId  
 )
```

Sets the master ID used for bus parking. Bus parking must be enabled and the option to use bus parking by park ID must be set for this park ID to take effect (see the `SetOptions` function). If the option to use bus parking by park ID is set but this function is not called, bus parking defaults to master 0.

Parameters:

InstancePtr is a pointer to the `XOpbArb` instance to be worked on.

MasterId is the ID of the master that will be parked if bus parking is enabled. This master's grant signal remains asserted as long as no other master requests the bus. The ID can range from 0 to N, where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

XST_SUCCESS if the park ID is successfully set, or XST_NO_FEATURE if bus parking is not supported by the device.

Note:

None.

```
XStatus ( XOpbArb *  
XOpbArb_SetPriorityLevel InstancePtr,  
 Xuint8 Level,  
 Xuint8 MasterId  
 )
```

Assigns a master ID to the given priority level. The use of priority levels by the device must be suspended before calling this function. Every master ID must be assigned to one and only one priority level. The driver enforces this before allowing use of priority levels by the device to be resumed.

Parameters:

InstancePtr is a pointer to the `XOpbArb` instance to be worked on.

Level is the priority level being set. The level can range from 0 (highest) to N (lowest), where N is the number of masters minus one. The device currently supports up to 16 masters.

MasterId is the ID of the master being assigned to the priority level. The ID can range from 0 to N, where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

- XST_SUCCESS if the slave is selected successfully.
- XST_OPBARB_NOT_SUSPENDED if priority levels have not been suspended. Before modifying the priority levels, use of priority levels by the device must be suspended.
- XST_OPBARB_NOT_FIXED_PRIORITY if the arbiter is in dynamic mode. It must be in fixed mode to modify the priority levels.

Note:

None.

```
void ( XOpbArb * )
XOpbArb_SuspendPriorityLevels InstancePtr
```

Suspends use of the priority levels by the device. Before modifying priority levels, the application must first suspend use of the levels by the device. This is to prevent possible OPB problems if no master is assigned a priority during the modification of priority levels. The application must resume use of priority levels by the device when all modifications are done. During the time priority levels are suspended, the device reverts to its default behavior of assigning priorities based on master IDs.

This function can be used when the device is configured for either fixed priority arbitration or dynamic priority arbitration. When used during dynamic priority arbitration, the application can configure the priority levels as a starting point for the LRU algorithm.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

None.

Note:

None.

plb2opb/v1_00_a/src/xplb2opb.h File Reference

Detailed Description

This component contains the implementation of the **XPlb2Opb** component. It is the driver for the PLB to OPB Bridge. The bridge converts PLB bus transactions to OPB bus transactions. The hardware acts as a slave on the PLB side and as a master on the OPB side. This interface is necessary for the processor to access OPB based peripherals.

This driver allows the user to access the Bridge registers to support the handling of bus errors and other access errors and determine an appropriate solution if possible.

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

The Xilinx PLB to OPB Bridge is a soft IP core designed for Xilinx FPGAs and contains the following features:

- PLB Slave interface
- 32-bit or 64-bit PLB Slave (configurable via a design parameter)
- Communicates with 32- or 64-bit PLB masters
- Non-burst transfers of 1-8 bytes
- Burst transfers, including word and double-word bursts of fixed or variable lengths, up to depth of burst buffer. Buffer depth configurable via a design parameter
- Limited support for byte, half-word, quad-word and octal-word bursts to maintain PLB compliance
- Cacheline transactions of 4, 8, and 16 words
- Support for transactions not utilized by the PPC405 Core can be eliminated via a design parameter
- PPC405 Core only utilizes single beat, 4, 8, or 16 word line transfers support for burst transactions can be eliminated via a design parameter
- Supports up to 8 PLB masters (number of PLB masters configurable via a design parameter)
- Programmable lower and upper address boundaries
- OPB Master interface with byte enable transfers *Note*: Does not support dynamic bus sizing without additional glue logic
- Data width configurable via a design parameter
- PLB and OPB clocks can have a 1:1, 1:2, 1:4 synchronous relationship
- Bus Error Address Registers (BEAR) and Bus Error Status Registers (BESR) to report errors
- DCR Slave interface provides access to BEAR/BESR
- BEAR, BESR, and DCR interface can be removed from the design via a design parameter
- Posted write buffer. Buffer depth configurable via a design parameter

Device Configuration

The device can be configured in various ways during the FPGA implementation process. The current configuration data contained in [xplb2opb_g.c](#). A table is defined where each entry contains configuration information for device. This information includes such things as the base address of the DCR mapped device, and the number of masters on the bus.

Note:

This driver is not thread-safe. Thread safety must be guaranteed by the layer above this driver if there is a need to access the device from multiple threads.

The Bridge registers reside on the DCR address bus.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	12/7/01	First release
1.00a	rpm	05/14/02	Made configuration typedef/lookup public

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xplb2opb_1.h"
```

Data Structures

```
struct XPlb2Opb
struct XPlb2Opb_Config
```

PLB-OPB bridge error status masks

```
#define XP2O_DRIVING_BEAR_MASK
#define XP2O_ERROR_READ_MASK
#define XP2O_ERROR_TYPE_MASK
#define XP2O_LOCK_ERR_MASK
```

Functions

```
XStatus XPlb2Opb_Initialize (XPlb2Opb *InstancePtr, Xuint16 DeviceId)
void XPlb2Opb_Reset (XPlb2Opb *InstancePtr)
XPlb2Opb_Config * XPlb2Opb_LookupConfig (Xuint16 DeviceId)
Xboolean XPlb2Opb_IsError (XPlb2Opb *InstancePtr)
void XPlb2Opb_ClearErrors (XPlb2Opb *InstancePtr, Xuint8 Master)
Xuint32 XPlb2Opb_GetErrorStatus (XPlb2Opb *InstancePtr, Xuint8 Master)
Xuint32 XPlb2Opb_GetErrorAddress (XPlb2Opb *InstancePtr)
```

Xuint32 XPlb2Opb_GetErrorByteEnables (XPlb2Opb *InstancePtr)
Xuint8 XPlb2Opb_GetMasterDrivingError (XPlb2Opb *InstancePtr)
Xuint8 XPlb2Opb_GetNumMasters (XPlb2Opb *InstancePtr)
void XPlb2Opb_EnableInterrupt (XPlb2Opb *InstancePtr)
void XPlb2Opb_DisableInterrupt (XPlb2Opb *InstancePtr)
XStatus XPlb2Opb_SelfTest (XPlb2Opb *InstancePtr, **Xuint32** TestAddress)

Define Documentation

#define

XP20_DRIVING_BEAR_MASK

XP20_DRIVING_BEAR_MASK	Indicates this master is driving the outstanding error
XP20_ERROR_READ_MASK	Indicates the error is a read error. It is a write error otherwise.
XP20_ERROR_TYPE_MASK	If set, the error was a timeout. Otherwise the error was an error acknowledge
XP20_LOCK_ERR_MASK	Indicates the error is locked and cannot be overwritten.

#define

XP20_ERROR_READ_MASK

XP20_DRIVING_BEAR_MASK	Indicates this master is driving the outstanding error
XP20_ERROR_READ_MASK	Indicates the error is a read error. It is a write error otherwise.
XP20_ERROR_TYPE_MASK	If set, the error was a timeout. Otherwise the error was an error acknowledge
XP20_LOCK_ERR_MASK	Indicates the error is locked and cannot be overwritten.

#define

XP20_ERROR_TYPE_MASK

XP20_DRIVING_BEAR_MASK	Indicates this master is driving the outstanding error
XP20_ERROR_READ_MASK	Indicates the error is a read error. It is a write error otherwise.
XP20_ERROR_TYPE_MASK	If set, the error was a timeout. Otherwise the error was an error acknowledge
XP20_LOCK_ERR_MASK	Indicates the error is locked and cannot be overwritten.

#define

XP20_LOCK_ERR_MASK

XP20_DRIVING_BEAR_MASK	Indicates this master is driving the outstanding error
XP20_ERROR_READ_MASK	Indicates the error is a read error. It is a write error otherwise.
XP20_ERROR_TYPE_MASK	If set, the error was a timeout. Otherwise the error was an error acknowledge
XP20_LOCK_ERR_MASK	Indicates the error is locked and cannot be overwritten.

Function Documentation

```
void
XPlb2Opb_ClearErrors ( XPlb2Opb *
                       InstancePtr,
                       Xuint8
                       Master
                       )
```

Clears any outstanding errors for the given master.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters on the bus

Returns:

None.

Note:

None.

```
void ( XPlb2Opb * )
XPlb2Opb_DisableInterrupt InstancePtr
```

Disables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void ( XPlb2Opb * )
XPlb2Opb_EnableInterrupt InstancePtr
```

Enables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
Xuint32 ( XPlb2Opb * )
XPlb2Opb_GetErrorAddress InstancePtr
```

Returns the OPB Address where the most recent error occurred. If there isn't an outstanding error, the last address in error is returned. 0x00000000 is the initial value coming out of reset.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

Address where error causing access occurred

Note:

Calling **XPlb2Opb_IsError()** is recommended to confirm that an error has occurred prior to calling **XPlb2Opb_GetErrorAddress()** to ensure that the data in the error address register is relevant.

```
Xuint32 ( XPlb2Opb * )  
XPlb2Opb_GetErrorByteEnables InstancePtr
```

Returns the byte-enables asserted during the access causing the error. The enables are parameters in the hardware making the return value dynamic. An example of a 32-bit bus with all 4 byte enables available, **XPlb2Opb_GetErrorByteEnables** will have the value 0xF0000000 returned from a 32-bit access error.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

The byte-enables asserted during the error causing access.

Note:

None.

```
Xuint32 ( XPlb2Opb *  
XPlb2Opb_GetErrorStatus InstancePtr,  
 Xuint8 Master  
 )
```

Returns the error status for the specified master.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters on the bus

Returns:

The current error status for the requested master on the PLB. The status is a bit-mask and the values are described in **xplb2opb.h**.

Note:

None.

Xuint8 (**XPlb2Opb ***)
XPlb2Opb_GetMasterDrivingError *InstancePtr*

Returns the ID of the master which is driving the error condition

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

The ID of the master that is driving the error

Note:

None.

Xuint8 (**XPlb2Opb ***)
XPlb2Opb_GetNumMasters *InstancePtr*

Returns the number of masters associated with the provided instance

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

The number of masters. This is a number from 1 to the maximum of 32.

Note:

The value returned from this call needs to be adjusted if it is to be used as the argument for other calls since the masters are numbered from 0 and this function returns values starting at 1.

```
XStatus ( XPlb2Opb *  
XPlb2Opb_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific **XPlb2Opb** instance. Looks for configuration data for the specified device, then initializes instance data.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XPlb2Opb** component. Passing in a device id associates the generic **XPlb2Opb** component to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if everything starts up as expected.
- XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

```
Xboolean ( XPlb2Opb *  
XPlb2Opb_IsError InstancePtr  
 )
```

Returns XTRUE if there is an error outstanding

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

Boolean XTRUE if there is an error, XFALSE if there is no current error.

Note:

None.

```
XPlb2Opb_Config* ( Xuint16  
XPlb2Opb_LookupConfig DeviceId  
 )
```

Looks up the device configuration based on the unique device ID. The table `PlbOpbConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for

Returns:

A pointer to the configuration data for the given device, or `XNULL` if no match is found.

Note:

None.

```
void ( XPlb2Opb * )  
XPlb2Opb_Reset InstancePtr
```

Forces a software-induced reset to occur in the bridge. Disables interrupts in the process.

Parameters:

InstancePtr is a pointer to the `XPlb2Opb` instance to be worked on.

Returns:

None.

Note:

Disables interrupts in the process.

```
XStatus ( XPlb2Opb *  
XPlb2Opb_SelfTest InstancePtr,  
Xuint32 TestAddress  
)
```

Runs a self-test on the driver/device.

This tests reads the BCR to verify that the proper value is there.

`XST_SUCCESS` is returned if expected value is there, `XST_PLB2OPB_FAIL_SELFTEST` is returned otherwise.

Parameters:

InstancePtr is a pointer to the `XPlb2Opb` instance to be worked on.

TestAddress is a location that could cause an error on read, not used - user definable for hw specific implementations.

Returns:

`XST_SUCCESS` if successful, or `XST_PLB2OPB_FAIL_SELFTEST` if the driver fails self-test.

Note:

This test assumes that the bus error interrupts are not enabled.

Generated on 24 Jun 2004 for Xilinx Device Drivers

XPlb2Opb Struct Reference

```
#include <xplb2opb.h>
```

Detailed Description

The XPlb2Opb driver instance data. The user is required to allocate a variable of this type for every PLB-to_OPB bridge device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [plb2opb/v1_00_a/src/xplb2opb.h](#)
-

plb2opb/v1_00_a/src/xplb2opb_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of PLB-to-OPB bridge devices in the system. Each bridge device should have an entry in this table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	11/16/01	First release
1.00a	rpm	05/14/02	Made configuration typedef/lookup public

```
#include "xplb2opb.h"  
#include "xparameters.h"
```

Variables

[XPlb2Opb_Config](#) [XPlb2Opb_ConfigTable](#) [XPAR_XPLB2OPB_NUM_INSTANCES]

Variable Documentation

[XPlb2Opb_Config](#) [XPlb2Opb_ConfigTable](#)
[XPAR_XPLB2OPB_NUM_INSTANCES]

The PLB-to-OPB bridge configuration table, sized by the number of instances defined in [xparameters.h](#).

XPlb2Opb_Config Struct Reference

```
#include <xplb2opb.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xuint8 NumMasters](#)

Field Documentation

[Xuint32 XPlb2Opb_Config::](#)

BaseAddress

Base address of
device

[Xuint16 XPlb2Opb_Config::](#)

DeviceId

Unique ID of
device

[Xuint8 XPlb2Opb_Config::](#)

NumMasters

Number of masters on the
bus

The documentation for this struct was generated from the following file:

- [plb2opb/v1_00_a/src/xplb2opb.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

plb2opb/v1_00_a/src/xplb2opb_i.h File Reference

Detailed Description

This file contains identifiers and low-level macros that can be used to access the device directly. See [xplb2opb.h](#) for the high-level driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	05/10/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

Defines

```
#define XPlb2Opb_mGetErrorDetectReg(BaseAddress)  
#define XPlb2Opb_mSetErrorDetectReg(BaseAddress, Mask)  
#define XPlb2Opb_mGetMasterDrivingReg(BaseAddress)  
#define XPlb2Opb_mGetReadWriteReg(BaseAddress)  
#define XPlb2Opb_mGetErrorTypeReg(BaseAddress)  
#define XPlb2Opb_mGetLockBitReg(BaseAddress)  
#define XPlb2Opb_mGetErrorAddressReg(BaseAddress)  
#define XPlb2Opb_mGetByteEnableReg(BaseAddress)  
#define XPlb2Opb_mSetControlReg(BaseAddress, Mask)  
#define XPlb2Opb_mGetControlReg(BaseAddress)
```

Define Documentation

```
#define XPlb2Opb_mGetByteEnableReg( BaseAddress )
```

Get the error address byte enable register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit error address byte enable register contents.

Note:

None.

```
#define XPlb2Opb_mGetControlReg( BaseAddress )
```

Get the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the control register.

Note:

None.

```
#define XPlb2Opb_mGetErrorAddressReg( BaseAddress )
```

Get the error address (or BEAR), which is the address that just caused the error.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit error address.

Note:

None.

```
#define XPlb2Opb_mGetErrorDetectReg( BaseAddress )
```

Get the error detect register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the error detect register.

Note:

None.

```
#define XPlb2Opb_mGetErrorTypeReg( BaseAddress )
```

Get the value of the error type register, which indicates whether the error is a timeout or a bus error.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the BESR Lock error register.

Note:

None.

#define XPlb2Opb_mGetLockBitReg(BaseAddress)

Get the value of the lock bit register, which indicates whether the master has locked the error registers.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the BESR Lock error register.

Note:

None.

#define XPlb2Opb_mGetMasterDrivingReg(BaseAddress)

Get the master driving the error, if any.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the BESR Master driving error register.

Note:

None.

#define XPlb2Opb_mGetReadWriteReg(BaseAddress)

Get the value of the Read-Not-Write register, which indicates whether the error is a read error or write error.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the BESR RNW error register.

Note:

None.

```
#define XPlb2Opb_mSetControlReg( BaseAddress,  
                                Mask          )
```

Set the control register to the given value.

Parameters:

BaseAddress is the base address of the device

Mask is the value to write to the control register.

Returns:

None.

Note:

None.

```
#define XPlb2Opb_mSetErrorDetectReg( BaseAddress,  
                                     Mask          )
```

Set the error detect register.

Parameters:

BaseAddress is the base address of the device

Mask is the 32-bit value to write to the error detect register.

Note:

None.

packet_fifo/v2_00_a/src/xpacket_fifo_v2_00_a.c

File Reference

Detailed Description

Contains functions for the [XPacketFifoV200a](#) component. See [xpacket_fifo_v2_00_a.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
2.00a	ecm	12/30/02	First release
2.00a	rmm	05/14/03	Fixed diab compiler warnings
2.00a	rpm	10/22/03	Created and made use of Level 0 driver
2.00a	rmm	02/24/04	Added WriteDRE function.

```
#include "xbasic_types.h"
#include "xio.h"
#include "xstatus.h"
#include "xpacket_fifo_v2_00_a.h"
```

Functions

XStatus XPacketFifoV200a_Initialize ([XPacketFifoV200a](#) *InstancePtr, [Xuint32](#) RegBaseAddress, [Xuint32](#) DataBaseAddress)

XStatus XPacketFifoV200a_SelfTest ([XPacketFifoV200a](#) *InstancePtr, [Xuint32](#) FifoType)

XStatus XPacketFifoV200a_Read ([XPacketFifoV200a](#) *InstancePtr, [Xuint8](#) *BufferPtr, [Xuint32](#) ByteCount)

XStatus **XPacketFifoV200a_Write** (**XPacketFifoV200a** *InstancePtr, **Xuint8** *BufferPtr, **Xuint32** ByteCount)

XStatus **XPacketFifoV200a_WriteDre** (**XPacketFifoV200a** *InstancePtr, **Xuint8** *BufferPtr, **Xuint32** ByteCount)

Function Documentation

```
XStatus XPacketFifoV200a_Initialize (XPacketFifoV200a *InstancePtr,  
Xuint32 RegBaseAddress,  
Xuint32 DataBaseAddress)
```

This function initializes a packet FIFO. Initialization resets the FIFO such that it's empty and ready to use.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

RegBaseAddress contains the base address of the registers for the packet FIFO.

DataBaseAddress contains the base address of the data for the packet FIFO.

Returns:

Always returns XST_SUCCESS.

Note:

None.

```
XStatus XPacketFifoV200a_Read (XPacketFifoV200a *InstancePtr,  
Xuint8 *BufferPtr,  
Xuint32 ByteCount)
```

Read data from a FIFO and puts it into a specified buffer. This function invokes the Level 0 driver function to read the FIFO.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

BufferPtr points to the memory buffer to write the data into. This buffer must be 32 bit aligned or an alignment exception could be generated. Since this buffer is a byte buffer, the data is assumed to be endian independent.

ByteCount contains the number of bytes to read from the FIFO. This number of bytes must be present in the FIFO or an error will be returned.

Returns:

- XST_SUCCESS if the operation was successful
- XST_PFIFO_LACK_OF_DATA if the number of bytes specified by the byte count is not present in the FIFO.

Note:

None.

```
XStatus ( XPacketFifoV200a *  
XPacketFifoV200a_SelfTest InstancePtr,  
Xuint32 FifoType  
)
```

This function performs a self-test on the specified packet FIFO. The self test resets the FIFO and reads a register to determine if it is the correct reset value. This test is destructive in that any data in the FIFO will be lost.

Parameters:

InstancePtr is a pointer to the packet FIFO to be operated on.

FifoType specifies the type of FIFO, read or write, for the self test. The FIFO type is specified by the values XPF_V200A_READ_FIFO_TYPE or XPF_V200A_WRITE_FIFO_TYPE.

Returns:

XST_SUCCESS is returned if the selftest is successful, or XST_PFIFO_BAD_REG_VALUE indicating that the value readback from the occupancy/vacancy count register after a reset does not match the specified reset value.

Note:

None.

```

XStatus ( XPacketFifoV200a *
XPacketFifoV200a_Write InstancePtr,
                          BufferPtr,
                          Xuint8 *,
                          Xuint32
                          ByteCount
                          )

```

Write data into a packet FIFO. This function invokes the Level 0 driver function to read the FIFO.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

BufferPtr points to the memory buffer that data is to be read from and written into the FIFO. Since this buffer is a byte buffer, the data is assumed to be endian independent. This buffer must be 32 bit aligned or an alignment exception could be generated.

ByteCount contains the number of bytes to read from the buffer and to write to the FIFO.

Returns:

- XST_SUCCESS is returned if the operation succeeded.
- XST_PFIFO_NO_ROOM is returned if there is not enough room in the FIFO to hold the specified bytes.

Note:

None.

```

XStatus ( XPacketFifoV200a *
XPacketFifoV200a_WriteDre InstancePtr,
                          BufferPtr,
                          Xuint8 *,
                          Xuint32
                          ByteCount
                          )

```

Write data into a packet FIFO configured with the Data Realignment engine (DRE). There are no alignment restrictions. The FIFO can be written on any byte boundary. The FIFO must be at least 32 bits wide.

Parameters:

InstancePtr contains a pointer to the FIFO to operate on.

BufferPtr points to the memory buffer that data is to be read from and written into the FIFO. Since this buffer is a byte buffer, the data is assumed to be endian independent.

ByteCount contains the number of bytes to read from the buffer and to write to the FIFO.

Returns:

XST_SUCCESS is returned if the operation succeeded. If there is not enough room in the FIFO to hold the specified bytes, XST_PFIFO_NO_ROOM is returned.

Note:

This function assumes that if the device inserting data into the FIFO is a byte device, the order of the bytes in each 32/64 bit word is from the most significant byte to the least significant byte.

pciarb/v1_00_a/src/xpciarb_i.h File Reference

Detailed Description

This header file contains the low-level device driver for the OPB PCI Arbiter device. Please refer to the [xpciarb.h](#) file for more information.

The hardware device has a parameter that allows the designer to remove the processor interface (i.e., registers and bus access) during hardware build time. This driver will obviously not function correctly if the processor interface to the device has been removed.

When enabling the use of the park register, the user should first write a valid one-hot encoded value to the park register, then set the control bit to enable the use of the register. This ensures that once the park register usage is enabled, the register contains a valid value. The user is responsible for understanding the register set of the device and using it properly with this low-level driver interface.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	01/06/04	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

Defines

```
#define XPCA_RESET_OFFSET  
#define XPCA_MIR_OFFSET  
#define XPCA_CR_OFFSET  
#define XPCA_PARK_OFFSET
```

```
#define XPCA_RESET_MASK
#define XPCA_CR_FIXED_ENABLE_MASK
#define XPCA_CR_PARK_ENABLE_MASK
#define XPCiArb_mReadReg(BaseAddress, RegOffset)
#define XPCiArb_mWriteReg(BaseAddress, RegOffset, Data)
```

Define Documentation

```
#define  
XPCA_CR_FIXED_ENABLE_MASK
```

Fixed priority
enable

```
#define  
XPCA_CR_OFFSET
```

Control
Register

```
#define  
XPCA_CR_PARK_ENABLE_MASK
```

Park
enable

```
#define  
XPCA_MIR_OFFSET
```

Device
MIR

```
#define  
XPCA_PARK_OFFSET
```

Park Master
Register

```
#define  
XPCA_RESET_MASK
```

Value used to reset the device

```
#define  
XPCA_RESET_OFFSET
```

Soft Reset and IPIF
MIR

```
#define XPCIARB_mReadReg( BaseAddress,  
                        RegOffset )
```

Read a register of the arbiter. This macro provides register access to all the arbiter registers using the register offsets defined above.

Parameters:

BaseAddress contains the base address of the device.

RegOffset is the offset of the register to read.

Returns:

The contents of the register.

Note:

C-style Signature: Xuint32 XPCIARB_mReadReg(Xuint32 BaseAddress, Xuint32 RegOffset)

```
#define XPCIARB_mWriteReg( BaseAddress,  
                        RegOffset,  
                        Data )
```

Write a register of the arbiter. This macro provides register access to all the arbiter registers using the register offsets defined above.

Parameters:

BaseAddress contains the base address of the device.

RegOffset is the offset of the register to write.

Data is the value to write to the register.

Returns:

None.

Note:

C-style Signature: void XPCIArb_mWriteReg(Xuint32 BaseAddress, Xuint32 RegOffset, Xuint32 Data)

pciarb/v1_00_a/src/xpciarb.h File Reference

Detailed Description

This file contains the device driver API definition for the OPB PCI Arbiter device. The device provides arbitration among several PCI Master devices. The user should refer to the hardware device specification for more details of operation.

Hardware Features

The Xilinx PCI Arbiter is a soft IP core designed for Xilinx FPGAs and contains the following features:

- Fixed or dynamic arbitration (dynamic is the default)
- Bus parking with programmable park master

The hardware device has a parameter that allows the designer to remove the processor interface (i.e., registers and bus access) during hardware build time. This driver will obviously not function correctly if the processor interface to the device has been removed.

Bus Parking

Bus parking is always enabled in the arbiter. The default PCI master that gets bus parking privileges is set at hardware build time by a hardware parameter. This default master can be overridden by software using the [XPciArb_EnableParkReg\(\)](#) function of this driver. When software disables use of the park register, using [XPciArb_DisableParkReg\(\)](#), the arbiter returns to using the default master as specified by the hardware parameter.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00a rpm 01/06/04 First release

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xpciarb_1.h"
```

Data Structures

```
struct XPCIArb
struct XPCIArb_Config
```

Functions

```
XStatus XPCIArb_Initialize (XPCIArb *InstancePtr, Xuint16 DeviceId)
void XPCIArb_Reset (XPCIArb *InstancePtr)
XPCIArb_Config * XPCIArb_LookupConfig (Xuint16 DeviceId)
void XPCIArb_EnableFixed (XPCIArb *InstancePtr)
void XPCIArb_DisableFixed (XPCIArb *InstancePtr)
void XPCIArb_EnableParkReg (XPCIArb *InstancePtr, unsigned int MasterId)
void XPCIArb_DisableParkReg (XPCIArb *InstancePtr)
XStatus XPCIArb_SelfTest (XPCIArb *InstancePtr)
```

Function Documentation

```
void XPCIArb_DisableFixed (XPCIArb *InstancePtr)
```

Disable fixed priority arbitration. The arbiter returns to its default dynamic arbitration scheme, where the least recently used master ID gets highest priority.

Parameters:

InstancePtr is a pointer to the **XPCIArb** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XPCIArb * )
XPciArb_DisableParkReg InstancePtr
```

Disable the use of the park register in the arbiter. The device returns to its default state of using the value of a hardware parameter as the master that gets bus parking privileges.

Parameters:

InstancePtr is a pointer to the **XPCIArb** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XPCIArb * )
XPciArb_EnableFixed InstancePtr
```

Enable fixed priority arbitration. The arbitration priority will remain constant with master ID 0 at the highest priority and master ID (C_NUM_PCI_MSTRS - 1) at the lowest priority, C_NUM_PCI_MSTRS is the hardware parameter that specifies the number of PCI masters. This function overrides the default state of the arbiter device, which is to use dynamic arbitration.

Parameters:

InstancePtr is a pointer to the **XPCIArb** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XPCIArb *
XPciArb_EnableParkReg InstancePtr,
unsigned int MasterId
)
```

Enable the use of the park register of the arbiter. This function sets the value of the park register and enables its use. The device will use the contents of the park register to determine which master gets to park on the bus. By default, the device uses the value of a hardware parameter as the master that gets bus parking privileges.

Parameters:

InstancePtr is a pointer to the **XPciArb** instance to be worked on.

MasterId is the master ID, from 0 to (C_NUM_PCI_MSTRS - 1), that will have PCI bus parking privileges.

Returns:

None

Note:

None.

```
XStatus ( XPciArb *  
XPciArb_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific **XPciArb** instance. Looks up the configuration for the given device instance and initialize the instance structure.

Parameters:

InstancePtr is a pointer to the **XPciArb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XPciArb** instance.

Returns:

- XST_SUCCESS if everything initializes as expected.
- XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

```
XPciArb_Config* ( Xuint16 )  
XPciArb_LookupConfig DeviceId
```

Looks up the device configuration based on the unique device ID. The table `XPCIArb_ConfigTable`, defined in `xpciarb_g.c`, contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for.

Returns:

A pointer to the configuration data for the device, or XNULL if no match is found.

Note:

None.

```
void ( XPCIArb * )  
XPCIArb_Reset InstancePtr
```

Forces a software reset to occur in the arbiter. The device returns to its default state of dynamic priority arbitration and bus parking on the PCI master specified by a hardware parameter.

Parameters:

InstancePtr is a pointer to the `XPCIArb` instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( XPCIArb * )  
XPCIArb_SelfTest InstancePtr
```

Run a self-test on the driver/device. The test simply writes to the control register of the device, enabling fixed priority arbitration, and verifies the value was written successfully. It then resets the device, returning the arbiter to its default state (dynamic priority arbitration).

Note that this is a destructive test in that a reset of the device is performed.

Parameters:

InstancePtr is a pointer to the **XPciArb** instance to be worked on.

Returns:

XST_SUCCESS if successful, or XST_FAILURE if the self test fails.

Note:

None.

pciARB/v1_00_a/src/xpciARB.c File Reference

Detailed Description

This file contains driver API functions that can be used to access the arbiter device. Please refer to the [xpciARB.h](#) header file for more information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	01/06/04	First release

```
#include "xparameters.h"  
#include "xio.h"  
#include "xpciARB.h"
```

Functions

XStatus **XPciArb_Initialize** (**XPciArb** *InstancePtr, **Xuint16** DeviceId)

void **XPciArb_Reset** (**XPciArb** *InstancePtr)

XPciArb_Config * **XPciArb_LookupConfig** (**Xuint16** DeviceId)

void **XPciArb_EnableFixed** (**XPciArb** *InstancePtr)

void **XPciArb_DisableFixed** (**XPciArb** *InstancePtr)

void **XPciArb_EnableParkReg** (**XPciArb** *InstancePtr, unsigned int MasterId)

void **XPciArb_DisableParkReg** (**XPciArb** *InstancePtr)

Variables

Function Documentation

```
void (XPciArb * )
XPciArb_DisableFixed InstancePtr
```

Disable fixed priority arbitration. The arbiter returns to its default dynamic arbitration scheme, where the least recently used master ID gets highest priority.

Parameters:

InstancePtr is a pointer to the [XPciArb](#) instance to be worked on.

Returns:

None.

Note:

None.

```
void (XPciArb * )
XPciArb_DisableParkReg InstancePtr
```

Disable the use of the park register in the arbiter. The device returns to its default state of using the value of a hardware parameter as the master that gets bus parking priveleges.

Parameters:

InstancePtr is a pointer to the [XPciArb](#) instance to be worked on.

Returns:

None.

Note:

None.

```
void (XPciArb * )
XPciArb_EnableFixed InstancePtr
```

Enable fixed priority arbitration. The arbitration priority will remain constant with master ID 0 at the highest priority and master ID (C_NUM_PCI_MSTRS - 1) at the lowest priority, C_NUM_PCI_MSTRS is the hardware parameter that specifies the number of PCI masters. This function overrides the default state of the arbiter device, which is to use dynamic arbitration.

Parameters:

InstancePtr is a pointer to the **XPciArb** instance to be worked on.

Returns:

None.

Note:

None.

```
void  
XPciArb_EnableParkReg ( XPciArb *  
                        InstancePtr,  
                        unsigned int MasterId  
                        )
```

Enable the use of the park register of the arbiter. This function sets the value of the park register and enables its use. The device will use the contents of the park register to determine which master gets to park on the bus. By default, the device uses the value of a hardware parameter as the master that gets bus parking privileges.

Parameters:

InstancePtr is a pointer to the **XPciArb** instance to be worked on.

MasterId is the master ID, from 0 to (C_NUM_PCI_MSTRS - 1), that will have PCI bus parking privileges.

Returns:

None

Note:

None.

```
XStatus  
XPciArb_Initialize ( XPciArb *  
                    InstancePtr,  
                    Xuint16 DeviceId  
                    )
```

Initializes a specific **XPCIarb** instance. Looks up the configuration for the given device instance and initialize the instance structure.

Parameters:

InstancePtr is a pointer to the **XPCIarb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XPCIarb** instance.

Returns:

- XST_SUCCESS if everything initializes as expected.
- XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

```
XPCIarb_Config* ( Xuint16 )  
XPCIarb_LookupConfig DeviceId
```

Looks up the device configuration based on the unique device ID. The table XPCIarb_ConfigTable, defined in **xpciarm_g.c**, contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for.

Returns:

A pointer to the configuration data for the device, or XNULL if no match is found.

Note:

None.

```
void ( XPCIarb * )  
XPCIarb_Reset InstancePtr
```

Forces a software reset to occur in the arbiter. The device returns to its default state of dynamic priority arbitration and bus parking on the PCI master specified by a hardware parameter.

Parameters:

InstancePtr is a pointer to the [XPciArb](#) instance to be worked on.

Returns:

None.

Note:

None.

Variable Documentation

[XPciArb_Config](#) XPciArb_ConfigTable ()

[]

The PCI Arbiter configuration table, sized by the number of instances defined in [xparameters.h](#).

XPciArb Struct Reference

```
#include <xpciarb.h>
```

Detailed Description

The driver's instance data. The user is required to allocate a variable of this type for every OPB PCI Arbiter device in the system. A pointer to a variable of this type is then passed to the driver API functions.

Data Fields

[Xuint32 BaseAddress](#)

[Xuint32 IsReady](#)

[Xuint8 NumMasters](#)

Field Documentation

[Xuint32 XPciArb::](#)

BaseAddress

Base address of device

[Xuint32 XPciArb::](#)

IsReady

Device is initialized and ready

Xuint8 XPciArb::

NumMasters

Number of masters for this
arbiter

The documentation for this struct was generated from the following file:

- pciarb/v1_00_a/src/[xpciadb.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

XPciArb_Config Struct Reference

```
#include <xpciarb.h>
```

Detailed Description

This typedef contains configuration information for a device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xuint8 NumMasters](#)

Field Documentation

[Xuint32 XPciArb_Config::](#)

BaseAddress

Register base
address

[Xuint16 XPciArb_Config::](#)

DeviceId

Unique ID of
device

[Xuint8 XPciArb_Config::](#)

NumMasters

The number of PCI
masters

The documentation for this struct was generated from the following file:

- pciarb/v1_00_a/src/[xpci**arb**.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

pciARB/v1_00_a/src/xpciARB_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of PCI Arbiter devices in the system. Each arbiter device should have an entry in this table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	01/06/04	First release

```
#include "xparameters.h"  
#include "xpciARB.h"
```

Variables

[XPciArb_Config](#) [XPciArb_ConfigTable](#) [XPAR_XPCIARB_NUM_INSTANCES]

Variable Documentation

[XPciArb_Config](#) [XPciArb_ConfigTable](#)
[XPAR_XPCIARB_NUM_INSTANCES]

The PCI Arbiter configuration table, sized by the number of instances defined in [xparameters.h](#).

pci/v1_00_b/src/xpci_intr.c File Reference

Detailed Description

Implements PCI interrupt processing functions for the **XPCI** component. See [xpci.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	03/25/02	Original code

```
#include "xpci.h"
```

Functions

```
void XPCI_InterruptGlobalEnable (XPCI *InstancePtr)  
void XPCI_InterruptGlobalDisable (XPCI *InstancePtr)  
void XPCI_InterruptEnable (XPCI *InstancePtr, XUINT32 Mask)  
void XPCI_InterruptDisable (XPCI *InstancePtr, XUINT32 Mask)  
void XPCI_InterruptClear (XPCI *InstancePtr, XUINT32 Mask)
```

```
XUINT32 XPCI_InterruptGetEnabled (XPCI *InstancePtr)
```

```
XUINT32 XPCI_InterruptGetStatus (XPCI *InstancePtr)
```

```
XUINT32 XPCI_InterruptGetPending (XPCI *InstancePtr)
```

```
XUINT32 XPCI_InterruptGetHighestPending (XPCI *InstancePtr)
```

```
void XPCI_InterruptPciEnable (XPCI *InstancePtr, XUINT32 Mask)
```

```
void XPCI_InterruptPciDisable (XPCI *InstancePtr, XUINT32 Mask)
```

```
void XPCI_InterruptPciClear (XPCI *InstancePtr, XUINT32 Mask)
```

```
XUINT32 XPCI_InterruptPciGetEnabled (XPCI *InstancePtr)
```

Xuint32 Xpci_InterruptPciGetStatus (Xpci *InstancePtr)

void **Xpci_AckSend** (Xpci *InstancePtr, **Xuint32** Vector)

Xuint32 Xpci_AckRead (Xpci *InstancePtr)

void **Xpci_SpecialCycle** (Xpci *InstancePtr, **Xuint32** Data)

Function Documentation

```
Xuint32 ( Xpci * )  
Xpci_AckRead InstancePtr
```

Read the contents of the PCI interrupt acknowledge vector register.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

System dependent interrupt vector.

Note:

None

```
void ( Xpci * )  
Xpci_AckSend InstancePtr,  
 Xuint32 Vector  
 )
```

Generate a PCI interrupt acknowledge bus cycle with the given vector.

Parameters:

InstancePtr is the PCI component to operate on.

Vector is a system dependent interrupt vector to place on the bus.

Note:

None

```
void  
Xpci_InterruptClear ( Xpci *  
                    InstancePtr,  
                    Xuint32 Mask  
                    )
```

Clear device level pending interrupts with the provided mask.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XPCI_IPIF_INT_MASK

Note:

None

```
void  
Xpci_InterruptDisable ( Xpci *  
                      InstancePtr,  
                      Xuint32 Mask  
                      )
```

Disable device interrupts. Any component interrupts enabled through [Xpci_InterruptPciEnable\(\)](#) and/or the DMA driver will no longer have any effect. The component interrupt settings will be retained however.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to disable. Bits set to 1 are disabled. The mask is formed by OR'ing bits from XPCI_IPIF_INT_MASK

Note:

None

```
void  
Xpci_InterruptEnable ( Xpci *  
                     InstancePtr,  
                     Xuint32 Mask  
                     )
```

Enable device interrupts. Device interrupts must be enabled by this function before component interrupts enabled by [XPci_InterruptPciEnable\(\)](#) and/or the DMA driver have any effect.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to enable. Bit positions of 1 are enabled. The mask is formed by OR'ing bits from XPCI_IPIF_INT_MASK.

Note:

None

Xuint32 (**XPci ***)
XPci_InterruptGetEnabled *InstancePtr*

Returns the device level interrupt enable mask as set by [XPci_InterruptEnable\(\)](#).

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of bits made from XPCI_IPIF_INT_MASK.

Note:

None

Xuint32 (**XPci ***)
XPci_InterruptGetHighestPending *InstancePtr*

Returns the highest priority pending device interrupt that has been enabled by [XPci_InterruptEnable\(\)](#).

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask is one set bit made from XPCI_IPIF_INT_MASK or zero if no interrupts are pending.

Note:

None

Xuint32 (**XPci ***)
XPci_InterruptGetPending *InstancePtr*

Returns the pending status of device level interrupt signals that have been enabled by **XPci_InterruptEnable()**. Any bit in the mask set to 1 indicates that an interrupt is pending from the given component

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of bits made from XPCI_IPIF_INT_MASK or zero if no interrupts are pending.

Note:

None

Xuint32 (**XPci ***)
XPci_InterruptGetStatus *InstancePtr*

Returns the status of device level interrupt signals. Any bit in the mask set to 1 indicates that the given component has asserted an interrupt condition.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of bits made from XPCI_IPIF_INT_MASK.

Note:

The interrupt status indicates the status of the device irregardless if the interrupts from the devices have been enabled or not through **XPci_InterruptEnable()**.

void (**XPci ***)
XPci_InterruptGlobalDisable *InstancePtr*

Disable the core's interrupt output signal. Interrupts enabled through [XPci_InterruptEnable\(\)](#) and [XPci_InterruptPciEnable\(\)](#) will no longer be passed through until the IPIF global enable bit is set by [XPci_InterruptGlobalEnable\(\)](#).

Parameters:

InstancePtr is the PCI component to operate on.

Note:

None

```
void XPci_InterruptGlobalEnable ( XPci * InstancePtr )
```

Enable the core's interrupt output signal. Interrupts enabled through [XPci_InterruptEnable\(\)](#) and [XPci_InterruptPciEnable\(\)](#) will not be passed through until the IPIF global enable bit is set by this function.

Parameters:

InstancePtr is the PCI component to operate on.

Note:

None

```
void XPci_InterruptPciClear ( XPci * InstancePtr, Xuint32 Mask )
```

Clear PCI bridge specific interrupt status bits with the provided mask.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XPCI_IR_MASK

Note:

None

```
void  
XpCi_InterruptPciDisable ( XpCi *  
                          InstancePtr,  
                          Xuint32 Mask  
                          )
```

Disable PCI bridge specific interrupts.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to disable. Bits set to 1 are disabled. The mask is formed by OR'ing bits from XPCI_IR_MASK

Note:

None

```
void  
XpCi_InterruptPciEnable ( XpCi *  
                          InstancePtr,  
                          Xuint32 Mask  
                          )
```

Enable PCI bridge specific interrupts. Before this function has any effect in generating interrupts, the function [XpCi_InterruptEnable\(\)](#) must be invoked with the XPCI_IPIF_INT_PCI bit set.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to enable. Bit positions of 1 are enabled. The mask is formed by OR'ing bits from XPCI_IR_MASK.

Note:

None

```
Xuint32  
XpCi_InterruptPciGetEnabled ( XpCi *  
                             InstancePtr
```

Get the PCI bridge specific interrupts enabled through **XPci_InterruptPciEnable()**. Bits set to 1 mean that interrupt source is enabled.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of enabled bits made from XPCI_IR_MASK.

Note:

None

```
Xuint32 ( XPci * )  
XPci_InterruptPciGetStatus InstancePtr
```

Get the status of PCI bridge specific interrupts that have been asserted Bits set to 1 are in an asserted state. Bits may be set to 1 irregardless of whether they have been enabled or not though **XPci_InterruptPciEnable()**. To get the pending interrupts, AND the results of this function with **XPci_InterruptPciGetEnabled()**.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of enabled bits made from XPCI_IR_MASK.

Note:

None

```
void ( XPci *  
XPci_SpecialCycle InstancePtr,  
Xuint32 Data  
)
```

Broadcasts a message to all listening PCI targets.

Parameters:

InstancePtr is the PCI component to operate on.

Data is the data to broadcast.

Note:

None

Generated on 24 Jun 2004 for Xilinx Device Drivers

XPci Struct Reference

```
#include <xpci.h>
```

Detailed Description

The XPci driver instance data. The user is required to allocate a variable of this type for every PCI device in the system that will be using this API. A pointer to a variable of this type is passed to the driver API functions defined here.

Data Fields

[Xuint32 RegBaseAddr](#)

[Xuint32 DmaRegBaseAddr](#)

[Xuint32 IsReady](#)

[Xuint8 DmaType](#)

Field Documentation

[Xuint32 XPci::](#)

DmaRegBaseAddr

Base address of DMA (if included)

[Xuint8 XPci::](#)

DmaType

Type of DMA (if enabled), see XPCI_DMA_TYPE constants in [xpci_1.h](#)

[h](#)

Xuint32 XPci::

IsReady

Device is initialized and ready

Xuint32 XPci::

RegBaseAddr

Base address of registers

The documentation for this struct was generated from the following file:

- [pci/v1_00_b/src/xpci.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

pci/v1_00_b/src/xpci_i.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined and more PCI documentation is in [xpci.h](#).

PCI configuration read/write macro functions can be changed so that data is swapped before being written to the configuration address/data registers. As delivered in this file, these macros do the swap. Change the definitions of [XIo_InPci\(\)](#) and [XIo_OutPci\(\)](#) in this file to suit system needs.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	05/19/02	First release
1.00b	rpm	11/05/03	Changed to use XIo_InSwap32 and XIo_OutSwap32 swap routines. This corresponds to the opb_pci core change that preserves byte address through the bridge.
1.00b	rpm	02/27/04	Added _ASMLANGUAGE ifndef statements. This is needed because the VxWorks BSP config.h file includes this file and the config.h file is included in .s source files.

```
#include "xbasic_types.h"
#include "xio.h"
#include "xipif_v1_23_b.h"
```

Registers

Register offsets for this device. Note that the following IPIF registers are implemented. Macros are defined to specifically access these registers without knowing which version of IPIF being used.

```
#define XPCI_PREOVRD_OFFSET
#define XPCI_IAR_OFFSET
#define XPCI_SC_DATA_OFFSET
#define XPCI_CAR_OFFSET
#define XPCI_CDR_OFFSET
#define XPCI_BUSNO_OFFSET
#define XPCI_STATCMD_OFFSET
#define XPCI_STATV3_OFFSET
#define XPCI_INHIBIT_OFFSET
#define XPCI_LMADDR_OFFSET
#define XPCI_LMA_R_OFFSET
#define XPCI_LMA_W_OFFSET
#define XPCI_SERR_R_OFFSET
#define XPCI_SERR_W_OFFSET
#define XPCI_PIADDR_OFFSET
#define XPCI_PIA_R_OFFSET
#define XPCI_PIA_W_OFFSET
```

Interrupt Status and Enable Register bitmaps and masks

Bit definitions for the interrupt status register and interrupt enable registers.

```
#define XPCI_IR_MASK
#define XPCI_IR_LM_SERR_R
#define XPCI_IR_LM_PERR_R
#define XPCI_IR_LM_TA_R
#define XPCI_IR_LM_SERR_W
#define XPCI_IR_LM_PERR_W
#define XPCI_IR_LM_TA_W
#define XPCI_IR_LM_MA_W
#define XPCI_IR_LM_BR_W
#define XPCI_IR_LM_BRD_W
#define XPCI_IR_LM_BRT_W
```

```
#define XPCI_IR_LM_BRANGE_W
#define XPCI_IR_PI_SERR_R
#define XPCI_IR_PI_SERR_W
```

Inhibit transfers on errors register bitmaps and masks.

These bits control whether subsequent PCI transactions are allowed after an error occurs. Bits set to 1 inhibit further transactions, while bits set to 0 allow further transactions after an error.

```
#define XPCI_INHIBIT_MASK
#define XPCI_INHIBIT_LOCAL_BUS_R
#define XPCI_INHIBIT_LOCAL_BUS_W
#define XPCI_INHIBIT_PCI_R
#define XPCI_INHIBIT_PCI_W
```

PCI configuration status & command register bitmaps and masks.

Bit definitions for the PCI configuration status & command register. The definition of this register is standard for PCI devices.

```
#define XPCI_STATCMD_IO_EN
#define XPCI_STATCMD_MEM_EN
#define XPCI_STATCMD_BUSM_EN
#define XPCI_STATCMD_SPECIALCYC
#define XPCI_STATCMD_MEMWR_INV_EN
#define XPCI_STATCMD_VGA_SNOOP_EN
#define XPCI_STATCMD_PARITY
#define XPCI_STATCMD_STEPPING
#define XPCI_STATCMD_SERR_EN
#define XPCI_STATCMD_BACK_EN
#define XPCI_STATCMD_INT_DISABLE
#define XPCI_STATCMD_INT_STATUS
#define XPCI_STATCMD_66MHZ_CAP
#define XPCI_STATCMD_MPERR
#define XPCI_STATCMD_DEVSEL_MSK
#define XPCI_STATCMD_DEVSEL_FAST
#define XPCI_STATCMD_DEVSEL_MED
#define XPCI_STATCMD_TGTABRT_SIG
```

```
#define XPCI_STATCMD_TGTABRT_RCV
#define XPCI_STATCMD_MSTABRT_RCV
#define XPCI_STATCMD_SERR_SIG
#define XPCI_STATCMD_PERR_DET
#define XPCI_STATCMD_ERR_MASK
```

V3 core transaction status register bitmaps and masks.

Bit definitions for the V3 core transaction status register. This register consists of status information on V3 core transactions.

```
#define XPCI_STATV3_MASK
#define XPCI_STATV3_DATA_XFER
#define XPCI_STATV3_TRANS_END
#define XPCI_STATV3_NORM_TERM
#define XPCI_STATV3_TGT_TERM
#define XPCI_STATV3_DISC_WODATA
#define XPCI_STATV3_DISC_WDATA
#define XPCI_STATV3_TGT_ABRT
#define XPCI_STATV3_MASTER_ABRT
#define XPCI_STATV3_PCI_RETRY_R
#define XPCI_STATV3_PCI_RETRY_W
#define XPCI_STATV3_WRITE_BUSY
```

Bus number and subordinate bus number register bitmaps and masks

```
#define XPCI_BUSNO_BUS_MASK
#define XPCI_BUSNO_SUBBUS_MASK
```

Local bus master address register bitmaps and masks

Bit definitions for the local bus master address definition register. This register defines the meaning of the address stored in the local bus master read (XPCI_LMA_R_OFFSET) and master write error (XPCI_LMA_W_OFFSET) registers.

```
#define XPCI_LMADDR_MASK
#define XPCI_LMADDR_SERR_R
#define XPCI_LMADDR_PERR_R
```

```
#define XPCI_LMADDR_TA_R
#define XPCI_LMADDR_SERR_W
#define XPCI_LMADDR_PERR_W
#define XPCI_LMADDR_TA_W
#define XPCI_LMADDR_MA_W
#define XPCI_LMADDR_BR_W
#define XPCI_LMADDR_BRD_W
#define XPCI_LMADDR_BRT_W
#define XPCI_LMADDR_BRANGE_W
```

PCI error address definition bitmaps and masks

Bit definitions for the PCI address definition register. This register defines the meaning of the address stored in the PCI read error address (XPCI_PIA_R_OFFSET) and PCI write error address (XPCI_PIA_W_OFFSET) registers.

```
#define XPCI_PIADDR_MASK
#define XPCI_PIADDR_ERRACK_R
#define XPCI_PIADDR_ERRACK_W
#define XPCI_PIADDR_RETRY_W
#define XPCI_PIADDR_TIMEOUT_W
#define XPCI_PIADDR_RANGE_W
```

PCI configuration header offsets

Defines the offsets in the standard PCI configuration header

```
#define XPCI_HDR_VENDOR
#define XPCI_HDR_DEVICE
#define XPCI_HDR_COMMAND
#define XPCI_HDR_STATUS
#define XPCI_HDR_REVID
#define XPCI_HDR_CLASSCODE
#define XPCI_HDR_CACHE_LINE_SZ
#define XPCI_HDR_LATENCY
#define XPCI_HDR_TYPE
#define XPCI_HDR_BIST
#define XPCI_HDR_BAR0
#define XPCI_HDR_BAR1
```

```
#define XPCI_HDR_BAR2
#define XPCI_HDR_BAR3
#define XPCI_HDR_BAR4
#define XPCI_HDR_BAR5
#define XPCI_HDR_CARDBUS_PTR
#define XPCI_HDR_SUB_VENDOR
#define XPCI_HDR_SUB_DEVICE
#define XPCI_HDR_ROM_BASE
#define XPCI_HDR_CAP_PTR
#define XPCI_HDR_INT_LINE
#define XPCI_HDR_INT_PIN
#define XPCI_HDR_MIN_GNT
#define XPCI_HDR_MAX_LAT
```

PCI BAR register definitions

Defines the masks and bits for the PCI XPAR_HDR_BARn registers. The bridge supports the first three BARs in the PCI configuration header.

```
#define XPCI_HDR_NUM_BAR
#define XPCI_HDR_BAR_ADDR_MASK
#define XPCI_HDR_BAR_PREFETCH_YES
#define XPCI_HDR_BAR_PREFETCH_NO
#define XPCI_HDR_BAR_TYPE_MASK
#define XPCI_HDR_BAR_TYPE_BELOW_4GB
#define XPCI_HDR_BAR_TYPE_BELOW_1MB
#define XPCI_HDR_BAR_TYPE_ABOVE_4GB
#define XPCI_HDR_BAR_TYPE_RESERVED
#define XPCI_HDR_BAR_SPACE_IO
#define XPCI_HDR_BAR_SPACE_MEMORY
```

DMA type constants

Defines the types of DMA engines.

```
#define XPCI_DMA_TYPE_NONE
#define XPCI_DMA_TYPE_SIMPLE
#define XPCI_DMA_TYPE_SG
```

Defines

```
#define XPci_mReset(BaseAddress)
#define XPci_mIntrGlobalEnable(BaseAddress)
#define XPci_mIntrGlobalDisable(BaseAddress)
#define XPci_mIntrEnable(BaseAddress, Mask)
#define XPci_mIntrDisable(BaseAddress, Mask)
#define XPci_mIntrClear(BaseAddress, Mask)
#define XPci_mIntrReadIER(BaseAddress)
#define XPci_mIntrReadISR(BaseAddress)
#define XPci_mIntrWriteISR(BaseAddress, Mask)
#define XPci_mIntrReadIPR(BaseAddress)
#define XPci_mIntrReadID(BaseAddress)
#define XPci_mIntrPciEnable(BaseAddress, Mask)
#define XPci_mIntrPciDisable(BaseAddress, Mask)
#define XPci_mIntrPciClear(BaseAddress, Mask)
#define XPci_mIntrPciReadIER(BaseAddress)
#define XPci_mIntrPciReadISR(BaseAddress)
#define XPci_mIntrPciWriteISR(BaseAddress, Mask)
#define XPci_mReadReg(BaseAddress, RegOffset)
#define XPci_mWriteReg(BaseAddress, RegOffset, Data)
#define XPci_mConfigIn(BaseAddress, ConfigAddress, ConfigData)
#define XPci_mConfigOut(BaseAddress, ConfigAddress, ConfigData)
#define XPci_mAckSend(BaseAddress, Vector)
#define XPci_mAckRead(BaseAddress)
#define XPci_mSpecialCycle(BaseAddress, Data)
#define XPci_mLocal2Pci(LocalAddr, TO)
#define XPci_mPci2Local(PciAddr, TO)
```

Define Documentation

```
#define  
XPCI_BUSNO_BUS_MASK
```

Mask for bus
number

#define

XPCI_BUSNO_OFFSET

bus/subordinate bus
numbers

#define

XPCI_BUSNO_SUBBUS_MASK

Mask for subordinate bus
no

#define

XPCI_CAR_OFFSET

Config addr reg
(port)

#define

XPCI_CDR_OFFSET

Config command
data

#define

XPCI_DMA_TYPE_NONE

No
DMA

#define

XPCI_DMA_TYPE_SG

Scatter-gather
DMA

#define

XPCI_DMA_TYPE_SIMPLE

Simple
DMA

#define

XPCI_HDR_BAR0

Base address

0

```
#define  
XPCI_HDR_BAR1
```

Base address

1

```
#define  
XPCI_HDR_BAR2
```

Base address

2

```
#define  
XPCI_HDR_BAR3
```

Base address

3

```
#define  
XPCI_HDR_BAR4
```

Base address

4

```
#define  
XPCI_HDR_BAR5
```

Base address

5

```
#define  
XPCI_HDR_BAR_ADDR_MASK
```

Base address

mask

```
#define  
XPCI_HDR_BAR_PREFETCH_NO
```

Range is not
prefetchable

```
#define  
XPCI_HDR_BAR_PREFETCH_YES
```

Range is
prefetchable

```
#define  
XPCI_HDR_BAR_SPACE_IO
```

IO space
indicator

```
#define  
XPCI_HDR_BAR_SPACE_MEMORY
```

Memory space
indicator

```
#define  
XPCI_HDR_BAR_TYPE_ABOVE_4GB
```

Locate anywhere above
4GB

```
#define  
XPCI_HDR_BAR_TYPE_BELOW_1MB
```

Reserved in PCI
2.2

```
#define  
XPCI_HDR_BAR_TYPE_BELOW_4GB
```

Locate anywhere below
4GB

```
#define  
XPCI_HDR_BAR_TYPE_MASK
```

Memory type
mask

```
#define  
XPCI_HDR_BAR_TYPE_RESERVED
```

Reserved

```
#define  
XPCI_HDR_BIST
```

Built in self
test

```
#define  
XPCI_HDR_CACHE_LINE_SZ
```

Cache line
size

```
#define  
XPCI_HDR_CAP_PTR
```

Capabilities
pointer

```
#define  
XPCI_HDR_CARDBUS_PTR
```

Cardbus CIS
pointer

```
#define  
XPCI_HDR_CLASSCODE
```

Class
code

```
#define  
XPCI_HDR_COMMAND
```

Command
register

```
#define  
XPCI_HDR_DEVICE
```

Device
ID

```
#define  
XPCI_HDR_INT_LINE
```

Interrupt
line

```
#define  
XPCI_HDR_INT_PIN
```

Interrupt
pin

```
#define  
XPCI_HDR_LATENCY
```

Latency
timer

```
#define  
XPCI_HDR_MAX_LAT
```

Priority level
request

```
#define  
XPCI_HDR_MIN_GNT
```

Timeslice
request

```
#define  
XPCI_HDR_NUM_BAR
```

Number of BARs in the PCI
header

```
#define  
XPCI_HDR_REVID
```

Revision
ID

```
#define  
XPCI_HDR_ROM_BASE
```

Expansion ROM base
address

```
#define  
XPCI_HDR_STATUS
```

Status
register

```
#define  
XPCI_HDR_SUB_DEVICE
```

Subsystem
ID

```
#define  
XPCI_HDR_SUB_VENDOR
```

Subsystem Vendor
ID

```
#define  
XPCI_HDR_TYPE
```

Header
type

```
#define  
XPCI_HDR_VENDOR
```

Vendor
ID

```
#define  
XPCI_IAR_OFFSET
```

PCI interrupt
ack

#define

XPCI_INHIBIT_LOCAL_BUS_R

Local bus master
reads

#define

XPCI_INHIBIT_LOCAL_BUS_W

Local bus mater
writes

#define

XPCI_INHIBIT_MASK

Mask for all bits defined
below

#define

XPCI_INHIBIT_OFFSET

Inhibit transfers on
errors

#define

XPCI_INHIBIT_PCI_R

PCI initiator
reads

#define

XPCI_INHIBIT_PCI_W

PCI initiator
writes

#define

XPCI_IR_LM_BR_W

Local bus master burst write
retry

#define

XPCI_IR_LM_BRANGE_W

Local bus master burst write
range

```
#define  
XPCI_IR_LM_BRD_W
```

Local bus master burst write retry
disconnect

```
#define  
XPCI_IR_LM_BRT_W
```

Local bus master burst write retry
timeout

```
#define  
XPCI_IR_LM_MA_W
```

Local bus master abort
write

```
#define  
XPCI_IR_LM_PERR_R
```

Local bus master read
PERR

```
#define  
XPCI_IR_LM_PERR_W
```

Local bus master write
PERR

```
#define  
XPCI_IR_LM_SERR_R
```

Local bus master read
SERR

```
#define  
XPCI_IR_LM_SERR_W
```

Local bus master write
SERR

```
#define  
XPCI_IR_LM_TA_R
```

Local bus master read target
abort

```
#define  
XPCI_IR_LM_TA_W
```

Local bus master write target
abort

```
#define  
XPCI_IR_MASK
```

Mask of all
bits

```
#define  
XPCI_IR_PI_SERR_R
```

PCI initiator read
SERR

```
#define  
XPCI_IR_PI_SERR_W
```

PCI initiator write
SERR

```
#define  
XPCI_LMA_R_OFFSET
```

Local bus master read error
address

```
#define  
XPCI_LMA_W_OFFSET
```

Local bus master write error
address

#define

XPCI_LMADDR_BR_W

Master burst write
retry

#define

XPCI_LMADDR_BRANGE_W

Master burst write
range

#define

XPCI_LMADDR_BRD_W

Master burst write retry
disconnect

#define

XPCI_LMADDR_BRT_W

Master burst write retry
timeout

#define

XPCI_LMADDR_MA_W

Master abort
write

#define

XPCI_LMADDR_MASK

Mask of all
bits

#define

XPCI_LMADDR_OFFSET

Local bus master address
definition

#define

XPCI_LMADDR_PERR_R

Master read
PERR

```
#define  
XPCI_LMADDR_PERR_W
```

Master write
PERR

```
#define  
XPCI_LMADDR_SERR_R
```

Master read
SERR

```
#define  
XPCI_LMADDR_SERR_W
```

Master write
SERR

```
#define  
XPCI_LMADDR_TA_R
```

Master read target
abort

```
#define  
XPCI_LMADDR_TA_W
```

Master write target
abort

```
#define XPCI_mAckRead( BaseAddress )
```

Read the contents of the PCI interrupt acknowledge vector register.

C-style signature: Xuint32 **XPci_mAckRead**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

System dependent interrupt vector.

Note:

None

```
#define XPci_mAckSend( BaseAddress,  
                    Vector      )
```

Generate a PCI interrupt acknowledge bus cycle with the given vector.

C-style signature: void **XPci_mAckSend**(Xuint32 BaseAddress, Xuint32 Vector)

Parameters:

BaseAddress is the base address of the PCI component.

Vector is the interrupt vector to place on the PCI bus.

Returns:

None

Note:

None

```
#define XPci_mConfigIn( BaseAddress,  
                      ConfigAddress,  
                      ConfigData  )
```

Low level PCI configuration read

C-style signature: void XPCI_mConfigIn(Xuint32 BaseAddress, Xuint32 ConfigAddress, Xuint32 ConfigData)

Parameters:

BaseAddress is the base address of the PCI component.

ConfigAddress is the PCI configuration space address in a packed format.

ConfigData is the data read from the ConfigAddress

Returns:

Data from configuration address

Note:

None

```
#define XPCI_mConfigOut( BaseAddress,  
                        ConfigAddress,  
                        ConfigData )
```

Low level PCI configuration write

C-style signature: void XPCI_mConfigOut(Xuint32 BaseAddress, Xuint32 ConfigAddress, Xuint32 ConfigData)

Parameters:

BaseAddress is the base address of the PCI component.

ConfigAddress is the PCI configuration space address in a packed format.

ConfigData is the data to write at the ConfigAddress

Returns:

None

Note:

None

```
#define XPCI_mIntrClear( BaseAddress,  
                        Mask          )
```

Clear pending interrupts in the device interrupt status register (DISR) This is a toggle on write register.

C-style signature: void **XPCI_mIntrClear**(Xuint32 BaseAddress, Xuint32 Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the group of interrupts to clear. Use a logical OR of constants in XPCI_IPIF_INT_MASK. Bits set to 1 are cleared, bits set to 0 are not affected.

Returns:

None

Note:

None

```
#define XPCI_mIntrDisable( BaseAddress,  
                          Mask          )
```

Disable interrupts in the device interrupt enable register (DIER)

C-style signature: void **XPCI_mIntrDisable**(Xuint32 BaseAddress, Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the group of interrupts to disable. Use a logical OR of constants in XPCI_IPIF_INT_MASK. Bits set to 1 are disabled, bits set to 0 are not affected.

Returns:

None

Note:

None

```
#define XPCI_mIntrEnable( BaseAddress,  
                        Mask      )
```

Enable interrupts in the device interrupt enable register (DIER)

C-style signature: void **XPCI_mIntrEnable**(Xuint32 BaseAddress, Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the group of interrupts to enable. Use a logical OR of constants in XPCI_IPIF_INT_MASK. Bits set to 1 are enabled, bits set to 0 are not affected.

Returns:

None

Note:

None

```
#define XPCI_mIntrGlobalDisable( BaseAddress )
```

Global interrupt disable. Disable all interrupts from this core. Any interrupts enabled by **XPCI_mIntrEnable()** or **XPCI_mIntrPciEnable()** are disabled, however their settings remain unchanged.

C-style signature: void **XPCI_mIntrGlobalDisable**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

None

Note:

None

```
#define XPCI_mIntrGlobalEnable( BaseAddress )
```

Global interrupt enable. Must be called before any interrupts enabled by **XPCI_mIntrEnable()** or **XPCI_mIntrPciEnable()** have any effect.

C-style signature: void **XPCI_mIntrGlobalEnable**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

None

Note:

None

```
#define XPCI_mIntrPciClear( BaseAddress,  
                          Mask          )
```

Clear PCI specific interrupts in the interrupt status register (IISR). This is a toggle on write register.

C-style signature: void **XPCI_mIntrPciClear**(Xuint32 BaseAddress, Xuint32 Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the group of interrupts to clear. Use a logical OR of constants in XPCI_IR_MASK. Bits set to 1 are cleared, bits set to 0 are not affected.

Returns:

None

Note:

None

```
#define XPCI_mIntrPciDisable( BaseAddress,  
                             Mask          )
```

Disable PCI specific interrupt sources in the PCI interrupt enable register (IIER)

C-style signature: void **XPCI_mIntrPciDisable**(Xuint32 BaseAddress, Xuint32 Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the group of interrupts to disable. Bits set to 1 are disabled, bits set to 0 are not affected. The mask is made up by OR'ing bits from XPCI_IR_MASK.

Returns:

None

Note:

None

```
#define XPCI_mIntrPciEnable( BaseAddress,  
                           Mask          )
```

Enable PCI specific interrupt sources in the PCI interrupt enable register (IIER)

C-style signature: void **XPCI_mIntrPciEnable**(Xuint32 BaseAddress, Xuint32 Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the group of interrupts to enable. Bits set to 1 are enabled, bits set to 0 are not affected. The mask is made up by OR'ing bits from XPCI_IR_MASK.

Returns:

None

Note:

None

```
#define XPCI_mIntrPciReadIER( BaseAddress )
```

Read the contents of the PCI specific interrupt enable register (IIER)

C-style signature: Xuint32 **XPci_mIntrPciReadIER**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

Contents of the pending interrupt register. The mask is made up of bits defined in XPCI_IR_MASK.

Note:

None

```
#define XPci_mIntrPciReadISR( BaseAddress )
```

Read the contents of the PCI specific interrupt status register (IISR)

C-style signature: Xuint32 **XPci_mIntrPciReadISR**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

Contents of the pending interrupt register. The mask is made up of bits defined in XPCI_IR_MASK.

Note:

None

```
#define XPci_mIntrPciWriteISR( BaseAddress,  
Mask )
```

Write to the PCI interrupt status register (IISR)

C-style signature: void **XPci_mIntrPciWriteISR**(Xuint32 BaseAddress, Xuint32 Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the value to write to the register and is assumed to be bits or'd together from the XPCI_IR_MASK.

Returns:

None

Note:

None

```
#define XPci_mIntrReadID( BaseAddress )
```

Read the contents of the device interrupt ID register (DIIR).

C-style signature: Xuint32 **XPci_mIntrReadID**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

Value of the register.

Note:

None

```
#define XPci_mIntrReadIER( BaseAddress )
```

Read the contents of the device interrupt enable register (DIER)

C-style signature: Xuint32 **XPci_mIntrReadIER**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

Value of the register.

Note:

None

```
#define XPci_mIntrReadIPR( BaseAddress )
```

Read the contents of the device interrupt pending register (DIPR).

C-style signature: Xuint32 **XPci_mIntrReadIPR**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

Value of the register.

Note:

None

```
#define XPci_mIntrReadISR( BaseAddress )
```

Read the contents of the device interrupt status register (DISR)

C-style signature: Xuint32 **XPci_mIntrReadISR**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

Value of the register.

Note:

None

```
#define XPci_mIntrWriteISR( BaseAddress,  
                           Mask          )
```

Write to the device interrupt status register (DISR)

C-style signature: void **XPci_mIntrWriteISR**(Xuint32 BaseAddress, Xuint32 Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the value to write to the register and is assumed to be bits or'd together from the XPCI_IPIF_INT_MASK.

Returns:

None

Note:

None

```
#define XPci_mLocal2Pci( LocalAddr,  
                        TO          )
```

Convert local bus address to a PCI address

C-style signature: Xuint32 **XPci_mLocal2Pci**(Xuint32 LocalAddress, Xuint32 TranslationOffset)

Parameters:

LocalAddr is the local address to find the equivalent PCI address for.

TO is the translation offset to apply

Returns:

Address in PCI space

Note:

IPIFBAR_n, IPIFHIGHADDR_n, and IPIFBAR2PCI_n, defined in [xparameters.h](#), are defined for each BAR. To make a proper conversion, LocalAddress must fall within range of a IPIFBAR_n and IPIFHIGHADDR_n pair and TO specified must be the matching IPIFBAR2PCI_n. Example: pciAddr = XPci_mLocal2Pci(XPAR_PCI_IPIFBAR_0, XPAR_PCI_IPIFBAR2PCI_0) finds the PCI equivalent address for the local address named by XPAR_PCI_IPIFBAR_0.

```
#define XPci_mPci2Local( PciAddr,  
                        TO      )
```

Convert PCI address to a local bus address

C-style signature: Xuint32 **XPci_mPci2Local**(Xuint32 PciAddress, Xuint32 TranslationOffset)

Parameters:

PciAddr is the PCI address to find the equivalent local address for.

TO is the translation offset to apply

Returns:

Address in local space

Note:

PCIBAR_n, PCIBAR_LEN_n, and PCIBAR2IPIF_n, defined in [xparameters.h](#), are defined for each BAR. To make a proper conversion, PciAddress must fall within range of a PCIBAR_n and PCIBAR_LEN_n pair and TO specified must be the matching PCIBAR2IPIF_n. Example: localAddr = XPci_mPci2Local(XPAR_PCI_PCIBAR_0,

XPAR_PCI_PCIBAR2IPIF_0) finds the local address that corresponds to XPAR_PCI_PCIBAR_0 on the PCI bus. Note that PCIBAR_LEN is expressed as a power of 2.

```
#define XPCI_mReadReg( BaseAddress,  
                    RegOffset  )
```

Low level register read function.

C-style signature: Xuint32 **XPCI_mReadReg**(Xuint32 BaseAddress, Xuint32 RegOffset)

Parameters:

BaseAddress is the base address of the PCI component.

RegOffset is the register offset.

Returns:

Value of the register.

Note:

None

```
#define XPCI_mReset( BaseAddress )
```

IPIF Low level PCI reset function. Reset the V3 core.

C-style signature: void **XPCI_mReset**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

None

Note:

The IPIF RESETR register is located at (base + 0x80) instead of (base + 0x40) where the IPIF component driver expects it. This macro adjusts for this difference.

```
#define XPCI_mSpecialCycle( BaseAddress,  
                           Data          )
```

Broadcasts a message to all listening PCI targets.

C-style signature: Xuint32 **XPCI_mSpecialCycle**(Xuint32 BaseAddress, Xuint32 Data)

Parameters:

BaseAddress is the base address of the PCI component.

Data is the data to broadcast.

Returns:

None

Note:

None

```
#define XPCI_mWriteReg( BaseAddress,  
                      RegOffset,  
                      Data          )
```

Low level register write function.

C-style signature: void **XPCI_mWriteReg**(Xuint32 BaseAddress, Xuint32 RegOffset, Xuint32 Data)

Parameters:

BaseAddress is the base address of the PCI component.

RegOffset is the register offset.

Data is the data to write.

Returns:

None

Note:

None

```
#define  
XPCI_PIA_R_OFFSET
```

PCI read error
address

```
#define  
XPCI_PIA_W_OFFSET
```

PCI write error
address

```
#define  
XPCI_PIADDR_ERRACK_R
```

PCI initiator read
ErrAck

```
#define  
XPCI_PIADDR_ERRACK_W
```

PCI initiator write
ErrAck

```
#define  
XPCI_PIADDR_MASK
```

Mask of all
bits

```
#define  
XPCI_PIADDR_OFFSET
```

PCI address
definition

```
#define  
XPCI_PIADDR_RANGE_W
```

PCI initiator write
range

```
#define  
XPCI_PIADDR_RETRY_W
```

PCI initiator write
retries

#define
XPCI_PIADDR_TIMEOUT_W

PCI initiator write
timeout

#define
XPCI_PREOVRD_OFFSET

Prefetch
override

#define
XPCI_SC_DATA_OFFSET

Special cycle
data

#define
XPCI_SERR_R_OFFSET

PCI initiator read SERR
address

#define
XPCI_SERR_W_OFFSET

PCI initiator write SERR
address

#define
XPCI_STATCMD_66MHZ_CAP

66MHz
capable

#define
XPCI_STATCMD_BACK_EN

Fast back-to-back
enable

```
#define  
XPCI_STATCMD_BUSM_EN
```

Bus master
enable

```
#define  
XPCI_STATCMD_DEVSEL_FAST
```

Device select timing
fast

```
#define  
XPCI_STATCMD_DEVSEL_MED
```

Device select timing
medium

```
#define  
XPCI_STATCMD_DEVSEL_MSK
```

Device select timing
mask

```
#define  
XPCI_STATCMD_ERR_MASK
```

Error bits or'd
together

```
#define  
XPCI_STATCMD_INT_DISABLE
```

Interrupt disable (PCI
v2.3)

```
#define  
XPCI_STATCMD_INT_STATUS
```

Interrupt status (PCI
v2.3)

```
#define  
XPCI_STATCMD_IO_EN
```

```
I/O access  
enable
```

```
#define  
XPCI_STATCMD_MEM_EN
```

```
Memory access  
enable
```

```
#define  
XPCI_STATCMD_MEMWR_INV_EN
```

```
Memory write &  
invalidate
```

```
#define  
XPCI_STATCMD_MPERR
```

```
Master data PERR  
detected
```

```
#define  
XPCI_STATCMD_MSTABRT_RCV
```

```
Received master  
abort
```

```
#define  
XPCI_STATCMD_OFFSET
```

```
PCI config status/  
command
```

```
#define  
XPCI_STATCMD_PARITY
```

```
Report parity  
errors
```

```
#define  
XPCI_STATCMD_PERR_DET
```

Detected
PERR

```
#define  
XPCI_STATCMD_SERR_EN
```

SERR report
enable

```
#define  
XPCI_STATCMD_SERR_SIG
```

Signaled
SERR

```
#define  
XPCI_STATCMD_SPECIALCYC
```

Special
cycles

```
#define  
XPCI_STATCMD_STEPPING
```

Stepping
control

```
#define  
XPCI_STATCMD_TGTABRT_RCV
```

Received target
abort

```
#define  
XPCI_STATCMD_TGTABRT_SIG
```

Signaled target
abort

```
#define  
XPCI_STATCMD_VGA_SNOOP_EN
```

VGA palette snoop
enable

#define
XPCI_STATV3_DATA_XFER

Data transfer. Read
only

#define
XPCI_STATV3_DISC_WDATA

Disconnect with data. Read
only

#define
XPCI_STATV3_DISC_WODATA

Disconnect without data. Read
only

#define
XPCI_STATV3_MASK

Mask of all
bits

#define
XPCI_STATV3_MASTER_ABRT

Master abort. Read
only

#define
XPCI_STATV3_NORM_TERM

Normal termination. Read
only

#define
XPCI_STATV3_OFFSET

V3 core transaction
status

#define

XPCI_STATV3_PCI_RETRY_R

PCI retry on read. Read/
write

#define

XPCI_STATV3_PCI_RETRY_W

PCI retry on write. Read/
write

#define

XPCI_STATV3_TGT_ABRT

Target abort. Read
only

#define

XPCI_STATV3_TGT_TERM

Target termination. Read
only

#define

XPCI_STATV3_TRANS_END

Transaction end. Read
only

#define

XPCI_STATV3_WRITE_BUSY

Write busy. Read
only

pci/v1_00_b/src/xpci.h File Reference

Detailed Description

This file contains the software API definition of the Xilinx PCI bridge (**XPci**) component. This component bridges between local bus IPIF and the Xilinx LogiCORE PCI64 Interface v3.0 core. It provides full bridge functionality between the local bus a 32 bit V2.2 compliant PCI bus.

Features

This driver allows the user to access the device's registers to perform PCI configuration read and write access, error detection and processing, and interrupt management.

The Xilinx PCI bridge controller is a soft IP core designed for Xilinx FPGAs and contains the following features:

- Supports 32 bit OPB local bus
- PCI V2.2 Compliant
- Robust error reporting and diagnostics
- DMA capable

Interrupt Management

The **XPci** component driver provides interrupt management functions. Implementation of callback handlers is left to the user. Refer to the provided PCI code fragments in the examples directory.

The PCI bridge IP core uses the IPIF to manage interrupts from devices within it. Devices in this core include the PCI bridge itself and an optional DMA engine. To manage interrupts from these devices, a three layer approach is utilized and is modeled on the IPIF.

Device specific interrupt control is at the lowest layer. This is where individual sources are managed. For example, PCI Master Abort or DMA complete interrupts are enabled/disabled/cleared here. The **XPci** function API that manages this layer is identified as `XPci_InterruptPci<operation>()`. DMA interrupts at this layer are managed by the `XDma_Channel` software component.

The middle layer is utilized to manage interrupts at a device level. For example, enabling PCI interrupts at this layer allows any PCI device specific interrupt enabled at the lowest layer to be passed up to the highest layer.

The XPCI function API that manages this layer is identified as `XPci_Interrupt<operation>()`.

The middle layer serves little purpose when there is no DMA engine and can largely be ignored. During initialization, use `XPci_InterruptEnable(..., XPCI_IPIF_INT_PCI)` to allow all PCI interrupts enabled at the lowest layer to pass through. After this operation, the middle layer can be forgotten.

The highest layer is simply a global interrupt enable/disable switch that allows all or none of the enabled interrupts to be passed on to an interrupt controller. The **XPci** function API that manages this level is identified as `XPci_InterruptGlobal<operation>()`.

DMA

The PCI bridge can include a DMA engine in HW. The **XPci** software driver can be used to query which type of DMA engine has been implemented and manage interrupts. The application is required to initialize an `XDma_Channel` component driver and provide an interrupt service routine to service DMA exceptions. Example DMA management code is provided in the examples directory.

Note:

This driver is intended to be used to bridge across multiple types of buses (PLB or OPB). While the register set will remain the same for all buses, their bit definitions may change slightly from bus to bus. The differences that arise out of this are clearly documented in this file.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	04/15/03	First release
1.00b	rpm	11/05/03	Changed to use <code>XIo_InSwap32</code> and <code>XIo_OutSwap32</code> swap routines. This corresponds to the <code>opb_pci</code> core change that preserves byte address through the bridge.
1.00b	rpm	05/11/04	Added <code>IdselAddrBit</code> to the configuration data structure. This allows <code>_Initialize()</code> to use correct PCI device ID when setting the PCI bar registers.

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xpci_1.h"
```

Data Structures

```
struct XPci  
struct XPci_Config
```

struct **XPCIError**

Functions

XStatus XPCI_Initialize (**XPCI** *InstancePtr, **Xuint16** DeviceId, int BusNo, int SubBusNo)

void **XPCI_Reset** (**XPCI** *InstancePtr)

Xuint32 XPCI_ConfigPack (unsigned Bus, unsigned Device, unsigned Function)

Xuint32 XPCI_ConfigIn (**XPCI** *InstancePtr, **Xuint32** ConfigAddress, **Xuint8** Offset)

void **XPCI_ConfigOut** (**XPCI** *InstancePtr, **Xuint32** ConfigAddress, **Xuint8** Offset, **Xuint32** ConfigData)

void **XPCI_ErrorGet** (**XPCI** *InstancePtr, **XPCIError** *ErrorDataPtr)

void **XPCI_ErrorClear** (**XPCI** *InstancePtr, **XPCIError** *ErrorDataPtr)

void **XPCI_InhibitAfterError** (**XPCI** *InstancePtr, **Xuint32** Mask)

void **XPCI_SetBusNumber** (**XPCI** *InstancePtr, int BusNo, int SubBusNo)

void **XPCI_GetBusNumber** (**XPCI** *InstancePtr, int *BusNoPtr, int *SubBusNoPtr)

void **XPCI_GetDmaImplementation** (**XPCI** *InstancePtr, **Xuint32** *BaseAddr, **Xuint8** *DmaType)

XPCI_Config * **XPCI_LookupConfig** (**Xuint16** DeviceId)

XStatus XPCI_ConfigIn8 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint8** *Data)

XStatus XPCI_ConfigIn16 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint16** *Data)

XStatus XPCI_ConfigIn32 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint32** *Data)

XStatus XPCI_ConfigOut8 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint8** Data)

XStatus XPCI_ConfigOut16 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint16** Data)

XStatus XPCI_ConfigOut32 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint32** Data)

void **XPCI_InterruptGlobalEnable** (**XPCI** *InstancePtr)

void **XPCI_InterruptGlobalDisable** (**XPCI** *InstancePtr)

void **XPCI_InterruptEnable** (**XPCI** *InstancePtr, **Xuint32** Mask)

void **XPCI_InterruptDisable** (**XPCI** *InstancePtr, **Xuint32** Mask)

void **XPCI_InterruptClear** (**XPCI** *InstancePtr, **Xuint32** Mask)

Xuint32 XPCI_InterruptGetEnabled (**XPCI** *InstancePtr)

Xuint32 XPCI_InterruptGetStatus (**XPCI** *InstancePtr)

Xuint32 XPCI_InterruptGetPending (**XPCI** *InstancePtr)

Xuint32 XPCI_InterruptGetHighestPending (**XPCI** *InstancePtr)

void **XPCI_InterruptPciEnable** (**XPCI** *InstancePtr, **Xuint32** Mask)

void **XPCI_InterruptPciDisable** (**XPCI** *InstancePtr, **Xuint32** Mask)

void **XPCI_InterruptPciClear** (**XPCI** *InstancePtr, **Xuint32** Mask)

Xuint32 XPCI_InterruptPciGetEnabled (**XPCI** *InstancePtr)

Xuint32 XpCi_InterruptPciGetStatus (XpCi *InstancePtr)
void XpCi_AckSend (XpCi *InstancePtr, Xuint32 Vector)
Xuint32 XpCi_AckRead (XpCi *InstancePtr)
void XpCi_SpecialCycle (XpCi *InstancePtr, Xuint32 Data)
Xuint32 XpCi_V3StatusCommandGet (XpCi *InstancePtr)
Xuint32 XpCi_V3TransactionStatusGet (XpCi *InstancePtr)
void XpCi_V3TransactionStatusClear (XpCi *InstancePtr, Xuint32 Data)
XStatus XpCi_SelfTest (XpCi *InstancePtr)

Function Documentation

Xuint32 (XpCi *
XpCi_AckRead *InstancePtr*)

Read the contents of the PCI interrupt acknowledge vector register.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

System dependent interrupt vector.

Note:

None

void (XpCi *
XpCi_AckSend *InstancePtr*,
Xuint32 *Vector*)

Generate a PCI interrupt acknowledge bus cycle with the given vector.

Parameters:

InstancePtr is the PCI component to operate on.

Vector is a system dependent interrupt vector to place on the bus.

Note:

None

```

Xuint32
XPCI_ConfigIn      ( XPCI *
                        InstancePtr,
                        Xuint32 ConfigAddress,
                        Xuint8  Offset
                      )

```

Perform a 32 bit configuration read transaction.

Parameters:

InstancePtr is the PCI component to operate on.

ConfigAddress contains the address of the PCI device to access. It should be properly formatted for writing to the PCI configuration access port. (see [XPCI_ConfigPack\(\)](#))

Offset is the register offset within the PCI device being accessed.

Returns:

32 bit data word from addressed device

Note:

This function performs the same type of operation that `XPCI_ConfigIn32`, does except the user must format the `ConfigAddress`

```

XStatus
XPCI_ConfigIn16   ( XPCI *
                        InstancePtr,
                        unsigned Bus,
                        unsigned Device,
                        unsigned Func,
                        unsigned Offset,
                        Xuint16 * Data
                      )

```

Perform a 16 bit read transaction in PCI configuration space. Together, the `Bus`, `Device`, `Func`, & `Offset` form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.

Bus is the target PCI Bus #.

Device is the target device number.

Func is the target device's function number.

Offset is the target device's configuration space I/O offset to address.

Data is the data read from the target.

Returns:

- o `XST_SUCCESS` Operation was successful.

- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```

XStatus          ( XPci *
XPci_ConfigIn32      InstancePtr,
                        unsigned Bus,
                        unsigned Device,
                        unsigned Func,
                        unsigned Offset,
                        Xuint32 * Data
                        )

```

Perform a 32 bit read transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

- InstancePtr* is the PCI component to operate on.
- Bus* is the target PCI Bus #.
- Device* is the target device number.
- Func* is the target device's function number.
- Offset* is the target device's configuration space I/O offset to address.
- Data* is the data read from the target.

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```

XStatus          ( XPci *
XPci_ConfigIn8      InstancePtr,
                        unsigned Bus,
                        unsigned Device,
                        unsigned Func,
                        unsigned Offset,
                        Xuint8 * Data
                        )

```

Perform a 8 bit read transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

- InstancePtr* is the PCI component to operate on.
- Bus* is the target PCI Bus #.
- Device* is the target device number.
- Func* is the target device's function number.
- Offset* is the target device's configuration space I/O offset to address.
- Data* is the data read from the target.

Returns:

- o XST_SUCCESS Operation was successfull.
- o XST_PCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
void XPci_ConfigOut ( XPCI * InstancePtr,
                    XUint32 ConfigAddress,
                    XUint8  Offset,
                    XUint32 ConfigData
                    )
```

Perform a 32 bit configuration write transaction.

Parameters:

- InstancePtr* is the PCI component to operate on.
- ConfigAddress* contains the address of the PCI device to access. It should be properly formatted for writing to the PCI configuration access port. (see [XPci_ConfigPack\(\)](#))
- Offset* is the register offset within the PCI device being accessed.
- ConfigData* is the data to write to the addressed device.

Note:

This function performs the same type of operation that XPCI_ConfigOutWord, does except the user must format the Car.

```
XStatus ( XPci *
XPci_ConfigOut16 InstancePtr,
    unsigned Bus,
    unsigned Device,
    unsigned Func,
    unsigned Offset,
    Xuint16 Data
)
```

Perform a 16 bit write transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.
Bus is the target PCI Bus #.
Device is the target device number.
Func is the target device's function number.
Offset is the target device's configuration space I/O offset to address.
Data is the data to write

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus ( XPci *
XPci_ConfigOut32 InstancePtr,
    unsigned Bus,
    unsigned Device,
    unsigned Func,
    unsigned Offset,
    Xuint32 Data
)
```

Perform a 32 bit write transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

- InstancePtr* is the PCI component to operate on.
- Bus* is the target PCI Bus #.
- Device* is the target device number.
- Func* is the target device's function number.
- Offset* is the target device's configuration space I/O offset to address.
- Data* is the data to write

Returns:

- o XST_SUCCESS Operation was successful.
- o XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus ( XPci *  
XPci_ConfigOut8 InstancePtr,  
    unsigned Bus,  
    unsigned Device,  
    unsigned Func,  
    unsigned Offset,  
    Xuint8 Data  
    )
```

Perform a 8 bit write transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

- InstancePtr* is the PCI component to operate on.
- Bus* is the target PCI Bus #.
- Device* is the target device number.
- Func* is the target device's function number.
- Offset* is the target device's configuration space I/O offset to address.
- Data* is the data to write

Returns:

- o XST_SUCCESS Operation was successful.
- o XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```

Xuint32
XPci_ConfigPack      ( unsigned   Bus,
                          unsigned   Device,
                          unsigned   Function
                          )

```

Pack configuration address data.

Parameters:

Bus is the PCI bus number. Valid range 0..255.

Device is the PCI device number. Valid range 0..31.

Function is the PCI function number. Valid range 0..7.

Returns:

Encoded Bus, Device & Function formatted to be written to PCI configuration address register.

Note:

None

```

void
XPci_ErrorClear      ( XPci *      InstancePtr,
                          XPciError * ErrorDataPtr
                          )

```

Clear errors associated with the PCI bridge. Which errors are cleared depend on the Reason attributes of the ErrorData parameter. For every bit set, that corresponding error is cleared.

XPci_ErrorGet() and **XPci_ErrorClear()** are designed to be used in tandem. Use ErrorGet to retrieve the errors, then ErrorClear to clear the error state.

```

XPci_ErrorGet(ThisInstance, &Errors) if (Errors->IsError) { // Handle error XPci_ErrorClear(ThisInstance,
&Errors); }

```

If it is desired to clear some but not all errors, or a specific set of errors, then prepare ErrorData Bitmap attributes appropriately. If it is desired to clear all errors indiscriminately, then use XPCI_CLEAR_ALL_ERRORS. This has the advantage of not requiring the caller to explicitly setup an **XPciError** structure.

Parameters:

InstancePtr is the PCI component to operate on.

ErrorDataPtr is used to determine which error conditions to clear. Only the Bitmap attributes are used. Addr attributes of this structure are ignored. If this parameter is set to XPCI_CLEAR_ALL_ERRORS then all errors are cleared.

Note:

If PciSerrReason attribute is set or XPCI_CLEAR_ALL_ERRORS is passed, then the IPIF interrupt status register bits associated with SERR are cleared. This has the same effect as acknowledging an interrupt. If you don't intend on doing this, then clear PciSerrReason before calling XPci_ErrorClear.

```
void XPci_ErrorGet ( XPCI * InstancePtr, XPCIError * ErrorDataPtr )
```

Get a snapshot of the PCI bridge's error state, summarize and place results in an **XPCIError** structure. Several bridge registers are read and their contents placed into the structure as follows. Register definitions and their bitmaps are located in [xpci_1.h](#):

Attribute	Source Register
LocalBusReason	XPCI_LMADDR_OFFSET
PciReason	XPCI_PIADDR_OFFSET
PciSerrReason	IPIF IISR
LocalBusReadAddr	XPCI_LMA_R_OFFSET
LocalBusWriteAddr	XPCI_LMA_W_OFFSET
PciSerrReadAddr	XPCI_SERR_R_OFFSET
PciSerrWriteAddr	XPCI_SERR_W_OFFSET
PciReadAddr	XPCI_PIA_R_OFFSET
PciWriteAddr	XPCI_PIA_W_OFFSET

LocalBusReadAddr, LocalBusWriteAddr, PciSerrReadAddr, PciSerrWriteAddr, PCIReadAddr, and PciWriteAddr are all error addresses whose contents are latched at the time of the error.

LocalBusReason and PciReason are present to allow the caller to precisely determine the source of the error. The summary below indicates which bits cause the associated error address to become valid and which interrupt bits from interrupt status register are the cause if the error was reported via an interrupt.

LocalBusReason:

Bit	Error addr is valid	Associated Interrupt bit
XPCI_LMADDR_SERR_R	LocalBusReadAddr	XPCI_IR_LM_SERR_R
XPCI_LMADDR_PERR_R	LocalBusReadAddr	XPCI_IR_LM_PERR_R
XPCI_LMADDR_TA_R	LocalBusReadAddr	XPCI_IR_LM_TA_R

XPCI_LMADDR_SERR_W	LocalBusWriteAddr	XPCI_IR_LM_SERR_W
XPCI_LMADDR_PERR_W	LocalBusWriteAddr	XPCI_IR_LM_PERR_W
XPCI_LMADDR_TA_W	LocalBusWriteAddr	XPCI_IR_LM_TA_W
XPCI_LMADDR_MA_W	LocalBusWriteAddr	XPCI_IR_LM_MA_W
XPCI_LMADDR_BR_W	LocalBusWriteAddr	XPCI_IR_LM_BR_W
XPCI_LMADDR_BRD_W	LocalBusWriteAddr	XPCI_IR_LM_BRD_W
XPCI_LMADDR_BRT_W	LocalBusWriteAddr	XPCI_IR_LM_BRT_W
XPCI_LMADDR_BRANGE_W	LocalBusWriteAddr	XPCI_IR_LM_BRANGE_W

PciReason:

Bit	Error addr is valid	Associated Interrupt bit
-----	-----	-----
XPCI_PiADDR_ERRACK_R	PciReadAddr	N/A
XPCI_PiADDR_ERRACK_W	PciWriteAddr	N/A
XPCI_PiADDR_RETRY_W	PciWriteAddr	N/A
XPCI_PiADDR_TIMEOUT_W	PciWriteAddr	N/A
XPCI_PiADDR_RANGE_W	PciWriteAddr	N/A

PciReasonSerr:

Bit	Error addr is valid	Associated Interrupt bit
-----	-----	-----
XPCI_IR_Pi_SERR_R	PciSerrReadAddr	XPCI_IR_Pi_SERR_R
XPCI_IR_Pi_SERR_W	PciSerrWriteAddr	XPCI_IR_Pi_SERR_W

If any of the above mentioned error reason bits are set, then attribute IsError is set to XTRUE. If no errors are detected, then it is set to XFALSE.

Parameters:

InstancePtr is the PCI component to operate on.

ErrorDataPtr is the error snapshot data returned from the PCI bridge.

Note:

None

```
void
XPCI_GetBusNumber ( XPCI * InstancePtr,
                   int * BusNoPtr,
                   int * SubBusNoPtr
                   )
```

Get the bus number and subordinate bus number of the pci bridge.

Parameters:

InstancePtr is the PCI component to operate on

BusNoPtr is storage to place the bus number

SubBusNoPtr is storage to place the subordinate bus number

Note:

None

```
void  
XPci_GetDmaImplementation ( XPci *  
                             InstancePtr,  
                             Xuint32 * BaseAddr,  
                             Xuint8 * DmaType  
                             )
```

Get the DMA engine implementation information for this instance.

Parameters:

InstancePtr is the PCI component to operate on.

BaseAddr is a return value indicating the base address of the DMA registers.

DmaType is a return value indicating the type of DMA implemented. The possible types are XPCI_DMA_TYPE_NONE for no DMA, XPCI_DMA_TYPE_SIMPLE for simple DMA, and XPCI_DMA_TYPE_SG for scatter-gather DMA.

Note:

None

```
void  
XPci_InhibitAfterError ( XPci *  
                          InstancePtr,  
                          Xuint32 Mask  
                          )
```

Change how the bridge handles subsequent PCI transactions after errors occur. Transactions can be prohibited once an error occurs then allowed again once the error is cleared. Or transactions are be allowed to continue despite an error condition.

Parameters:

InstancePtr is the PCI component to operate on.

Mask defines the type of transactions affected. OR together bits from XPCI_INHIBIT_* to form the mask. Bits set to 1 will cause transactions to be inhibited when an error exists. Bits set to 0 will allow transactions to proceed.

Note:

None

```
XStatus      ( XPCI *
XPCI_Initialize      InstancePtr,
                    Xuint16 DeviceId,
                    int     BusNo,
                    int     SubBusNo
                    )
```

Initialize the **XPCI** instance provided by the caller based on the given DeviceID.

Initialization includes setting up the bar registers in the bridge's PCI header to match the IPIF settings. Not performing this step will cause the the IPIF to not respond to PCI bus hits.

Parameters:

InstancePtr is a pointer to an **XPCI** instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the **XPCI** API must be made with this pointer.

DeviceId is the unique id of the device controlled by this **XPCI** component. Passing in a device id associates the generic **XPCI** instance to a specific device, as chosen by the caller or application developer.

BusNo is the initial PCI bus number to assign to the host bridge. This value can be changed later with a call to **XPCI_SetBusNumber()**

SubBusNo is the initial PCI sub-bus number to assign to the host bridge This value can be changed later with a call to **XPCI_SetBusNumber()**

Returns:

- o XST_SUCCESS Initialization was successful.
- o XST_DEVICE_NOT_FOUND Device configuration data was not found for a device with the supplied device ID.

Note:

None

```
void  
XPci_InterruptClear ( XPci *  
                    InstancePtr,  
                    Xuint32 Mask  
                    )
```

Clear device level pending interrupts with the provided mask.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XPCI_IPIF_INT_MASK

Note:

None

```
void  
XPci_InterruptDisable ( XPci *  
                       InstancePtr,  
                       Xuint32 Mask  
                       )
```

Disable device interrupts. Any component interrupts enabled through [XPci_InterruptPciEnable\(\)](#) and/or the DMA driver will no longer have any effect. The component interrupt settings will be retained however.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to disable. Bits set to 1 are disabled. The mask is formed by OR'ing bits from XPCI_IPIF_INT_MASK

Note:

None

```
void  
XPci_InterruptEnable ( XPci *  
                      InstancePtr,  
                      Xuint32 Mask  
                      )
```

Enable device interrupts. Device interrupts must be enabled by this function before component interrupts enabled by [XPci_InterruptPciEnable\(\)](#) and/or the DMA driver have any effect.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to enable. Bit positions of 1 are enabled. The mask is formed by OR'ing bits from XPCI_IPIF_INT_MASK.

Note:

None

Xuint32 (**XPci ***)
XPci_InterruptGetEnabled *InstancePtr*

Returns the device level interrupt enable mask as set by [XPci_InterruptEnable\(\)](#).

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of bits made from XPCI_IPIF_INT_MASK.

Note:

None

Xuint32 (**XPci ***)
XPci_InterruptGetHighestPending *InstancePtr*

Returns the highest priority pending device interrupt that has been enabled by [XPci_InterruptEnable\(\)](#).

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask is one set bit made from XPCI_IPIF_INT_MASK or zero if no interrupts are pending.

Note:

None

Xuint32 (**XPci ***)
XPci_InterruptGetPending *InstancePtr*

Returns the pending status of device level interrupt signals that have been enabled by [XPci_InterruptEnable\(\)](#). Any bit in the mask set to 1 indicates that an interrupt is pending from the given component

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of bits made from XPCI_IPIF_INT_MASK or zero if no interrupts are pending.

Note:

None

```
Xuint32 ( XPci * )  
XPci_InterruptGetStatus InstancePtr
```

Returns the status of device level interrupt signals. Any bit in the mask set to 1 indicates that the given component has asserted an interrupt condition.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of bits made from XPCI_IPIF_INT_MASK.

Note:

The interrupt status indicates the status of the device irregardless if the interrupts from the devices have been enabled or not through [XPci_InterruptEnable\(\)](#).

```
void ( XPci * )  
XPci_InterruptGlobalDisable InstancePtr
```

Disable the core's interrupt output signal. Interrupts enabled through [XPci_InterruptEnable\(\)](#) and [XPci_InterruptPciEnable\(\)](#) will no longer be passed through until the IPIF global enable bit is set by [XPci_InterruptGlobalEnable\(\)](#).

Parameters:

InstancePtr is the PCI component to operate on.

Note:

None

```
void ( XPCI * )
XPci_InterruptGlobalEnable InstancePtr
```

Enable the core's interrupt output signal. Interrupts enabled through [XPci_InterruptEnable\(\)](#) and [XPci_InterruptPciEnable\(\)](#) will not be passed through until the IPIF global enable bit is set by this function.

Parameters:

InstancePtr is the PCI component to operate on.

Note:

None

```
void ( XPCI *
XPci_InterruptPciClear InstancePtr,
Xuint32 Mask
)
```

Clear PCI bridge specific interrupt status bits with the provided mask.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XPCI_IR_MASK

Note:

None

```
void ( XPCI *
XPci_InterruptPciDisable InstancePtr,
Xuint32 Mask
)
```

Disable PCI bridge specific interrupts.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to disable. Bits set to 1 are disabled. The mask is formed by OR'ing bits from XPCI_IR_MASK

Note:

None

```
void XPCI_InterruptPciEnable ( XPCI * InstancePtr, Xuint32 Mask )
```

Enable PCI bridge specific interrupts. Before this function has any effect in generating interrupts, the function [XPCI_InterruptEnable\(\)](#) must be invoked with the XPCI_IPIF_INT_PCI bit set.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to enable. Bit positions of 1 are enabled. The mask is formed by OR'ing bits from XPCI_IR_MASK.

Note:

None

```
Xuint32 XPCI_InterruptPciGetEnabled ( XPCI * InstancePtr )
```

Get the PCI bridge specific interrupts enabled through [XPCI_InterruptPciEnable\(\)](#). Bits set to 1 mean that interrupt source is enabled.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of enabled bits made from XPCI_IR_MASK.

Note:

None

```
Xuint32 XPCI_InterruptPciGetStatus ( XPCI * InstancePtr )
```

Get the status of PCI bridge specific interrupts that have been asserted Bits set to 1 are in an asserted state. Bits may be set to 1 irregardless of whether they have been enabled or not though [XPci_InterruptPciEnable\(\)](#). To get the pending interrupts, AND the results of this function with [XPci_InterruptPciGetEnabled\(\)](#).

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of enabled bits made from XPCI_IR_MASK.

Note:

None

XPci_Config* (**Xuint16**)
XPci_LookupConfig *DeviceId*

Lookup the device configuration based on the unique device ID. The table ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the device identifier to lookup.

Returns:

- o **XEmc** configuration structure pointer if DeviceID is found.
- o XNULL if DeviceID is not found.

void (**XPci ***)
XPci_Reset *InstancePtr*

Reset the PCI IP core. This is a destructive operation that could cause loss of data, local bus errors, or PCI bus errors if reset occurs while a transaction is pending.

Parameters:

InstancePtr is the PCI component to operate on.

Note:

None

XStatus (**XPci ***)
XPci_SelfTest *InstancePtr*

Broadcasts a message to all listening PCI targets.

Parameters:

InstancePtr is the PCI component to operate on.

Data is the data to broadcast.

Note:

None

```
Xuint32 ( XPCI * )  
XPCI_V3StatusCommandGet InstancePtr
```

Read the contents of the V3 bridge's status & command register. This same register can be retrieved by a PCI configuration access. The register can be written only with a PCI configuration access.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Contents of the V3 bridge's status and command register

Note:

None

```
void ( XPCI *  
XPCI_V3TransactionStatusClear InstancePtr,  
 Xuint32 Data )
```

Clear status bits in the V3 bridge's transaction status register. The contents of this register can be decoded using XPCI_STATV3_* constants defined in [xpci_1.h](#).

Parameters:

InstancePtr is the PCI component to operate on.

Data is the contents to write to the register. Or XPCI_STATV3_* constants for those bits to be cleared. Bits in the register that are read-only are not affected.

Note:

None

Xuint32 (**XPci ***)

XPci_V3TransactionStatusGet *InstancePtr*

Read the contents of the V3 bridge's transaction status register. The contents of this register can be decoded using XPCI_STATV3_* constants defined in [xpci_1.h](#).

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Contents of the V3 bridge's transaction status register.

Note:

None

XPciError Struct Reference

```
#include <xpci.h>
```

Detailed Description

XPciError is used to retrieve a snapshot of the bridge's error state. Most of the attributes of this structure are copies of various bridge registers. See [XPci_ErrorGet\(\)](#) and [XPci_ErrorClear\(\)](#).

Data Fields

Xboolean IsError

Xuint32 LocalBusReason

Xuint32 PciReason

Xuint32 PciSerrReason

Xuint32 LocalBusReadAddr

Xuint32 LocalBusWriteAddr

Xuint32 PciReadAddr

Xuint32 PciWriteAddr

Xuint32 PciSerrReadAddr

Xuint32 PciSerrWriteAddr

Field Documentation

Xboolean XPciError::

IsError

Global error
indicator

Xuint32 XPciError::

LocalBusReadAddr

Local bus master read error
address

Xuint32 XPciError::

LocalBusReason

Local bus master address
definition

Xuint32 XPciError::

LocalBusWriteAddr

Local bus master write error
address

Xuint32 XPciError::

PciReadAddr

PCI read error
address

Xuint32 XPciError::

PciReason

PCI address
definition

Xuint32 XPciError::

PciSerrReadAddr

PCI initiator read SERR
address

Xuint32 XPciError::

PciSerrReason

PCI System error
definiton

Xuint32 XPciError::

PciSerrWriteAddr

PCI initiator write SERR
address

Xuint32 XPciError::

PciWriteAddr

PCI write error
address

The documentation for this struct was generated from the following file:

- [pci/v1_00_b/src/xpci.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

pci/v1_00_b/src/xpci_config.c File Reference

Detailed Description

Implements advanced PCI configuration functions for the **XPCI** component. See [xpci.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	03/25/02	Original code

```
#include "xpci.h"
```

Functions

XStatus XPCI_ConfigIn8 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint8** *Data)

XStatus XPCI_ConfigIn16 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint16** *Data)

XStatus XPCI_ConfigIn32 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint32** *Data)

XStatus XPCI_ConfigOut8 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint8** Data)

XStatus XPCI_ConfigOut16 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint16** Data)

XStatus XPCI_ConfigOut32 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint32** Data)

Function Documentation

```
XStatus ( XPCI *  
XPCI_ConfigIn16 InstancePtr,  
    unsigned Bus,  
    unsigned Device,  
    unsigned Func,  
    unsigned Offset,  
    Xuint16 * Data  
)
```

Perform a 16 bit read transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.

Bus is the target PCI Bus #.

Device is the target device number.

Func is the target device's function number.

Offset is the target device's configuration space I/O offset to address.

Data is the data read from the target.

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus ( XPCI *  
XPCI_ConfigIn32 InstancePtr,  
    unsigned Bus,  
    unsigned Device,  
    unsigned Func,  
    unsigned Offset,  
    Xuint32 * Data  
)
```

Perform a 32 bit read transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

- InstancePtr* is the PCI component to operate on.
- Bus* is the target PCI Bus #.
- Device* is the target device number.
- Func* is the target device's function number.
- Offset* is the target device's configuration space I/O offset to address.
- Data* is the data read from the target.

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus ( XPci *  
XPci_ConfigIn8 InstancePtr,  
                unsigned Bus,  
                unsigned Device,  
                unsigned Func,  
                unsigned Offset,  
                Xuint8 * Data  
                )
```

Perform a 8 bit read transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

- InstancePtr* is the PCI component to operate on.
- Bus* is the target PCI Bus #.
- Device* is the target device number.
- Func* is the target device's function number.
- Offset* is the target device's configuration space I/O offset to address.
- Data* is the data read from the target.

Returns:

- XST_SUCCESS Operation was successful.
- XST_PCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus ( XPci *  
XPci_ConfigOut16 InstancePtr,  
                unsigned Bus,  
                unsigned Device,  
                unsigned Func,  
                unsigned Offset,  
                Xuint16 Data  
                )
```

Perform a 16 bit write transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.

Bus is the target PCI Bus #.

Device is the target device number.

Func is the target device's function number.

Offset is the target device's configuration space I/O offset to address.

Data is the data to write

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```

XStatus ( XPci *
XPci_ConfigOut32 InstancePtr,
    unsigned Bus,
    unsigned Device,
    unsigned Func,
    unsigned Offset,
    Xuint32 Data
)

```

Perform a 32 bit write transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.

Bus is the target PCI Bus #.

Device is the target device number.

Func is the target device's function number.

Offset is the target device's configuration space I/O offset to address.

Data is the data to write

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```

XStatus ( XPci *
XPci_ConfigOut8 InstancePtr,
    unsigned Bus,
    unsigned Device,
    unsigned Func,
    unsigned Offset,
    Xuint8 Data
)

```

Perform a 8 bit write transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.

Bus is the target PCI Bus #.

Device is the target device number.

Func is the target device's function number.

Offset is the target device's configuration space I/O offset to address.

Data is the data to write

Returns:

- XST_SUCCESS Operation was successfull.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

pci/v1_00_b/src/xpci_selftest.c File Reference

Detailed Description

Implements self test for the [XPci](#) component. See [xpci.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	03/25/02	Original code

```
#include "xpci.h"
```

Functions

[XStatus XPci_SelfTest](#) ([XPci](#) *InstancePtr)

Function Documentation

XStatus	(XPci *)
XPci_SelfTest	<i>InstancePtr</i>

Run a self-test on the driver/device. This includes the following tests:

- Configuration read of the bridge device and vendor ID.

Parameters:

InstancePtr is a pointer to the **XPci** instance to be worked on. This parameter must have been previously initialized with **XPci_Initialize()**.

Returns:

- XST_SUCCESS If test passed
- XST_FAILURE If test failed

Note:

None

Generated on 24 Jun 2004 for Xilinx Device Drivers

pci/v1_00_b/src/xpci_v3.c File Reference

Detailed Description

Implements V3 core processing functions for the **XPCI** component. See [xpci.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	03/25/02	Original code

```
#include "xpci.h"
```

Functions

Xuint32 [XPCI_V3StatusCommandGet](#) (**XPCI** *InstancePtr)

Xuint32 [XPCI_V3TransactionStatusGet](#) (**XPCI** *InstancePtr)

void [XPCI_V3TransactionStatusClear](#) (**XPCI** *InstancePtr, **Xuint32** Data)

Function Documentation

Xuint32 [XPCI_V3StatusCommandGet](#) (**XPCI** **InstancePtr*)

Read the contents of the V3 bridge's status & command register. This same register can be retrieved by a PCI configuration access. The register can be written only with a PCI configuration access.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Contents of the V3 bridge's status and command register

Note:

None

```
void  
Xpci_V3TransactionStatusClear ( Xpci *  
                                InstancePtr,  
                                Xuint32 Data  
                                )
```

Clear status bits in the V3 bridge's transaction status register. The contents of this register can be decoded using XPCI_STATV3_* constants defined in [xpci_l.h](#).

Parameters:

InstancePtr is the PCI component to operate on.

Data is the contents to write to the register. Or XPCI_STATV3_* constants for those bits to be cleared. Bits in the register that are read-only are not affected.

Note:

None

```
Xuint32  
Xpci_V3TransactionStatusGet ( Xpci *  
                               )  
                               InstancePtr
```

Read the contents of the V3 bridge's transaction status register. The contents of this register can be decoded using XPCI_STATV3_* constants defined in [xpci_l.h](#).

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Contents of the V3 bridge's transaction status register.

Note:

None

pciarb/v1_00_a/src/xpciarb_selftest.c File Reference

Detailed Description

Contains a diagnostic self-test function for the [XPciArb](#) driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	

1.00a	rpm	01/06/04	First release

```
#include "xstatus.h"  
#include "xpciarb.h"
```

Functions

[XStatus XPciArb_SelfTest](#) ([XPciArb](#) *InstancePtr)

Function Documentation

XStatus	(XPciArb *)
XPciArb_SelfTest	<i>InstancePtr</i>	

Run a self-test on the driver/device. The test simply writes to the control register of the device, enabling fixed priority arbitration, and verifies the value was written successfully. It then resets the device, returning the arbiter to its default state (dynamic priority arbitration).

Note that this is a destructive test in that a reset of the device is performed.

Parameters:

InstancePtr is a pointer to the **XPciArb** instance to be worked on.

Returns:

XST_SUCCESS if successful, or XST_FAILURE if the self test fails.

Note:

None.

plb2opb/v1_00_a/src/xplb2opb.c File Reference

Detailed Description

Contains required functions for the [XPlb2Opb](#) component. See [xplb2opb.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	12/7/01	First release
1.00a	rpm	05/14/02	Made configuration typedef/lookup public

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xplb2opb.h"  
#include "xplb2opb_i.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

Functions

[XStatus XPlb2Opb_Initialize](#) ([XPlb2Opb](#) *InstancePtr, [Xuint16](#) DeviceId)
[Xboolean XPlb2Opb_IsError](#) ([XPlb2Opb](#) *InstancePtr)
void [XPlb2Opb_ClearErrors](#) ([XPlb2Opb](#) *InstancePtr, [Xuint8](#) Master)
[Xuint32 XPlb2Opb_GetErrorStatus](#) ([XPlb2Opb](#) *InstancePtr, [Xuint8](#) Master)
[Xuint32 XPlb2Opb_GetErrorAddress](#) ([XPlb2Opb](#) *InstancePtr)
[Xuint32 XPlb2Opb_GetErrorByteEnables](#) ([XPlb2Opb](#) *InstancePtr)
[Xuint8 XPlb2Opb_GetMasterDrivingError](#) ([XPlb2Opb](#) *InstancePtr)

Xuint8 XPlb2Opb_GetNumMasters (**XPlb2Opb** *InstancePtr)

void **XPlb2Opb_EnableInterrupt** (**XPlb2Opb** *InstancePtr)

void **XPlb2Opb_DisableInterrupt** (**XPlb2Opb** *InstancePtr)

void **XPlb2Opb_Reset** (**XPlb2Opb** *InstancePtr)

XPlb2Opb_Config * **XPlb2Opb_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
void  
XPlb2Opb_ClearErrors      ( XPlb2Opb *  
                           InstancePtr,  
                           Xuint8   Master  
                           )
```

Clears any outstanding errors for the given master.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters on the bus

Returns:

None.

Note:

None.

```
void  
XPlb2Opb_DisableInterrupt ( XPlb2Opb *  
                           InstancePtr )
```

Disables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void ( XPlb2Opb * )  
XPlb2Opb_EnableInterrupt InstancePtr
```

Enables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
Xuint32 ( XPlb2Opb * )  
XPlb2Opb_GetErrorAddress InstancePtr
```

Returns the OPB Address where the most recent error occurred. If there isn't an outstanding error, the last address in error is returned. 0x00000000 is the initial value coming out of reset.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

Address where error causing access occurred

Note:

Calling **XPlb2Opb_IsError()** is recommended to confirm that an error has occurred prior to calling **XPlb2Opb_GetErrorAddress()** to ensure that the data in the error address register is relevant.

Xuint32 (**XPlb2Opb** * *InstancePtr*)
XPlb2Opb_GetErrorByteEnables

Returns the byte-enables asserted during the access causing the error. The enables are parameters in the hardware making the return value dynamic. An example of a 32-bit bus with all 4 byte enables available, XPlb2Opb_GetErrorByteEnables will have the value 0xF0000000 returned from a 32-bit access error.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

The byte-enables asserted during the error causing access.

Note:

None.

Xuint32 (**XPlb2Opb** * *InstancePtr*,
Xuint8 *Master*)
XPlb2Opb_GetErrorStatus

Returns the error status for the specified master.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters on the bus

Returns:

The current error status for the requested master on the PLB. The status is a bit-mask and the values are described in **xplb2opb.h**.

Note:

None.

Xuint8 (**XPlb2Opb** *)
XPlb2Opb_GetMasterDrivingError *InstancePtr*

Returns the ID of the master which is driving the error condition

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

The ID of the master that is driving the error

Note:

None.

Xuint8 (**XPlb2Opb** *)
XPlb2Opb_GetNumMasters *InstancePtr*

Returns the number of masters associated with the provided instance

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

The number of masters. This is a number from 1 to the maximum of 32.

Note:

The value returned from this call needs to be adjusted if it is to be used as the argument for other calls since the masters are numbered from 0 and this function returns values starting at 1.

```
XStatus ( XPlb2Opb *  
XPlb2Opb_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific **XPlb2Opb** instance. Looks for configuration data for the specified device, then initializes instance data.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XPlb2Opb** component. Passing in a device id associates the generic **XPlb2Opb** component to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if everything starts up as expected.
- XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

```
Xboolean ( XPlb2Opb *  
XPlb2Opb_IsError InstancePtr  
 )
```

Returns XTRUE if there is an error outstanding

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

Boolean XTRUE if there is an error, XFALSE if there is no current error.

Note:

None.

```
XPlb2Opb_Config* ( Xuint16 )  
XPlb2Opb_LookupConfig DeviceId
```

Looks up the device configuration based on the unique device ID. The table PlbOpbConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for

Returns:

A pointer to the configuration data for the given device, or XNULL if no match is found.

Note:

None.

```
void ( XPlb2Opb * )  
XPlb2Opb_Reset InstancePtr
```

Forces a software-induced reset to occur in the bridge. Disables interrupts in the process.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

None.

Note:

Disables interrupts in the process.

plb2opb/v1_00_a/src/xplb2opb_i.h File Reference

Detailed Description

This file contains data which is shared between files and internal to the **XPlb2Opb** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	02/28/02	First release
1.00a	rpm	05/14/02	Moved identifiers to xplb2opb_1.h

```
#include "xplb2opb\_1.h"
```

Variables

[XPlb2Opb_Config](#) [XPlb2Opb_ConfigTable](#) []

Variable Documentation

[XPlb2Opb_Config](#) [XPlb2Opb_ConfigTable](#) ()

[]

The PLB-to-OPB bridge configuration table, sized by the number of instances defined in [xparameters.h](#).

plb2opb/v1_00_a/src/xplb2opb_selftest.c File Reference

Detailed Description

Contains diagnostic self-test functions for the **XPlb2Opb** component. See [xplb2opb.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	12/7/01	First release

```
#include "xstatus.h"  
#include "xplb2opb.h"  
#include "xplb2opb_i.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

Functions

XStatus **XPlb2Opb_SelfTest** (**XPlb2Opb** *InstancePtr, **Xuint32** TestAddress)

Function Documentation

```
XStatus          ( XPlb2Opb *  
XPlb2Opb_SelfTest      InstancePtr,  
                        Xuint32      TestAddress  
                        )
```

Runs a self-test on the driver/device.

This tests reads the BCR to verify that the proper value is there.

XST_SUCCESS is returned if expected value is there, XST_PLB2OPB_FAIL_SELFTEST is returned otherwise.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

TestAddress is a location that could cause an error on read, not used - user definable for hw specific implementations.

Returns:

XST_SUCCESS if successful, or XST_PLB2OPB_FAIL_SELFTEST if the driver fails self-test.

Note:

This test assumes that the bus error interrupts are not enabled.

plbarb/v1_01_a/src/xplbarb.h File Reference

Detailed Description

This component contains the implementation of the **XPlbArb** component. It is the driver for the PLB (Processor Local Bus) Arbiter. The arbiter performs bus arbitration on the PLB transactions.

This driver allows the user to access the PLB Arbiter registers to support the handling of bus errors and other access errors and determine an appropriate solution if possible.

The Arbiter Hardware generates an interrupt in error conditions which is not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

Hardware Features

The Xilinx PLB Arbiter is a soft IP core designed for Xilinx FPGAs and contains the following features:

- PLB address and data steering support for up to eight masters. Number of PLB masters is configurable via a design parameter
- 64-bit and/or 32-bit support for masters and slaves
- PLB address pipelining
- PLB arbitration support for up to eight masters. Number of PLB masters is configurable via a design parameter
- Three cycle arbitration
- Four levels of dynamic master request priority
- PLB watchdog timer
- PLB architecture compliant

Device Configuration

The device can be configured in various ways during the FPGA implementation process. The configuration data for each device is contained in [xplbarb_g.c](#). A table is defined where each entry contains configuration information for a device. This information includes such things as the base address of the DCR mapped device, and the number of masters on the bus.

Note:

This driver is not thread-safe. Thread safety must be guaranteed by the layer above this driver if there is a need to access the device from multiple threads.

The Arbiter registers reside on the DCR address bus.

Any and all outstanding errors are cleared in the initialization function.

Clears the Errors for the specified master

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters - 1 on the bus

Returns:

None.

Note:

None.

```
void XPlbArb_DisableInterrupt ( XPlbArb * InstancePtr )
```

Disables the interrupt output from the arbiter

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

None

Note:

The Arbiter hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt source with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void XPlbArb_EnableInterrupt ( XPlbArb * InstancePtr )
```

Enables the interrupt output from the arbiter

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

None.

Note:

The Arbiter hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt source with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

Xuint32 (**XPlbArb ***)
XPlbArb_GetErrorAddress *InstancePtr*

Returns the PLB Address where the most recent error occurred. If there isn't an outstanding error, the last address in error is returned. 0x00000000 is the initial value coming out of reset.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

Address where error causing access occurred

Note:

Calling **XPlbArb_IsError()** is recommended to confirm that an error has occurred prior to calling this function to ensure that the data in the error address register is relevant.

Xuint32 (**XPlbArb ***
XPlbArb_GetErrorStatus *InstancePtr,*
Xuint8 *Master*
)

Returns the Error status for the specified master. These are bit masks.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters on the bus

Returns:

The current error status for the requested master on the PLB. The status is a bit-mask that can contain the following bit values:

XPA_DRIVING_BEAR_MASK	Indicates that an error has occurred and this master is driving the error address
XPA_ERROR_READ_MASK	Indicates the error was a read error (it is a write error otherwise).
XPA_ERROR_STATUS_LOCK_MASK	Indicates the error status and address are locked and cannot be overwritten.
XPA_PEAR_SIZE_MASK	Size of access that caused error
XPA_PEAR_TYPE_MASK	Type of access that caused error

Note:

None.

Xuint8 (**XPlbArb ***)
XPlbArb_GetNumMasters *InstancePtr*

Returns the number of masters associated with the arbiter.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

The number of masters. This is a number from 1 to the maximum of 32.

Note:

The value returned from this call needs to be adjusted if it is to be used as the argument for other calls since the masters are numbered from 0 and this function returns values starting at 1.

```
XStatus ( XPlbArb *  
XPlbArb_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific **XPlbArb** instance. Looks up the configuration for the given device instance and initialize the instance structure.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XPlbArb** component.

Returns:

- XST_SUCCESS if everything starts up as expected.
- XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

```
Xboolean ( XPlbArb *  
XPlbArb_IsError InstancePtr  
 )
```

Returns XTRUE if there is an error outstanding

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

Boolean XTRUE if there is an error, XFALSE if there is no current error.

Note:

None.

XPlbArb_Config* (**Xuint16**)
XPlbArb_LookupConfig *DeviceId*

Looks up the device configuration based on the unique device ID. The table XPlbArb_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for.

Returns:

A pointer to the configuration data for the device, or XNULL if no match is found.

Note:

None.

void (**XPlbArb ***)
XPlbArb_Reset *InstancePtr*

Forces a software reset to occur in the arbiter. Disables interrupts in the process.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

None.

Note:

Disables interrupts in the process.

XStatus (**XPlbArb ***
XPlbArb_SelfTest *InstancePtr*,
Xuint32 *TestAddress*
)

Runs a self-test on the driver/device.

This tests reads the PACR to verify that the proper value is there.

XST_SUCCESS is returned if expected value is there, XST_PLBARB_FAIL_SELFTEST is returned otherwise.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

TestAddress is a location that could cause an error on read, not used - user definable for hw specific implementations.

Returns:

XST_SUCCESS if successful, or XST_PLBARB_FAIL_SELFTEST if the driver fails the self test.

Note:

None.

Generated on 24 Jun 2004 for Xilinx Device Drivers

XPlbArb Struct Reference

```
#include <xplbarb.h>
```

Detailed Description

The XPlbArb driver instance data. The user is required to allocate a variable of this type for every PLB arbiter device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `plbarb/v1_01_a/src/xplbarb.h`
-

plbarb/v1_01_a/src/xplbarb_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of PLB Arbiter devices in the system. Each arbiter device should have an entry in this table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	11/16/01	First release
1.01a	rpm	05/13/02	Updated to match hw version, removed _g.h

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

```
#include "xplbarb.h"
```

Variables

[XPlbArb_Config XPlbArb_ConfigTable](#) [XPAR_XPLBARB_NUM_INSTANCES]

Variable Documentation

[XPlbArb_Config XPlbArb_ConfigTable](#)
[XPAR_XPLBARB_NUM_INSTANCES]

The PLB Arbiter configuration table, sized by the number of instances defined in [xparameters.h](#).

XPlbArb_Config Struct Reference

```
#include <xplbarb.h>
```

Detailed Description

This typedef contains configuration information for the device. This information would typically be extracted from Configuration ROM (CROM).

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xuint8 NumMasters](#)

Field Documentation

[Xuint32 XPlbArb_Config::](#)

BaseAddress

Register base
address

[Xuint16 XPlbArb_Config::](#)

DeviceId

Unique ID of
device

Xuint8 XPlbArb_Config::

NumMasters

Number of masters on the
bus

The documentation for this struct was generated from the following file:

- [plbarb/v1_01_a/src/xplbarb.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

plbarb/v1_01_a/src/xplbarb.c File Reference

Detailed Description

Contains required functions for the [XPlbArb](#) component. See [xplbarb.h](#) for more information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	12/7/01	First release
1.01a	rpm	05/13/02	Updated to match hw version, publicized LookupConfig

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xplbarb.h"  
#include "xplbarb_i.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

Functions

```
XStatus XPlbArb_Initialize (XPlbArb *InstancePtr, Xuint16 DeviceId)  
Xboolean XPlbArb_IsError (XPlbArb *InstancePtr)  
void XPlbArb_ClearErrors (XPlbArb *InstancePtr, Xuint8 Master)  
Xuint32 XPlbArb_GetErrorStatus (XPlbArb *InstancePtr, Xuint8 Master)  
Xuint32 XPlbArb_GetErrorAddress (XPlbArb *InstancePtr)  
Xuint8 XPlbArb_GetNumMasters (XPlbArb *InstancePtr)  
void XPlbArb_EnableInterrupt (XPlbArb *InstancePtr)  
void XPlbArb_DisableInterrupt (XPlbArb *InstancePtr)  
void XPlbArb_Reset (XPlbArb *InstancePtr)  
XPlbArb_Config * XPlbArb_LookupConfig (Xuint16 DeviceId)
```

Function Documentation

```
void XPlbArb_ClearErrors ( XPlbArb * InstancePtr, Xuint8 Master )
```

Clears the Errors for the specified master

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters - 1 on the bus

Returns:

None.

Note:

None.

```
void XPlbArb_DisableInterrupt ( XPlbArb * InstancePtr )
```

Disables the interrupt output from the arbiter

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

None

Note:

The Arbiter hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt source with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void XPlbArb_EnableInterrupt ( XPlbArb * InstancePtr )
```

Enables the interrupt output from the arbiter

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

None.

Note:

The Arbiter hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt source with a handler which

then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

Xuint32 (**XPlbArb ***)
XPlbArb_GetErrorAddress *InstancePtr*

Returns the PLB Address where the most recent error occurred. If there isn't an outstanding error, the last address in error is returned. 0x00000000 is the initial value coming out of reset.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

Address where error causing access occurred

Note:

Calling **XPlbArb_IsError()** is recommended to confirm that an error has occurred prior to calling this function to ensure that the data in the error address register is relevant.

Xuint32 (**XPlbArb ***
XPlbArb_GetErrorStatus *InstancePtr*,
Xuint8 *Master*
)

Returns the Error status for the specified master. These are bit masks.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters on the bus

Returns:

The current error status for the requested master on the PLB. The status is a bit-mask that can contain the following bit values:

XPA_DRIVING_BEAR_MASK	Indicates that an error has occurred and this master is driving the error address
XPA_ERROR_READ_MASK	Indicates the error was a read error (it is a write error otherwise).
XPA_ERROR_STATUS_LOCK_MASK	Indicates the error status and address are locked and cannot be overwritten.
XPA_PEAR_SIZE_MASK	Size of access that caused error
XPA_PEAR_TYPE_MASK	Type of access that caused error

Note:

None.

Xuint8 (**XPlbArb ***)
XPlbArb_GetNumMasters *InstancePtr*

Returns the number of masters associated with the arbiter.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

The number of masters. This is a number from 1 to the maximum of 32.

Note:

The value returned from this call needs to be adjusted if it is to be used as the argument for other calls since the masters are numbered from 0 and this function returns values starting at 1.

XStatus (**XPlbArb ***
XPlbArb_Initialize *InstancePtr*,
Xuint16 *DeviceId*
)

Initializes a specific **XPlbArb** instance. Looks up the configuration for the given device instance and initialize the instance structure.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XPlbArb** component.

Returns:

- o XST_SUCCESS if everything starts up as expected.
- o XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

Xboolean (**XPlbArb ***)
XPlbArb_IsError *InstancePtr*

Returns XTRUE if there is an error outstanding

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

Boolean XTRUE if there is an error, XFALSE if there is no current error.

Note:

None.

XPlbArb_Config* (**Xuint16**)
XPlbArb_LookupConfig *DeviceId*

Looks up the device configuration based on the unique device ID. The table XPlbArb_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for.

Returns:

A pointer to the configuration data for the device, or XNULL if no match is found.

Note:

None.

void (**XPlbArb ***)
XPlbArb_Reset *InstancePtr*

Forces a software reset to occur in the arbiter. Disables interrupts in the process.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

None.

Note:

Disables interrupts in the process.

plbarb/v1_01_a/src/xplbarb_i.h File Reference

Detailed Description

This file contains data which is shared between files and internal to the **XPlbArb** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	02/28/02	First release
1.01a	rpm	05/13/02	Updated to match hw version, moved identifiers to xplbarb_1.h

```
#include "xplbarb\_1.h"
```

Variables

[XPlbArb_Config](#) [XPlbArb_ConfigTable](#) []

Variable Documentation

[XPlbArb_Config](#) [XPlbArb_ConfigTable](#) ()

[]

The PLB Arbiter configuration table, sized by the number of instances defined in [xparameters.h](#).

plbarb/v1_01_a/src/xplbarb_l.h File Reference

Detailed Description

This file contains internal identifiers and low-level macros that can be used to access the device directly. See [xplbarb.h](#) for a description of the high-level driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01a	rpm	05/10/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

Defines

```
#define XPlbArb_mGetPesrMerrReg(BaseAddress)  
#define XPlbArb_mSetPesrMerrReg(BaseAddress, Mask)  
#define XPlbArb_mGetPesrMDriveReg(BaseAddress)  
#define XPlbArb_mGetPesrRnwReg(BaseAddress)  
#define XPlbArb_mGetPesrLockReg(BaseAddress)  
#define XPlbArb_mGetPearAddrReg(BaseAddress)  
#define XPlbArb_mGetPearByteEnReg(BaseAddress)  
#define XPlbArb_mGetControlReg(BaseAddress)  
#define XPlbArb_mEnableInterrupt(BaseAddress)  
#define XPlbArb_mDisableInterrupt(BaseAddress)  
#define XPlbArb_mReset(BaseAddress)
```

Define Documentation

```
#define XPlbArb_mDisableInterrupt( BaseAddress )
```

Disable interrupts in the bridge. Preserve the contents of the ctrl register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XPlbArb_mEnableInterrupt( BaseAddress )
```

Enable interrupts in the bridge. Preserve the contents of the ctrl register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XPlbArb_mGetControlReg( BaseAddress )
```

Get the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit control register contents.

Note:

None.

```
#define XPlbArb_mGetPearAddrReg( BaseAddress )
```

Get the error address (or PEAR), which is the address that just caused the error.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit error address.

Note:

None.

```
#define XPlbArb_mGetPearByteEnReg( BaseAddress )
```

Get the error address byte enable register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit error address byte enable register contents.

Note:

None.

#define XPlbArb_mGetPesrLockReg(BaseAddress)

Get the value of the lock bit register, which indicates whether the master has locked the error registers.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the PESR Lock error register.

Note:

None.

#define XPlbArb_mGetPesrMDriveReg(BaseAddress)

Get the master driving the error, if any.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the PESR Master driving error register.

Note:

None.

#define XPlbArb_mGetPesrMerrReg(BaseAddress)

Get the error status register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the error status register.

Note:

None.

```
#define XPlbArb_mGetPesrRnwReg( BaseAddress )
```

Get the value of the Read-Not-Write register, which indicates whether the error is a read error or write error.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the PESR RNW error register.

Note:

None.

```
#define XPlbArb_mReset( BaseAddress )
```

Reset the bridge. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XPlbArb_mSetPesrMerrReg( BaseAddress,  
                                Mask )
```

Set the error status register.

Parameters:

BaseAddress is the base address of the device

Mask is the 32-bit value to write to the error status register.

Note:

None.

plbarb/v1_01_a/src/xplbarb_selftest.c File Reference

Detailed Description

Contains diagnostic self-test functions for the **XPlbArb** component. See [xplbarb.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	11/16/01	First release
1.01a	rpm	05/13/02	Updated to match hw version, removed _g.h

```
#include "xstatus.h"  
#include "xplbarb.h"  
#include "xplbarb_i.h"  
#include "xio_dcr.h"
```

Functions

[XStatus XPlbArb_SelfTest](#) ([XPlbArb](#) *InstancePtr, [Xuint32](#) TestAddress)

Function Documentation

```
XStatus          ( XPlbArb *  
XPlbArb_SelfTest      InstancePtr,  
                        Xuint32  TestAddress  
                        )
```

Runs a self-test on the driver/device.

This tests reads the PACR to verify that the proper value is there.

XST_SUCCESS is returned if expected value is there, XST_PLBARB_FAIL_SELFTEST is returned otherwise.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

TestAddress is a location that could cause an error on read, not used - user definable for hw specific implementations.

Returns:

XST_SUCCESS if successful, or XST_PLBARB_FAIL_SELFTEST if the driver fails the self test.

Note:

None.

rapidio/v1_01_a/src/xrapidio_i.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	12/13/02	First release
1.01a	xd	06/02/05	Changed to support plb_rapidio_lvds v1.00c, which has different base addresses for register interface and TX/RX Packet buffers.

```
#include "xbasic_types.h"  
#include "xio.h"
```

Defines

```
#define XRapidIo_mReadReg(RegBaseAddress, RegOffset)  
#define XRapidIo_mWriteReg(RegBaseAddress, RegOffset, Data)  
#define XRapidIo_mReset(RegBaseAddress)  
#define XRapidIo_mGetLinkStatus(RegBaseAddress)
```

Functions

unsigned **XRapidIo_SendPkt** (**Xuint32** RegBaseAddress, **Xuint32** PacketBufBaseAddress, **Xuint8** *PktPtr, unsigned ByteCount)
unsigned **XRapidIo_RecvPkt** (**Xuint32** RegBaseAddress, **Xuint32** PacketBufBaseAddress, **Xuint8** *PktPtr)

Define Documentation

```
#define XRapidIo_mGetLinkStatus( RegBaseAddress )
```

Get the status of the PHY link.

Parameters:

RegBaseAddress is the base address of the device register interface

Returns:

A value representing the contents of the PHY LINK register.

Note:

C-Style Signature: Xuint8 **XRapidIo_mGetLinkStatus**(Xuint32 RegBaseAddress)

```
#define XRapidIo_mReadReg( RegBaseAddress,  
                          RegOffset      )
```

Read a 32-bit value from a register.

Parameters:

RegBaseAddress is the base address of the device register interface

RegOffset is the offset of the register to read

Returns:

The 32-bit register value

Note:

C-Style Signature: Xuint32 **XRapidIo_mReadReg**(Xuint32 RegBaseAddress,int RegOffset)

```
#define XRapidIo_mReset( RegBaseAddress )
```

Reset the device using the IPIF reset register. Also reset the static bin counters.

Parameters:

RegBaseAddress is the base address of the device register interface

Returns:

None.

Note:

C-Style Signature: void **XRapidIo_mReset**(Xuint32
RegBaseAddress)

```
#define XRapidIo_mWriteReg( RegBaseAddress,  
                           RegOffset,  
                           Data          )
```

Write a 32-bit value to a register.

Parameters:

RegBaseAddress is the base address of the device register
interface

RegOffset is the offset of the register to write

Data is the value to write

Returns:

None.

Note:

C-Style Signature: void XRapidIo_mWriteReg(Xuint32 RegBaseAddress,int RegOffset,
Xuint32 Data)

Function Documentation

```
unsigned
XRapidIo_RecvPkt      ( Xuint32  RegBaseAddress,
                        Xuint32  PacketBufBaseAddress,
                        Xuint8 * PktPtr
                        )
```

Receive a packet. Wait for a packet to arrive.

Parameters:

RegBaseAddress is the base address of the device register interface
PacketBufBaseAddress is the base address of the device packet buffers
PktPtr is a pointer to a 64-bit word-aligned buffer where the packet will be stored.

Returns:

The size, in bytes, of the packet received, or 0 if the incoming buffer is not 64-bit address aligned.

Note:

TODO: enforce 64-bit address alignment?

```
unsigned
XRapidIo_SendPkt      ( Xuint32  RegBaseAddress,
                        Xuint32  PacketBufBaseAddress,
                        Xuint8 * PktPtr,
                        unsigned ByteCount
                        )
```

Send a RapidIO packet. The byte count is the total packet size, including header. This function blocks waiting for the packet to be transmitted.

Parameters:

RegBaseAddress is the base address of the device register interface
PacketBufBaseAddress is the base address of the device packet buffers
PktPtr is a pointer to 64-bit word-aligned packet
ByteCount is the number of bytes in the packet

Returns:

The number of bytes sent. If an error occurs, such as the data is not 64-bit word aligned, a value of 0 is returned.

Note:

The data is written to the packet buffer in 32-bit chunks.

Generated on 24 Jun 2004 for Xilinx Device Drivers

rapidio/v1_01_a/src/xrapidio_l.c File Reference

Detailed Description

This file contains low-level polled functions to send and receive RapidIO frames.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	12/13/02	First release
1.01a	xd	06/02/05	Changed to support plb_rapidio_lvds v1.00c, which has different base addresses for register interface and TX/RX Packet buffers.

```
#include "xrapidio_l.h"  
#include "xio.h"
```

Functions

unsigned **XRapidIo_SendPkt** (**Xuint32** RegBaseAddress, **Xuint32** PacketBufBaseAddress, **Xuint8** *PktPtr, unsigned ByteCount)
unsigned **XRapidIo_RecvPkt** (**Xuint32** RegBaseAddress, **Xuint32** PacketBufBaseAddress, **Xuint8** *PktPtr)

Function Documentation

```
unsigned
XRapidIo_RecvPkt      ( Xuint32  RegBaseAddress,
                        Xuint32  PacketBufBaseAddress,
                        Xuint8 * PktPtr
                        )
```

Receive a packet. Wait for a packet to arrive.

Parameters:

RegBaseAddress is the base address of the device register interface
PacketBufBaseAddress is the base address of the device packet buffers
PktPtr is a pointer to a 64-bit word-aligned buffer where the packet will be stored.

Returns:

The size, in bytes, of the packet received, or 0 if the incoming buffer is not 64-bit address aligned.

Note:

TODO: enforce 64-bit address alignment?

```
unsigned
XRapidIo_SendPkt      ( Xuint32  RegBaseAddress,
                        Xuint32  PacketBufBaseAddress,
                        Xuint8 * PktPtr,
                        unsigned ByteCount
                        )
```

Send a RapidIO packet. The byte count is the total packet size, including header. This function blocks waiting for the packet to be transmitted.

Parameters:

RegBaseAddress is the base address of the device register interface
PacketBufBaseAddress is the base address of the device packet buffers
PktPtr is a pointer to 64-bit word-aligned packet
ByteCount is the number of bytes in the packet

Returns:

The number of bytes sent. If an error occurs, such as the data is not 64-bit word aligned, a value of 0 is returned.

Note:

The data is written to the packet buffer in 32-bit chunks.

Generated on 24 Jun 2004 for Xilinx Device Drivers

sysace/v1_00_a/src/xsysace_i.h File Reference

Detailed Description

Defines identifiers and low-level macros/functions for the **XSysAce** driver. These identifiers include register offsets and bit masks. A high-level driver interface is defined in [xsysace.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	06/14/02	work in progress

```
#include "xbasic_types.h"  
#include "xio.h"
```

Register Offsets

System ACE register offsets

```
#define XSA_BMR_OFFSET  
#define XSA_SR_OFFSET  
#define XSA_ER_OFFSET  
#define XSA_CLR_OFFSET  
#define XSA_MLR_OFFSET  
#define XSA_SCCR_OFFSET  
#define XSA_VR_OFFSET  
#define XSA_CR_OFFSET  
#define XSA_FSR_OFFSET  
#define XSA_DBR_OFFSET
```

Status Values

Status Register masks

```
#define XSA_SR_CFGLOCK_MASK
#define XSA_SR_MPULOCK_MASK
#define XSA_SR_CFGERROR_MASK
#define XSA_SR_CFCERROR_MASK
#define XSA_SR_CFDTECT_MASK
#define XSA_SR_DATABUFRDY_MASK
#define XSA_SR_DATABUFMODE_MASK
#define XSA_SR_CFGDONE_MASK
#define XSA_SR_RDYFORCMD_MASK
#define XSA_SR_CFGMODE_MASK
#define XSA_SR_CFGADDR_MASK
#define XSA_SR_CFBSY_MASK
#define XSA_SR_CFRDY_MASK
#define XSA_SR_CFDWF_MASK
#define XSA_SR_CFDSC_MASK
#define XSA_SR_CFDRQ_MASK
#define XSA_SR_CFCORR_MASK
#define XSA_SR_CFERR_MASK
```

Error Values

Error Register masks.

```
#define XSA_ER_CARD_RESET
#define XSA_ER_CARD_READY
#define XSA_ER_CARD_READ
#define XSA_ER_CARD_WRITE
#define XSA_ER_SECTOR_READY
#define XSA_ER_CFG_ADDR
#define XSA_ER_CFG_FAIL
#define XSA_ER_CFG_READ
#define XSA_ER_CFG_INSTR
#define XSA_ER_CFG_INIT
#define XSA_ER_RESERVED
```

```
#define XSA_ER_BAD_BLOCK
#define XSA_ER_UNCORRECTABLE
#define XSA_ER_SECTOR_ID
#define XSA_ER_ABORT
#define XSA_ER_GENERAL
```

Sector Count/Command Values

Sector Count Command Register masks

```
#define XSA_SCCR_COUNT_MASK
#define XSA_SCCR_RESET_MASK
#define XSA_SCCR_IDENTIFY_MASK
#define XSA_SCCR_READDATA_MASK
#define XSA_SCCR_WRITEDATA_MASK
#define XSA_SCCR_ABORT_MASK
#define XSA_SCCR_CMD_MASK
```

Control Values

Control Register masks

```
#define XSA_CR_FORCELOCK_MASK
#define XSA_CR_LOCKREQ_MASK
#define XSA_CR_FORCECFGADDR_MASK
#define XSA_CR_FORCECFGMODE_MASK
#define XSA_CR_CFGMODE_MASK
#define XSA_CR_CFGSTART_MASK
#define XSA_CR_CFGSEL_MASK
#define XSA_CR_CFGRESET_MASK
#define XSA_CR_DATARDYIRQ_MASK
#define XSA_CR_ERRORIRQ_MASK
#define XSA_CR_CFGDONEIRQ_MASK
#define XSA_CR_RESETIRQ_MASK
#define XSA_CR_CFGPROG_MASK
#define XSA_CR_CFGADDR_MASK
#define XSA_CR_CFGADDR_SHIFT
```

FAT Status

FAT filesystem status masks. The first valid partition of the CF is a FAT partition.

```
#define XSA_FAT_VALID_BOOT_REC
#define XSA_FAT_VALID_PART_REC
#define XSA_FAT_12_BOOT_REC
#define XSA_FAT_12_PART_REC
#define XSA_FAT_16_BOOT_REC
#define XSA_FAT_16_PART_REC
#define XSA_FAT_12_CALC
#define XSA_FAT_16_CALC
```

Defines

```
#define XSA_BMR_16BIT_MASK
#define XSA_CLR_LBA_MASK
#define XSA_MLR_LBA_MASK
#define XSA_DATA_BUFFER_SIZE
#define XSA_CF_SECTOR_SIZE
#define XSysAce_mGetControlReg(BaseAddress)
#define XSysAce_mSetControlReg(BaseAddress, Data)
#define XSysAce_mOrControlReg(BaseAddress, Data)
#define XSysAce_mAndControlReg(BaseAddress, Data)
#define XSysAce_mGetErrorReg(BaseAddress)
#define XSysAce_mGetStatusReg(BaseAddress)
#define XSysAce_mSetCfgAddr(BaseAddress, Address)
#define XSysAce_mWaitForLock(BaseAddress)
#define XSysAce_mEnableIntr(BaseAddress, Mask)
#define XSysAce_mDisableIntr(BaseAddress, Mask)
#define XSysAce_mIsReadyForCmd(BaseAddress)
#define XSysAce_mIsMpuLocked(BaseAddress)
#define XSysAce_mIsCfgDone(BaseAddress)
#define XSysAce_mIsIntrEnabled(BaseAddress)
```

Functions

```
int XSysAce_ReadSector (Xuint32 BaseAddress, Xuint32 SectorId, Xuint8 *BufferPtr)
```

```
int XSysAce_WriteSector (Xuint32 BaseAddress, Xuint32 SectorId, Xuint8 *BufferPtr)
Xuint32 XSysAce_RegRead32 (Xuint32 Address)
Xuint16 XSysAce_RegRead16 (Xuint32 Address)
void XSysAce_RegWrite32 (Xuint32 Address, Xuint32 Data)
void XSysAce_RegWrite16 (Xuint32 Address, Xuint16 Data)
int XSysAce_ReadDataBuffer (Xuint32 BaseAddress, Xuint8 *BufferPtr, int NumBytes)
int XSysAce_WriteDataBuffer (Xuint32 BaseAddress, Xuint8 *BufferPtr, int NumBytes)
```

Define Documentation

```
#define
XSA_BMR_16BIT_MASK
```

16-bit access to ACE
controller

```
#define
XSA_BMR_OFFSET
```

Bus mode
(BUSMODEREG)

```
#define
XSA_CF_SECTOR_SIZE
```

Number of bytes in a CF
sector

```
#define
XSA_CLR_LBA_MASK
```

Config LBA Register - address
mask

```
#define
XSA_CLR_OFFSET
```

Config LBA
(CFGLBAREG)

#define

XSA_CR_CFGADDR_MASK

Config address
mask

#define

XSA_CR_CFGADDR_SHIFT

Config address
shift

#define

XSA_CR_CFGDONEIRQ_MASK

Enable CFG done
IRQ

#define

XSA_CR_CFGMODE_MASK

CFG
mode

#define

XSA_CR_CFGPROG_MASK

Inverted CFGPROG
pin

#define

XSA_CR_CFGRESET_MASK

CFG
reset

#define

XSA_CR_CFGSEL_MASK

CFG
select

#define

XSA_CR_CFGSTART_MASK

CFG
start

```
#define  
XSA_CR_DATARDYIRQ_MASK
```

Enable data ready
IRQ

```
#define  
XSA_CR_ERRORIRQ_MASK
```

Enable error
IRQ

```
#define  
XSA_CR_FORCECFGADDR_MASK
```

Force CFG
address

```
#define  
XSA_CR_FORCECFGMODE_MASK
```

Force CFG
mode

```
#define  
XSA_CR_FORCELOCK_MASK
```

Force lock
request

```
#define  
XSA_CR_LOCKREQ_MASK
```

MPU lock
request

```
#define  
XSA_CR_OFFSET
```

Control
(CONTROLREG)

#define
XSA_CR_RESETIRQ_MASK

Reset IRQ
line

#define
XSA_DATA_BUFFER_SIZE

Size of System ACE data
buffer

#define
XSA_DBR_OFFSET

Data buffer
(DATABUFREG)

#define
XSA_ER_ABORT

CF command
aborted

#define
XSA_ER_BAD_BLOCK

CF bad block
detected

#define
XSA_ER_CARD_READ

CF read command
failed

#define
XSA_ER_CARD_READY

CF card failed to
ready

#define

XSA_ER_CARD_RESET

CF card failed to
reset

#define

XSA_ER_CARD_WRITE

CF write command
failed

#define

XSA_ER_CFG_ADDR

Cfg address is
invalid

#define

XSA_ER_CFG_FAIL

Failed to configure a
device

#define

XSA_ER_CFG_INIT

CFGINIT pin error - did not go high within 500ms of
start

#define

XSA_ER_CFG_INSTR

Invalid instruction during
cfg

#define

XSA_ER_CFG_READ

Cfg read of CF
failed

#define

XSA_ER_GENERAL

CF general
error

```
#define  
XSA_ER_OFFSET
```

Error
(ERRORREG)

```
#define  
XSA_ER_RESERVED
```

reserved

```
#define  
XSA_ER_SECTOR_ID
```

CF sector ID not
found

```
#define  
XSA_ER_SECTOR_READY
```

CF sector failed to
ready

```
#define  
XSA_ER_UNCORRECTABLE
```

CF uncorrectable
error

```
#define  
XSA_FAT_12_BOOT_REC
```

FAT12 in master boot
rec

```
#define  
XSA_FAT_12_CALC
```

Calculated FAT12 from
clusters

#define

XSA_FAT_12_PART_REC

FAT12 in partition boot
rec

#define

XSA_FAT_16_BOOT_REC

FAT16 in master boot
rec

#define

XSA_FAT_16_CALC

Calculated FAT16 from
clusters

#define

XSA_FAT_16_PART_REC

FAT16 in partition boot
rec

#define

XSA_FAT_VALID_BOOT_REC

Valid master boot
record

#define

XSA_FAT_VALID_PART_REC

Valid partition boot
record

#define

XSA_FSR_OFFSET

FAT status
(FATSTATREG)

#define

XSA_MLR_LBA_MASK

MPU LBA Register - address
mask

#define

XSA_MLR_OFFSET

MPU LBA
(MPULBAREG)

#define

XSA_SCCR_ABORT_MASK

Abort CF
command

#define

XSA_SCCR_CMD_MASK

Command
mask

#define

XSA_SCCR_COUNT_MASK

Sector count
mask

#define

XSA_SCCR_IDENTIFY_MASK

Identify CF card
command

#define

XSA_SCCR_OFFSET

Sector cnt
(SECCNTCMDREG)

#define

XSA_SCCR_READDATA_MASK

Read CF card
command

```
#define  
XSA_SCCR_RESET_MASK
```

Reset CF card
command

```
#define  
XSA_SCCR_WRITEDATA_MASK
```

Write CF card
command

```
#define  
XSA_SR_CFBSY_MASK
```

CF busy (BSY
bit)

```
#define  
XSA_SR_CFCERROR_MASK
```

CF error
status

```
#define  
XSA_SR_CFCORR_MASK
```

CF correctable error (CORR
bit)

```
#define  
XSA_SR_CFDETECT_MASK
```

CF detect
flag

```
#define  
XSA_SR_CFDRQ_MASK
```

CF data request
(DRQ)

```
#define  
XSA_SR_CFDSC_MASK
```

CF ready (DSC
bit)

```
#define  
XSA_SR_CFDWF_MASK
```

CF data write fault (DWF
bit)

```
#define  
XSA_SR_CFERR_MASK
```

CF error (ERR
bit)

```
#define  
XSA_SR_CFGADDR_MASK
```

Configuration
address

```
#define  
XSA_SR_CFGDONE_MASK
```

Configuration done
status

```
#define  
XSA_SR_CFGERROR_MASK
```

Config port error
status

```
#define  
XSA_SR_CFGLOCK_MASK
```

Config port lock
status

#define

XSA_SR_CFGMODE_MASK

Configuration mode
status

#define

XSA_SR_CFRDY_MASK

CF ready (RDY
bit)

#define

XSA_SR_DATABUFMODE_MASK

Data buffer mode
status

#define

XSA_SR_DATABUFRDY_MASK

Data buffer ready
status

#define

XSA_SR_MPULOCK_MASK

MPU port lock
status

#define

XSA_SR_OFFSET

Status
(STATUSREG)

#define

XSA_SR_RDYFORCMD_MASK

Ready for CF
command

#define

XSA_VR_OFFSET

Version
(VERSIONREG)

```
#define XSysAce_mAndControlReg( BaseAddress,  
                                Data          )
```

Set the contents of the control register to the value specified AND'ed with its current contents.

Parameters:

BaseAddress is the base address of the device.

Data is the 32-bit value to AND with the register.

Returns:

None.

Note:

None.

```
#define XSysAce_mDisableIntr( BaseAddress,  
                               Mask        )
```

Disable ACE controller interrupts.

Parameters:

BaseAddress is the base address of the device.

Returns:

None.

Note:

None.

```
#define XSysAce_mEnableIntr( BaseAddress,  
                              Mask        )
```

Enable ACE controller interrupts.

Parameters:

BaseAddress is the base address of the device.

Returns:

None.

Note:

None.

```
#define XSysAce_mGetControlReg( BaseAddress )
```

Get the contents of the control register.

Parameters:

BaseAddress is the base address of the device.

Returns:

The 32-bit value of the control register.

Note:

None.

```
#define XSysAce_mGetErrorReg( BaseAddress )
```

Get the contents of the error register.

Parameters:

BaseAddress is the base address of the device.

Returns:

The 32-bit value of the register.

Note:

None.

#define XSysAce_mGetStatusReg(BaseAddress)

Get the contents of the status register.

Parameters:

BaseAddress is the base address of the device.

Returns:

The 32-bit value of the register.

Note:

None.

#define XSysAce_mIsCfgDone(BaseAddress)

Is the CompactFlash configuration of the target FPGA chain complete?

Parameters:

BaseAddress is the base address of the device.

Returns:

XTRUE if it is ready, XFALSE otherwise.

Note:

None.

#define XSysAce_mIsIntrEnabled(BaseAddress)

Have interrupts been enabled by the user? We look for the interrupt reset bit to be clear (meaning interrupts are armed, even though none may be individually enabled).

Parameters:

BaseAddress is the base address of the device.

Returns:

XTRUE if it is enabled, XFALSE otherwise.

Note:

None.

Set the contents of the control register to the value specified OR'ed with its current contents.

Parameters:

BaseAddress is the base address of the device.

Data is the 32-bit value to OR with the register.

Returns:

None.

Note:

None.

```
#define XSysAce_mSetCfgAddr( BaseAddress,  
                             Address      )
```

Set the configuration address, or file, of the CompactFlash. This address indicates which .ace bitstream to use to configure the target FPGA chain.

Parameters:

BaseAddress is the base address of the device.

Address ranges from 0 to 7 and represents the eight possible .ace bitstreams that can reside on the CompactFlash.

Returns:

None.

Note:

Used cryptic var names to avoid conflict with caller's var names.

```
#define XSysAce_mSetControlReg( BaseAddress,  
                                Data        )
```

Set the contents of the control register.

Parameters:

BaseAddress is the base address of the device.

Data is the 32-bit value to write to the register.

Returns:

None.

Note:

None.

```
#define XSysAce_mWaitForLock( BaseAddress )
```

Request then wait for the MPU lock. This is not a forced lock, so we must contend with the configuration controller.

Parameters:

BaseAddress is the base address of the device.

Returns:

None.

Note:

None.

Function Documentation

```
int XSysAce_ReadDataBuffer ( Xuint32 BaseAddress,  
                             Xuint8 * BufferPtr,  
                             int Size  
                             )
```

Read the specified number of bytes from the data buffer of the ACE controller. The data buffer, which is 32 bytes, can only be read two bytes at a time. Once the data buffer is read, we wait for it to be filled again before reading the next buffer's worth of data.

Parameters:

- BaseAddress* is the base address of the device
- BufferPtr* is a pointer to a buffer in which to store data.
- Size* is the number of bytes to read

Returns:

The total number of bytes read, or 0 if an error occurred.

Note:

If *Size* is not aligned with the size of the data buffer (32 bytes), this function will read the entire data buffer, dropping the extra bytes on the floor since the user did not request them. This is necessary to get the data buffer to be ready again.

```
int XSysAce_ReadSector ( Xuint32 BaseAddress,  
                        Xuint32 SectorId,  
                        Xuint8 * BufferPtr  
                        )
```

Read a CompactFlash sector. This is a blocking, low-level function which does not return until the specified sector is read.

Parameters:

- BaseAddress* is the base address of the device
- SectorId* is the id of the sector to read
- BufferPtr* is a pointer to a buffer where the data will be stored.

Returns:

The number of bytes read. If this number is not equal to the sector size, 512 bytes, then an error occurred.

Note:

None.

Xuint16 (**Xuint32**)

XSysAce_RegRead16 *Address*

Read a 16-bit value from the given address. Based on a compile-time constant, do the read in one 16-bit read or two 8-bit reads.

Parameters:

Address is the address to read from.

Returns:

The 16-bit value of the address.

Note:

No need for endian conversion in 8-bit mode since this function gets the bytes into their proper lanes in the 16-bit word.

Xuint32 (**Xuint32**)

XSysAce_RegRead32 *Address*

Read a 32-bit value from the given address. Based on a compile-time constant, do the read in two 16-bit reads or four 8-bit reads.

Parameters:

Address is the address to read from.

Returns:

The 32-bit value of the address.

Note:

No need for endian conversion in 8-bit mode since this function gets the bytes into their proper lanes in the 32-bit word.

void (**Xuint32**
XSysAce_RegWrite16 *Address*,
Xuint16 *Data*
)

Write a 16-bit value to the given address. Based on a compile-time constant, do the write in one 16-bit write or two 8-bit writes.

Parameters:

Address is the address to write to.

Data is the value to write

Returns:

None.

Note:

No need for endian conversion in 8-bit mode since this function writes the bytes into their proper lanes based on address.

```
void XSysAce_RegWrite32 ( Xuint32 Address, Xuint32 Data )
```

Write a 32-bit value to the given address. Based on a compile-time constant, do the write in two 16-bit writes or four 8-bit writes.

Parameters:

Address is the address to write to.

Data is the value to write

Returns:

None.

Note:

No need for endian conversion in 8-bit mode since this function writes the bytes into their proper lanes based on address.

```
int XSysAce_WriteDataBuffer ( Xuint32 BaseAddress,  
                             Xuint8 * BufferPtr,  
                             int Size  
                             )
```

Write the specified number of bytes to the data buffer of the ACE controller. The data buffer, which is 32 bytes, can only be written two bytes at a time. Once the data buffer is written, we wait for it to be empty again before writing the next buffer's worth of data. If the size of the incoming buffer is not aligned with the System ACE data buffer size (32 bytes), then this routine pads out the data buffer with zeros so the entire data buffer is written. This is necessary for the ACE controller to process the data buffer.

Parameters:

BaseAddress is the base address of the device

BufferPtr is a pointer to a buffer used to write to the controller.

Size is the number of bytes to write

Returns:

The total number of bytes written (not including pad bytes), or 0 if an error occurs.

Note:

None.

```
int XSysAce_WriteSector ( Xuint32 BaseAddress,  
                          Xuint32 SectorId,  
                          Xuint8 * BufferPtr  
                          )
```

Write a CompactFlash sector. This is a blocking, low-level function which does not return until the specified sector is written in its entirety.

Parameters:

BaseAddress is the base address of the device

SectorId is the id of the sector to write

BufferPtr is a pointer to a buffer used to write the sector.

Returns:

The number of bytes written. If this number is not equal to the sector size, 512 bytes, then an error occurred.

Note:

None.

XSysAce Struct Reference

```
#include <xsysace.h>
```

Detailed Description

The XSysAce driver instance data. The user is required to allocate a variable of this type for every System ACE device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [sysace/v1_00_a/src/xsysace.h](#)
-

sysace/v1_00_a/src/xsysace.h File Reference

Detailed Description

The Xilinx System ACE driver. This driver supports the Xilinx System Advanced Configuration Environment (ACE) controller. It currently supports only the CompactFlash solution. The driver makes use of the Microprocessor (MPU) interface to communicate with the device.

The driver provides a user the ability to access the CompactFlash through the System ACE device. The user can read and write CompactFlash sectors, identify the flash device, and reset the flash device. Also, the driver provides a user the ability to configure FPGA devices by selecting a configuration file (.ace file) resident on the CompactFlash, or directly configuring the FPGA devices via the MPU port and the configuration JTAG port of the controller.

Bus Mode

The System ACE device supports both 8-bit and 16-bit access to its registers. The driver defaults to 8-bit access, but can be changed to use 16-bit access at compile-time. The compile-time constant `XPAR_XSYSACE_MEM_WIDTH` must be defined equal to 16 to make the driver use 16-bit access. This constant is typically defined in [xparameters.h](#).

Endianness

The System ACE device is little-endian. If being accessed by a big-endian processor, the endian conversion will be done by the device driver. The endian conversion is encapsulated inside the `XSysAce_RegRead/Write` functions so that it can be removed if the endian conversion is moved to hardware.

Hardware Access

The device driver expects the System ACE controller to be a memory-mapped device. Access to the System ACE controller is typically achieved through the External Memory Controller (EMC) IP core. The EMC is simply a pass-through device that allows access to the off-chip System ACE device. There is no software-based setup or configuration necessary for the EMC.

The System ACE registers are expected to be byte-addressable. If for some reason this is not possible, the register offsets defined in [xsysace_1.h](#) must be changed accordingly.

Reading or Writing CompactFlash

The smallest unit that can be read from or written to CompactFlash is one sector. A sector is 512 bytes. The functions provided by this driver allow the user to specify a starting sector ID and the number of sectors to be read or written. At most 256 sectors can be read or written in one operation. The user must ensure that the buffer passed to the functions is big enough to hold $(512 * \text{NumSectors})$, where NumSectors is the number of sectors specified.

Interrupt Mode

By default, the device and driver are in polled mode. The user is required to enable interrupts using [XSysAce_EnableInterrupt\(\)](#). In order to use interrupts, it is necessary for the user to connect the driver's interrupt handler, [XSysAce_InterruptHandler\(\)](#), to the interrupt system of the application. This function does not save and restore the processor context. An event handler must also be set by the user, using [XSysAce_SetEventHandler\(\)](#), for the driver such that the handler is called when interrupt events occur. The handler is called from interrupt context and allows application-specific processing to be performed.

In interrupt mode, the only available interrupt is data buffer ready, so the size of a data transfer between interrupts is 32 bytes (the size of the data buffer).

Polled Mode

The sector read and write functions are blocking when in polled mode. This choice was made over non-blocking since sector transfer rates are high (>20Mbps) and the user can limit the number of sectors transferred in a single operation to 1 when in polled mode, plus the API for non-blocking polled functions was a bit awkward. Below is some more information on the sector transfer rates given the current state of technology (year 2002). Although the seek times for CompactFlash cards is high, this average hit needs to be taken every time a new read/write operation is invoked by the user. So the additional few microseconds to transfer an entire sector along with seeking is miniscule.

- Microdrives are slower than CompactFlash cards by a significant factor, especially if the MD is asleep.
 - Microdrive:
 - Power-up/wake-up time is approx. 150 to 1000 ms.
 - Average seek time is approx. 15 to 20 ms.
 - CompactFlash:
 - Power-up/reset time is approx. 50 to 400 ms and wake-up time is approx. 3 ms.

- "Seek time" here means how long it takes the internal controller to process the command until the sector data is ready for transfer by the ACE controller. This time is approx. 2 ms per sector.
- Once the sector data is ready in the CF device buffer (i.e., "seek time" is over) the ACE controller can read 2 bytes from the MD/CF device every 11 clock cycles, assuming no wait cycles happen. For instance, if the clock is 33 MHz, then then the max. rate that the ACE controller can transfer is 6 MB/sec. However, due to other overhead (e.g., time for data buffer transfers over MPU port, etc.), a better estimate is 3-5 MB/sec.

Mutual Exclusion

This driver is not thread-safe. The System ACE device has a single data buffer and therefore only one operation can be active at a time. The device driver does not prevent the user from starting an operation while a previous operation is still in progress. It is up to the user to provide this mutual exclusion.

Errors

Error causes are defined in [xsysace_1.h](#) using the prefix XSA_ER_*. The user can use [XSysAce_GetErrors\(\)](#) to retrieve all outstanding errors.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	06/17/02	work in progress

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xsysace_1.h"
```

Data Structures

```
struct XSysAce
struct XSysAce_CFParameters
struct XSysAce_Config
```

Asynchronous Events

Asynchronous events passed to the event handler when in interrupt mode.

Note that when an error event occurs, the only way to clear this condition is to reset the CompactFlash or the System ACE configuration controller, depending on where the error occurred. The driver does not reset either and leaves this task to the user.

```
#define XSA_EVENT_CFG_DONE
#define XSA_EVENT_DATA_DONE
#define XSA_EVENT_ERROR
```

Typedefs

```
typedef void(* XSysAce_EventHandler)(void *CallBackRef, int Event)
```

Functions

```
XStatus XSysAce_Initialize (XSysAce *InstancePtr, Xuint16 DeviceId)
```

```
XStatus XSysAce_Lock (XSysAce *InstancePtr, Xboolean Force)
```

```
void XSysAce_Unlock (XSysAce *InstancePtr)
```

```
Xuint32 XSysAce_GetErrors (XSysAce *InstancePtr)
```

```
XSysAce_Config * XSysAce_LookupConfig (Xuint16 DeviceId)
```

```
XStatus XSysAce_ResetCF (XSysAce *InstancePtr)
```

```
XStatus XSysAce_AbortCF (XSysAce *InstancePtr)
```

```
XStatus XSysAce_IdentifyCF (XSysAce *InstancePtr, XSysAce_CFParameters
*ParamPtr)
```

```
Xboolean XSysAce_IsCFReady (XSysAce *InstancePtr)
```

```
XStatus XSysAce_SectorRead (XSysAce *InstancePtr, Xuint32 StartSector, int
NumSectors, Xuint8 *BufferPtr)
```

```
XStatus XSysAce_SectorWrite (XSysAce *InstancePtr, Xuint32 StartSector, int
NumSectors, Xuint8 *BufferPtr)
```

```
Xuint16 XSysAce_GetFatStatus (XSysAce *InstancePtr)
```

```
void XSysAce_ResetCfg (XSysAce *InstancePtr)
```

```
void XSysAce_SetCfgAddr (XSysAce *InstancePtr, unsigned int Address)
```

```
void XSysAce_SetStartMode (XSysAce *InstancePtr, Xboolean ImmedOnReset,
Xboolean SetStart)
```

```
Xboolean XSysAce_IsCfgDone (XSysAce *InstancePtr)
```

```
Xuint32 XSysAce_GetCfgSector (XSysAce *InstancePtr)
```

```
XStatus XSysAce_ProgramChain (XSysAce *InstancePtr, Xuint8 *BufferPtr, int
NumBytes)
```

```
void XSysAce_EnableInterrupt (XSysAce *InstancePtr)
void XSysAce_DisableInterrupt (XSysAce *InstancePtr)
void XSysAce_SetEventHandler (XSysAce *InstancePtr, XSysAce_EventHandler
    FuncPtr, void *CallbackRef)
void XSysAce_InterruptHandler (void *InstancePtr)
XStatus XSysAce_SelfTest (XSysAce *InstancePtr)
Xuint16 XSysAce_GetVersion (XSysAce *InstancePtr)
```

Define Documentation

```
#define
XSA_EVENT_CFG_DONE
```

Configuration of JTAG chain is done

```
#define
XSA_EVENT_DATA_DONE
```

Data transfer to/from CompactFlash is done

```
#define
XSA_EVENT_ERROR
```

An error occurred. Use **XSysAce_GetErrors**() to determine the cause of the error (s).

Typedef Documentation

```
typedef void(* XSysAce_EventHandler)(void *CallbackRef, int
Event)
```

Callback when an asynchronous event occurs during interrupt mode.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

Event is the event that occurred. See [xsysace.h](#) and the event identifiers prefixed with XSA_EVENT_* for a description of possible events.

Function Documentation

```
XStatus ( XSysAce * )  
XSysAce_AbortCF InstancePtr
```

Abort the CompactFlash operation currently in progress.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

- XST_SUCCESS if the abort was done successfully
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the CompactFlash is not ready for a command

Note:

According to the ASIC designer, the abort command has not been well tested.

```
void ( XSysAce * )  
XSysAce_DisableInterrupt InstancePtr
```

Disable all System ACE interrupts and hold the interrupt request line of the device in reset.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance that just interrupted.

Returns:

None.

Note:

None.

```
void ( XSysAce * )  
XSysAce_EnableInterrupt InstancePtr
```

Enable System ACE interrupts. There are three interrupts that can be enabled. The error interrupt enable serves as the driver's means to determine whether interrupts have been enabled or not. The configuration-done interrupt is not enabled here, instead it is enabled during a reset - which can cause a configuration process to start. The data-buffer-ready interrupt is not enabled here either. It is enabled when a read or write operation is started. The reason for not enabling the latter two interrupts are because the status bits may be set as a leftover of an earlier occurrence of the interrupt.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to work on.

Returns:

None.

Note:

None.

```
Xuint32 ( XSysAce * )  
XSysAce_GetCfgSector InstancePtr
```

Get the sector ID of the CompactFlash sector being used for configuration of the target FPGA chain. This sector ID (or logical block address) only affects transfers between the ACE configuration logic and the CompactFlash card. This function is typically used for debug purposes to determine which sector was being accessed when an error occurred.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

The sector ID (logical block address) being used for data transfers between the ACE configuration logic and the CompactFlash. Sector IDs range from 0 to 0x10000000.

Note:

None.

```
Xuint32 ( XSysAce * )  
XSysAce\_GetErrors InstancePtr
```

Get all outstanding errors. Errors include the inability to read or write CompactFlash and the inability to successfully configure FPGA devices along the target FPGA chain.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

A 32-bit mask of error values. See [xsysace_1.h](#) for a description of possible values. The error identifiers are prefixed with XSA_ER_*.

Note:

None.

```
Xuint16 ( XSysAce * )  
XSysAce\_GetFatStatus InstancePtr
```

Get the status of the FAT filesystem on the first valid partition of the CompactFlash device such as the boot record and FAT types found.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

A 16-bit mask of status values. These values are defined in **xsysace_1.h** with the prefix XSA_FAT_*.

Note:

None.

```
Xuint16 ( XSysAce * )  
XSysAce_GetVersion InstancePtr
```

Retrieve the version of the System ACE device.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

A 16-bit version where the 4 most significant bits are the major version number, the next four bits are the minor version number, and the least significant 8 bits are the revision or build number.

Note:

None.

```
XStatus ( XSysAce *  
XSysAce_IdentifyCF InstancePtr,  
XSysAce_CFPParameters * ParamPtr  
)
```

Identify the CompactFlash device. Retrieves the parameters for the CompactFlash storage device. Note that this is a polled read of one sector of data. The data is read from the CompactFlash into a byte buffer, which is then copied into the **XSysAce_CFPParameters** structure passed in by the user. The copy is necessary since we don't know how the compiler packs the **XSysAce_CFPParameters** structure.

An MPU lock, obtained using **XSysAce_Lock()**, must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

ParamPtr is a pointer to a **XSysAce_CFPParameters** structure where the information for the CompactFlash device will be stored. See **xsysace.h** for details on the **XSysAce_CFPParameters** structure.

Returns:

- XST_SUCCESS if the identify was done successfully
- XST_FAILURE if an error occurs. Use **XSysAce_GetErrors()** to determine cause.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the CompactFlash is not ready for a command

Note:

None.

```
XStatus ( XSysAce *  
XSysAce_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize a specific **XSysAce** instance. The configuration information for the given device ID is found and the driver instance data is initialized appropriately.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XSysAce** instance.

Returns:

XST_SUCCESS if successful, or XST_DEVICE_NOT_FOUND if the device was not found in the configuration table in **xsysace_g.c**.

Note:

We do not want to reset the configuration controller here since this could cause a reconfiguration of the JTAG target chain, depending on how the CFGMODEPIN of the device is wired.

```
void                               ( void *                               )
XSysAce_InterruptHandler           InstancePtr
```

The interrupt handler for the System ACE driver. This handler must be connected by the user to an interrupt controller or source. This function does not save or restore context.

This function continues reading or writing to the compact flash if such an operation is in progress, and notifies the upper layer software through the event handler once the operation is complete or an error occurs. On an error, any command currently in progress is aborted.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance that just interrupted.

Returns:

None.

Note:

None.

```
Xboolean                           ( XSysAce *                           )
XSysAce_IsCfgDone                   InstancePtr
```

Check to see if the configuration of the target FPGA chain is complete. This function is typically only used in polled mode. In interrupt mode, an event (XSA_EVENT_CFG_DONE) is returned to the user in the asynchronous event handler.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

XTRUE if the configuration is complete. XFALSE otherwise.

Note:

None.

```
Xboolean ( XSysAce * )  
XSysAce_IsCFReady InstancePtr
```

Check to see if the CompactFlash is ready for a command. The CompactFlash may delay after one operation before it is ready for the next. This function helps the user determine when it is ready before invoking a CompactFlash operation such as [XSysAce_SectorRead\(\)](#) or [XSysAce_SectorWrite\(\)](#);

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

XTRUE if the CompactFlash is ready for a command, and XFALSE otherwise.

Note:

None.

```
XStatus ( XSysAce *  
XSysAce_Lock InstancePtr,  
Xboolean Force  
)
```

Attempt to lock access to the CompactFlash. The CompactFlash may be accessed by the MPU port as well as the JTAG configuration port within the System ACE device. This function requests exclusive access to the CompactFlash for the MPU port. This is a non-blocking request. If access cannot be locked (because the configuration controller has the lock), an appropriate status is returned. In this case, the user should call this function again until successful.

If the user requests a forced lock, the JTAG configuration controller will be put into a reset state in case it currently has a lock on the CompactFlash. This effectively aborts any operation the configuration controller had in progress and makes the configuration controller restart its process the next time it is able to get a lock.

A lock must be granted to the user before attempting to read or write the CompactFlash device.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Force is a boolean value that, when set to XTRUE, will force the MPU lock to occur in the System ACE. When set to XFALSE, the lock is requested and the device arbitrates between the MPU request and JTAG requests. Forcing the MPU lock resets the configuration controller, thus aborting any configuration operations in progress.

Returns:

XST_SUCCESS if the lock was granted, or XST_DEVICE_BUSY if the lock was not granted because the configuration controller currently has access to the CompactFlash.

Note:

If the lock is not granted to the MPU immediately, this function removes its request for a lock so that a lock is not later granted at a time when the application is (a) not ready for the lock, or (b) cannot be informed asynchronously about the granted lock since there is no such interrupt event.

XSysAce_Config* (**Xuint16**)
XSysAce_LookupConfig *DeviceId*

Looks up the device configuration based on the unique device ID. The table `XSysAce_ConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for.

Returns:

A pointer to the configuration data for the device, or `XNULL` if no match is found.

Note:

None.

```
XStatus ( XSysAce *  
XSysAce_ProgramChain InstancePtr,  
 Xuint8 * BufferPtr,  
 int NumBytes  
 )
```

Program the target FPGA chain through the configuration JTAG port. This allows the user to program the devices on the target FPGA chain from the MPU port instead of from CompactFlash. The user specifies a buffer and the number of bytes to write. The buffer should be equivalent to an ACE (.ace) file.

Note that when loading the ACE file via the MPU port, the first sector of the ACE file is discarded. The CF filesystem controller in the System ACE device knows to skip the first sector when the ACE file comes from the CF, but the CF filesystem controller is bypassed when the ACE file comes from the MPU port. For this reason, this function skips the first sector of the buffer passed in.

In polled mode, the write is blocking. In interrupt mode, the write is non-blocking and an event, `XSA_EVENT_CFG_DONE`, is returned to the user in the asynchronous event handler when the configuration is complete.

An MPU lock, obtained using `XSysAce_Lock()`, must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the `XSysAce` instance to be worked on.

BufferPtr is a pointer to a buffer that will be used to program the configuration JTAG devices.

NumBytes is the number of bytes in the buffer. We assume that there is at least one sector of data in the .ace file, which is the information sector.

Returns:

- XST_SUCCESS if the write was successful. In interrupt mode, this does not mean the write is complete, only that it has begun. An event is returned to the user when the write is complete.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_FAILURE if an error occurred during the write. The user should call [XSysAce_GetErrors\(\)](#) to determine the cause of the error.

Note:

None.

```
XStatus ( XSysAce * )  
XSysAce_ResetCF InstancePtr
```

Reset the CompactFlash device. This function does not reset the System ACE controller. An ATA soft-reset of the CompactFlash is performed.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

- XST_SUCCESS if the reset was done successfully
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the CompactFlash is not ready for a command

Note:

None.

```
void ( XSysAce * )  
XSysAce_ResetCf InstancePtr
```

Reset the JTAG configuration controller. This comprises a reset of the JTAG configuration controller and the CompactFlash controller (if it is currently being accessed by the configuration controller). Note that the MPU controller is not reset, meaning the MPU registers remain unchanged. The configuration controller is reset then released from reset in this function.

The CFGDONE status (and therefore interrupt) is cleared when the configuration controller is reset. If interrupts have been enabled, we go ahead and enable the CFGDONE interrupt here. This means that if and when a configuration process starts as a result of this reset, an interrupt will be received when it is complete.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

None.

Note:

This function is not thread-safe.

```
XStatus ( XSysAce *  
XSysAce_SectorRead InstancePtr,  
 Xuint32 StartSector,  
 int NumSectors,  
 Xuint8 * BufferPtr  
 )
```

Read at least one sector of data from the CompactFlash. The user specifies the starting sector ID and the number of sectors to be read. The minimum unit that can be read from the CompactFlash is a sector, which is 512 bytes.

In polled mode, this read is blocking. If there are other tasks in the system that must run, it is best to keep the number of sectors to be read to a minimum (e.g., 1). In interrupt mode, this read is non-blocking and an event, XSA_EVENT_DATA_DONE, is returned to the user in the asynchronous event handler when the read is complete. The user must call [XSysAce_EnableInterrupt\(\)](#) to put the driver/device into interrupt mode.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

StartSector is the starting sector ID from where data will be read. Sector IDs range from 0 (first sector) to 0x10000000.

NumSectors is the number of sectors to read. The range can be from 1 to 256.

BufferPtr is a pointer to a buffer where the data will be stored. The user must ensure it is big enough to hold (512 * NumSectors) bytes.

Returns:

- XST_SUCCESS if the read was successful. In interrupt mode, this does not mean the read is complete, only that it has begun. An event is returned to the user when the read is complete.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the ACE controller is not ready for a command
- XST_FAILURE if an error occurred during the read. The user should call **XSysAce_GetErrors()** to determine the cause of the error.

Note:

None.

```
XStatus ( XSysAce *  
XSysAce_SectorWrite InstancePtr,  
 Xuint32 StartSector,  
 int NumSectors,  
 Xuint8 * BufferPtr  
 )
```

Write data to the CompactFlash. The user specifies the starting sector ID and the number of sectors to be written. The minimum unit that can be written to the CompactFlash is a sector, which is 512 bytes.

In polled mode, this write is blocking. If there are other tasks in the system that must run, it is best to keep the number of sectors to be written to a minimum (e.g., 1). In interrupt mode, this write is non-blocking and an event, XSA_EVENT_DATA_DONE, is returned to the user in the asynchronous event handler when the write is complete. The user must call **XSysAce_EnableInterrupt()** to put the driver/device into interrupt mode.

An MPU lock, obtained using **XSysAce_Lock()**, must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

StartSector is the starting sector ID from where data will be written. Sector IDs range from 0 (first sector) to 0x10000000.

NumSectors is the number of sectors to write. The range can be from 1 to 256.

BufferPtr is a pointer to the data buffer to be written. This buffer must have at least (512 * NumSectors) bytes.

Returns:

- XST_SUCCESS if the write was successful. In interrupt mode, this does not mean the write is complete, only that it has begun. An event is returned to the user when the write is complete.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the ACE controller is not ready for a command
- XST_FAILURE if an error occurred during the write. The user should call **XSysAce_GetErrors()** to determine the cause of the error.

Note:

None.

```
XStatus ( XSysAce * )  
XSysAce_SelfTest InstancePtr
```

A self-test that simply proves communication to the ACE controller from the device driver by obtaining an MPU lock, verifying it, then releasing it.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

XST_SUCCESS if self-test passes, or XST_FAILURE if an error occurs.

Note:

None.

```
void ( XSysAce *  
XSysAce_SetCfgAddr InstancePtr,  
 unsigned int Address  
 )
```

Select the configuration address (or file) from the CompactFlash to be used for configuration of the target FPGA chain.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Address is the address or file number to be used as the bitstream to configure the target FPGA devices. There are 8 possible files, so the value of this parameter can range from 0 to 7.

Returns:

None.

Note:

None.

```
void XSysAce_SetEventHandler ( XSysAce * InstancePtr,
                             XSysAce_EventHandler FuncPtr,
                             void * CallbackRef
                             )
```

Set the callback function for handling events. The upper layer software should call this function during initialization. The events are passed asynchronously to the upper layer software. The events are described in [xsysace.h](#) and are named XSA_EVENT_.*.

Note that the callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is a reference pointer to be passed back to the upper layer.

Returns:

None.

Note:

None.

```

void
XSysAce_SetStartMode ( XSysAce * InstancePtr,
                       Xboolean ImmedOnReset,
                       Xboolean StartCfg
                       )

```

Set the start mode for configuration of the target FPGA chain from CompactFlash. The configuration process only starts after a reset. The user can indicate that the configuration should start immediately after a reset, or the configuration process can be delayed until the user commands it to start (using this function). The configuration controller can be reset using [XSysAce_ResetCfg\(\)](#).

The user can select which configuration file on the CompactFlash to use using the [XSysAce_SetCfgAddr\(\)](#) function. If the user intends to configure the target FPGA chain directly from the MPU port, this function is not needed. Instead, the user would simply call [XSysAce_ProgramChain\(\)](#).

The user can use [XSysAce_IsCfgDone\(\)](#) when in polled mode to determine if the configuration is complete. If in interrupt mode, the event XSA_EVENT_CFG_DONE will be returned asynchronously to the user when the configuration is complete. The user must call [XSysAce_EnableInterrupt\(\)](#) to put the device/driver into interrupt mode.

Parameters:

- InstancePtr* is a pointer to the [XSysAce](#) instance to be worked on.
- ImmedOnReset* can be set to XTRUE to indicate the configuration process will start immediately after a reset of the ACE configuration controller, or it can be set to XFALSE to indicate the configuration process is delayed after a reset until the user starts it (using this function).
- StartCfg* is a boolean indicating whether to start the configuration process or not. When ImmedOnReset is set to XTRUE, this value is ignored. When ImmedOnReset is set to XFALSE, then this value controls when the configuration process is started. When set to XTRUE the configuration process starts (assuming a reset of the device has occurred), and when set to XFALSE the configuration process does not start.

Returns:

None.

Note:

None.

```
void ( XSysAce * )  
XSysAce_Unlock InstancePtr
```

Release the MPU lock to the CompactFlash. If a lock is not currently granted to the MPU port, this function has no effect.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

None.

Note:

None.

sysace/v1_00_a/src/xsysace.c File Reference

Detailed Description

The Xilinx System ACE driver component. This driver supports the Xilinx System Advanced Configuration Environment (ACE) controller. It currently supports only the CompactFlash solution. See [xsysace.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	06/17/02	work in progress
1.00a	rmm	05/14/03	Fixed diab compiler warnings relating to asserts

```
#include "xparameters.h"  
#include "xsysace.h"  
#include "xsysace_1.h"
```

Functions

XStatus **XSysAce_Initialize** (**XSysAce** *InstancePtr, **Xuint16** DeviceId)

XStatus **XSysAce_Lock** (**XSysAce** *InstancePtr, **Xboolean** Force)

void **XSysAce_Unlock** (**XSysAce** *InstancePtr)

Xuint32 **XSysAce_GetErrors** (**XSysAce** *InstancePtr)

XSysAce_Config * **XSysAce_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
Xuint32 ( XSysAce * )  
XSysAce_GetErrors InstancePtr
```

Get all outstanding errors. Errors include the inability to read or write CompactFlash and the inability to successfully configure FPGA devices along the target FPGA chain.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

A 32-bit mask of error values. See [xsysace_l.h](#) for a description of possible values. The error identifiers are prefixed with XSA_ER_*.

Note:

None.

```
XStatus ( XSysAce *  
XSysAce_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize a specific **XSysAce** instance. The configuration information for the given device ID is found and the driver instance data is initialized appropriately.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XSysAce** instance.

Returns:

XST_SUCCESS if successful, or XST_DEVICE_NOT_FOUND if the device was not found in the configuration table in [xsysace_g.c](#).

Note:

We do not want to reset the configuration controller here since this could cause a reconfiguration of the JTAG target chain, depending on how the CFGMODEPIN of the device is wired.

```
XStatus ( XSysAce *  
XSysAce_Lock InstancePtr,  
 Xboolean Force  
 )
```

Attempt to lock access to the CompactFlash. The CompactFlash may be accessed by the MPU port as well as the JTAG configuration port within the System ACE device. This function requests exclusive access to the CompactFlash for the MPU port. This is a non-blocking request. If access cannot be locked (because the configuration controller has the lock), an appropriate status is returned. In this case, the user should call this function again until successful.

If the user requests a forced lock, the JTAG configuration controller will be put into a reset state in case it currently has a lock on the CompactFlash. This effectively aborts any operation the configuration controller had in progress and makes the configuration controller restart its process the next time it is able to get a lock.

A lock must be granted to the user before attempting to read or write the CompactFlash device.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Force is a boolean value that, when set to XTRUE, will force the MPU lock to occur in the System ACE. When set to XFALSE, the lock is requested and the device arbitrates between the MPU request and JTAG requests. Forcing the MPU lock resets the configuration controller, thus aborting any configuration operations in progress.

Returns:

XST_SUCCESS if the lock was granted, or XST_DEVICE_BUSY if the lock was not granted because the configuration controller currently has access to the CompactFlash.

Note:

If the lock is not granted to the MPU immediately, this function removes its request for a lock so that a lock is not later granted at a time when the application is (a) not ready for the lock, or (b) cannot be informed asynchronously about the granted lock since there is no such interrupt event.

```
XSysAce_Config* ( Xuint16 )  
XSysAce_LookupConfig DeviceId
```

Looks up the device configuration based on the unique device ID. The table `XSysAce_ConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for.

Returns:

A pointer to the configuration data for the device, or `XNULL` if no match is found.

Note:

None.

```
void ( XSysAce * )  
XSysAce_Unlock InstancePtr
```

Release the MPU lock to the CompactFlash. If a lock is not currently granted to the MPU port, this function has no effect.

Parameters:

InstancePtr is a pointer to the `XSysAce` instance to be worked on.

Returns:

None.

Note:

None.

XSysAce_Config Struct Reference

```
#include <xsysace.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

Field Documentation

[Xuint32 XSysAce_Config::](#)

BaseAddress

Register base
address

[Xuint16 XSysAce_Config::](#)

DeviceId

Unique ID of
device

The documentation for this struct was generated from the following file:

- `sysace/v1_00_a/src/xsysace.h`

sysace/v1_00_a/src/xsysace_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of System ACE devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	06/17/02	work in progress

```
#include "xsysace.h"  
#include "xparameters.h"
```

Variables

[XSysAce_Config](#) [XSysAce_ConfigTable](#) [XPAR_XSYSACE_NUM_INSTANCES]

Variable Documentation

[XSysAce_Config](#) [XSysAce_ConfigTable](#)
[XPAR_XSYSACE_NUM_INSTANCES]

The configuration table for System ACE devices in the system. Each device should have an entry in this table.

XSysAce_CFPParameters Struct Reference

```
#include <xsysace.h>
```

Detailed Description

Typedef for CompactFlash identify drive parameters. Use [XSysAce_IdentifyCF\(\)](#) to retrieve this information from the CompactFlash storage device.

Data Fields

[Xuint16 Signature](#)

[Xuint16 NumCylinders](#)

[Xuint16 NumHeads](#)

[Xuint16 NumBytesPerTrack](#)

[Xuint16 NumBytesPerSector](#)

[Xuint16 NumSectorsPerTrack](#)

[Xuint32 NumSectorsPerCard](#)

[Xuint16 VendorUnique](#)

[Xuint8 SerialNo](#) [20]

[Xuint16 BufferType](#)

[Xuint16 BufferSize](#)

[Xuint16 NumEccBytes](#)

[Xuint8 FwVersion](#) [8]

[Xuint8 ModelNo](#) [40]

[Xuint16 MaxSectors](#)

[Xuint16 DblWord](#)

[Xuint16 Capabilities](#)

[Xuint16 PioMode](#)

[Xuint16 DmaMode](#)

Xuint16 TranslationValid
Xuint16 CurNumCylinders
Xuint16 CurNumHeads
Xuint16 CurSectorsPerTrack
Xuint32 CurSectorsPerCard
Xuint16 MultipleSectors
Xuint32 LbaSectors
Xuint16 SecurityStatus
 Xuint8 VendorUniqueBytes [62]
Xuint16 PowerDesc

Field Documentation

Xuint16 XSysAce_CFPParameters::

BufferSize

Buffer size in 512-byte
increments

Xuint16 XSysAce_CFPParameters::

BufferType

Buffer
type

Xuint16 XSysAce_CFPParameters::

Capabilities

Device
capabilities

Xuint16 XSysAce_CFPParameters::

CurNumCylinders

Current number of
cylinders

Xuint16 XSysAce_CFPParameters::

CurNumHeads

Current number of heads

Xuint32 XSysAce_CFPParameters::CurSectorsPerCard

Current capacity in sectors

Xuint16 XSysAce_CFPParameters::CurSectorsPerTrack

Current number of sectors per track

Xuint16 XSysAce_CFPParameters::DbfWord

Double Word not supported

Xuint16 XSysAce_CFPParameters::DmaMode

DMA data transfer cycle timing mode

Xuint8 XSysAce_CFPParameters::FwVersion [8]

ASCII firmware version

Xuint32 XSysAce_CFPParameters::LbaSectors

Number of addressable sectors in LBA mode

Xuint16 XSysAce_CFPParameters::MaxSectors

Max sectors on R/W Multiple
cmds

Xuint8 XSysAce_CFPParameters::ModelNo
[40]

ASCII model
number

Xuint16 XSysAce_CFPParameters::
MultipleSectors

Multiple sector
setting

Xuint16 XSysAce_CFPParameters::
NumBytesPerSector

Number of unformatted bytes per
sector

Xuint16 XSysAce_CFPParameters::
NumBytesPerTrack

Number of unformatted bytes per
track

Xuint16 XSysAce_CFPParameters::
NumCylinders

Default number of
cylinders

Xuint16 XSysAce_CFPParameters::
NumEccBytes

Number of ECC bytes on R/W Long
cmds

Xuint16 XSysAce_CFPParameters::
NumHeads

Default number of heads

Xuint32 XSysAce_CFPParameters::NumSectorsPerCard

Default number of sectors per card

Xuint16 XSysAce_CFPParameters::NumSectorsPerTrack

Default number of sectors per track

Xuint16 XSysAce_CFPParameters::PioMode

PIO data transfer cycle timing mode

Xuint16 XSysAce_CFPParameters::PowerDesc

Power requirement description

Xuint16 XSysAce_CFPParameters::SecurityStatus

Security status

Xuint8 XSysAce_CFPParameters::SerialNo [20]

ASCII serial number

Xuint16 XSysAce_CFPParameters::Signature

CompactFlash signature is
0x848a

Xuint16 XSysAce_CFPParameters::

TranslationValid

Translation parameters are
valid

Xuint16 XSysAce_CFPParameters::

VendorUnique

Vendor
unique

Xuint8 XSysAce_CFPParameters::VendorUniqueBytes

[62]

Vendor unique
bytes

The documentation for this struct was generated from the following file:

- [sysace/v1_00_a/src/xsysace.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

spi/v1_00_b/src/xspi.h File Reference

Detailed Description

This component contains the implementation of the **XSpi** component. It is the driver for an SPI master or slave device. User documentation for the driver functions is contained in this file in the form of comment blocks at the front of each function.

SPI is a 4-wire serial interface. It is a full-duplex, synchronous bus that facilitates communication between one master and one slave. The device is always full-duplex, which means that for every byte sent, one is received, and vice-versa. The master controls the clock, so it can regulate when it wants to send or receive data. The slave is under control of the master, it must respond quickly since it has no control of the clock and must send/receive data as fast or as slow as the master does.

The application software between master and slave must implement a higher layer protocol so that slaves know what to transmit to the master and when.

Multiple Masters

More than one master can exist, but arbitration is the responsibility of the higher layer software. The device driver does not perform any type of arbitration.

Multiple Slaves

Multiple slaves are supported by adding additional slave select (SS) signals to each device, one for each slave on the bus. The driver ensures that only one slave can be selected at any one time.

FIFOs

The SPI hardware is parameterized such that it can be built with or without FIFOs. When using FIFOs, both send and receive must have FIFOs. The driver will not function correctly if one direction has a FIFO but the other direction does not. The frequency of the interrupts which occur is proportional to the data rate such that high data rates without the FIFOs could cause the software to consume large amounts of processing time. The driver is designed to work with or without the FIFOs.

Interrupts

The user must connect the interrupt handler of the driver, `XSpi_InterruptHandler` to an interrupt system such that it will be called when an interrupt occurs. This function does not save and restore the processor context such that the user must provide this processing.

The driver handles the following interrupts:

- Data Transmit Register/FIFO Empty
- Data Transmit Register/FIFO Underrun
- Data Receive Register/FIFO Overrun
- Mode Fault Error
- Slave Mode Fault Error

The Data Transmit Register/FIFO Empty interrupt indicates that the SPI device has transmitted all the data available to transmit, and now its data register (or FIFO) is empty. The driver uses this interrupt to indicate progress while sending data. The driver may have more data to send, in which case the data transmit register (or FIFO) is filled for subsequent transmission. When this interrupt arrives and all the data has been sent, the driver invokes the status callback with a value of `XST_SPI_TRANSFER_DONE` to inform the upper layer software that all data has been sent.

The Data Transmit Register/FIFO Underrun interrupt indicates that, as slave, the SPI device was required to transmit but there was no data available to transmit in the transmit register (or FIFO). This may not be an error if the master is not expecting data, but in the case where the master is expecting data this serves as a notification of such a condition. The driver reports this condition to the upper layer software through the status handler.

The Data Receive Register/FIFO Overrun interrupt indicates that the SPI device received data and subsequently dropped the data because the data receive register (or FIFO) was full. The interrupt applies to both master and slave operation. The driver reports this condition to the upper layer software through the status handler. This likely indicates a problem with the higher layer protocol, or a problem with the slave performance.

The Mode Fault Error interrupt indicates that while configured as a master, the device was selected as a slave by another master. This can be used by the application for arbitration in a multimaster environment or to indicate a problem with arbitration. When this interrupt occurs, the driver invokes the status callback with a status value of `XST_SPI_MODE_FAULT`. It is up to the application to resolve the conflict.

The Slave Mode Fault Error interrupt indicates that a slave device was selected as a slave by a master, but the slave device was disabled. This can be used during system debugging or by the slave application to learn when the slave application has not prepared for a master operation in a timely fashion. This likely indicates a problem with the higher layer protocol, or a problem with the slave performance.

Note that during the FPGA implementation process, the interrupt registers of the IPIF can be parameterized away. This driver is currently dependent on those interrupt registers and will not function without them.

Polled Operation

Currently there is no support for polled operation.

Device Busy

Some operations are disallowed when the device is busy. The driver tracks whether a device is busy. The device is considered busy when a data transfer request is outstanding, and is considered not busy only when that transfer completes (or is aborted with a mode fault error). This applies to both master and slave devices.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are

stored in the `xspi_g.c` file. A table is defined where each entry contains configuration information for an SPI device. This information includes such things as the base address of the memory-mapped device, the base address of the IPIF module within the device, the number of slave select bits in the device, and whether the device has FIFOs and is configured as slave-only.

RTOS Independence

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	10/11/01	First release
1.00b	jhl	03/14/02	Repartitioned driver for smaller files.

```
#include "xbasic_types.h"
#include "xstatus.h"
```

Data Structures

```
struct XSpi
struct XSpi_Config
struct XSpi_Stats
```

Configuration options

The following options may be specified or retrieved for the device and enable/disable additional features of the SPI. Each of the options are bit fields, so more than one may be specified.

```
#define XSP_MASTER_OPTION
#define XSP_CLK_ACTIVE_LOW_OPTION
#define XSP_CLK_PHASE_1_OPTION
#define XSP_LOOPBACK_OPTION
#define XSP_MANUAL_SSELECT_OPTION
```

Typedefs

```
typedef void(* XSpi_StatusHandler )(void *CallBackRef, Xuint32 StatusEvent, unsigned int ByteCount)
```

Functions

```
XStatus XSpi_Initialize (XSpi *InstancePtr, Xuint16 DeviceId)
```

```

XStatus XSpi_Start (XSpi *InstancePtr)
XStatus XSpi_Stop (XSpi *InstancePtr)
    void XSpi_Reset (XSpi *InstancePtr)
XStatus XSpi_SetSlaveSelect (XSpi *InstancePtr, Xuint32 SlaveMask)
Xuint32 XSpi_GetSlaveSelect (XSpi *InstancePtr)
XStatus XSpi_Transfer (XSpi *InstancePtr, Xuint8 *SendBufPtr, Xuint8 *RecvBufPtr, unsigned int ByteCount)
    void XSpi_SetStatusHandler (XSpi *InstancePtr, void *CallbackRef, XSpi_StatusHandler FuncPtr)
    void XSpi_InterruptHandler (void *InstancePtr)
XSpi_Config * XSpi_LookupConfig (Xuint16 DeviceId)
XStatus XSpi_SelfTest (XSpi *InstancePtr)
    void XSpi_GetStats (XSpi *InstancePtr, XSpi_Stats *StatsPtr)
    void XSpi_ClearStats (XSpi *InstancePtr)
XStatus XSpi_SetOptions (XSpi *InstancePtr, Xuint32 Options)
Xuint32 XSpi_GetOptions (XSpi *InstancePtr)

```

Define Documentation

```
#define
```

```
XSP_CLK_ACTIVE_LOW_OPTION
```

The Master option configures the SPI device as a master. By default, the device is a slave.

The Active Low Clock option configures the device's clock polarity. Setting this option means the clock is active low and the SCK signal idles high. By default, the clock is active high and SCK idles low.

The Clock Phase option configures the SPI device for one of two transfer formats. A clock phase of 0, the default, means data is valid on the first SCK edge (rising or falling) after the slave select (SS) signal has been asserted. A clock phase of 1 means data is valid on the second SCK edge (rising or falling) after SS has been asserted.

The Loopback option configures the SPI device for loopback mode. Data is looped back from the transmitter to the receiver.

The Manual Slave Select option, which is default, causes the device not to automatically drive the slave select. The driver selects the device at the start of a transfer and deselects it at the end of a transfer. If this option is off, then the device automatically toggles the slave select signal between bytes in a transfer.

```
#define
```

```
XSP_CLK_PHASE_1_OPTION
```

The Master option configures the SPI device as a master. By default, the device is a slave.

The Active Low Clock option configures the device's clock polarity. Setting this option means the clock is active low and the SCK signal idles high. By default, the clock is active high and SCK idles low.

The Clock Phase option configures the SPI device for one of two transfer formats. A clock phase of 0, the default, means data is valid on the first SCK edge (rising or falling) after the slave select (SS) signal has been asserted. A clock phase of 1 means data is valid on the second SCK edge (rising or falling) after SS has been asserted.

The Loopback option configures the SPI device for loopback mode. Data is looped back from the transmitter to the receiver.

The Manual Slave Select option, which is default, causes the device not to automatically drive the slave select. The driver selects the device at the start of a transfer and deselects it at the end of a transfer. If this option is off, then the device automatically toggles the slave select signal between bytes in a transfer.

#define XSP_LOOPBACK_OPTION

The Master option configures the SPI device as a master. By default, the device is a slave.

The Active Low Clock option configures the device's clock polarity. Setting this option means the clock is active low and the SCK signal idles high. By default, the clock is active high and SCK idles low.

The Clock Phase option configures the SPI device for one of two transfer formats. A clock phase of 0, the default, means data is valid on the first SCK edge (rising or falling) after the slave select (SS) signal has been asserted. A clock phase of 1 means data is valid on the second SCK edge (rising or falling) after SS has been asserted.

The Loopback option configures the SPI device for loopback mode. Data is looped back from the transmitter to the receiver.

The Manual Slave Select option, which is default, causes the device not to automatically drive the slave select. The driver selects the device at the start of a transfer and deselects it at the end of a transfer. If this option is off, then the device automatically toggles the slave select signal between bytes in a transfer.

#define

XSP_MANUAL_SSELECT_OPTION

The Master option configures the SPI device as a master. By default, the device is a slave.

The Active Low Clock option configures the device's clock polarity. Setting this option means the clock is active low and the SCK signal idles high. By default, the clock is active high and SCK idles low.

The Clock Phase option configures the SPI device for one of two transfer formats. A clock phase of 0, the default, means data is valid on the first SCK edge (rising or falling) after the slave select (SS) signal has been asserted. A clock phase of 1 means data is valid on the second SCK edge (rising or falling) after SS has been asserted.

The Loopback option configures the SPI device for loopback mode. Data is looped back from the transmitter to the receiver.

The Manual Slave Select option, which is default, causes the device not to automatically drive the slave select. The driver selects the device at the start of a transfer and deselects it at the end of a transfer. If this option is off, then the device automatically toggles the slave select signal between bytes in a transfer.

#define

XSP_MASTER_OPTION

The Master option configures the SPI device as a master. By default, the device is a slave.

The Active Low Clock option configures the device's clock polarity. Setting this option means the clock is active low and the SCK signal idles high. By default, the clock is active high and SCK idles low.

The Clock Phase option configures the SPI device for one of two transfer formats. A clock phase of 0, the default, means data is valid on the first SCK edge (rising or falling) after the slave select (SS) signal has been asserted. A clock phase of 1 means data is valid on the second SCK edge (rising or falling) after SS has been asserted.

The Loopback option configures the SPI device for loopback mode. Data is looped back from the transmitter to the receiver.

The Manual Slave Select option, which is default, causes the device not to automatically drive the slave select. The driver selects the device at the start of a transfer and deselects it at the end of a transfer. If this option is off, then the device automatically toggles the slave select signal between bytes in a transfer.

Typedef Documentation

```
typedef void(* XSpi_StatusHandler)(void *CallbackRef, Xuint32 StatusEvent, unsigned int  
ByteCount)
```

The handler data type allows the user to define a callback function to handle the asynchronous processing of the SPI driver. The application using this driver is expected to define a handler of this type to support interrupt driven mode. The handler executes in an interrupt context such that minimal processing should be performed.

Parameters:

- CallbackRef* A callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked. Its type is unimportant to the driver component, so it is a void pointer.
- StatusEvent* Indicates one or more status events that occurred. See the [XSpi_SetStatusHandler\(\)](#) for details on the status events that can be passed in the callback.
- ByteCount* Indicates how many bytes of data were successfully transferred. This may be less than the number of bytes requested if the status event indicates an error.

Function Documentation

```
void  
XSpi_ClearStats ( XSpi \*  
InstancePtr )
```

Clears the statistics for the SPI device.

Parameters:

InstancePtr is a pointer to the [XSpi](#) instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint32  
XSpi_GetOptions ( XSpi \*  
InstancePtr )
```

This function gets the options for the SPI device. The options control how the device behaves relative to the SPI bus.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

Options contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. One or more bit values may be contained in the mask. See the bit definitions named XSP_*_OPTIONS in the file **xspi.h**.

Note:

None.

```
Xuint32 ( XSpi * )  
XSpi_GetSlaveSelect InstancePtr
```

Gets the current slave select bit mask for the SPI device.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

The value returned is a 32-bit mask with a 1 in the bit position of the slave currently selected. The value may be zero if no slaves are selected.

Note:

None.

```
void ( XSpi *  
XSpi_GetStats InstancePtr,  
 XSpi_Stats * StatsPtr  
 )
```

Gets a copy of the statistics for an SPI device.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

StatsPtr is a pointer to a **XSpi_Stats** structure which will get a copy of current statistics.

Returns:

None.

Note:

None.

```
XStatus          ( XSpi *  
XSpi_Initialize      InstancePtr,  
                    Xuint16 DeviceId  
                    )
```

Initializes a specific **XSpi** instance such that the driver is ready to use.

The state of the device after initialization is:

- Device is disabled
- Slave mode
- Active high clock polarity
- Clock phase 0

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XSpi** instance. Passing in a device id associates the generic **XSpi** instance to a specific device, as chosen by the caller or application developer.

Returns:

The return value is `XST_SUCCESS` if successful. On error, a code indicating the specific error is returned.

Possible error codes are:

- `XST_DEVICE_IS_STARTED` if the device is started. It must be stopped to re-initialize.
- `XST_DEVICE_NOT_FOUND` if the device was not found in the configuration such that initialization could not be accomplished.

Note:

None.

```
void              ( void *  
XSpi_InterruptHandler      InstancePtr  
                          )
```

The interrupt handler for SPI interrupts. This function must be connected by the user to an interrupt source. This function does not save and restore the processor context such that the user must provide this processing.

The interrupts that are handled are:

- Mode Fault Error. This interrupt is generated if this device is selected as a slave when it is configured as a master. The driver aborts any data transfer that is in progress by resetting FIFOs (if present) and resetting its buffer pointers. The upper layer software is informed of the error.
- Data Transmit Register (FIFO) Empty. This interrupt is generated when the transmit register or FIFO is empty. The driver uses this interrupt during a transmission to continually send/receive data until there is no more data to send/receive.
- Data Transmit Register (FIFO) Underrun. This interrupt is generated when the SPI device, when configured as a slave, attempts to read an empty DTR/FIFO. An empty DTR/FIFO usually means that software is not giving the device data in a timely manner. No action is taken by the driver other than to inform the upper layer software of the error.

- Data Receive Register (FIFO) Overrun. This interrupt is generated when the SPI device attempts to write a received byte to an already full DRR/FIFO. A full DRR/FIFO usually means software is not emptying the data in a timely manner. No action is taken by the driver other than to inform the upper layer software of the error.
- Slave Mode Fault Error. This interrupt is generated if a slave device is selected as a slave while it is disabled. No action is taken by the driver other than to inform the upper layer software of the error.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

None.

Note:

The slave select register is being set to deselect the slave when a transfer is complete. This is being done regardless of whether it is a slave or a master since the hardware does not drive the slave select as a slave.

XSpi_Config* (**Xuint16**)
XSpi_LookupConfig *DeviceId*

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId contains the ID of the device to look up the configuration for.

Returns:

A pointer to the configuration found or XNULL if the specified device ID was not found. See **xspi.h** for the definition of **XSpi_Config**.

Note:

None.

void (**XSpi ***)
XSpi_Reset *InstancePtr*

Resets the SPI device. Reset must only be called after the driver has been initialized. The configuration of the device after reset is the same as its configuration after initialization. Refer to the `XSpi_Initialize` function for more details. This is a hard reset of the device. Any data transfer that is in progress is aborted.

The upper layer software is responsible for re-configuring (if necessary) and restarting the SPI device after the reset.

Parameters:

InstancePtr is a pointer to the `XSpi` instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( XSpi *  
XSpi_SelfTest InstancePtr
```

Runs a self-test on the driver/device. The self-test is destructive in that a reset of the device is performed in order to check the reset values of the registers and to get the device into a known state. A simple loopback test is also performed to verify that transmit and receive are working properly. The device is changed to master mode for the loopback test, since only a master can initiate a data transfer.

Upon successful return from the self-test, the device is reset.

Parameters:

InstancePtr is a pointer to the `XSpi` instance to be worked on.

Returns:

- XST_SUCCESS if successful, or one of the following error codes otherwise.
 - XST_REGISTER_ERROR indicates a register did not read or write correctly
 - XST_LOOPBACK_ERROR if a loopback error occurred.

Note:

None.

```
XStatus ( XSpi *  
XSpi_SetOptions InstancePtr,  
 Xuint32 Options  
 )
```

This function sets the options for the SPI device driver. The options control how the device behaves relative to the SPI bus. The device must be idle rather than busy transferring data before setting these device options.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Options contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. One or more bit values may be contained in the mask. See the bit definitions named XSP_*_OPTIONS in the file **xspi.h**.

Returns:

XST_SUCCESS if options are successfully set. Otherwise, returns:

- XST_DEVICE_BUSY if the device is currently transferring data. The transfer must complete or be aborted before setting options.
- XST_SPI_SLAVE_ONLY if the caller attempted to configure a slave-only device as a master.

Note:

This function makes use of internal resources that are shared between the **XSpi_Stop()** and **XSpi_SetOptions()** functions. So if one task might be setting device options while another is trying to stop the device, the user is required to provide protection of this shared data (typically using a semaphore).

```
XStatus XSpi_SetSlaveSelect(XSpi *  
InstancePtr,  
Xuint32 SlaveMask  
)
```

Selects or deselect the slave with which the master communicates. Each slave that can be selected is represented in the slave select register by a bit. The argument passed to this function is the bit mask with a 1 in the bit position of the slave being selected. Only one slave can be selected.

The user is not allowed to deselect the slave while a transfer is in progress. If no transfer is in progress, the user can select a new slave, which implicitly deselects the current slave. In order to explicitly deselect the current slave, a zero can be passed in as the argument to the function.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

SlaveMask is a 32-bit mask with a 1 in the bit position of the slave being selected. Only one slave can be selected. The SlaveMask can be zero if the slave is being deselected.

Returns:

XST_SUCCESS if the slave is selected or deselected successfully. Otherwise, returns:

- XST_DEVICE_BUSY if a transfer is in progress, slave cannot be changed
- XST_SPI_TOO_MANY_SLAVES if more than one slave is being selected.

Note:

This function only sets the slave which will be selected when a transfer occurs. The slave is not selected when the SPI is idle. The slave select has no affect when the device is configured as a slave.

```

void
XSpi_SetStatusHandler
    ( XSpi *
      InstancePtr,
      void *
      CallbackRef,
      XSpi_StatusHandler FuncPtr
    )

```

Sets the status callback function, the status handler, which the driver calls when it encounters conditions that should be reported to the higher layer software. The handler executes in an interrupt context, so it must minimize the amount of processing performed such as transferring data to a thread context. One of the following status events is passed to the status handler.

XST_SPI_MODE_FAULT	A mode fault error occurred, meaning another master tried to select this device as a slave when this device was configured to be a master. Any transfer in progress is aborted.
XST_SPI_TRANSFER_DONE	The requested data transfer is done
XST_SPI_TRANSMIT_UNDERRUN	As a slave device, the master clocked data but there were none available in the transmit register/FIFO. This typically means the slave application did not issue a transfer request fast enough, or the processor/driver could not fill the transmit register/FIFO fast enough.
XST_SPI_RECEIVE_OVERRUN	The SPI device lost data. Data was received but the receive data register/FIFO was full. This indicates that the device is receiving data
XST_SPI_SLAVE_MODE_FAULT	faster than the processor/driver can consume it. A slave SPI device was selected as a slave while it was disabled. This indicates the master is already transferring data (which is being dropped until the slave application issues a transfer).

Parameters:

- InstancePtr* is a pointer to the **XSpi** instance to be worked on.
- CallbackRef* is the upper layer callback reference passed back when the callback function is invoked.
- FuncPtr* is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context, so it should do its work quickly and queue potentially time-consuming work to a task-level thread.

XStatus (**XSpi ***)
XSpi_Start *InstancePtr*

This function enables interrupts for the SPI device. It is up to the user to connect the SPI interrupt handler to the interrupt controller before this Start function is called. The GetIntrHandler function is used for that purpose. If the device is configured with FIFOs, the FIFOs are reset at this time.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

XST_SUCCESS if the device is successfully started, or XST_DEVICE_IS_STARTED if the device was already started.

Note:

None.

XStatus (**XSpi ***)
XSpi_Stop *InstancePtr*

This function stops the SPI device by disabling interrupts and disabling the device itself. Interrupts are disabled only within the device itself. If desired, the caller is responsible for disabling interrupts in the interrupt controller and disconnecting the interrupt handler from the interrupt controller.

If the device is in progress of transferring data on the SPI bus, this function returns a status indicating the device is busy. The user will be notified via the status handler when the transfer is complete, and at that time can again try to stop the device. As a master, we do not allow the device to be stopped while a transfer is in progress because the slave may be left in a bad state. As a slave, we do not allow the device to be stopped while a transfer is in progress because the master is not done with its transfer yet.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

XST_SUCCESS if the device is successfully started, or XST_DEVICE_BUSY if a transfer is in progress and cannot be stopped.

Note:

This function makes use of internal resources that are shared between the **XSpi_Stop()** and **XSpi_SetOptions()** functions. So if one task might be setting device options while another is trying to stop the device, the user is required to provide protection of this shared data (typically using a semaphore).

```

XStatus          ( XSpi *
XSpi_Transfer    InstancePtr,
                   Xuint8 * SendBufPtr,
                   Xuint8 * RecvBufPtr,
                   unsigned int ByteCount
                   )

```

Transfers the specified data on the SPI bus. If the SPI device is configured to be a master, this function initiates bus communication and sends/receives the data to/from the selected SPI slave. If the SPI device is configured to be a slave, this function prepares the data to be sent/received when selected by a master. For every byte sent, a byte is received.

The caller has the option of providing two different buffers for send and receive, or one buffer for both send and receive, or no buffer for receive. The receive buffer must be at least as big as the send buffer to prevent unwanted memory writes. This implies that the byte count passed in as an argument must be the smaller of the two buffers if they differ in size. Here are some sample usages:

```
XSpi_Transfer(InstancePtr, SendBuf, RecvBuf, ByteCount)
```

The caller wishes to send and receive, and provides two different buffers for send and receive.

```
XSpi_Transfer(InstancePtr, SendBuf, NULL, ByteCount)
```

The caller wishes only to send and does not care about the received data. The driver ignores the received data in this case.

```
XSpi_Transfer(InstancePtr, SendBuf, SendBuf, ByteCount)
```

The caller wishes to send and receive, but provides the same buffer for doing both. The driver sends the data and overwrites the send buffer with received data as it transfers the data.

```
XSpi_Transfer(InstancePtr, RecvBuf, RecvBuf, ByteCount)
```

The caller wishes to only receive and does not care about sending data. In this case, the caller must still provide a send buffer, but it can be the same as the receive buffer if the caller does not care what it sends. The device must send N bytes of data if it wishes to receive N bytes of data.

Although this function takes a buffer as an argument, the driver can only transfer a limited number of bytes at time. It transfers only one byte at a time if there are no FIFOs, or it can transfer the number of bytes up to the size of the FIFO. A call to this function only starts the transfer, then subsequent transfer of the data is performed by the interrupt service routine until the entire buffer has been transferred. The status callback function is called when the entire buffer has been sent/received.

This function is non-blocking. As a master, the SetSlaveSelect function must be called prior to this function.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

SendBufPtr is a pointer to a buffer of data which is to be sent. This buffer must not be NULL.

RecvBufPtr is a pointer to a buffer which will be filled with received data. This argument can be NULL if the caller does not wish to receive data.

ByteCount contains the number of bytes to send/receive. The number of bytes received always equals the number of bytes sent.

Returns:

- XST_SUCCESS if the buffers are successfully handed off to the driver for transfer. Otherwise, returns:
- XST_DEVICE_IS_STOPPED if the device must be started before transferring data.
 - XST_DEVICE_BUSY indicates that a data transfer is already in progress. This is determined by the driver.
 - XST_SPI_NO_SLAVE indicates the device is configured as a master and a slave has not yet been selected.

Note:

This function is not thread-safe. The higher layer software must ensure that no two threads are transferring data on the SPI bus at the same time.

XSpi Struct Reference

```
#include <xspi.h>
```

Detailed Description

The XSpi driver instance data. The user is required to allocate a variable of this type for every SPI device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [spi/v1_00_b/src/xspi.h](#)
-

spi/v1_00_b/src/xspi_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of SPI devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	10/11/01	First release
1.00b	jhl	03/14/02	Repartitioned driver for smaller files.
1.00b	rpm	04/24/02	Condensed config typedef - got rid of versions and multiple base addresses.

```
#include "xspi.h"  
#include "xparameters.h"
```

Variables

[XSpi_Config](#) [XSpi_ConfigTable](#) [XPAR_XSPI_NUM_INSTANCES]

Variable Documentation

[XSpi_Config](#) [XSpi_ConfigTable](#)
[XPAR_XSPI_NUM_INSTANCES]

This table contains configuration information for each SPI device in the system.

XSpi_Config Struct Reference

```
#include <xspi.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

Xuint16 DeviceId

Xuint32 BaseAddress

Xboolean HasFifos

Xboolean SlaveOnly

Xuint8 NumSlaveBits

Field Documentation

Xuint32 XSpi_Config::

BaseAddress

Base address of the device

Xuint16 XSpi_Config::

DeviceId

Unique ID of device

Xboolean XSpi_Config::

HasFifos

Does device have
FIFOs?

Xuint8 XSpi_Config::

NumSlaveBits

Number of slave select bits on the
device

Xboolean XSpi_Config::

SlaveOnly

Is the device slave
only?

The documentation for this struct was generated from the following file:

- [spi/v1_00_b/src/xspi.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

XSpi_Stats Struct Reference

```
#include <xspi.h>
```

Detailed Description

XSpi statistics

Data Fields

Xuint32 ModeFaults

Xuint32 XmitUnderruns

Xuint32 RecvOverruns

Xuint32 SlaveModeFaults

Xuint32 BytesTransferred

Xuint32 NumInterrupts

Field Documentation

Xuint32 XSpi_Stats::

BytesTransferred

Number of bytes transferred

Xuint32 XSpi_Stats::

ModeFaults

Number of mode fault errors

Xuint32 XSpi_Stats::**NumInterrupts**

Number of transmit/receive interrupts

Xuint32 XSpi_Stats::**RecvOverruns**

Number of receive overruns

Xuint32 XSpi_Stats::**SlaveModeFaults**

Number of selects as a slave while disabled

Xuint32 XSpi_Stats::**XmitUnderruns**

Number of transmit underruns

The documentation for this struct was generated from the following file:

- [spi/v1_00_b/src/xspi.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

spi/v1_00_b/src/xspi_i.h File Reference

Detailed Description

This header file contains internal identifiers. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	10/11/01	First release
1.00b	jhl	03/14/02	Repartitioned driver for smaller files.
1.00b	rpm	04/24/02	Moved register definitions to xspi_1.h

```
#include "xbasic_types.h"  
#include "xspi_1.h"
```

Functions

```
void XSpi_Abort (XSpi *InstancePtr)
```

Variables

```
XSpi_Config XSpi_ConfigTable []
```

Function Documentation

```
void XSpi_Abort ( XSpi * InstancePtr )
```

Aborts a transfer in progress by setting the stop bit in the control register, then resetting the FIFOs if present. The byte counts are cleared and the busy flag is set to false.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

None.

Note:

This function does a read/modify/write of the control register. The user of this function needs to take care of critical sections.

Variable Documentation

XSpi_Config XSpi_ConfigTable ()

[]

This table contains configuration information for each SPI device in the system.

spi/v1_00_b/src/xspi_1.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xspi.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	rpm	04/24/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

Defines

```
#define XSpi_mSetControlReg(BaseAddress, Mask)  
#define XSpi_mGetControlReg(BaseAddress)  
#define XSpi_mGetStatusReg(BaseAddress)  
#define XSpi_mSetSlaveSelectReg(BaseAddress, Mask)  
#define XSpi_mGetSlaveSelectReg(BaseAddress)  
#define XSpi_mEnable(BaseAddress)  
#define XSpi_mDisable(BaseAddress)  
#define XSpi_mSendByte(BaseAddress, Data)  
#define XSpi_mRecvByte(BaseAddress)
```

Define Documentation

#define XSpi_mDisable(BaseAddress)

Disable the device. Preserves the current contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

#define XSpi_mEnable(BaseAddress)

Enable the device and uninhibit master transactions. Preserves the current contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

#define XSpi_mGetControlReg(BaseAddress)

Get the contents of the control register. Use the XSP_CR_* constants defined above to interpret the bit-mask returned.

Parameters:

BaseAddress is the base address of the device

Returns:

A 16-bit value representing the contents of the control register.

Note:

None.

```
#define XSpi_mGetSlaveSelectReg( BaseAddress )
```

Get the contents of the slave select register. Each bit in the mask corresponds to a slave select line. Only one slave should be selected at any one time.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value in the slave select register

Note:

None.

```
#define XSpi_mGetStatusReg( BaseAddress )
```

Get the contents of the status register. Use the XSP_SR_* constants defined above to interpret the bit-mask returned.

Parameters:

BaseAddress is the base address of the device

Returns:

An 8-bit value representing the contents of the status register.

Note:

None.

```
#define XSpi_mRecvByte( BaseAddress )
```

Receive one byte from the device's receive FIFO/register. It is assumed that the byte is already available.

Parameters:

BaseAddress is the base address of the device

Returns:

The byte retrieved from the receive FIFO/register.

Note:

None.

```
#define XSpi_mSendByte( BaseAddress,  
                        Data          )
```

Send one byte to the currently selected slave. The byte that is received from the slave is saved in the receive FIFO/register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XSpi_mSetControlReg( BaseAddress,  
                             Mask        )
```

Set the contents of the control register. Use the XSP_CR_* constants defined above to create the bit-mask to be written to the register.

Parameters:

BaseAddress is the base address of the device

Mask is the 16-bit value to write to the control register

Returns:

None.

Note:

None.

```
#define XSpi_mSetSlaveSelectReg( BaseAddress,  
                                Mask      )
```

Set the contents of the slave select register. Each bit in the mask corresponds to a slave select line. Only one slave should be selected at any one time.

Parameters:

BaseAddress is the base address of the device

Mask is the 32-bit value to write to the slave select register

Returns:

None.

Note:

None.

spi/v1_00_b/src/xspi.c File Reference

Detailed Description

Contains required functions of the **XSpi** driver component. See [xspi.h](#) for a detailed description of the device and driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes

1.00a	rpm	10/11/01	First release
1.00b	jhl	03/14/02	Repartitioned driver for smaller files.
1.00b	rpm	04/25/02	Collapsed IPIF and reg base addresses into one
1.00b	rmm	05/14/03	Fixed diab compiler warnings relating to asserts

```
#include "xparameters.h"  
#include "xspi.h"  
#include "xspi_i.h"  
#include "xipif_v1_23_b.h"  
#include "xio.h"
```

Functions

XStatus XSpi_Initialize (**XSpi** *InstancePtr, **Xuint16** DeviceId)
XStatus XSpi_Start (**XSpi** *InstancePtr)
XStatus XSpi_Stop (**XSpi** *InstancePtr)
void **XSpi_Reset** (**XSpi** *InstancePtr)
XStatus XSpi_Transfer (**XSpi** *InstancePtr, **Xuint8** *SendBufPtr, **Xuint8** *RecvBufPtr, unsigned int ByteCount)
XStatus XSpi_SetSlaveSelect (**XSpi** *InstancePtr, **Xuint32** SlaveMask)
Xuint32 XSpi_GetSlaveSelect (**XSpi** *InstancePtr)
void **XSpi_SetStatusHandler** (**XSpi** *InstancePtr, void *CallBackRef, **XSpi_StatusHandler** FuncPtr)
void **XSpi_InterruptHandler** (void *InstancePtr)
void **XSpi_Abort** (**XSpi** *InstancePtr)
XSpi_Config * **XSpi_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
void ( XSpi * )
XSpi_Abort InstancePtr
```

Aborts a transfer in progress by setting the stop bit in the control register, then resetting the FIFOs if present. The byte counts are cleared and the busy flag is set to false.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

None.

Note:

This function does a read/modify/write of the control register. The user of this function needs to take care of critical sections.

```
Xuint32 ( XSpi * )
XSpi_GetSlaveSelect InstancePtr
```

Gets the current slave select bit mask for the SPI device.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

The value returned is a 32-bit mask with a 1 in the bit position of the slave currently selected. The value may be zero if no slaves are selected.

Note:

None.

```
XStatus ( XSpi *
XSpi_Initialize InstancePtr,
Xuint16 DeviceId
)
```

Initializes a specific **XSpi** instance such that the driver is ready to use.

The state of the device after initialization is:

- Device is disabled
- Slave mode
- Active high clock polarity
- Clock phase 0

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XSpi** instance. Passing in a device id associates the generic **XSpi** instance to a specific device, as chosen by the caller or application developer.

Returns:

The return value is XST_SUCCESS if successful. On error, a code indicating the specific error is returned.

Possible error codes are:

- XST_DEVICE_IS_STARTED if the device is started. It must be stopped to re-initialize.
- XST_DEVICE_NOT_FOUND if the device was not found in the configuration such that initialization could not be accomplished.

Note:

None.

```
void XSpi_InterruptHandler ( void * InstancePtr )
```

The interrupt handler for SPI interrupts. This function must be connected by the user to an interrupt source. This function does not save and restore the processor context such that the user must provide this processing.

The interrupts that are handled are:

- Mode Fault Error. This interrupt is generated if this device is selected as a slave when it is configured as a master. The driver aborts any data transfer that is in progress by resetting FIFOs (if present) and resetting its buffer pointers. The upper layer software is informed of the error.
- Data Transmit Register (FIFO) Empty. This interrupt is generated when the transmit register or FIFO is empty. The driver uses this interrupt during a transmission to continually send/receive data until there is no more data to send/receive.
- Data Transmit Register (FIFO) Underrun. This interrupt is generated when the SPI device, when configured as a slave, attempts to read an empty DTR/FIFO. An empty DTR/FIFO usually means that software is not giving the device data in a timely manner. No action is taken by the driver other than to inform the upper layer software of the error.
- Data Receive Register (FIFO) Overrun. This interrupt is generated when the SPI device attempts to write a received byte to an already full DRR/FIFO. A full DRR/FIFO usually means software is not emptying the data in a timely manner. No action is taken by the driver other than to inform the upper layer software of the error.
- Slave Mode Fault Error. This interrupt is generated if a slave device is selected as a slave while it is disabled. No action is taken by the driver other than to inform the upper layer software of the error.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

None.

Note:

The slave select register is being set to deselect the slave when a transfer is complete. This is being done regardless of whether it is a slave or a master since the hardware does not drive the slave select as a slave.

```
XSpi_Config* ( Xuint16 )  
XSpi_LookupConfig DeviceId
```

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId contains the ID of the device to look up the configuration for.

Returns:

A pointer to the configuration found or XNULL if the specified device ID was not found. See [xspi.h](#) for the definition of [XSpi_Config](#).

Note:

None.

```
void ( XSpi * )  
XSpi_Reset InstancePtr
```

Resets the SPI device. Reset must only be called after the driver has been initialized. The configuration of the device after reset is the same as its configuration after initialization. Refer to the `XSpi_Initialize` function for more details. This is a hard reset of the device. Any data transfer that is in progress is aborted.

The upper layer software is responsible for re-configuring (if necessary) and restarting the SPI device after the reset.

Parameters:

InstancePtr is a pointer to the [XSpi](#) instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( XSpi *  
XSpi_SetSlaveSelect InstancePtr,  
 Xuint32 SlaveMask  
 )
```

Selects or deselects the slave with which the master communicates. Each slave that can be selected is represented in the slave select register by a bit. The argument passed to this function is the bit mask with a 1 in the bit position of the slave being selected. Only one slave can be selected.

The user is not allowed to deselect the slave while a transfer is in progress. If no transfer is in progress, the user can select a new slave, which implicitly deselects the current slave. In order to explicitly deselect the current slave, a zero can be passed in as the argument to the function.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

SlaveMask is a 32-bit mask with a 1 in the bit position of the slave being selected. Only one slave can be selected. The SlaveMask can be zero if the slave is being deselected.

Returns:

XST_SUCCESS if the slave is selected or deselected successfully. Otherwise, returns:

- XST_DEVICE_BUSY if a transfer is in progress, slave cannot be changed
- XST_SPI_TOO_MANY_SLAVES if more than one slave is being selected.

Note:

This function only sets the slave which will be selected when a transfer occurs. The slave is not selected when the SPI is idle. The slave select has no affect when the device is configured as a slave.

```
void  
XSpi_SetStatusHandler  
  
    ( XSpi * InstancePtr,  
  
      void * CallbackRef,  
      XSpi_StatusHandler FuncPtr  
    )
```

Sets the status callback function, the status handler, which the driver calls when it encounters conditions that should be reported to the higher layer software. The handler executes in an interrupt context, so it must minimize the amount of processing performed such as transferring data to a thread context. One of the following status events is passed to the status handler.

XST_SPI_MODE_FAULT	A mode fault error occurred, meaning another master tried to select this device as a slave when this device was configured to be a master. Any transfer in progress is aborted.
XST_SPI_TRANSFER_DONE	The requested data transfer is done
XST_SPI_TRANSMIT_UNDERRUN	As a slave device, the master clocked data but there were none available in the transmit register/FIFO. This typically means the slave application did not issue a transfer request fast enough, or the processor/driver could not fill the transmit register/FIFO fast enough.
XST_SPI_RECEIVE_OVERRUN	The SPI device lost data. Data was received

but the receive data register/FIFO was full. This indicates that the device is receiving data

XST_SPI_SLAVE_MODE_FAULT

faster than the processor/driver can consume it. A slave SPI device was selected as a slave while it was disabled. This indicates the master is already transferring data (which is being dropped until the slave application issues a transfer).

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context, so it should do its work quickly and queue potentially time-consuming work to a task-level thread.

XStatus (**XSpi ***)
XSpi_Start *InstancePtr*

This function enables interrupts for the SPI device. It is up to the user to connect the SPI interrupt handler to the interrupt controller before this Start function is called. The GetIntrHandler function is used for that purpose. If the device is configured with FIFOs, the FIFOs are reset at this time.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

XST_SUCCESS if the device is successfully started, or XST_DEVICE_IS_STARTED if the device was already started.

Note:

None.

XStatus (**XSpi ***)
XSpi_Stop *InstancePtr*

This function stops the SPI device by disabling interrupts and disabling the device itself. Interrupts are disabled only within the device itself. If desired, the caller is responsible for disabling interrupts in the interrupt controller and disconnecting the interrupt handler from the interrupt controller.

If the device is in progress of transferring data on the SPI bus, this function returns a status indicating the device is busy. The user will be notified via the status handler when the transfer is complete, and at that time can again try to stop the device. As a master, we do not allow the device to be stopped while a transfer is in progress because the slave may be left in a bad state. As a slave, we do not allow the device to be stopped while a transfer is in progress because the master is not done with its transfer yet.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

XST_SUCCESS if the device is successfully started, or XST_DEVICE_BUSY if a transfer is in progress and cannot be stopped.

Note:

This function makes use of internal resources that are shared between the **XSpi_Stop()** and **XSpi_SetOptions()** functions. So if one task might be setting device options while another is trying to stop the device, the user is required to provide protection of this shared data (typically using a semaphore).

```
XStatus
XSpi_Transfer ( XSpi *
                 InstancePtr,
                 Xuint8 *
                 SendBufPtr,
                 Xuint8 *
                 RecvBufPtr,
                 unsigned int ByteCount
                 )
```

Transfers the specified data on the SPI bus. If the SPI device is configured to be a master, this function initiates bus communication and sends/receives the data to/from the selected SPI slave. If the SPI device is configured to be a slave, this function prepares the data to be sent/received when selected by a master. For every byte sent, a byte is received.

The caller has the option of providing two different buffers for send and receive, or one buffer for both send and receive, or no buffer for receive. The receive buffer must be at least as big as the send buffer to prevent unwanted memory writes. This implies that the byte count passed in as an argument must be the smaller of the two buffers if they differ in size. Here are some sample usages:

```
XSpi_Transfer(InstancePtr, SendBuf, RecvBuf, ByteCount)
```

The caller wishes to send and receive, and provides two different buffers for send and receive.

```
XSpi_Transfer(InstancePtr, SendBuf, NULL, ByteCount)
```

The caller wishes only to send and does not care about the received data. The driver ignores the received data in this case.

XSpi_Transfer(InstancePtr, SendBuf, SendBuf, ByteCount)

The caller wishes to send and receive, but provides the same buffer for doing both. The driver sends the data and overwrites the send buffer with received data as it transfers the data.

XSpi_Transfer(InstancePtr, RecvBuf, RecvBuf, ByteCount)

The caller wishes to only receive and does not care about sending data. In this case, the caller must still provide a send buffer, but it can be the same as the receive buffer if the caller does not care what it sends. The device must send N bytes of data if it wishes to receive N bytes of data.

Although this function takes a buffer as an argument, the driver can only transfer a limited number of bytes at time. It transfers only one byte at a time if there are no FIFOs, or it can transfer the number of bytes up to the size of the FIFO. A call to this function only starts the transfer, then subsequent transfer of the data is performed by the interrupt service routine until the entire buffer has been transferred. The status callback function is called when the entire buffer has been sent/received.

This function is non-blocking. As a master, the SetSlaveSelect function must be called prior to this function.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

SendBufPtr is a pointer to a buffer of data which is to be sent. This buffer must not be NULL.

RecvBufPtr is a pointer to a buffer which will be filled with received data. This argument can be NULL if the caller does not wish to receive data.

ByteCount contains the number of bytes to send/receive. The number of bytes received always equals the number of bytes sent.

Returns:

- XST_SUCCESS if the buffers are successfully handed off to the driver for transfer. Otherwise, returns:
- o XST_DEVICE_IS_STOPPED if the device must be started before transferring data.
 - o XST_DEVICE_BUSY indicates that a data transfer is already in progress. This is determined by the driver.
 - o XST_SPI_NO_SLAVE indicates the device is configured as a master and a slave has not yet been selected.

Note:

This function is not thread-safe. The higher layer software must ensure that no two threads are transferring data on the SPI bus at the same time.

spi/v1_00_b/src/xspi_stats.c File Reference

Detailed Description

This component contains the implementation of statistics functions for the **XSpi** driver component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	03/14/02	First release
1.00b	rpm	04/25/02	Changed macro naming convention

```
#include "xspi.h"  
#include "xspi_i.h"
```

Functions

```
void XSpi_GetStats (XSpi *InstancePtr, XSpi_Stats *StatsPtr)
```

```
void XSpi_ClearStats (XSpi *InstancePtr)
```

Function Documentation

```
void XSpi_ClearStats (XSpi *InstancePtr)
```

Clears the statistics for the SPI device.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

None.

Note:

None.

```
void      ( XSpi *
XSpi_GetStats      InstancePtr,
                  XSpi_Stats * StatsPtr
                )
```

Gets a copy of the statistics for an SPI device.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

StatsPtr is a pointer to a **XSpi_Stats** structure which will get a copy of current statistics.

Returns:

None.

Note:

None.

spi/v1_00_b/src/xspi_options.c File Reference

Detailed Description

Contains functions for the configuration of the **XSpi** driver component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	2/27/02	First release
1.00b	rpm	04/25/02	Collapsed IPIF and reg base addresses into one

```
#include "xspi.h"  
#include "xspi_i.h"  
#include "xio.h"
```

Data Structures

```
struct OptionsMap
```

Functions

```
XStatus XSpi_SetOptions (XSpi *InstancePtr, Xuint32 Options)  
Xuint32 XSpi_GetOptions (XSpi *InstancePtr)
```

Function Documentation

```
Xuint32 ( XSpi * )  
XSpi_GetOptions InstancePtr
```

This function gets the options for the SPI device. The options control how the device behaves relative to the SPI bus.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

Options contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. One or more bit values may be contained in the mask. See the bit definitions named XSP_*_OPTIONS in the file **xspi.h**.

Note:

None.

```
XStatus ( XSpi *  
XSpi_SetOptions InstancePtr,  
Xuint32 Options  
)
```

This function sets the options for the SPI device driver. The options control how the device behaves relative to the SPI bus. The device must be idle rather than busy transferring data before setting these device options.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Options contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. One or more bit values may be contained in the mask. See the bit definitions named XSP_*_OPTIONS in the file **xspi.h**.

Returns:

XST_SUCCESS if options are successfully set. Otherwise, returns:

- XST_DEVICE_BUSY if the device is currently transferring data. The transfer must complete or be aborted before setting options.
- XST_SPI_SLAVE_ONLY if the caller attempted to configure a slave-only device as a master.

Note:

This function makes use of internal resources that are shared between the **XSpi_Stop()** and **XSpi_SetOptions()** functions. So if one task might be setting device options while another is trying to stop the device, the user is required to provide protection of this shared data (typically using a semaphore).

Generated on 24 Jun 2004 for Xilinx Device Drivers

spi/v1_00_b/src/xspi_selftest.c File Reference

Detailed Description

This component contains the implementation of selftest functions for the **XSpi** driver component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	2/27/02	First release
1.00b	rpm	04/25/02	Collapsed IPIF and reg base addresses into one

```
#include "xspi.h"  
#include "xspi_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

XStatus XSpi_SelfTest (**XSpi** *InstancePtr)

Function Documentation

XStatus (**XSpi** *
XSpi_SelfTest *InstancePtr*)

Runs a self-test on the driver/device. The self-test is destructive in that a reset of the device is performed in order to check the reset values of the registers and to get the device into a known state. A simple loopback test is also performed to verify that transmit and receive are working properly. The device is changed to master mode for the loopback test, since only a master can initiate a data transfer.

Upon successful return from the self-test, the device is reset.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

XST_SUCCESS if successful, or one of the following error codes otherwise.

- XST_REGISTER_ERROR indicates a register did not read or write correctly
- XST_LOOPBACK_ERROR if a loopback error occurred.

Note:

None.

sysace/v1_00_a/src/xsysace_compactflash.c File Reference

Detailed Description

Contains functions to reset, read, and write the CompactFlash device via the System ACE controller.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	06/17/02	work in progress

```
#include "xsysace.h"  
#include "xsysace_1.h"
```

Functions

```
XStatus XSysAce_ResetCF (XSysAce *InstancePtr)  
XStatus XSysAce_AbortCF (XSysAce *InstancePtr)  
XStatus XSysAce_IdentifyCF (XSysAce *InstancePtr, XSysAce_CFParameters *ParamPtr)  
Xboolean XSysAce_IsCFReady (XSysAce *InstancePtr)  
XStatus XSysAce_SectorRead (XSysAce *InstancePtr, Xuint32 StartSector, int NumSectors,  
    Xuint8 *BufferPtr)  
XStatus XSysAce_SectorWrite (XSysAce *InstancePtr, Xuint32 StartSector, int NumSectors,  
    Xuint8 *BufferPtr)  
Xuint16 XSysAce_GetFatStatus (XSysAce *InstancePtr)
```

Function Documentation

XStatus (**XSysAce ***)

XSysAce_AbortCF *InstancePtr*

Abort the CompactFlash operation currently in progress.

An MPU lock, obtained using **XSysAce_Lock()**, must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

- XST_SUCCESS if the abort was done successfully
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the CompactFlash is not ready for a command

Note:

According to the ASIC designer, the abort command has not been well tested.

Xuint16 (**XSysAce ***)

XSysAce_GetFatStatus *InstancePtr*

Get the status of the FAT filesystem on the first valid partition of the CompactFlash device such as the boot record and FAT types found.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

A 16-bit mask of status values. These values are defined in **xsysace_1.h** with the prefix XSA_FAT_*.

Note:

None.

```

XStatus ( XSysAce *
XSysAce_IdentifyCF InstancePtr,
XSysAce_CFPParameters * ParamPtr
)

```

Identify the CompactFlash device. Retrieves the parameters for the CompactFlash storage device. Note that this is a polled read of one sector of data. The data is read from the CompactFlash into a byte buffer, which is then copied into the **XSysAce_CFPParameters** structure passed in by the user. The copy is necessary since we don't know how the compiler packs the **XSysAce_CFPParameters** structure.

An MPU lock, obtained using **XSysAce_Lock()**, must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

ParamPtr is a pointer to a **XSysAce_CFPParameters** structure where the information for the CompactFlash device will be stored. See [xsysace.h](#) for details on the **XSysAce_CFPParameters** structure.

Returns:

- XST_SUCCESS if the identify was done successfully
- XST_FAILURE if an error occurs. Use **XSysAce_GetErrors()** to determine cause.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the CompactFlash is not ready for a command

Note:

None.

```

Xboolean ( XSysAce *
XSysAce_IsCFReady InstancePtr
)

```

Check to see if the CompactFlash is ready for a command. The CompactFlash may delay after one operation before it is ready for the next. This function helps the user determine when it is ready before invoking a CompactFlash operation such as [XSysAce_SectorRead\(\)](#) or [XSysAce_SectorWrite\(\)](#);

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

XTRUE if the CompactFlash is ready for a command, and XFALSE otherwise.

Note:

None.

XStatus ([XSysAce](#) *)
XSysAce_ResetCF *InstancePtr*

Reset the CompactFlash device. This function does not reset the System ACE controller. An ATA soft-reset of the CompactFlash is performed.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

- XST_SUCCESS if the reset was done successfully
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the CompactFlash is not ready for a command

Note:

None.

```

XStatus ( XSysAce *
XSysAce_SectorRead      InstancePtr,
                          Xuint32   StartSector,
                          int       NumSectors,
                          Xuint8 *  BufferPtr
                          )

```

Read at least one sector of data from the CompactFlash. The user specifies the starting sector ID and the number of sectors to be read. The minimum unit that can be read from the CompactFlash is a sector, which is 512 bytes.

In polled mode, this read is blocking. If there are other tasks in the system that must run, it is best to keep the number of sectors to be read to a minimum (e.g., 1). In interrupt mode, this read is non-blocking and an event, XSA_EVENT_DATA_DONE, is returned to the user in the asynchronous event handler when the read is complete. The user must call **XSysAce_EnableInterrupt()** to put the driver/device into interrupt mode.

An MPU lock, obtained using **XSysAce_Lock()**, must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

StartSector is the starting sector ID from where data will be read. Sector IDs range from 0 (first sector) to 0x10000000.

NumSectors is the number of sectors to read. The range can be from 1 to 256.

BufferPtr is a pointer to a buffer where the data will be stored. The user must ensure it is big enough to hold (512 * NumSectors) bytes.

Returns:

- XST_SUCCESS if the read was successful. In interrupt mode, this does not mean the read is complete, only that it has begun. An event is returned to the user when the read is complete.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the ACE controller is not ready for a command
- XST_FAILURE if an error occurred during the read. The user should call **XSysAce_GetErrors()** to determine the cause of the error.

Note:

None.

```

XStatus ( XSysAce *
XSysAce_SectorWrite InstancePtr,
Xuint32 StartSector,
int NumSectors,
Xuint8 * BufferPtr
)

```

Write data to the CompactFlash. The user specifies the starting sector ID and the number of sectors to be written. The minimum unit that can be written to the CompactFlash is a sector, which is 512 bytes.

In polled mode, this write is blocking. If there are other tasks in the system that must run, it is best to keep the number of sectors to be written to a minimum (e.g., 1). In interrupt mode, this write is non-blocking and an event, XSA_EVENT_DATA_DONE, is returned to the user in the asynchronous event handler when the write is complete. The user must call **XSysAce_EnableInterrupt()** to put the driver/device into interrupt mode.

An MPU lock, obtained using **XSysAce_Lock()**, must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

StartSector is the starting sector ID from where data will be written. Sector IDs range from 0 (first sector) to 0x10000000.

NumSectors is the number of sectors to write. The range can be from 1 to 256.

BufferPtr is a pointer to the data buffer to be written. This buffer must have at least (512 * NumSectors) bytes.

Returns:

- XST_SUCCESS if the write was successful. In interrupt mode, this does not mean the write is complete, only that it has begun. An event is returned to the user when the write is complete.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the ACE controller is not ready for a command
- XST_FAILURE if an error occurred during the write. The user should call **XSysAce_GetErrors()** to determine the cause of the error.

Note:

None.

sysace/v1_00_a/src/xsysace_intr.c File Reference

Detailed Description

Contains functions related to System ACE interrupt mode. The driver's interrupt handler, [XSysAce_InterruptHandler\(\)](#), must be connected by the user to the interrupt controller.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	06/17/02	work in progress

```
#include "xsysace.h"  
#include "xsysace_1.h"
```

Functions

```
void XSysAce\_EnableInterrupt (XSysAce *InstancePtr)  
void XSysAce\_DisableInterrupt (XSysAce *InstancePtr)  
void XSysAce\_InterruptHandler (void *InstancePtr)  
void XSysAce\_SetEventHandler (XSysAce *InstancePtr, XSysAce\_EventHandler FuncPtr, void  
*CallBackRef)
```

Function Documentation

```
void XSysAce\_DisableInterrupt (XSysAce *  
InstancePtr)
```

Disable all System ACE interrupts and hold the interrupt request line of the device in reset.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance that just interrupted.

Returns:

None.

Note:

None.

```
void ( XSysAce * )  
XSysAce_EnableInterrupt InstancePtr
```

Enable System ACE interrupts. There are three interrupts that can be enabled. The error interrupt enable serves as the driver's means to determine whether interrupts have been enabled or not. The configuration-done interrupt is not enabled here, instead it is enabled during a reset - which can cause a configuration process to start. The data-buffer-ready interrupt is not enabled here either. It is enabled when a read or write operation is started. The reason for not enabling the latter two interrupts are because the status bits may be set as a leftover of an earlier occurrence of the interrupt.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to work on.

Returns:

None.

Note:

None.

```
void ( void * )  
XSysAce_InterruptHandler InstancePtr
```

The interrupt handler for the System ACE driver. This handler must be connected by the user to an interrupt controller or source. This function does not save or restore context.

This function continues reading or writing to the compact flash if such an operation is in progress, and notifies the upper layer software through the event handler once the operation is complete or an error occurs. On an error, any command currently in progress is aborted.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance that just interrupted.

Returns:

None.

Note:

None.

```
void XSysAce_SetEventHandler ( XSysAce * InstancePtr,
                             XSysAce_EventHandler FuncPtr,
                             void * CallbackRef
                             )
```

Set the callback function for handling events. The upper layer software should call this function during initialization. The events are passed asynchronously to the upper layer software. The events are described in **xsysace.h** and are named XSA_EVENT_*

Note that the callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is a reference pointer to be passed back to the upper layer.

Returns:

None.

Note:

None.

Generated on 24 Jun 2004 for Xilinx Device Drivers

sysace/v1_00_a/src/xsysace_jtagcfg.c File Reference

Detailed Description

Contains functions to control the configuration of the target FPGA chain on the System ACE via the JTAG configuration port.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	06/17/02	work in progress

```
#include "xsysace.h"  
#include "xsysace_1.h"
```

Functions

```
void XSysAce_ResetCfg (XSysAce *InstancePtr)  
void XSysAce_SetCfgAddr (XSysAce *InstancePtr, unsigned int Address)  
void XSysAce_SetStartMode (XSysAce *InstancePtr, Xboolean ImmedOnReset, Xboolean  
    StartCfg)  
XStatus XSysAce_ProgramChain (XSysAce *InstancePtr, Xuint8 *BufferPtr, int NumBytes)  
Xboolean XSysAce_IsCfgDone (XSysAce *InstancePtr)  
Xuint32 XSysAce_GetCfgSector (XSysAce *InstancePtr)
```

Function Documentation

Xuint32 (**XSysAce ***)

XSysAce_GetCfgSector *InstancePtr*

Get the sector ID of the CompactFlash sector being used for configuration of the target FPGA chain. This sector ID (or logical block address) only affects transfers between the ACE configuration logic and the CompactFlash card. This function is typically used for debug purposes to determine which sector was being accessed when an error occurred.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

The sector ID (logical block address) being used for data transfers between the ACE configuration logic and the CompactFlash. Sector IDs range from 0 to 0x10000000.

Note:

None.

Xboolean (**XSysAce ***)

XSysAce_IsCfgDone *InstancePtr*

Check to see if the configuration of the target FPGA chain is complete. This function is typically only used in polled mode. In interrupt mode, an event (XSA_EVENT_CFG_DONE) is returned to the user in the asynchronous event handler.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

XTRUE if the configuration is complete. XFALSE otherwise.

Note:

None.

XStatus (**XSysAce ***

XSysAce_ProgramChain *InstancePtr,*
Xuint8 * *BufferPtr,*
int *NumBytes*
)

Program the target FPGA chain through the configuration JTAG port. This allows the user to program the devices on the target FPGA chain from the MPU port instead of from CompactFlash. The user specifies a buffer and the number of bytes to write. The buffer should be equivalent to an ACE (.ace) file.

Note that when loading the ACE file via the MPU port, the first sector of the ACE file is discarded. The CF filesystem controller in the System ACE device knows to skip the first sector when the ACE file comes from the CF, but the CF filesystem controller is bypassed when the ACE file comes from the MPU port. For this reason, this function skips the first sector of the buffer passed in.

In polled mode, the write is blocking. In interrupt mode, the write is non-blocking and an event, XSA_EVENT_CFG_DONE, is returned to the user in the asynchronous event handler when the configuration is complete.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

BufferPtr is a pointer to a buffer that will be used to program the configuration JTAG devices.

NumBytes is the number of bytes in the buffer. We assume that there is at least one sector of data in the .ace file, which is the information sector.

Returns:

- XST_SUCCESS if the write was successful. In interrupt mode, this does not mean the write is complete, only that it has begun. An event is returned to the user when the write is complete.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_FAILURE if an error occurred during the write. The user should call [XSysAce_GetErrors\(\)](#) to determine the cause of the error.

Note:

None.

```
void XSysAce_ResetCfg ( XSysAce * InstancePtr )
```

Reset the JTAG configuration controller. This comprises a reset of the JTAG configuration controller and the CompactFlash controller (if it is currently being accessed by the configuration controller). Note that the MPU controller is not reset, meaning the MPU registers remain unchanged. The configuration controller is reset then released from reset in this function.

The CFGDONE status (and therefore interrupt) is cleared when the configuration controller is reset. If interrupts have been enabled, we go ahead and enable the CFGDONE interrupt here. This means that if and when a configuration process starts as a result of this reset, an interrupt will be received when it is complete.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

None.

Note:

This function is not thread-safe.

```
void  
XSysAce_SetCfgAddr ( XSysAce *  
                    InstancePtr,  
                    unsigned int Address  
                    )
```

Select the configuration address (or file) from the CompactFlash to be used for configuration of the target FPGA chain.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Address is the address or file number to be used as the bitstream to configure the target FPGA devices. There are 8 possible files, so the value of this parameter can range from 0 to 7.

Returns:

None.

Note:

None.

```

void
XSysAce_SetStartMode
( XSysAce * InstancePtr,
  Xboolean ImmedOnReset,
  Xboolean StartCfg
)

```

Set the start mode for configuration of the target FPGA chain from CompactFlash. The configuration process only starts after a reset. The user can indicate that the configuration should start immediately after a reset, or the configuration process can be delayed until the user commands it to start (using this function). The configuration controller can be reset using [XSysAce_ResetCfg\(\)](#).

The user can select which configuration file on the CompactFlash to use using the [XSysAce_SetCfgAddr\(\)](#) function. If the user intends to configure the target FPGA chain directly from the MPU port, this function is not needed. Instead, the user would simply call [XSysAce_ProgramChain\(\)](#).

The user can use [XSysAce_IsCfgDone\(\)](#) when in polled mode to determine if the configuration is complete. If in interrupt mode, the event XSA_EVENT_CFG_DONE will be returned asynchronously to the user when the configuration is complete. The user must call [XSysAce_EnableInterrupt\(\)](#) to put the device/driver into interrupt mode.

Parameters:

- InstancePtr* is a pointer to the [XSysAce](#) instance to be worked on.
- ImmedOnReset* can be set to XTRUE to indicate the configuration process will start immediately after a reset of the ACE configuration controller, or it can be set to XFALSE to indicate the configuration process is delayed after a reset until the user starts it (using this function).
- StartCfg* is a boolean indicating whether to start the configuration process or not. When ImmedOnReset is set to XTRUE, this value is ignored. When ImmedOnReset is set to XFALSE, then this value controls when the configuration process is started. When set to XTRUE the configuration process starts (assuming a reset of the device has occurred), and when set to XFALSE the configuration process does not start.

Returns:

None.

Note:

None.

sysace/v1_00_a/src/xsysace_selftest.c File Reference

Detailed Description

Contains diagnostic functions for the System ACE device and driver. This includes a self-test to make sure communication to the device is possible and the ability to retrieve the ACE controller version.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	06/17/02	work in progress

```
#include "xsysace.h"  
#include "xsysace_1.h"
```

Functions

[XStatus XSysAce_SelfTest](#) ([XSysAce](#) *InstancePtr)

[Xuint16 XSysAce_GetVersion](#) ([XSysAce](#) *InstancePtr)

Function Documentation

```
Xuint16 (XSysAce *  
XSysAce\_GetVersion InstancePtr)
```

Retrieve the version of the System ACE device.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

A 16-bit version where the 4 most significant bits are the major version number, the next four bits are the minor version number, and the least significant 8 bits are the revision or build number.

Note:

None.

XStatus	(XSysAce *)
XSysAce_SelfTest	<i>InstancePtr</i>

A self-test that simply proves communication to the ACE controller from the device driver by obtaining an MPU lock, verifying it, then releasing it.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

XST_SUCCESS if self-test passes, or XST_FAILURE if an error occurs.

Note:

None.

sysace/v1_00_a/src/xsysace_1.c File Reference

Detailed Description

This file contains low-level functions to read and write CompactFlash sectors and ACE controller registers. These functions can be used when only the low-level functionality of the driver is desired. The user would typically use the high-level driver functions defined in [xsysace.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	06/14/02	work in progress
1.00a	rpm	09/16/03	Added include of xparameters.h in order to get the XPAR_XSYSACE_MEM_WIDTH definition.
1.00a	rpm	02/17/04	Fixed WriteSector function command

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

```
#include "xsysace\_1.h"
```

Functions

[Xuint32](#) [XSysAce_RegRead32](#) ([Xuint32](#) Address)

[Xuint16](#) [XSysAce_RegRead16](#) ([Xuint32](#) Address)

void [XSysAce_RegWrite32](#) ([Xuint32](#) Address, [Xuint32](#) Data)

void [XSysAce_RegWrite16](#) ([Xuint32](#) Address, [Xuint16](#) Data)

int [XSysAce_ReadSector](#) ([Xuint32](#) BaseAddress, [Xuint32](#) SectorId, [Xuint8](#) *BufferPtr)

int [XSysAce_WriteSector](#) ([Xuint32](#) BaseAddress, [Xuint32](#) SectorId, [Xuint8](#) *BufferPtr)

int [XSysAce_ReadDataBuffer](#) ([Xuint32](#) BaseAddress, [Xuint8](#) *BufferPtr, int Size)

int [XSysAce_WriteDataBuffer](#) ([Xuint32](#) BaseAddress, [Xuint8](#) *BufferPtr, int Size)

Function Documentation

```
int  
XSysAce_ReadDataBuffer ( Xuint32  
                        BaseAddress,  
                        Xuint8 * BufferPtr,  
                        int      Size  
                        )
```

Read the specified number of bytes from the data buffer of the ACE controller. The data buffer, which is 32 bytes, can only be read two bytes at a time. Once the data buffer is read, we wait for it to be filled again before reading the next buffer's worth of data.

Parameters:

BaseAddress is the base address of the device

BufferPtr is a pointer to a buffer in which to store data.

Size is the number of bytes to read

Returns:

The total number of bytes read, or 0 if an error occurred.

Note:

If *Size* is not aligned with the size of the data buffer (32 bytes), this function will read the entire data buffer, dropping the extra bytes on the floor since the user did not request them. This is necessary to get the data buffer to be ready again.

```
int  
XSysAce_ReadSector ( Xuint32  
                   BaseAddress,  
                   Xuint32 SectorId,  
                   Xuint8 * BufferPtr  
                   )
```

Read a CompactFlash sector. This is a blocking, low-level function which does not return until the specified sector is read.

Parameters:

BaseAddress is the base address of the device

SectorId is the id of the sector to read

BufferPtr is a pointer to a buffer where the data will be stored.

Returns:

The number of bytes read. If this number is not equal to the sector size, 512 bytes, then an error occurred.

Note:

None.

Xuint16 (**Xuint32**)
XSysAce_RegRead16 *Address*

Read a 16-bit value from the given address. Based on a compile-time constant, do the read in one 16-bit read or two 8-bit reads.

Parameters:

Address is the address to read from.

Returns:

The 16-bit value of the address.

Note:

No need for endian conversion in 8-bit mode since this function gets the bytes into their proper lanes in the 16-bit word.

Xuint32 (**Xuint32**)
XSysAce_RegRead32 *Address*

Write a 32-bit value to the given address. Based on a compile-time constant, do the write in two 16-bit writes or four 8-bit writes.

Parameters:

Address is the address to write to.

Data is the value to write

Returns:

None.

Note:

No need for endian conversion in 8-bit mode since this function writes the bytes into their proper lanes based on address.

```
int XSysAce_WriteDataBuffer ( Xuint32 BaseAddress,
                             Xuint8 * BufferPtr,
                             int Size
                             )
```

Write the specified number of bytes to the data buffer of the ACE controller. The data buffer, which is 32 bytes, can only be written two bytes at a time. Once the data buffer is written, we wait for it to be empty again before writing the next buffer's worth of data. If the size of the incoming buffer is not aligned with the System ACE data buffer size (32 bytes), then this routine pads out the data buffer with zeros so the entire data buffer is written. This is necessary for the ACE controller to process the data buffer.

Parameters:

BaseAddress is the base address of the device

BufferPtr is a pointer to a buffer used to write to the controller.

Size is the number of bytes to write

Returns:

The total number of bytes written (not including pad bytes), or 0 if an error occurs.

Note:

None.

```
int  
XSysAce_WriteSector ( Xuint32  
                      BaseAddress,  
                      Xuint32 SectorId,  
                      Xuint8 * BufferPtr  
                    )
```

Write a CompactFlash sector. This is a blocking, low-level function which does not return until the specified sector is written in its entirety.

Parameters:

BaseAddress is the base address of the device

SectorId is the id of the sector to write

BufferPtr is a pointer to a buffer used to write the sector.

Returns:

The number of bytes written. If this number is not equal to the sector size, 512 bytes, then an error occurred.

Note:

None.

tmrctr/v1_00_b/src/xtmrctr.h File Reference

Detailed Description

The Xilinx timer/counter component. This component supports the Xilinx timer/counter. More detailed description of the driver operation can be found in the [xtmrctr.c](#) file.

The Xilinx timer/counter supports the following features:

- Polled mode.
- Interrupt driven mode
- enabling and disabling specific timers
- PWM operation

The driver does not currently support the PWM operation of the device.

The timer counter operates in 2 primary modes, compare and capture. In either mode, the timer counter may count up or down, with up being the default.

Compare mode is typically used for creating a single time period or multiple repeating time periods in the auto reload mode, such as a periodic interrupt. When started, the timer counter loads an initial value, referred to as the compare value, into the timer counter and starts counting down or up. The timer counter expires when it rolls over/under depending upon the mode of counting. An external compare output signal may be configured such that a pulse is generated with this signal when it hits the compare value.

Capture mode is typically used for measuring the time period between external events. This mode uses an external capture input signal to cause the value of the timer counter to be captured. When started, the timer counter loads an initial value, referred to as the compare value,

The timer can be configured to either cause an interrupt when the count reaches the compare value in compare mode or latch the current count value in the capture register when an external input is asserted in capture mode. The external capture input can be enabled/disabled using the `XTmrCtr_SetOptions` function. While in compare mode, it is also possible to drive an external output when the compare value is reached in the count register. The external compare output can be enabled/disabled using the `XTmrCtr_SetOptions` function.

Interrupts

It is the responsibility of the application to connect the interrupt handler of the timer/counter to the interrupt source. The interrupt handler function, `XTmrCtr_InterruptHandler`, is visible such that the user can connect it to the interrupt source. Note that this interrupt handler does not provide interrupt context save and restore processing, the user must

perform this processing.

The driver services interrupts and passes timeouts to the upper layer software through callback functions. The upper layer software must register its callback functions during initialization. The driver requires callback functions for timers.

Note:

The default settings for the timers are:

- o Interrupt generation disabled
- o Count up mode
- o Compare mode
- o Hold counter (will not reload the timer)
- o External compare output disabled
- o External capture input disabled
- o Pulse width modulation disabled
- o Timer disabled, waits for Start function to be called

A timer counter device may contain multiple timer counters. The symbol `XTC_DEVICE_TIMER_COUNT` defines the number of timer counters in the device. The device currently contains 2 timer counters.

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

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

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xtmrctr_1.h"
```

Data Structures

```
struct XTmrCtr  
struct XTmrCtr_Config  
struct XTmrCtrStats
```

Configuration options

These options are used in `XTmrCtr_SetOptions()` and `XTmrCtr_GetOptions()`

```
#define XTC_ENABLE_ALL_OPTION
#define XTC_DOWN_COUNT_OPTION
#define XTC_CAPTURE_MODE_OPTION
#define XTC_INT_MODE_OPTION
#define XTC_AUTO_RELOAD_OPTION
#define XTC_EXT_COMPARE_OPTION
```

Typedefs

```
typedef void(* XTmrCtr_Handler )(void *CallBackRef, Xuint8 TmrCtrNumber)
```

Functions

```
XStatus XTmrCtr_Initialize (XTmrCtr *InstancePtr, Xuint16 DeviceId)
void XTmrCtr_Start (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
void XTmrCtr_Stop (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
Xuint32 XTmrCtr_GetValue (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
void XTmrCtr_SetResetValue (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber, Xuint32 ResetValue)
Xuint32 XTmrCtr_GetCaptureValue (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
Xboolean XTmrCtr_IsExpired (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
void XTmrCtr_Reset (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
void XTmrCtr_SetOptions (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber, Xuint32 Options)
Xuint32 XTmrCtr_GetOptions (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
void XTmrCtr_GetStats (XTmrCtr *InstancePtr, XTmrCtrStats *StatsPtr)
void XTmrCtr_ClearStats (XTmrCtr *InstancePtr)
XStatus XTmrCtr_SelfTest (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
void XTmrCtr_SetHandler (XTmrCtr *InstancePtr, XTmrCtr_Handler FuncPtr, void *CallBackRef)
void XTmrCtr_InterruptHandler (void *InstancePtr)
```

Define Documentation

```
#define
XTC_AUTO_RELOAD_OPTION
```

Used to configure the timer counter device.

XTC_ENABLE_ALL_OPTION	Enables all timer counters at once.
XTC_DOWN_COUNT_OPTION	Configures the timer counter to count down from start value, the default is to count up.
XTC_CAPTURE_MODE_OPTION	Configures the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.
XTC_INT_MODE_OPTION	Enables the timer counter interrupt output.
XTC_AUTO_RELOAD_OPTION	In compare mode, configures the timer counter to reload from the compare value. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.
XTC_EXT_COMPARE_OPTION	Enables the external compare output signal.

#define

XTC_CAPTURE_MODE_OPTION

Used to configure the timer counter device.

XTC_ENABLE_ALL_OPTION	Enables all timer counters at once.
XTC_DOWN_COUNT_OPTION	Configures the timer counter to count down from start value, the default is to count up.
XTC_CAPTURE_MODE_OPTION	Configures the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.
XTC_INT_MODE_OPTION	Enables the timer counter interrupt output.
XTC_AUTO_RELOAD_OPTION	In compare mode, configures the timer counter to reload from the compare value. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.
XTC_EXT_COMPARE_OPTION	Enables the external compare output signal.

#define

XTC_DOWN_COUNT_OPTION

Used to configure the timer counter device.

XTC_ENABLE_ALL_OPTION	Enables all timer counters at once.
XTC_DOWN_COUNT_OPTION	Configures the timer counter to count down from start value, the default is to count up.
XTC_CAPTURE_MODE_OPTION	Configures the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.
XTC_INT_MODE_OPTION	Enables the timer counter interrupt output.
XTC_AUTO_RELOAD_OPTION	In compare mode, configures the timer counter to reload from the compare value. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.
XTC_EXT_COMPARE_OPTION	Enables the external compare output signal.

#define

XTC_ENABLE_ALL_OPTION

Used to configure the timer counter device.

XTC_ENABLE_ALL_OPTION	Enables all timer counters at once.
XTC_DOWN_COUNT_OPTION	Configures the timer counter to count down from start value, the default is to count up.
XTC_CAPTURE_MODE_OPTION	Configures the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.
XTC_INT_MODE_OPTION	Enables the timer counter interrupt output.
XTC_AUTO_RELOAD_OPTION	In compare mode, configures the timer counter to reload from the compare value. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.
XTC_EXT_COMPARE_OPTION	Enables the external compare output signal.

#define

XTC_EXT_COMPARE_OPTION

Used to configure the timer counter device.

XTC_ENABLE_ALL_OPTION	Enables all timer counters at once.
XTC_DOWN_COUNT_OPTION	Configures the timer counter to count down from start value, the default is to count up.
XTC_CAPTURE_MODE_OPTION	Configures the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.
XTC_INT_MODE_OPTION	Enables the timer counter interrupt output.
XTC_AUTO_RELOAD_OPTION	In compare mode, configures the timer counter to reload from the compare value. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.
XTC_EXT_COMPARE_OPTION	Enables the external compare output signal.

#define

XTC_INT_MODE_OPTION

Used to configure the timer counter device.

XTC_ENABLE_ALL_OPTION	Enables all timer counters at once.
XTC_DOWN_COUNT_OPTION	Configures the timer counter to count down from start value, the default is to count up.
XTC_CAPTURE_MODE_OPTION	Configures the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.
XTC_INT_MODE_OPTION	Enables the timer counter interrupt output.
XTC_AUTO_RELOAD_OPTION	In compare mode, configures the timer counter to reload from the compare value. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.
XTC_EXT_COMPARE_OPTION	Enables the external compare output signal.

```
typedef void(* XTmrCtr_Handler)(void *CallbackRef, Xuint8
TmrCtrNumber)
```

Signature for the callback function.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked. Its type is unimportant to the driver, so it is a void pointer.

TmrCtrNumber is the number of the timer/counter within the device. The device typically contains at least two timer/counters. The timer number is a zero based number with a range of 0 to (XTC_DEVICE_TIMER_COUNT - 1).

Function Documentation

```
void ( XTmrCtr * )
XTmrCtr_ClearStats InstancePtr
```

Clear the **XTmrCtrStats** structure for this driver.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint32 ( XTmrCtr * InstancePtr,
XTmrCtr_GetCaptureValue Xuint8 TmrCtrNumber
)
```

Returns the timer counter value that was captured the last time the external capture input was asserted.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The current capture value for the indicated timer counter.

Note:

None.

```
Xuint32 XTmrCtr_GetOptions ( XTmrCtr * InstancePtr,  
                             Xuint8 TmrCtrNumber  
                             )
```

Get the options for the specified timer counter.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The currently set options. An option which is set to a '1' is enabled and set to a '0' is disabled. The options are bit masks such that multiple options may be set or cleared. The options are described in [xtmrctr.h](#).

Note:

None.

```
void XTmrCtr_GetStats ( XTmrCtr * InstancePtr,  
                       XTmrCtrStats * StatsPtr  
                       )
```

Get a copy of the **XTmrCtrStats** structure, which contains the current statistics for this driver.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

StatsPtr is a pointer to a **XTmrCtrStats** structure which will get a copy of current statistics.

Returns:

None.

Note:

None.

```
Xuint32          ( XTmrCtr * InstancePtr,  
XTmrCtr_GetValue      Xuint8   TmrCtrNumber  
                      )
```

Get the current value of the specified timer counter. The timer counter may be either incrementing or decrementing based upon the current mode of operation.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The current value for the timer counter.

Note:

None.

```
XStatus          ( XTmrCtr *  
XTmrCtr_Initialize      InstancePtr,  
                          Xuint16   DeviceId  
                          )
```

Initializes a specific timer/counter instance/driver. Initialize fields of the **XTmrCtr** structure, then reset the timer/counter

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XTmrCtr** component. Passing in a device id associates the generic **XTmrCtr** component to a specific device, as chosen by the caller or application developer.

Returns:

- o XST_SUCCESS if initialization was successful
- o XST_DEVICE_IS_STARTED if the device has already been started
- o XST_DEVICE_NOT_FOUND if the device doesn't exist

Note:

None.

```
void XTmrCtr_InterruptHandler ( void * InstancePtr )
```

Interrupt Service Routine (ISR) for the driver. This function only performs processing for the device and does not save and restore the interrupt context.

Parameters:

InstancePtr contains a pointer to the timer/counter instance for the interrupt.

Returns:

None.

Note:

None.

```
Xboolean XTmrCtr_IsExpired ( XTmrCtr * InstancePtr, Xuint8 TmrCtrNumber )
```

Checks if the specified timer counter of the device has expired. In capture mode, expired is defined as a capture occurred. In compare mode, expired is defined as the timer counter rolled over/under for up/down counting.

When interrupts are enabled, the expiration causes an interrupt. This function is typically used to poll a timer counter to determine when it has expired.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

XTRUE if the timer has expired, and XFALSE otherwise.

Note:

None.

```
void  
XTmrCtr_Reset  
  
    ( XTmrCtr * InstancePtr,  
      Xuint8   TmrCtrNumber  
    )
```

Resets the specified timer counter of the device. A reset causes the timer counter to set it's value to the reset value.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```
XStatus  
XTmrCtr_SelfTest  
  
    ( XTmrCtr * InstancePtr,  
      Xuint8   TmrCtrNumber  
    )
```

Runs a self-test on the driver/device. This test verifies that the specified timer counter of the device can be enabled and increments.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

XST_SUCCESS if self-test was successful, or XST_FAILURE if the timer is not incrementing.

Note:

This is a destructive test using the provided timer. The current settings of the timer are returned to the initialized values and all settings at the time this function is called are overwritten.

```
void  
XTmrCtr_SetHandler ( XTmrCtr *  
                    InstancePtr,  
                    XTmrCtr_Handler FuncPtr,  
                    void *  
                    CallbackRef  
                    )
```

Sets the timer callback function, which the driver calls when the specified timer times out.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context so the function that is called should either be short or pass the more extensive processing off to another task to allow the interrupt to return and normal processing to continue.

```
void  
XTmrCtr_SetOptions ( XTmrCtr *  
                    InstancePtr,  
                    Xuint8  
                    TmrCtrNumber,  
                    Xuint32  
                    Options  
                    )
```

Enables the specified options for the specified timer counter. This function sets the options without regard to the current options of the driver. To prevent a loss of the current options, the user should call [XTmrCtr_GetOptions](#) () prior to this function and modify the retrieved options to pass into this function to prevent loss of the current options.

Parameters:

- InstancePtr* is a pointer to the [XTmrCtr](#) instance to be worked on.
- TmrCtrNumber* is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).
- Options* contains the desired options to be set or cleared. Setting the option to '1' enables the option, clearing the to '0' disables the option. The options are bit masks such that multiple options may be set or cleared. The options are described in [xtmrctr.h](#).

Returns:

None.

Note:

None.

```
void  
XTmrCtr_SetResetValue ( XTmrCtr * InstancePtr,  
                        Xuint8 TmrCtrNumber,  
                        Xuint32 ResetValue  
                        )
```

Set the reset value for the specified timer counter. This is the value that is loaded into the timer counter when it is reset. This value is also loaded when the timer counter is started.

Parameters:

- InstancePtr* is a pointer to the [XTmrCtr](#) instance to be worked on.
- TmrCtrNumber* is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).
- ResetValue* contains the value to be used to reset the timer counter.

Returns:

None.

Note:

None.

```
void  
XTmrCtr_Start ( XTmrCtr * InstancePtr,  
                Xuint8   TmrCtrNumber  
              )
```

Starts the specified timer counter of the device such that it starts running. The timer counter is reset before it is started and the reset value is loaded into the timer counter.

If interrupt mode is specified in the options, it is necessary for the caller to connect the interrupt handler of the timer/counter to the interrupt source, typically an interrupt controller, and enable the interrupt within the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```
void  
XTmrCtr_Stop ( XTmrCtr * InstancePtr,  
               Xuint8   TmrCtrNumber  
             )
```

Stops the timer counter by disabling it.

It is the callers' responsibility to disconnect the interrupt handler of the timer_counter from the interrupt source, typically an interrupt controller, and disable the interrupt within the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

tmrctr/v1_00_b/src/xtmrctr.c File Reference

Detailed Description

Contains required functions for the [XTmrCtr](#) driver.

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

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xparameters.h"  
#include "xtmrctr.h"  
#include "xtmrctr_i.h"
```

Functions

XStatus [XTmrCtr_Initialize](#) ([XTmrCtr](#) *InstancePtr, **Xuint16** DeviceId)

void [XTmrCtr_Start](#) ([XTmrCtr](#) *InstancePtr, **Xuint8** TmrCtrNumber)

void [XTmrCtr_Stop](#) ([XTmrCtr](#) *InstancePtr, **Xuint8** TmrCtrNumber)

Xuint32 [XTmrCtr_GetValue](#) ([XTmrCtr](#) *InstancePtr, **Xuint8** TmrCtrNumber)

void [XTmrCtr_SetResetValue](#) ([XTmrCtr](#) *InstancePtr, **Xuint8** TmrCtrNumber, **Xuint32** ResetValue)

Xuint32 [XTmrCtr_GetCaptureValue](#) ([XTmrCtr](#) *InstancePtr, **Xuint8** TmrCtrNumber)

void [XTmrCtr_Reset](#) ([XTmrCtr](#) *InstancePtr, **Xuint8** TmrCtrNumber)

Xboolean [XTmrCtr_IsExpired](#) ([XTmrCtr](#) *InstancePtr, **Xuint8** TmrCtrNumber)

Function Documentation

```
Xuint32  
XTmrCtr_GetCaptureValue ( XTmrCtr * InstancePtr,  
                          Xuint8   TmrCtrNumber  
                          )
```

Returns the timer counter value that was captured the last time the external capture input was asserted.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The current capture value for the indicated timer counter.

Note:

None.

```
Xuint32  
XTmrCtr_GetValue ( XTmrCtr * InstancePtr,  
                  Xuint8   TmrCtrNumber  
                  )
```

Get the current value of the specified timer counter. The timer counter may be either incrementing or decrementing based upon the current mode of operation.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The current value for the timer counter.

Note:

None.

```
XStatus ( XTmrCtr *  
XTmrCtr_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific timer/counter instance/driver. Initialize fields of the **XTmrCtr** structure, then reset the timer/counter

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XTmrCtr** component. Passing in a device id associates the generic **XTmrCtr** component to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if the device doesn't exist

Note:

None.

```
Xboolean ( XTmrCtr *  
XTmrCtr_IsExpired InstancePtr,  
 Xuint8 TmrCtrNumber  
 )
```

Checks if the specified timer counter of the device has expired. In capture mode, expired is defined as a capture occurred. In compare mode, expired is defined as the timer counter rolled over/under for up/down counting.

When interrupts are enabled, the expiration causes an interrupt. This function is typically used to poll a timer counter to determine when it has expired.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

XTRUE if the timer has expired, and XFALSE otherwise.

Note:

None.

```
void XTmrCtr_Reset ( XTmrCtr * InstancePtr, Xuint8 TmrCtrNumber )
```

Resets the specified timer counter of the device. A reset causes the timer counter to set it's value to the reset value.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```

void          ( XTmrCtr *
XTmrCtr_SetResetValue  InstancePtr,

                Xuint8    TmrCtrNumber,
                Xuint32   ResetValue
                )

```

Set the reset value for the specified timer counter. This is the value that is loaded into the timer counter when it is reset. This value is also loaded when the timer counter is started.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

ResetValue contains the value to be used to reset the timer counter.

Returns:

None.

Note:

None.

```

void          ( XTmrCtr *
XTmrCtr_Start  InstancePtr,

                Xuint8    TmrCtrNumber
                )

```

Starts the specified timer counter of the device such that it starts running. The timer counter is reset before it is started and the reset value is loaded into the timer counter.

If interrupt mode is specified in the options, it is necessary for the caller to connect the interrupt handler of the timer/counter to the interrupt source, typically an interrupt controller, and enable the interrupt within the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```
void  
XTmrCtr_Stop ( XTmrCtr * InstancePtr,  
               Xuint8   TmrCtrNumber  
             )
```

Stops the timer counter by disabling it.

It is the callers' responsibility to disconnect the interrupt handler of the timer_counter from the interrupt source, typically an interrupt controller, and disable the interrupt within the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

XTmrCtr Struct Reference

```
#include <xtmrctr.h>
```

Detailed Description

The XTmrCtr driver instance data. The user is required to allocate a variable of this type for every timer/counter device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [tmrctr/v1_00_b/src/xtmrctr.h](#)
-

tmrctr/v1_00_b/src/xtmrctr_i.h File Reference

Detailed Description

This file contains data which is shared between files internal to the **XTmrCtr** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	02/06/02	First release

```
#include "xbasic_types.h"
```

Variables

XTmrCtr_Config **XTmrCtr_ConfigTable** []

Variable Documentation

XTmrCtr_Config **XTmrCtr_ConfigTable** ()

[]

The timer/counter configuration table, sized by the number of instances defined in **xparameters.h**.

XTmrCtr_Config Struct Reference

```
#include <xtmrctr.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

Field Documentation

[Xuint32 XTmrCtr_Config::](#)

BaseAddress

Register base
address

[Xuint16 XTmrCtr_Config::](#)

DeviceId

Unique ID of
device

The documentation for this struct was generated from the following file:

- [tmrctr/v1_00_b/src/xtmrctr.h](#)

tmrctr/v1_00_b/src/xtmrctr_1.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. High-level driver functions are defined in [xtmrctr.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	04/24/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

Register Offset Definitions

Register offsets within a timer counter, there are multiple timer counters within a single device

```
#define XTC_TCSR_OFFSET  
#define XTC_TLR_OFFSET  
#define XTC_TCR_OFFSET
```

Control Status Register Bit Definitions

Control Status Register bit masks Used to configure the timer counter device.

```
#define XTC_CSR_ENABLE_ALL_MASK  
#define XTC_CSR_ENABLE_PWM_MASK
```

```
#define XTC_CSR_INT_OCCURED_MASK
#define XTC_CSR_ENABLE_TMR_MASK
#define XTC_CSR_ENABLE_INT_MASK
#define XTC_CSR_LOAD_MASK
#define XTC_CSR_AUTO_RELOAD_MASK
#define XTC_CSR_EXT_CAPTURE_MASK
#define XTC_CSR_EXT_GENERATE_MASK
#define XTC_CSR_DOWN_COUNT_MASK
#define XTC_CSR_CAPTURE_MODE_MASK
```

Defines

```
#define XTC_DEVICE_TIMER_COUNT
#define XTimerCtr_mReadReg(BaseAddress, TmrCtrNumber, RegOffset)
#define XTmrCtr_mWriteReg(BaseAddress, TmrCtrNumber, RegOffset, ValueToWrite)
#define XTmrCtr_mSetControlStatusReg(BaseAddress, TmrCtrNumber, RegisterValue)
#define XTmrCtr_mGetControlStatusReg(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mGetTimerCounterReg(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mSetLoadReg(BaseAddress, TmrCtrNumber, RegisterValue)
#define XTmrCtr_mGetLoadReg(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mEnable(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mDisable(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mEnableIntr(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mDisableIntr(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mLoadTimerCounterReg(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mHasEventOccurred(BaseAddress, TmrCtrNumber)
```

Define Documentation

```
#define  
XTC_CSR_AUTO_RELOAD_MASK
```

In compare mode, configures the timer counter to reload from the Load Register. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.

#define

XTC_CSR_CAPTURE_MODE_MASK

Enables the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.

#define

XTC_CSR_DOWN_COUNT_MASK

Configures the timer counter to count down from start value, the default is to count up.

#define

XTC_CSR_ENABLE_ALL_MASK

Enables all timer
counters

#define

XTC_CSR_ENABLE_INT_MASK

Enables the interrupt
output.

#define

XTC_CSR_ENABLE_PWM_MASK

Enables the Pulse Width
Modulation

#define

XTC_CSR_ENABLE_TMR_MASK

Enables only the specific
timer

#define

XTC_CSR_EXT_CAPTURE_MASK

Enables the external input to the timer
counter.

#define

XTC_CSR_EXT_GENERATE_MASK

Enables the external generate output for the timer.

```
#define  
XTC_CSR_INT_OCCURED_MASK
```

If set and '1' is written to this bit position, bit is cleared.

```
#define  
XTC_CSR_LOAD_MASK
```

Loads the timer using the load value provided earlier in the Load Register, XTC_TLR_OFFSET.

```
#define  
XTC_DEVICE_TIMER_COUNT
```

Defines the number of timer counters within a single hardware device. This number is not currently parameterized in the hardware but may be in the future.

```
#define  
XTC_TCR_OFFSET
```

timer counter
register

```
#define  
XTC_TCSR_OFFSET
```

control/status
register

```
#define  
XTC_TLR_OFFSET
```

load
register

```
#define XTimerCtr_mReadReg( BaseAddress,  
                           TmrCtrNumber,  
                           RegOffset      )
```

Read one of the timer counter registers.

Parameters:

BaseAddress contains the base address of the timer counter device.

TmrCtrNumber contains the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

RegOffset contains the offset from the 1st register of the timer counter to select the specific register of the timer counter.

Returns:

The value read from the register, a 32 bit value.

Note:

None.

Xuint32 XTmrCtr_mReadReg(Xuint32 BaseAddress, Xuint8 TimerNumber, unsigned RegOffset)

```
#define XTmrCtr_mDisable( BaseAddress,  
                        TmrCtrNumber )
```

Disable a timer counter such that it stops running.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

void **XTmrCtr_mDisable**(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)

```
#define XTmrCtr_mDisableIntr( BaseAddress,  
                             TmrCtrNumber )
```

Disable the interrupt for a timer counter.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

void **XTmrCtr_mDisableIntr**(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)

```
#define XTmrCtr_mEnable( BaseAddress,  
                        TmrCtrNumber )
```

Enable a timer counter such that it starts running.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

void **XTmrCtr_mEnable**(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)

```
#define XTmrCtr_mEnableIntr( BaseAddress,  
                             TmrCtrNumber )
```

Enable the interrupt for a timer counter.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

void **XTmrCtr_mEnableIntr**(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)

```
#define XTmrCtr_mGetControlStatusReg( BaseAddress,  
                                     TmrCtrNumber )
```

Get the Control Status Register of a timer counter.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The value read from the register, a 32 bit value.

Note:

None.

Xuint32 **XTmrCtr_mGetControlStatusReg**(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)

```
#define XTmrCtr_mGetLoadReg( BaseAddress,  
                             TmrCtrNumber )
```

Get the Load Register of a timer counter.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The value read from the register, a 32 bit value.

Note:

None.

Xuint32 **XTmrCtr_mGetLoadReg**(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)

```
#define XTmrCtr_mGetTimerCounterReg( BaseAddress,  
                                     TmrCtrNumber )
```

Get the Timer Counter Register of a timer counter.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The value read from the register, a 32 bit value.

Note:

None.

Xuint32 **XTmrCtr_mGetTimerCounterReg**(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)

```
#define XTmrCtr_mHasEventOccurred( BaseAddress,  
                                    TmrCtrNumber )
```

Determine if a timer counter event has occurred. Events are defined to be when a capture has occurred or the counter has roller over.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Note:

None.

Xboolean **XTmrCtr_mHasEventOccurred**(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)

```
#define XTmrCtr_mLoadTimerCounterReg( BaseAddress,  
                                     TmrCtrNumber )
```

Cause the timer counter to load it's Timer Counter Register with the value in the Load Register.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

void **XTmrCtr_mLoadTimerCounterReg**(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)

```
#define XTmrCtr_mSetControlStatusReg( BaseAddress,  
                                     TmrCtrNumber,  
                                     RegisterValue )
```

Set the Control Status Register of a timer counter to the specified value.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

RegisterValue is the 32 bit value to be written to the register.

Returns:

None.

Note:

None.

```
void XTmrCtr_mSetControlStatusReg(Xuint32 BaseAddress, Xuint8 TmrCtrNumber, Xuint32 RegisterValue)
```

```
#define XTmrCtr_mSetLoadReg( BaseAddress,  
                             TmrCtrNumber,  
                             RegisterValue )
```

Set the Load Register of a timer counter to the specified value.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

RegisterValue is the 32 bit value to be written to the register.

Returns:

None.

Note:

None.

```
void XTmrCtr_mSetLoadReg(Xuint32 BaseAddress, Xuint8 TmrCtrNumber, Xuint32 RegisterValue)
```

```
#define XTmrCtr_mWriteReg( BaseAddress,  
                          TmrCtrNumber,  
                          RegOffset,  
                          ValueToWrite  )
```

Write a specified value to a register of a timer counter.

Parameters:

BaseAddress is the base address of the timer counter device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

RegOffset contain the offset from the 1st register of the timer counter to select the specific register of the timer counter.

ValueToWrite is the 32 bit value to be written to the register.

Returns:

None

```
void XTmrCtr_mWriteReg(Xuint32 BaseAddress, Xuint8 TimerNumber, unsigned RegOffset,  
Xuint32 ValueToWrite)
```

Generated on 24 Jun 2004 for Xilinx Device Drivers

XTmrCtrStats Struct Reference

```
#include <xtmrctr.h>
```

Detailed Description

Timer/Counter statistics

Data Fields

[Xuint32 Interrupts](#)

Field Documentation

[Xuint32 XTmrCtrStats::](#)

Interrupts

The number of interrupts that have occurred

The documentation for this struct was generated from the following file:

- tmrctr/v1_00_b/src/[xtmrctr.h](#)
-

Clear the **XTmrCtrStats** structure for this driver.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

Returns:

None.

Note:

None.

```
void  
XTmrCtr_GetStats ( XTmrCtr *  
                  InstancePtr,  
                  XTmrCtrStats * StatsPtr  
                  )
```

Get a copy of the **XTmrCtrStats** structure, which contains the current statistics for this driver.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

StatsPtr is a pointer to a **XTmrCtrStats** structure which will get a copy of current statistics.

Returns:

None.

Note:

None.

tmrctr/v1_00_b/src/xtmrctr_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of timer/counter devices in the system. Each timer/counter device should have an entry in this table.

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

```
#include "xtmrctr.h"  
#include "xparameters.h"
```

Variables

[XTmrCtr_Config](#) [XTmrCtr_ConfigTable](#) [XPAR_XTMRCTR_NUM_INSTANCES]

Variable Documentation

[XTmrCtr_Config](#) [XTmrCtr_ConfigTable](#)
[XPAR_XTMRCTR_NUM_INSTANCES]

The timer/counter configuration table, sized by the number of instances defined in [xparameters.h](#).

tmrctr/v1_00_b/src/xtmrctr_options.c File Reference

Detailed Description

Contains configuration options functions for the **XTmrCtr** component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	02/06/02	First release

```
#include "xbasic_types.h"  
#include "xtmrctr.h"  
#include "xtmrctr_i.h"
```

Data Structures

struct **Mapping**

Functions

```
void XTmrCtr_SetOptions (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber, Xuint32 Options)  
Xuint32 XTmrCtr_GetOptions (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
```

Function Documentation

```

Xuint32
XTmrCtr_GetOptions
    ( XTmrCtr *
      InstancePtr,
      Xuint8
      TmrCtrNumber
    )

```

Get the options for the specified timer counter.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The currently set options. An option which is set to a '1' is enabled and set to a '0' is disabled. The options are bit masks such that multiple options may be set or cleared. The options are described in [xtmrctr.h](#).

Note:

None.

```

void
XTmrCtr_SetOptions
    ( XTmrCtr *
      InstancePtr,
      Xuint8
      TmrCtrNumber,
      Xuint32
      Options
    )

```

Enables the specified options for the specified timer counter. This function sets the options without regard to the current options of the driver. To prevent a loss of the current options, the user should call **XTmrCtr_GetOptions()** prior to this function and modify the retrieved options to pass into this function to prevent loss of the current options.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Options

contains the desired options to be set or cleared. Setting the option to '1' enables the option, clearing the to '0' disables the option. The options are bit masks such that multiple options may be set or cleared. The options are described in [xtmrctr.h](#).

Returns:

None.

Note:

None.

tmrctr/v1_00_b/src/xtmrctr_intr.c File Reference

Detailed Description

Contains interrupt-related functions for the [XTmrCtr](#) component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	02/06/02	First release

```
#include "xbasic_types.h"  
#include "xtmrctr.h"  
#include "xtmrctr_1.h"
```

Functions

```
void XTmrCtr\_SetHandler (XTmrCtr *InstancePtr, XTmrCtr\_Handler FuncPtr, void  
    *CallbackRef)  
void XTmrCtr\_InterruptHandler (void *InstancePtr)
```

Function Documentation

```
void XTmrCtr\_InterruptHandler (void *  
    InstancePtr)
```

Interrupt Service Routine (ISR) for the driver. This function only performs processing for the device and does not save and restore the interrupt context.

Parameters:

InstancePtr contains a pointer to the timer/counter instance for the nterrupt.

Returns:

None.

Note:

None.

```
void XTmrCtr_SetHandler ( XTmrCtr * InstancePtr,
                          XTmrCtr_Handler FuncPtr,
                          void * CallbackRef
                        )
```

Sets the timer callback function, which the driver calls when the specified timer times out.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context so the function that is called should either be short or pass the more extensive processing off to another task to allow the interrupt to return and normal processing to continue.

tmrctr/v1_00_b/src/xtmrctr_selftest.c File Reference

Detailed Description

Contains diagnostic/self-test functions for the [XTmrCtr](#) component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	02/06/02	First release

```
#include "xbasic_types.h"
#include "xio.h"
#include "xtmrctr.h"
#include "xtmrctr_i.h"
```

Functions

[XStatus](#) [XTmrCtr_SelfTest](#) ([XTmrCtr](#) *InstancePtr, [Xuint8](#) TmrCtrNumber)

Function Documentation

```
XStatus
XTmrCtr_SelfTest
    ( XTmrCtr *
      InstancePtr,
      Xuint8
      TmrCtrNumber
    )
```

Runs a self-test on the driver/device. This test verifies that the specified timer counter of the device can be enabled and increments.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

XST_SUCCESS if self-test was successful, or XST_FAILURE if the timer is not incrementing.

Note:

This is a destructive test using the provided timer. The current settings of the timer are returned to the initialized values and all settings at the time this function is called are overwritten.

uartlite/v1_00_b/src/xuartlite_stats.c File Reference

Detailed Description

This file contains the statistics functions for the UART Lite component ([XUartLite](#)).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xbasic_types.h"  
#include "xuartlite.h"  
#include "xuartlite_i.h"
```

Functions

```
void XUartLite_GetStats (XUartLite *InstancePtr, XUartLite_Stats *StatsPtr)  
void XUartLite_ClearStats (XUartLite *InstancePtr)
```

Function Documentation

```
void XUartLite_ClearStats ( XUartLite * InstancePtr )
```

This function zeros the statistics for the given instance.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

```
void XUartLite_GetStats ( XUartLite * InstancePtr, XUartLite_Stats * StatsPtr )
```

Returns a snapshot of the current statistics in the structure specified.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

StatsPtr is a pointer to a XUartLiteStats structure to where the statistics are to be copied.

Returns:

None.

Note:

None.

XUartLite Struct Reference

```
#include <xuartlite.h>
```

Detailed Description

The XUartLite driver instance data. The user is required to allocate a variable of this type for every UART Lite device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [uartlite/v1_00_b/src/xuartlite.h](#)
-

uartlite/v1_00_b/src/xuartlite.h File Reference

Detailed Description

This component contains the implementation of the **XUartLite** component which is the driver for the Xilinx UART Lite device. This UART is a minimal hardware implementation with minimal features. Most of the features, including baud rate, parity, and number of data bits are only configurable when the hardware device is built, rather than at run time by software.

The device has 16 byte transmit and receive FIFOs and supports interrupts. The device does not have any way to disable the receiver such that the receive FIFO may contain unwanted data. The FIFOs are not flushed when the driver is initialized, but a function is provided to allow the user to reset the FIFOs if desired.

The driver defaults to no interrupts at initialization such that interrupts must be enabled if desired. An interrupt is generated when the transmit FIFO transitions from having data to being empty or when any data is contained in the receive FIFO.

In order to use interrupts, it's necessary for the user to connect the driver interrupt handler, XUartLite_InterruptHandler, to the interrupt system of the application. This function does not save and restore the processor context such that the user must provide it. Send and receive handlers may be set for the driver such that the handlers are called when transmit and receive interrupts occur. The handlers are called from interrupt context and are designed to allow application specific processing to be performed.

The functions, XUartLite_Send and XUartLite_Recv, are provided in the driver to allow data to be sent and received. They are designed to be used in polled or interrupt modes.

The driver provides a status for each received byte indicating any parity frame or overrun error. The driver provides statistics which allow visibility into these errors.

RTOS Independence

This driver is intended to be RTOS and processor independent. It works with physical addresses only.

Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

Note:

The driver is partitioned such that a minimal implementation may be used. More features require additional files to be linked in.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

Data Structures

```
struct XUartLite  
struct XUartLite_Buffer  
struct XUartLite_Config  
struct XUartLite_Stats
```

Typedefs

```
typedef void(* XUartLite_Handler)(void *CallbackRef, unsigned int ByteCount)
```

Functions

```
XStatus XUartLite_Initialize (XUartLite *InstancePtr, Xuint16 DeviceId)  
void XUartLite_ResetFifos (XUartLite *InstancePtr)  
unsigned int XUartLite_Send (XUartLite *InstancePtr, Xuint8 *DataBufferPtr, unsigned int  
    NumBytes)  
unsigned int XUartLite_Recv (XUartLite *InstancePtr, Xuint8 *DataBufferPtr, unsigned int  
    NumBytes)  
Xboolean XUartLite_IsSending (XUartLite *InstancePtr)  
void XUartLite_GetStats (XUartLite *InstancePtr, XUartLite_Stats *StatsPtr)
```

```
void XUartLite_ClearStats (XUartLite *InstancePtr)
XStatus XUartLite_SelfTest (XUartLite *InstancePtr)
void XUartLite_EnableInterrupt (XUartLite *InstancePtr)
void XUartLite_DisableInterrupt (XUartLite *InstancePtr)
void XUartLite_SetRecvHandler (XUartLite *InstancePtr, XUartLite_Handler FuncPtr,
void *CallbackRef)
void XUartLite_SetSendHandler (XUartLite *InstancePtr, XUartLite_Handler FuncPtr,
void *CallbackRef)
void XUartLite_InterruptHandler (XUartLite *InstancePtr)
```

Typedef Documentation

```
typedef void(* XUartLite_Handler)(void *CallbackRef, unsigned int
ByteCount)
```

Callback function. The first argument is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked. The second argument is the ByteCount which is the number of bytes that actually moved from/to the buffer provided in the _Send/_Receive call.

Function Documentation

```
void XUartLite_ClearStats ( XUartLite * InstancePtr )
```

This function zeros the statistics for the given instance.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

```
void XUartLite_DisableInterrupt ( XUartLite * InstancePtr )
```

This function disables the UART interrupt. After calling this function, data may still be received by the UART but no interrupt will be generated since the hardware device has no way to disable the receiver.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

```
void XUartLite_EnableInterrupt ( XUartLite * InstancePtr )
```

This function enables the UART interrupt such that an interrupt will occur when data is received or data has been transmitted. The device contains 16 byte receive and transmit FIFOs such that an interrupt is generated anytime there is data in the receive FIFO and when the transmit FIFO transitions from not empty to empty.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

```
void XUartLite_GetStats ( XUartLite * InstancePtr, XUartLite_Stats * StatsPtr )
```

Returns a snapshot of the current statistics in the structure specified.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

StatsPtr is a pointer to a XUartLiteStats structure to where the statistics are to be copied.

Returns:

None.

Note:

None.

```
XStatus XUartLite_Initialize(XUartLite *InstancePtr,  
Xuint16 DeviceId)
```

Initialize a **XUartLite** instance. The receive and transmit FIFOs of the UART are not flushed, so the user may want to flush them. The hardware device does not have any way to disable the receiver such that any valid data may be present in the receive FIFO. This function disables the UART interrupt. The baudrate and format of the data are fixed in the hardware at hardware build time.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XUartLite** instance. Passing in a device id associates the generic **XUartLite** instance to a specific device, as chosen by the caller or application developer.

Returns:

- o XST_SUCCESS if everything starts up as expected.
- o XST_DEVICE_NOT_FOUND if the device is not found in the configuration table.

Note:

None.

```
void ( XUartLite * )
XUartLite_InterruptHandler InstancePtr
```

This function is the interrupt handler for the UART lite driver. It must be connected to an interrupt system by the user such that it is called when an interrupt for any UART lite occurs. This function does not save or restore the processor context such that the user must ensure this occurs.

Parameters:

InstancePtr contains a pointer to the instance of the UART that the interrupt is for.

Returns:

None.

Note:

None.

```
Xboolean ( XUartLite * )
XUartLite_IsSending InstancePtr
```

This function determines if the specified UART is sending data. If the transmitter register is not empty, it is sending data.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

A value of XTRUE if the UART is sending data, otherwise XFALSE.

Note:

None.

```
unsigned int ( XUartLite * InstancePtr,
XUartLite_Recv XUint8 * DataBufferPtr,
unsigned int NumBytes )
```

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer in the FIFO. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue receiving data until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

BufferPtr is pointer to buffer for data to be received into

NumBytes is the number of bytes to be received. A value of zero will stop a previous receive operation that is in progress in interrupt mode.

Returns:

The number of bytes received.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

```
void XUartLite_ResetFifos ( XUartLite * InstancePtr )
```

This function resets the FIFOs, both transmit and receive, of the UART such that they are emptied. Since the UART does not have any way to disable it from receiving data, it may be necessary for the application to reset the FIFOs to get rid of any unwanted data.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( XUartLite * )  
XUartLite_SelfTest InstancePtr
```

Runs a self-test on the device hardware. Since there is no way to perform a loopback in the hardware, this test can only check the state of the status register to verify it is correct. This test assumes that the hardware device is still in its reset state, but has been initialized with the Initialize function.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

- XST_SUCCESS if the self-test was successful.
- XST_FAILURE if the self-test failed, the status register value was not correct

Note:

None.

```
unsigned int ( XUartLite * InstancePtr,  
XUartLite_Send Xuint8 * DataBufferPtr,  
unsigned int NumBytes )
```

This functions sends the specified buffer of data using the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART. If the UART is busy sending data, it will return and indicate zero bytes were sent.

In a polled mode, this function will only send as much data as the UART can buffer in the FIFO. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue sending data until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

BufferPtr is pointer to a buffer of data to be sent.
NumBytes contains the number of bytes to be sent. A value of zero will stop a previous send operation that is in progress in interrupt mode. Any data that was already put into the transmit FIFO will be sent.

Returns:

The number of bytes actually sent.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

```
void XUartLite_SetRecvHandler ( XUartLite * InstancePtr,  
                               XUartLite_Handler FuncPtr,  
                               void * CallbackRef  
                               )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

```
void XUartLite_SetSendHandler ( XUartLite * InstancePtr,  
                               XUartLite_Handler FuncPtr,  
                               void * CallbackRef  
                               )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

XUartLite_Buffer Struct Reference

```
#include <xuartlite.h>
```

Detailed Description

The following data type is used to manage the buffers that are handled when sending and receiving data in the interrupt mode. It is intended for internal use only.

The documentation for this struct was generated from the following file:

- [uartlite/v1_00_b/src/xuartlite.h](#)
-

XUartLite_Config Struct Reference

```
#include <xuartlite.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 RegBaseAddr](#)

[Xuint32 BaudRate](#)

[Xuint8 UseParity](#)

[Xuint8 ParityOdd](#)

[Xuint8 DataBits](#)

Field Documentation

[Xuint32 XUartLite_Config::](#)

BaudRate

Fixed baud
rate

[Xuint8 XUartLite_Config::](#)

DataBits

Fixed data
bits

Xuint16 XUartLite_Config::

DeviceId

Unique ID of
device

Xuint8 XUartLite_Config::

ParityOdd

Parity generated is odd when XTRUE, even when
XFALSE

Xuint32 XUartLite_Config::

RegBaseAddr

Register base
address

Xuint8 XUartLite_Config::

UseParity

Parity generator enabled when
XTRUE

The documentation for this struct was generated from the following file:

- [uartlite/v1_00_b/src/xuartlite.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

XUartLite_Stats Struct Reference

```
#include <xuartlite.h>
```

Detailed Description

Statistics for the [XUartLite](#) driver

Data Fields

[Xuint32 TransmitInterrupts](#)

[Xuint32 ReceiveInterrupts](#)

[Xuint32 CharactersTransmitted](#)

[Xuint32 CharactersReceived](#)

[Xuint32 ReceiveOverrunErrors](#)

[Xuint32 ReceiveParityErrors](#)

[Xuint32 ReceiveFramingErrors](#)

Field Documentation

Xuint32 XUartLite_Stats::

CharactersReceived

Number of characters
received

Xuint32 XUartLite_Stats::

CharactersTransmitted

Number of characters
transmitted

Xuint32 XUartLite_Stats::

ReceiveFramingErrors

Number of receive framing errors

Xuint32 XUartLite_Stats::

ReceiveInterrupts

Number of receive interrupts

Xuint32 XUartLite_Stats::

ReceiveOverrunErrors

Number of receive overruns

Xuint32 XUartLite_Stats::

ReceiveParityErrors

Number of receive parity errors

Xuint32 XUartLite_Stats::

TransmitInterrupts

Number of transmit interrupts

The documentation for this struct was generated from the following file:

- [uartlite/v1_00_b/src/xuartlite.h](#)

uartlite/v1_00_b/src/xuartlite_i.h File Reference

Detailed Description

Contains data which is shared between the files of the **XUartLite** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files
1.00b	rpm	04/24/02	Moved register definitions to xuartlite_1.h and updated macro naming convention

```
#include "xuartlite.h"  
#include "xuartlite\_1.h"
```

Functions

unsigned int [XUartLite_SendBuffer](#) ([XUartLite](#) *InstancePtr)
unsigned int [XUartLite_ReceiveBuffer](#) ([XUartLite](#) *InstancePtr)

Variables

[XUartLite_Config XUartLite_ConfigTable](#) []

Function Documentation

unsigned int (**XUartLite ***)
XUartLite_ReceiveBuffer *InstancePtr*

This function receives a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartLite** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if there is no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer, either in the receiver or in the FIFO if present and enabled. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

The number of bytes received.

Note:

None.

unsigned int (**XUartLite ***)
XUartLite_SendBuffer *InstancePtr*

This function sends a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartLite** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function sends the specified buffer of data to the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART.

In a polled mode, this function will only send as much data as the UART can buffer, either in the transmitter or in the FIFO if present and enabled. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

NumBytes is the number of bytes actually sent (put into the UART transmitter and/or FIFO).

Note:

None.

Variable Documentation

XUartLite_Config XUartLite_ConfigTable ()

[]

The configuration table for UART Lite devices

uartlite/v1_00_b/src/xuartlite_i.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xuartlite.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b rpm		04/25/02	First release
1.00b rpm		07/07/03	Removed references to XUartLite_mGetControlReg macro since the control register is write-only

```
#include "xbasic_types.h"  
#include "xio.h"
```

Defines

```
#define XUartLite_mSetControlReg(BaseAddress, Mask)  
#define XUartLite_mGetStatusReg(BaseAddress)  
#define XUartLite_mIsReceiveEmpty(BaseAddress)  
#define XUartLite_mIsTransmitFull(BaseAddress)  
#define XUartLite_mIsIntrEnabled(BaseAddress)  
#define XUartLite_mEnableIntr(BaseAddress)  
#define XUartLite_mDisableIntr(BaseAddress)
```

Functions

void **XUartLite_SendByte** (**Xuint32** BaseAddress, **Xuint8** Data)

Xuint8 XUartLite_RecvByte (**Xuint32** BaseAddress)

Define Documentation

```
#define XUartLite_mDisableIntr( BaseAddress )
```

Disable the device interrupt. We cannot read the control register, so we just clear all bits. Since the only other ones are the FIFO reset bits, this works without side effects.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XUartLite_mEnableIntr( BaseAddress )
```

Enable the device interrupt. We cannot read the control register, so we just write the enable interrupt bit and clear all others. Since the only other ones are the FIFO reset bits, this works without side effects.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

#define XUartLite_mGetStatusReg(BaseAddress)

Get the contents of the status register. Use the XUL_SR_* constants defined above to interpret the bit-mask returned.

Parameters:

BaseAddress is the base address of the device

Returns:

A 32-bit value representing the contents of the status register.

Note:

None.

#define XUartLite_mIsIntrEnabled(BaseAddress)

Check to see if the interrupt is enabled.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if the interrupt is enabled, XFALSE otherwise.

Note:

None.

#define XUartLite_mIsReceiveEmpty(BaseAddress)

Check to see if the receiver has data.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if the receiver is empty, XFALSE if there is data present.

Note:

None.

```
#define XUartLite_mIsTransmitFull( BaseAddress )
```

Check to see if the transmitter is full.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if the transmitter is full, XFALSE otherwise.

Note:

None.

```
#define XUartLite_mSetControlReg( BaseAddress,  
                                Mask      )
```

Set the contents of the control register. Use the XUL_CR_* constants defined above to create the bit-mask to be written to the register.

Parameters:

BaseAddress is the base address of the device

Mask is the 32-bit value to write to the control register

Returns:

None.

Note:

None.

Function Documentation

```
Xuint8  
XUartLite_RecvByte ( Xuint32  
                    BaseAddress )
```

This functions receives a single byte using the UART. It is blocking in that it waits for the receiver to become non-empty before it reads from the receive register.

Parameters:

BaseAddress is the base address of the device

Returns:

The byte of data received.

Note:

None.

```
void XUartLite_SendByte ( Xuint32 BaseAddress, Xuint8 Data )
```

This functions sends a single byte using the UART. It is blocking in that it waits for the transmitter to become non-full before it writes the byte to the transmit register.

Parameters:

BaseAddress is the base address of the device

Data is the byte of data to send

Returns:

None.

Note:

None.

uartlite/v1_00_b/src/xuartlite_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of UART Lite devices in the system. Each device in the system should have an entry in the table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xuartlite.h"  
#include "xparameters.h"
```

Variables

[XUartLite_Config XUartLite_ConfigTable](#) [XPAR_XUARTLITE_NUM_INSTANCES]

Variable Documentation

[XUartLite_Config XUartLite_ConfigTable](#)
[XPAR_XUARTLITE_NUM_INSTANCES]

The configuration table for UART Lite devices

uartlite/v1_00_b/src/xuartlite_intr.c File Reference

Detailed Description

This file contains interrupt-related functions for the UART Lite component (**XUartLite**).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xbasic_types.h"  
#include "xuartlite.h"  
#include "xuartlite_i.h"  
#include "xio.h"
```

Functions

```
void XUartLite_SetRecvHandler (XUartLite *InstancePtr, XUartLite_Handler FuncPtr, void  
    *CallbackRef)  
void XUartLite_SetSendHandler (XUartLite *InstancePtr, XUartLite_Handler FuncPtr, void  
    *CallbackRef)  
void XUartLite_InterruptHandler (XUartLite *InstancePtr)  
void XUartLite_DisableInterrupt (XUartLite *InstancePtr)  
void XUartLite_EnableInterrupt (XUartLite *InstancePtr)
```

Function Documentation

```
void ( XUartLite * )
XUartLite_DisableInterrupt InstancePtr
```

This function disables the UART interrupt. After calling this function, data may still be received by the UART but no interrupt will be generated since the hardware device has no way to disable the receiver.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XUartLite * )
XUartLite_EnableInterrupt InstancePtr
```

This function enables the UART interrupt such that an interrupt will occur when data is received or data has been transmitted. The device contains 16 byte receive and transmit FIFOs such that an interrupt is generated anytime there is data in the receive FIFO and when the transmit FIFO transitions from not empty to empty.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XUartLite * )
XUartLite_InterruptHandler InstancePtr
```

This function is the interrupt handler for the UART lite driver. It must be connected to an interrupt system by the user such that it is called when an interrupt for any UART lite occurs. This function does not save or restore the processor context such that the user must ensure this occurs.

Parameters:

InstancePtr contains a pointer to the instance of the UART that the interrupt is for.

Returns:

None.

Note:

None.

```
void XUartLite_SetRecvHandler ( XUartLite * InstancePtr,
                               XUartLite_Handler FuncPtr,
                               void * CallbackRef
                               )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

```
void XUartLite_SetSendHandler ( XUartLite * InstancePtr,
                               XUartLite_Handler FuncPtr,
                               void * CallbackRef
                               )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

uartlite/v1_00_b/src/xuartlite.c File Reference

Detailed Description

Contains required functions for the **XUartLite** driver. See the **xuartlite.h** header file for more details on this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files
1.00b	rmm	05/13/03	Fixed diab compiler warnings relating to asserts

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xparameters.h"  
#include "xuartlite.h"  
#include "xuartlite_i.h"  
#include "xio.h"
```

Functions

XStatus XUartLite_Initialize (**XUartLite** *InstancePtr, **Xuint16** DeviceId)

unsigned int **XUartLite_Send** (**XUartLite** *InstancePtr, **Xuint8** *DataBufferPtr, unsigned int NumBytes)

unsigned int **XUartLite_Recv** (**XUartLite** *InstancePtr, **Xuint8** *DataBufferPtr, unsigned int NumBytes)

void **XUartLite_ResetFifos** (**XUartLite** *InstancePtr)

Xboolean XUartLite_IsSending (**XUartLite** *InstancePtr)

unsigned int **XUartLite_SendBuffer** (**XUartLite** *InstancePtr)
unsigned int **XUartLite_ReceiveBuffer** (**XUartLite** *InstancePtr)

Function Documentation

```
XStatus ( XUartLite *  
XUartLite_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize a **XUartLite** instance. The receive and transmit FIFOs of the UART are not flushed, so the user may want to flush them. The hardware device does not have any way to disable the receiver such that any valid data may be present in the receive FIFO. This function disables the UART interrupt. The baudrate and format of the data are fixed in the hardware at hardware build time.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XUartLite** instance. Passing in a device id associates the generic **XUartLite** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if everything starts up as expected.
- XST_DEVICE_NOT_FOUND if the device is not found in the configuration table.

Note:

None.

```
Xboolean ( XUartLite *  
XUartLite_IsSending InstancePtr  
 )
```

This function determines if the specified UART is sending data. If the transmitter register is not empty, it is sending data.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

A value of XTRUE if the UART is sending data, otherwise XFALSE.

Note:

None.

```
unsigned int ( XUartLite * )
XUartLite_ReceiveBuffer InstancePtr
```

This function receives a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartLite** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if there is no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer, either in the receiver or in the FIFO if present and enabled. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

The number of bytes received.

Note:

None.

```
unsigned int          ( XUartLite *  
XUartLite_Recv      InstancePtr,  
  
                    Xuint8 *  
                    DataBufferPtr,  
                    unsigned int NumBytes  
                    )
```

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer in the FIFO. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue receiving data until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

BufferPtr is pointer to buffer for data to be received into

NumBytes is the number of bytes to be received. A value of zero will stop a previous receive operation that is in progress in interrupt mode.

Returns:

The number of bytes received.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

```
void          ( XUartLite *  
XUartLite_ResetFifos InstancePtr
```

This function resets the FIFOs, both transmit and receive, of the UART such that they are emptied. Since the UART does not have any way to disable it from receiving data, it may be necessary for the application to reset the FIFOs to get rid of any unwanted data.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

```
unsigned int          ( XUartLite *  
XUartLite_Send      InstancePtr,  
  
                    XUint8 *  
                    DataBufferPtr,  
                    unsigned int NumBytes  
                    )
```

This functions sends the specified buffer of data using the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART. If the UART is busy sending data, it will return and indicate zero bytes were sent.

In a polled mode, this function will only send as much data as the UART can buffer in the FIFO. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue sending data until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

BufferPtr is pointer to a buffer of data to be sent.

NumBytes contains the number of bytes to be sent. A value of zero will stop a previous send operation that is in progress in interrupt mode. Any data that was already put into the transmit FIFO will be sent.

Returns:

The number of bytes actually sent.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

unsigned int (**XUartLite ***)
XUartLite_SendBuffer *InstancePtr*

This function sends a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartLite** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function sends the specified buffer of data to the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART.

In a polled mode, this function will only send as much data as the UART can buffer, either in the transmitter or in the FIFO if present and enabled. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

NumBytes is the number of bytes actually sent (put into the UART transmitter and/or FIFO).

Note:

None.

uartlite/v1_00_b/src/xuartlite_1.c File Reference

Detailed Description

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.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	rpm	04/25/02	First release

```
#include "xuartlite_1.h"
```

Functions

```
void XUartLite_SendByte (Xuint32 BaseAddress, Xuint8 Data)  
Xuint8 XUartLite_RecvByte (Xuint32 BaseAddress)
```

Function Documentation

```
Xuint8  
XUartLite_RecvByte ( Xuint32  
                    BaseAddress )
```

This functions receives a single byte using the UART. It is blocking in that it waits for the receiver to become non-empty before it reads from the receive register.

Parameters:

BaseAddress is the base address of the device

Returns:

The byte of data received.

Note:

None.

```
void XUartLite_SendByte ( Xuint32 BaseAddress, Xuint8 Data )
```

This functions sends a single byte using the UART. It is blocking in that it waits for the transmitter to become non-full before it writes the byte to the transmit register.

Parameters:

BaseAddress is the base address of the device

Data is the byte of data to send

Returns:

None.

Note:

None.

uartlite/v1_00_b/src/xuartlite_selftest.c File Reference

Detailed Description

This file contains the self-test functions for the UART Lite component ([XUartLite](#)).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xuartlite.h"  
#include "xuartlite_i.h"  
#include "xio.h"
```

Functions

[XStatus XUartLite_SelfTest](#) ([XUartLite](#) *InstancePtr)

Function Documentation

XStatus	(XUartLite *)
XUartLite_SelfTest	<i>InstancePtr</i>

Runs a self-test on the device hardware. Since there is no way to perform a loopback in the hardware, this test can only check the state of the status register to verify it is correct. This test assumes that the hardware device is still in its reset state, but has been initialized with the Initialize function.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

- XST_SUCCESS if the self-test was successful.
- XST_FAILURE if the self-test failed, the status register value was not correct

Note:

None.

uartns550/v1_00_b/src/xuartns550_stats.c File Reference

Detailed Description

This file contains the statistics functions for the 16450/16550 UART driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.

```
#include "xuartns550.h"  
#include "xuartns550_i.h"
```

Functions

```
void XUartNs550_GetStats (XUartNs550 *InstancePtr, XUartNs550Stats *StatsPtr)  
void XUartNs550_ClearStats (XUartNs550 *InstancePtr)
```

Function Documentation

```
void XUartNs550_ClearStats ( XUartNs550 * InstancePtr )
```

This function zeros the statistics for the given instance.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

None.

Note:

None.

```
void XUartNs550_GetStats ( XUartNs550 * InstancePtr, XUartNs550Stats * StatsPtr )
```

This functions returns a snapshot of the current statistics in the area provided.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

StatsPtr is a pointer to a **XUartNs550Stats** structure to where the statistics are to be copied to.

Returns:

None.

Note:

None.

uartns550/v1_00_b/src/xuartns550.h File Reference

Detailed Description

This driver supports the following features in the Xilinx 16450/16550 compatible UART.

- Dynamic data format (baud rate, data bits, stop bits, parity)
- Polled mode
- Interrupt driven mode
- Transmit and receive FIFOs (16 bytes each for the 16550)
- Access to the external modem control lines and the two discrete outputs

The only difference between the 16450 and the 16550 is the addition of transmit and receive FIFOs in the 16550.

Baud Rate

The UART has an internal baud rate generator that is clocked at a specified input clock frequency. Not all baud rates can be generated from some clock frequencies. The requested baud rate is checked using the provided clock for the system, and checked against the acceptable error range. An error may be returned from some functions indicating the baud rate was in error because it could not be generated.

Interrupts

The device does not have any way to disable the receiver such that the receive FIFO may contain unwanted data. The FIFOs are not flushed when the driver is initialized, but a function is provided to allow the user to reset the FIFOs if desired.

The driver defaults to no interrupts at initialization such that interrupts must be enabled if desired. An interrupt is generated for any of the following conditions.

- Transmit FIFO is empty
- Data in the receive FIFO equal to the receive threshold

- Data in the receiver when FIFOs are disabled
- Any receive status error or break condition detected
- Data in the receive FIFO for 4 character times without receiver activity
- A change of a modem signal

The application can control which interrupts are enabled using the SetOptions function.

In order to use interrupts, it is necessary for the user to connect the driver interrupt handler, **XUartNs550_InterruptHandler()**, to the interrupt system of the application. This function does not save and restore the processor context such that the user must provide it. A handler must be set for the driver such that the handler is called when interrupt events occur. The handler is called from interrupt context and is designed to allow application specific processing to be performed.

The functions, **XUartNs550_Send()** and **XUartNs550_Recv()**, are provided in the driver to allow data to be sent and received. They are designed to be used in polled or interrupt modes.

Note:

The default configuration for the UART after initialization is:

- 19,200 bps or XPAR_DEFAULT_BAUD_RATE if defined
- 8 data bits
- 1 stop bit
- no parity
- FIFO's are enabled with a receive threshold of 8 bytes

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	03/11/02	Repartitioned the driver for smaller files.

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xuartns550_1.h"
```

Data Structures

- struct **XUartNs550**
- struct **XUartNs550_Config**

```
struct XUartNs550Buffer
struct XUartNs550Format
struct XUartNs550Stats
```

Configuration options

```
#define XUN_OPTION_SET_BREAK
#define XUN_OPTION_LOOPBACK
#define XUN_OPTION_DATA_INTR
#define XUN_OPTION_MODEM_INTR
#define XUN_OPTION_FIFOS_ENABLE
#define XUN_OPTION_RESET_TX_FIFO
#define XUN_OPTION_RESET_RX_FIFO
#define XUN_OPTION_ASSERT_OUT2
#define XUN_OPTION_ASSERT_OUT1
#define XUN_OPTION_ASSERT_RTS
#define XUN_OPTION_ASSERT_DTR
```

Data format values

```
#define XUN_FORMAT_8_BITS
#define XUN_FORMAT_7_BITS
#define XUN_FORMAT_6_BITS
#define XUN_FORMAT_5_BITS
#define XUN_FORMAT_EVEN_PARITY
#define XUN_FORMAT_ODD_PARITY
#define XUN_FORMAT_NO_PARITY
#define XUN_FORMAT_2_STOP_BIT
#define XUN_FORMAT_1_STOP_BIT
```

Modem status values

```
#define XUN_MODEM_DCD_DELTA_MASK
#define XUN_MODEM_DSR_DELTA_MASK
#define XUN_MODEM_CTS_DELTA_MASK
#define XUN_MODEM_RINGING_MASK
#define XUN_MODEM_DSR_MASK
#define XUN_MODEM_CTS_MASK
#define XUN_MODEM_DCD_MASK
```

```
#define XUN_MODEM_RING_STOP_MASK
```

Callback events

```
#define XUN_EVENT_RECV_DATA  
#define XUN_EVENT_RECV_TIMEOUT  
#define XUN_EVENT_SENT_DATA  
#define XUN_EVENT_RECV_ERROR  
#define XUN_EVENT_MODEM
```

Error values

```
#define XUN_ERROR_BREAK_MASK  
#define XUN_ERROR_FRAMING_MASK  
#define XUN_ERROR_PARITY_MASK  
#define XUN_ERROR_OVERRUN_MASK  
#define XUN_ERROR_NONE
```

Typedefs

```
typedef void(* XUartNs550_Handler )(void *CallBackRef, Xuint32 Event, unsigned int  
    EventData)
```

Functions

```
XStatus XUartNs550_Initialize (XUartNs550 *InstancePtr, Xuint16 DeviceId)
```

```
unsigned int XUartNs550_Send (XUartNs550 *InstancePtr, Xuint8 *BufferPtr, unsigned  
    int NumBytes)
```

```
unsigned int XUartNs550_Recv (XUartNs550 *InstancePtr, Xuint8 *BufferPtr, unsigned  
    int NumBytes)
```

```
XUartNs550_Config * XUartNs550_LookupConfig (Xuint16 DeviceId)
```

```
XStatus XUartNs550_SetOptions (XUartNs550 *InstancePtr, Xuint16 Options)
```

```
Xuint16 XUartNs550_GetOptions (XUartNs550 *InstancePtr)
```

```
XStatus XUartNs550_SetFifoThreshold (XUartNs550 *InstancePtr, Xuint8  
    TriggerLevel)
```

```
Xuint8 XUartNs550_GetFifoThreshold (XUartNs550 *InstancePtr)
```

```
Xboolean XUartNs550_IsSending (XUartNs550 *InstancePtr)
```

```
Xuint8 XUartNs550_GetLastErrors (XUartNs550 *InstancePtr)
```

```
Xuint8 XUartNs550_GetModemStatus (XUartNs550 *InstancePtr)
```

```
XStatus XUartNs550_SetDataFormat (XUartNs550 *InstancePtr,  
    XUartNs550Format *Format)  
void XUartNs550_GetDataFormat (XUartNs550 *InstancePtr,  
    XUartNs550Format *Format)  
void XUartNs550_SetHandler (XUartNs550 *InstancePtr, XUartNs550_Handler  
    FuncPtr, void *CallbackRef)  
void XUartNs550_InterruptHandler (XUartNs550 *InstancePtr)  
void XUartNs550_GetStats (XUartNs550 *InstancePtr, XUartNs550Stats  
    *StatsPtr)  
void XUartNs550_ClearStats (XUartNs550 *InstancePtr)  
XStatus XUartNs550_SelfTest (XUartNs550 *InstancePtr)
```

Define Documentation

#define

XUN_ERROR_BREAK_MASK

These constants specify the errors that may be retrieved from the driver using the XUartNs550_GetLastErrors function. All of them are bit masks, except no error, such that multiple errors may be specified.

XUN_ERROR_BREAK_MASK	Break detected
XUN_ERROR_FRAMING_MASK	Receive framing error
XUN_ERROR_PARITY_MASK	Receive parity error
XUN_ERROR_OVERRUN_MASK	Receive overrun error
XUN_ERROR_NONE	No error

#define

XUN_ERROR_FRAMING_MASK

These constants specify the errors that may be retrieved from the driver using the XUartNs550_GetLastErrors function. All of them are bit masks, except no error, such that multiple errors may be specified.

XUN_ERROR_BREAK_MASK	Break detected
XUN_ERROR_FRAMING_MASK	Receive framing error
XUN_ERROR_PARITY_MASK	Receive parity error
XUN_ERROR_OVERRUN_MASK	Receive overrun error
XUN_ERROR_NONE	No error

#define

XUN_ERROR_NONE

These constants specify the errors that may be retrieved from the driver using the XUartNs550_GetLastErrors function. All of them are bit masks, except no error, such that multiple errors may be specified.

XUN_ERROR_BREAK_MASK	Break detected
XUN_ERROR_FRAMING_MASK	Receive framing error
XUN_ERROR_PARITY_MASK	Receive parity error
XUN_ERROR_OVERRUN_MASK	Receive overrun error
XUN_ERROR_NONE	No error

#define

XUN_ERROR_OVERRUN_MASK

These constants specify the errors that may be retrieved from the driver using the XUartNs550_GetLastErrors function. All of them are bit masks, except no error, such that multiple errors may be specified.

XUN_ERROR_BREAK_MASK	Break detected
XUN_ERROR_FRAMING_MASK	Receive framing error
XUN_ERROR_PARITY_MASK	Receive parity error
XUN_ERROR_OVERRUN_MASK	Receive overrun error
XUN_ERROR_NONE	No error

#define

XUN_ERROR_PARITY_MASK

These constants specify the errors that may be retrieved from the driver using the XUARTNs550_GetLastErrors function. All of them are bit masks, except no error, such that multiple errors may be specified.

XUN_ERROR_BREAK_MASK	Break detected
XUN_ERROR_FRAMING_MASK	Receive framing error
XUN_ERROR_PARITY_MASK	Receive parity error
XUN_ERROR_OVERRUN_MASK	Receive overrun error
XUN_ERROR_NONE	No error

#define

XUN_EVENT_MODEM

These constants specify the handler events that are passed to a handler from the driver. These constants are not bit masks such that only one will be passed at a time to the handler.

XUN_EVENT_RECV_DATA	Data has been received
XUN_EVENT_RECV_TIMEOUT	A receive timeout occurred
XUN_EVENT_SENT_DATA	Data has been sent
XUN_EVENT_RECV_ERROR	A receive error was detected
XUN_EVENT_MODEM	A change in modem status

#define

XUN_EVENT_RECV_DATA

These constants specify the handler events that are passed to a handler from the driver. These constants are not bit masks such that only one will be passed at a time to the handler.

XUN_EVENT_RECV_DATA	Data has been received
XUN_EVENT_RECV_TIMEOUT	A receive timeout occurred
XUN_EVENT_SENT_DATA	Data has been sent
XUN_EVENT_RECV_ERROR	A receive error was detected
XUN_EVENT_MODEM	A change in modem status

#define

XUN_EVENT_RECV_ERROR

These constants specify the handler events that are passed to a handler from the driver. These constants are not bit masks such that only one will be passed at a time to the handler.

XUN_EVENT_RECV_DATA	Data has been received
XUN_EVENT_RECV_TIMEOUT	A receive timeout occurred
XUN_EVENT_SENT_DATA	Data has been sent
XUN_EVENT_RECV_ERROR	A receive error was detected
XUN_EVENT_MODEM	A change in modem status

#define

XUN_EVENT_RECV_TIMEOUT

These constants specify the handler events that are passed to a handler from the driver. These constants are not bit masks such that only one will be passed at a time to the handler.

XUN_EVENT_RECV_DATA	Data has been received
XUN_EVENT_RECV_TIMEOUT	A receive timeout occurred
XUN_EVENT_SENT_DATA	Data has been sent
XUN_EVENT_RECV_ERROR	A receive error was detected
XUN_EVENT_MODEM	A change in modem status

#define

XUN_EVENT_SENT_DATA

These constants specify the handler events that are passed to a handler from the driver. These constants are not bit masks such that only one will be passed at a time to the handler.

XUN_EVENT_RECV_DATA	Data has been received
XUN_EVENT_RECV_TIMEOUT	A receive timeout occurred
XUN_EVENT_SENT_DATA	Data has been sent
XUN_EVENT_RECV_ERROR	A receive error was detected
XUN_EVENT_MODEM	A change in modem status

#define

XUN_FORMAT_1_STOP_BIT

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define

XUN_FORMAT_2_STOP_BIT

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define

XUN_FORMAT_5_BITS

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define

XUN_FORMAT_6_BITS

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define

XUN_FORMAT_7_BITS

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define

XUN_FORMAT_8_BITS

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define

XUN_FORMAT_EVEN_PARITY

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define

XUN_FORMAT_NO_PARITY

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define

XUN_FORMAT_ODD_PARITY

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define

XUN_MODEM_CTS_DELTA_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringling has stopped

#define

XUN_MODEM_CTS_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringing has stopped

#define

XUN_MODEM_DCD_DELTA_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringing has stopped

#define

XUN_MODEM_DCD_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringing has stopped

#define

XUN_MODEM_DSR_DELTA_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringling has stopped

#define

XUN_MODEM_DSR_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringling has stopped

#define

XUN_MODEM_RING_STOP_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringling has stopped

#define

XUN_MODEM_RINGING_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringling has stopped

#define

XUN_OPTION_ASSERT_DTR

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal

XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define

XUN_OPTION_ASSERT_OUT1

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define

XUN_OPTION_ASSERT_OUT2

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal

XUN_OPTION_ASSERT_DTR Assert DTR signal

#define

XUN_OPTION_ASSERT_RTS

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define

XUN_OPTION_DATA_INTR

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define

XUN_OPTION_FIFOS_ENABLE

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define

XUN_OPTION_LOOPBACK

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define

XUN_OPTION_MODEM_INTR

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define

XUN_OPTION_RESET_RX_FIFO

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define

XUN_OPTION_RESET_TX_FIFO

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define

XUN_OPTION_SET_BREAK

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

```
typedef void(* XUartNs550_Handler)(void *CallbackRef, Xuint32 Event, unsigned int
EventData)
```

This data type defines a handler which the application must define when using interrupt mode. The handler will be called from the driver in an interrupt context to handle application specific processing.

Parameters:

- CallbackRef* is a callback reference passed in by the upper layer when setting the handler, and is passed back to the upper layer when the handler is called.
- Event* contains one of the event constants indicating why the handler is being called.
- EventData* contains the number of bytes sent or received at the time of the call for send and receive events and contains the modem status for modem events.

Function Documentation

```
void ( XUartNs550 * )
XUartNs550_ClearStats InstancePtr
```

This function zeros the statistics for the given instance.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XUartNs550 *
XUartNs550_GetDataFormat InstancePtr,
XUartNs550Format * FormatPtr
)
```

Gets the data format for the specified UART. The data format includes the baud rate, number of data bits, number of stop bits, and parity.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

FormatPtr is a pointer to a format structure that will contain the data format after this call completes.

Returns:

None.

Note:

None.

Xuint8 (**XUartNs550** *)
XUartNs550_GetFifoThreshold *InstancePtr*

This function gets the receive FIFO trigger level. The receive trigger level indicates the number of bytes in the receive FIFO that cause a receive data event (interrupt) to be generated.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The current receive FIFO trigger level. Constants which define each trigger level are contained in the file **xuartns550.h** and named XUN_FIFO_TRIGGER_*.

Note:

None.

Xuint8 (**XUartNs550** *)
XUartNs550_GetLastErrors *InstancePtr*

This function returns the last errors that have occurred in the specified UART. It also clears the errors such that they cannot be retrieved again. The errors include parity error, receive overrun error, framing error, and break detection.

The last errors is an accumulation of the errors each time an error is discovered in the driver. A status is checked for each received byte and this status is accumulated in the last errors.

If this function is called after receiving a buffer of data, it will indicate any errors that occurred for the bytes of the buffer. It does not indicate which bytes contained errors.

Parameters:

InstancePtr is a pointer to the [XUartNs550](#) instance to be worked on.

Returns:

The last errors that occurred. The errors are bit masks that are contained in the file [xuartns550.h](#) and named XUN_ERROR_*.

Note:

None.

Xuint8 ([XUartNs550](#) *)
XUartNs550_GetModemStatus *InstancePtr*

This function gets the modem status from the specified UART. The modem status indicates any changes of the modem signals. This function allows the modem status to be read in a polled mode. The modem status is updated whenever it is read such that reading it twice may not yield the same results.

Parameters:

InstancePtr is a pointer to the [XUartNs550](#) instance to be worked on.

Returns:

The modem status which are bit masks that are contained in the file [xuartns550.h](#) and named XUN_MODEM_*.

Note:

The bit masks used for the modem status are the exact bits of the modem status register with no abstraction.

Xuint16 (**XUartNs550** *)

XUartNs550_GetOptions *InstancePtr*

Gets the options for the specified driver instance. The options are implemented as bit masks such that multiple options may be enabled or disabled simultaneously.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The current options for the UART. The options are bit masks that are contained in the file **xuartns550.h** and named XUN_OPTION_*.

Returns:

None.

```
void ( XUartNs550 *  
XUartNs550_GetStats InstancePtr,  
 XUartNs550Stats * StatsPtr  
 )
```

This functions returns a snapshot of the current statistics in the area provided.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

StatsPtr is a pointer to a **XUartNs550Stats** structure to where the statistics are to be copied to.

Returns:

None.

Note:

None.

XStatus (**XUartNs550** *)

XUartNs550_Initialize *InstancePtr*,
 Xuint16 *DeviceId*
)

Initializes a specific **XUartNs550** instance such that it is ready to be used. The data format of the device is setup for 8 data bits, 1 stop bit, and no parity by default. The baud rate is set to a default value specified by `XPAR_DEFAULT_BAUD_RATE` if the symbol is defined, otherwise it is set to 19.2K baud. If the device has FIFOs (16550), they are enabled and the a receive FIFO threshold is set for 8 bytes. The default operating mode of the driver is polled mode.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XUartNs550** instance. Passing in a device id associates the generic **XUartNs550** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_NOT_FOUND if the device ID could not be found in the configuration table
- XST_UART_BAUD_ERROR if the baud rate is not possible because the input clock frequency is not divisible with an acceptable amount of error

Note:

None.

```
void ( XUartNs550 * )  
XUartNs550_InterruptHandler InstancePtr
```

This function is the interrupt handler for the 16450/16550 UART driver. It must be connected to an interrupt system by the user such that it is called when an interrupt for any 16450/16550 UART occurs. This function does not save or restore the processor context such that the user must ensure this occurs.

Parameters:

InstancePtr contains a pointer to the instance of the UART that the interrupt is for.

Returns:

None.

Note:

None.

Xboolean (**XUartNs550 ***)

XUartNs550_IsSending *InstancePtr*

This function determines if the specified UART is sending data. If the transmitter register is not empty, it is sending data.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

A value of XTRUE if the UART is sending data, otherwise XFALSE.

Note:

None.

XUartNs550_Config* (**Xuint16**)

XUartNs550_LookupConfig *DeviceId*

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId contains the ID of the device to look up the configuration for.

Returns:

A pointer to the configuration found or XNULL if the specified device ID was not found.

Note:

None.

unsigned int (**XUartNs550 ***
XUartNs550_Recv *InstancePtr,*
Xuint8 * *BufferPtr,*
unsigned int *NumBytes*
)

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer, either in the receiver or in the FIFO if present and enabled. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue receiving data until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

BufferPtr is pointer to buffer for data to be received into

NumBytes is the number of bytes to be received. A value of zero will stop a previous receive operation that is in progress in interrupt mode.

Returns:

The number of bytes received.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

XStatus (**XUartNs550** *)
XUartNs550_SelfTest *InstancePtr*

This functions runs a self-test on the driver and hardware device. This self test performs a local loopback and verifies data can be sent and received.

The statistics are cleared at the end of the test. The time for this test to execute is proportional to the baud rate that has been set prior to calling this function.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

- XST_SUCCESS if the test was successful
- XST_UART_TEST_FAIL if the test failed looping back the data

Note:

This function can hang if the hardware is not functioning properly.

```
unsigned int          ( XUartNs550 *  
XUartNs550_Send      InstancePtr,  
                     Xuint8 *    BufferPtr,  
                     unsigned int NumBytes  
                     )
```

This function sends the specified buffer of data using the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART. If the UART is busy sending data, it will return and indicate zero bytes were sent.

In a polled mode, this function will only send as much data as the UART can buffer, either in the transmitter or in the FIFO if present and enabled. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue sending data until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

BufferPtr is pointer to a buffer of data to be sent.

NumBytes contains the number of bytes to be sent. A value of zero will stop a previous send operation that is in progress in interrupt mode. Any data that was already put into the transmit FIFO will be sent.

Returns:

The number of bytes actually sent.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

This function and the **XUartNs550_SetOptions()** function modify shared data such that there may be a need for mutual exclusion in a multithreaded environment and if **XUartNs550_SetOptions()** is called from a handler.

```

XStatus ( XUartNs550 *
XUartNs550_SetDataFormat InstancePtr,
XUartNs550Format * FormatPtr
)

```

Sets the data format for the specified UART. The data format includes the baud rate, number of data bits, number of stop bits, and parity. It is the caller's responsibility to ensure that the UART is not sending or receiving data when this function is called.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

FormatPtr is a pointer to a format structure containing the data format to be set.

Returns:

- XST_SUCCESS if the data format was successfully set.
- XST_UART_BAUD_ERROR indicates the baud rate could not be set because of the amount of error with the baud rate and the input clock frequency.
- XST_INVALID_PARAM if one of the parameters was not valid.

Note:

The data types in the format type, data bits and parity, are 32 bit fields to prevent a compiler warning that is a bug with the GNU PowerPC compiler. The asserts in this function will cause a warning if these fields are bytes.

The baud rates tested include: 1200, 2400, 4800, 9600, 19200, 38400, 57600 and 115200.

```

XStatus ( XUartNs550 *
XUartNs550_SetFifoThreshold InstancePtr,
Xuint8 TriggerLevel
)

```

This functions sets the receive FIFO trigger level. The receive trigger level specifies the number of bytes in the receive FIFO that cause a receive data event (interrupt) to be generated. The FIFOs must be enabled to set the trigger level.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

TriggerLevel contains the trigger level to set. Constants which define each trigger level are contained in the file **xuartns550.h** and named XUN_FIFO_TRIGGER_*.

Returns:

- XST_SUCCESS if the trigger level was set

- XST_UART_CONFIG_ERROR if the trigger level could not be set, either the hardware does not support the FIFOs or FIFOs are not enabled

Note:

None.

```
void XUartNs550_SetHandler ( XUartNs550 * InstancePtr,
                             XUartNs550_Handler FuncPtr,
                             void * CallbackRef
                             )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the CallbackRef since the driver doesn't know what it is (nor should it)

```
XStatus XUartNs550_SetOptions ( XUartNs550 * InstancePtr,
                                Xuint16 Options
                                )
```

Sets the options for the specified driver instance. The options are implemented as bit masks such that multiple options may be enabled or disabled simultaneously.

The `GetOptions` function may be called to retrieve the currently enabled options. The result is ORed in the desired new settings to be enabled and ANDed with the inverse to clear the settings to be disabled. The resulting value is then used as the options for the `SetOption` function call.

Parameters:

InstancePtr is a pointer to the [XUartNs550](#) instance to be worked on.

Options contains the options to be set which are bit masks contained in the file [xuartns550.h](#) and named `XUN_OPTION_*`.

Returns:

- `XST_SUCCESS` if the options were set successfully.
- `XST_UART_CONFIG_ERROR` if the options could not be set because the hardware does not support FIFOs

Note:

None.

uartns550/v1_00_b/src/xuartns550.c File Reference

Detailed Description

This file contains the required functions for the 16450/16550 UART driver. Refer to the header file [xuartns550.h](#) for more detailed information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.
1.00b	rmm	05/14/03	Fixed diab compiler warnings relating to asserts.

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xuartns550.h"  
#include "xuartns550_i.h"  
#include "xio.h"
```

Functions

XStatus XUartNs550_Initialize (XUartNs550 *InstancePtr, **Xuint16** DeviceId)
unsigned int **XUartNs550_Send** (XUartNs550 *InstancePtr, **Xuint8** *BufferPtr, unsigned int NumBytes)
unsigned int **XUartNs550_Recv** (XUartNs550 *InstancePtr, **Xuint8** *BufferPtr, unsigned int NumBytes)

unsigned int **XUartNs550_SendBuffer** (**XUartNs550** *InstancePtr)
unsigned int **XUartNs550_ReceiveBuffer** (**XUartNs550** *InstancePtr)
XUartNs550_Config * **XUartNs550_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
XStatus ( XUartNs550 *  
XUartNs550_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initializes a specific **XUartNs550** instance such that it is ready to be used. The data format of the device is setup for 8 data bits, 1 stop bit, and no parity by default. The baud rate is set to a default value specified by XPAR_DEFAULT_BAUD_RATE if the symbol is defined, otherwise it is set to 19.2K baud. If the device has FIFOs (16550), they are enabled and the a receive FIFO threshold is set for 8 bytes. The default operating mode of the driver is polled mode.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XUartNs550** instance. Passing in a device id associates the generic **XUartNs550** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_NOT_FOUND if the device ID could not be found in the configuration table
- XST_UART_BAUD_ERROR if the baud rate is not possible because the input clock frequency is not divisible with an acceptable amount of error

Note:

None.

```
XUartNs550_Config* ( Xuint16 )  
XUartNs550_LookupConfig DeviceId
```

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId contains the ID of the device to look up the configuration for.

Returns:

A pointer to the configuration found or XNULL if the specified device ID was not found.

Note:

None.

unsigned int (**XUartNs550 ***)
XUartNs550_ReceiveBuffer *InstancePtr*

This function receives a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartNs550** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if there is no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer, either in the receiver or in the FIFO if present and enabled. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The number of bytes received.

Note:

None.

```

unsigned int          ( XUartNs550 *
XUartNs550_Recv      InstancePtr,
                       Xuint8 *   BufferPtr,
                       unsigned int NumBytes
                       )

```

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer, either in the receiver or in the FIFO if present and enabled. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue receiving data until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

BufferPtr is pointer to buffer for data to be received into

NumBytes is the number of bytes to be received. A value of zero will stop a previous receive operation that is in progress in interrupt mode.

Returns:

The number of bytes received.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

```

unsigned int          ( XUartNs550 *
XUartNs550_Send      InstancePtr,
                       Xuint8 *   BufferPtr,
                       unsigned int NumBytes
                       )

```

This functions sends the specified buffer of data using the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART. If the UART is busy sending data, it will return and indicate zero bytes were sent.

In a polled mode, this function will only send as much data as the UART can buffer, either in the transmitter or in the FIFO if present and enabled. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue sending data until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the [XUartNs550](#) instance to be worked on.

BufferPtr is pointer to a buffer of data to be sent.

NumBytes contains the number of bytes to be sent. A value of zero will stop a previous send operation that is in progress in interrupt mode. Any data that was already put into the transmit FIFO will be sent.

Returns:

The number of bytes actually sent.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

This function and the [XUartNs550_SetOptions\(\)](#) function modify shared data such that there may be a need for mutual exclusion in a multithreaded environment and if [XUartNs550_SetOptions\(\)](#) if called from a handler.

```
unsigned int          ( XUartNs550 *  
XUartNs550_SendBuffer InstancePtr )
```

This function sends a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartNs550** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function sends the specified buffer of data to the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART.

In a polled mode, this function will only send as much data as the UART can buffer, either in the transmitter or in the FIFO if present and enabled. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

NumBytes is the number of bytes actually sent (put into the UART transmitter and/or FIFO).

Note:

None.

uartns550/v1_00_b/src/xuartns550_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between the files of the driver. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.

```
#include "xuartns550.h"
```

Functions

```
unsigned int XUartNs550\_SendBuffer (XUartNs550 *InstancePtr)  
unsigned int XUartNs550\_ReceiveBuffer (XUartNs550 *InstancePtr)
```

Variables

```
XUartNs550\_Config XUartNs550\_ConfigTable []
```

Function Documentation

unsigned int (**XUartNs550 ***)
XUartNs550_ReceiveBuffer *InstancePtr*

This function receives a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartNs550** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if there is no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer, either in the receiver or in the FIFO if present and enabled. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The number of bytes received.

Note:

None.

unsigned int (**XUartNs550 ***)
XUartNs550_SendBuffer *InstancePtr*

This function sends a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartNs550** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function sends the specified buffer of data to the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART.

In a polled mode, this function will only send as much data as the UART can buffer, either in the transmitter or in the FIFO if present and enabled. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

NumBytes is the number of bytes actually sent (put into the UART transmitter and/or FIFO).

Note:

None.

Variable Documentation

XUartNs550_Config XUartNs550_ConfigTable ()

[]

The configuration table for UART 16550/16450 devices in the table. Each device should have an entry in this table.

XUartNs550 Struct Reference

```
#include <xuartns550.h>
```

Detailed Description

The XUartNs550 driver instance data. The user is required to allocate a variable of this type for every UART 16550/16450 device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [uartns550/v1_00_b/src/xuartns550.h](#)
-

XUartNs550_Config Struct Reference

```
#include <xuartns550.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xuint32 InputClockHz](#)

Field Documentation

[Xuint32 XUartNs550_Config::](#)

BaseAddress

Base address of device
(IPIF)

[Xuint16 XUartNs550_Config::](#)

DeviceId

Unique ID of
device

[Xuint32 XUartNs550_Config::](#)

InputClockHz

Input clock
frequency

The documentation for this struct was generated from the following file:

- [uartns550/v1_00_b/src/xuartns550.h](#)
-

Generated on 24 Jun 2004 for Xilinx Device Drivers

uartns550/v1_00_b/src/xuartns550_i.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. High-level driver functions are defined in [xuartns550.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	04/24/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

Defines

```
#define XUartNs550_mReadReg(BaseAddress, RegOffset)  
#define XUartNs550_mWriteReg(BaseAddress, RegOffset, RegisterValue)  
#define XUartNs550_mGetLineStatusReg(BaseAddress)  
#define XUartNs550_mGetLineControlReg(BaseAddress)  
#define XUartNs550_mSetLineControlReg(BaseAddress, RegisterValue)  
#define XUartNs550_mEnableIntr(BaseAddress)  
#define XUartNs550_mDisableIntr(BaseAddress)  
#define XUartNs550_mIsReceiveData(BaseAddress)  
#define XUartNs550_mIsTransmitEmpty(BaseAddress)
```

Functions

void **XUartNs550_SendByte** (**Xuint32** BaseAddress, **Xuint8** Data)

Xuint8 XUartNs550_RecvByte (**Xuint32** BaseAddress)

Define Documentation

#define XUartNs550_mDisableIntr(BaseAddress)

Disable the transmit and receive interrupts of the UART.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

#define XUartNs550_mEnableIntr(BaseAddress)

Enable the transmit and receive interrupts of the UART.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

#define XUartNs550_mGetLineControlReg(BaseAddress)

Get the UART Line Status Register.

Parameters:

BaseAddress contains the base address of the device.

Returns:

The value read from the register.

Note:

None.

```
#define XUartNs550_mGetLineStatusReg( BaseAddress )
```

Get the UART Line Status Register.

Parameters:

BaseAddress contains the base address of the device.

Returns:

The value read from the register.

Note:

None.

```
#define XUartNs550_mIsReceiveData( BaseAddress )
```

Determine if there is receive data in the receiver and/or FIFO.

Parameters:

BaseAddress contains the base address of the device.

Returns:

XTRUE if there is receive data, XFALSE otherwise.

Note:

None.

Set the UART Line Status Register.

Parameters:

BaseAddress contains the base address of the device.

RegisterValue is the value to be written to the register.

Returns:

None.

Note:

None.

```
#define XUartNs550_mWriteReg( BaseAddress,  
                             RegOffset,  
                             RegisterValue )
```

Write to a UART register.

Parameters:

BaseAddress contains the base address of the device.

RegOffset contains the offset from the 1st register of the device to select the specific register.

Returns:

The value read from the register.

Note:

None.

Function Documentation

```
Xuint8 XUartNs550_RecvByte ( Xuint32 BaseAddress )
```

This function receives a byte from the UART. It operates in a polling mode and blocks until a byte of data is received.

Parameters:

BaseAddress contains the base address of the UART.

Returns:

The data byte received by the UART.

Note:

None.

```
void XUartNs550_SendByte ( Xuint32 BaseAddress, Xuint8 Data )
```

This function sends a data byte with the UART. This function operates in the polling mode and blocks until the data has been put into the UART transmit holding register.

Parameters:

BaseAddress contains the base address of the UART.

Data contains the data byte to be sent.

Returns:

None.

Note:

None.

XUartNs550Format Struct Reference

```
#include <xuartns550.h>
```

Detailed Description

This data type allows the data format of the device to be set and retrieved.

Data Fields

[Xuint32 BaudRate](#)

[Xuint32 DataBits](#)

[Xuint32 Parity](#)

[Xuint8 StopBits](#)

Field Documentation

[Xuint32 XUartNs550Format::](#)

BaudRate

In bps, ie
1200

[Xuint32 XUartNs550Format::](#)

DataBits

Number of data
bits

Xuint32 XUartNs550Format::

Parity

Parity

Xuint8 XUartNs550Format::

StopBits

Number of stop
bits

The documentation for this struct was generated from the following file:

- [uartns550/v1_00_b/src/xuartns550.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

XUartNs550Stats Struct Reference

```
#include <xuartns550.h>
```

Detailed Description

UART statistics

Data Fields

[Xuint16 TransmitInterrupts](#)

[Xuint16 ReceiveInterrupts](#)

[Xuint16 StatusInterrupts](#)

[Xuint16 ModemInterrupts](#)

[Xuint16 CharactersTransmitted](#)

[Xuint16 CharactersReceived](#)

[Xuint16 ReceiveOverrunErrors](#)

[Xuint16 ReceiveParityErrors](#)

[Xuint16 ReceiveFramingErrors](#)

[Xuint16 ReceiveBreakDetected](#)

Field Documentation

[Xuint16 XUartNs550Stats::](#)

CharactersReceived

Number of characters
received

Xuint16 XUartNs550Stats::

CharactersTransmitted

Number of characters transmitted

Xuint16 XUartNs550Stats::

ModemInterrupts

Number of modem interrupts

Xuint16 XUartNs550Stats::

ReceiveBreakDetected

Number of receive breaks

Xuint16 XUartNs550Stats::

ReceiveFramingErrors

Number of receive framing errors

Xuint16 XUartNs550Stats::

ReceiveInterrupts

Number of receive interrupts

Xuint16 XUartNs550Stats::

ReceiveOverrunErrors

Number of receive overruns

Xuint16 XUartNs550Stats::

ReceiveParityErrors

Number of receive parity errors

Xuint16 XUartNs550Stats::

StatusInterrupts

Number of status
interrupts

Xuint16 XUartNs550Stats::

TransmitInterrupts

Number of transmit
interrupts

The documentation for this struct was generated from the following file:

- [uartns550/v1_00_b/src/xuartns550.h](#)

Generated on 24 Jun 2004 for Xilinx Device Drivers

uartns550/v1_00_b/src/xuartns550_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of NS16550 devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.

```
#include "xuartns550.h"  
#include "xparameters.h"
```

Variables

[XUartNs550_Config XUartNs550_ConfigTable](#) []

Variable Documentation

[XUartNs550_Config XUartNs550_ConfigTable](#)

[]

The configuration table for UART 16550/16450 devices in the table. Each device should have an entry in this table.

uartns550/v1_00_b/src/xuartns550_format.c File Reference

Detailed Description

This file contains the data format functions for the 16450/16550 UART driver. The data format functions allow the baud rate, number of data bits, number of stop bits and parity to be set and retrieved.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.
1.00b	rmm	05/15/03	Fixed diab compiler warnings.

```
#include "xstatus.h"  
#include "xuartns550.h"  
#include "xuartns550_i.h"  
#include "xio.h"
```

Functions

[XStatus XUartNs550_SetDataFormat](#) ([XUartNs550](#) *InstancePtr, [XUartNs550Format](#) *FormatPtr)
void [XUartNs550_GetDataFormat](#) ([XUartNs550](#) *InstancePtr, [XUartNs550Format](#) *FormatPtr)

Function Documentation

```
void XUartNs550_GetDataFormat ( XUartNs550 * InstancePtr, XUartNs550Format * FormatPtr )
```

Gets the data format for the specified UART. The data format includes the baud rate, number of data bits, number of stop bits, and parity.

Parameters:

InstancePtr is a pointer to the [XUartNs550](#) instance to be worked on.

FormatPtr is a pointer to a format structure that will contain the data format after this call completes.

Returns:

None.

Note:

None.

```
XStatus XUartNs550_SetDataFormat ( XUartNs550 * InstancePtr, XUartNs550Format * FormatPtr )
```

Sets the data format for the specified UART. The data format includes the baud rate, number of data bits, number of stop bits, and parity. It is the caller's responsibility to ensure that the UART is not sending or receiving data when this function is called.

Parameters:

InstancePtr is a pointer to the [XUartNs550](#) instance to be worked on.

FormatPtr is a pointer to a format structure containing the data format to be set.

Returns:

- XST_SUCCESS if the data format was successfully set.
- XST_UART_BAUD_ERROR indicates the baud rate could not be set because of the amount of error with the baud rate and the input clock frequency.
- XST_INVALID_PARAM if one of the parameters was not valid.

Note:

The data types in the format type, data bits and parity, are 32 bit fields to prevent a compiler warning that is a bug with the GNU PowerPC compiler. The asserts in this function will

cause a warning if these fields are bytes.

The baud rates tested include: 1200, 2400, 4800, 9600, 19200, 38400, 57600 and 115200.

Generated on 24 Jun 2004 for Xilinx Device Drivers

uartns550/v1_00_b/src/xuartns550_options.c File Reference

Detailed Description

The implementation of the options functions for the [XUartNs550](#) driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.

```
#include "xuartns550.h"  
#include "xuartns550_i.h"  
#include "xio.h"
```

Data Structures

```
struct Mapping
```

Functions

```
Xuint16 XUartNs550\_GetOptions (XUartNs550 *InstancePtr)  
XStatus XUartNs550\_SetOptions (XUartNs550 *InstancePtr, Xuint16 Options)  
  Xuint8 XUartNs550\_GetFifoThreshold (XUartNs550 *InstancePtr)  
XStatus XUartNs550\_SetFifoThreshold (XUartNs550 *InstancePtr, Xuint8 TriggerLevel)  
  Xuint8 XUartNs550\_GetLastErrors (XUartNs550 *InstancePtr)  
  Xuint8 XUartNs550\_GetModemStatus (XUartNs550 *InstancePtr)
```

Function Documentation

Xuint8 (**XUartNs550 ***)
XUartNs550_GetFifoThreshold *InstancePtr*

This function gets the receive FIFO trigger level. The receive trigger level indicates the number of bytes in the receive FIFO that cause a receive data event (interrupt) to be generated.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The current receive FIFO trigger level. Constants which define each trigger level are contained in the file **xuartns550.h** and named XUN_FIFO_TRIGGER_*.

Note:

None.

Xuint8 (**XUartNs550 ***)
XUartNs550_GetLastErrors *InstancePtr*

This function returns the last errors that have occurred in the specified UART. It also clears the errors such that they cannot be retrieved again. The errors include parity error, receive overrun error, framing error, and break detection.

The last errors is an accumulation of the errors each time an error is discovered in the driver. A status is checked for each received byte and this status is accumulated in the last errors.

If this function is called after receiving a buffer of data, it will indicate any errors that occurred for the bytes of the buffer. It does not indicate which bytes contained errors.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The last errors that occurred. The errors are bit masks that are contained in the file **xuartns550.h** and named XUN_ERROR_*.

Note:

None.

Xuint8 (**XUartNs550 ***)
XUartNs550_GetModemStatus *InstancePtr*

This function gets the modem status from the specified UART. The modem status indicates any changes of the modem signals. This function allows the modem status to be read in a polled mode. The modem status is updated whenever it is read such that reading it twice may not yield the same results.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The modem status which are bit masks that are contained in the file **xuartns550.h** and named XUN_MODEM_*.

Note:

The bit masks used for the modem status are the exact bits of the modem status register with no abstraction.

Xuint16 (**XUartNs550 ***)
XUartNs550_GetOptions *InstancePtr*

Gets the options for the specified driver instance. The options are implemented as bit masks such that multiple options may be enabled or disabled simulataneously.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The current options for the UART. The options are bit masks that are contained in the file **xuartns550.h** and named XUN_OPTION_*.

Returns:

None.

Xboolean (**XUartNs550** *)
XUartNs550_IsSending *InstancePtr*

This function determines if the specified UART is sending data. If the transmitter register is not empty, it is sending data.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

A value of XTRUE if the UART is sending data, otherwise XFALSE.

Note:

None.

XStatus (**XUartNs550** *
XUartNs550_SetFifoThreshold *InstancePtr*,
Xuint8 *TriggerLevel*
)

This functions sets the receive FIFO trigger level. The receive trigger level specifies the number of bytes in the receive FIFO that cause a receive data event (interrupt) to be generated. The FIFOs must be enabled to set the trigger level.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

TriggerLevel contains the trigger level to set. Constants which define each trigger level are contained in the file **xuartns550.h** and named XUN_FIFO_TRIGGER_*.

Returns:

- XST_SUCCESS if the trigger level was set
- XST_UART_CONFIG_ERROR if the trigger level could not be set, either the hardware does not support the FIFOs or FIFOs are not enabled

Note:

None.

```
XStatus ( XUartNs550 *  
XUartNs550_SetOptions InstancePtr,  
 Xuint16 Options  
 )
```

Sets the options for the specified driver instance. The options are implemented as bit masks such that multiple options may be enabled or disabled simultaneously.

The GetOptions function may be called to retrieve the currently enabled options. The result is ORed in the desired new settings to be enabled and ANDed with the inverse to clear the settings to be disabled. The resulting value is then used as the options for the SetOption function call.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Options contains the options to be set which are bit masks contained in the file **xuartns550.h** and named XUN_OPTION_*.

Returns:

- o XST_SUCCESS if the options were set successfully.
- o XST_UART_CONFIG_ERROR if the options could not be set because the hardware does not support FIFOs

Note:

None.

uartns550/v1_00_b/src/xuartns550_intr.c File Reference

Detailed Description

This file contains the functions that are related to interrupt processing for the 16450/16550 UART driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00b	jhl	03/11/02	Repartitioned driver for smaller files.
-------	-----	----------	---

```
#include "xuartns550.h"  
#include "xuartns550_i.h"  
#include "xio.h"
```

Functions

void [XUartNs550_SetHandler](#) ([XUartNs550](#) *InstancePtr, [XUartNs550_Handler](#) FuncPtr, void *CallbackRef)

void [XUartNs550_InterruptHandler](#) ([XUartNs550](#) *InstancePtr)

Function Documentation

```
void  
XUartNs550\_InterruptHandler (XUartNs550 *  
InstancePtr)
```

This function is the interrupt handler for the 16450/16550 UART driver. It must be connected to an interrupt system by the user such that it is called when an interrupt for any 16450/16550 UART occurs. This function does not save or restore the processor context such that the user must ensure this occurs.

Parameters:

InstancePtr contains a pointer to the instance of the UART that the interrupt is for.

Returns:

None.

Note:

None.

```
void XUartNs550_SetHandler ( XUartNs550 * InstancePtr,
                             XUartNs550_Handler FuncPtr,
                             void * CallbackRef
                             )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

uartns550/v1_00_b/src/xuartns550_l.c File Reference

Detailed Description

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.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	04/24/02	First release

```
#include "xuartns550_1.h"
```

Functions

```
void XUartNs550_SendByte (Xuint32 BaseAddress, Xuint8 Data)  
Xuint8 XUartNs550_RecvByte (Xuint32 BaseAddress)
```

Function Documentation

```
Xuint8  
XUartNs550_RecvByte ( Xuint32  
                    BaseAddress )
```

This function receives a byte from the UART. It operates in a polling mode and blocks until a byte of data is received.

Parameters:

BaseAddress contains the base address of the UART.

Returns:

The data byte received by the UART.

Note:

None.

```
void XUartNs550_SendByte ( Xuint32 BaseAddress, Xuint8 Data )
```

This function sends a data byte with the UART. This function operates in the polling mode and blocks until the data has been put into the UART transmit holding register.

Parameters:

BaseAddress contains the base address of the UART.

Data contains the data byte to be sent.

Returns:

None.

Note:

None.

uartns550/v1_00_b/src/xuartns550_selftest.c File Reference

Detailed Description

This file contains the self-test functions for the 16450/16550 UART driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.

```
#include "xstatus.h"  
#include "xuartns550.h"  
#include "xuartns550_i.h"  
#include "xio.h"
```

Functions

[XStatus XUartNs550_SelfTest](#) ([XUartNs550](#) *InstancePtr)

Function Documentation

XStatus	(XUartNs550 *)
XUartNs550_SelfTest	<i>InstancePtr</i>	

This functions runs a self-test on the driver and hardware device. This self test performs a local loopback and verifies data can be sent and received.

The statistics are cleared at the end of the test. The time for this test to execute is proportional to the baud rate that has been set prior to calling this function.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

- XST_SUCCESS if the test was successful
- XST_UART_TEST_FAIL if the test failed looping back the data

Note:

This function can hang if the hardware is not functioning properly.

common/v1_00_a/src/xutil.h File Reference

Detailed Description

This file contains utility functions such as memory test functions.

Memory test description

A subset of the memory tests can be selected or all of the tests can be run in order. If there is an error detected by a subtest, the test stops and the failure code is returned. Further tests are not run even if all of the tests are selected.

Subtest descriptions:

`XUT_ALLMEMTESTS:`
Runs all of the following tests

`XUT_INCREMENT:`
Incrementing Value Test.
This test starts at 'XUT_MEMTEST_INIT_VALUE' and uses the incrementing value as the test value for memory.

`XUT_WALKONES:`
Walking Ones Test.
This test uses a walking '1' as the test value for memory.
location 1 = 0x00000001
location 2 = 0x00000002
...

`XUT_WALKZEROS:`
Walking Zero's Test.
This test uses the inverse value of the walking ones test as the test value for memory.

```
location 1 = 0xFFFFFFFF
location 2 = 0xFFFFFFFFD
...
```

XUT_INVERSEADDR:

Inverse Address Test.

This test uses the inverse of the address of the location under test as the test value for memory.

XUT_FIXEDPATTERN:

Fixed Pattern Test.

This test uses the provided patterns as the test value for memory.

If zero is provided as the pattern the test uses '0xDEADBEEF'.

WARNING

The tests are **DESTRUCTIVE**. Run before any initialized memory spaces have been set up.

The address, Addr, provided to the memory tests is not checked for validity except for the XNULL case. It is possible to provide a code-space pointer for this test to start with and ultimately destroy executable code causing random failures.

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	11/01/01	First release

```
#include "xbasic_types.h"
```

```
#include "xstatus.h"
```

Memory subtests

```
#define XUT_ALLMEMTESTS
#define XUT_INCREMENT
#define XUT_WALKONES
#define XUT_WALKZEROS
#define XUT_INVERSEADDR
#define XUT_FIXEDPATTERN
#define XUT_MAXTEST
```

Functions

```
XStatus XUtil_MemoryTest32 (Xuint32 *Addr, Xuint32 Words, Xuint32 Pattern, Xuint8 Subtest)
XStatus XUtil_MemoryTest16 (Xuint16 *Addr, Xuint32 Words, Xuint16 Pattern, Xuint8 Subtest)
XStatus XUtil_MemoryTest8 (Xuint8 *Addr, Xuint32 Words, Xuint8 Pattern, Xuint8 Subtest)
```

Define Documentation

```
#define
XUT_ALLMEMTESTS
```

See the detailed description of the subtests in the file description.

```
#define
XUT_FIXEDPATTERN
```

See the detailed description of the subtests in the file description.

```
#define
XUT_INCREMENT
```

See the detailed description of the subtests in the file description.

```
#define
XUT_INVERSEADDR
```

See the detailed description of the subtests in the file description.

#define
XUT_MAXTEST

See the detailed description of the subtests in the file description.

#define
XUT_WALKONES

See the detailed description of the subtests in the file description.

#define
XUT_WALKZEROS

See the detailed description of the subtests in the file description.

Function Documentation

XStatus
XUtil_MemoryTest16 (**Xuint16** * *Addr*,
Xuint32 *Words*,
Xuint16 *Pattern*,
Xuint8 *Subtest*
)

Performs a destructive 16-bit wide memory test.

Parameters:

Addr is a pointer to the region of memory to be tested.

Words is the length of the block.

Pattern is the constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

Subtest is the test selected. See [xutil.h](#) for possible values.

Returns:

- XST_MEMTEST_FAILED is returned for a failure
- XST_SUCCESS is returned for a pass

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundry of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

```
XStatus          ( Xuint32 *  
XUtil_MemoryTest32  Addr,  
                   Xuint32  Words,  
                   Xuint32  Pattern,  
                   Xuint8   Subtest  
                   )
```


Performs a destructive 8-bit wide memory test.

Parameters:

Addr is a pointer to the region of memory to be tested.

Words is the length of the block.

Pattern is the constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

Subtest is the test selected. See [xutil.h](#) for possible values.

Returns:

- XST_MEMTEST_FAILED is returned for a failure
- XST_SUCCESS is returned for a pass

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundry of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

common/v1_00_a/src/xutil_memtest.c File Reference

Detailed Description

Contains the memory test utility functions.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	11/01/01	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xutil.h"
```

Functions

```
XStatus XUtil_MemoryTest32 (Xuint32 *Addr, Xuint32 Words, Xuint32 Pattern, Xuint8 Subtest)  
XStatus XUtil_MemoryTest16 (Xuint16 *Addr, Xuint32 Words, Xuint16 Pattern, Xuint8 Subtest)  
XStatus XUtil_MemoryTest8 (Xuint8 *Addr, Xuint32 Words, Xuint8 Pattern, Xuint8 Subtest)
```

Function Documentation

```
XStatus          ( Xuint16 *
XUtil_MemoryTest16   Addr,
                      Xuint32  Words,
                      Xuint16  Pattern,
                      Xuint8   Subtest
                      )
```

Performs a destructive 16-bit wide memory test.

Parameters:

Addr is a pointer to the region of memory to be tested.

Words is the length of the block.

Pattern is the constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

Subtest is the test selected. See [xutil.h](#) for possible values.

Returns:

- XST_MEMTEST_FAILED is returned for a failure
- XST_SUCCESS is returned for a pass

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2^{**} width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

```
XStatus          ( Xuint32 *
XUtil_MemoryTest32   Addr,
                      Xuint32  Words,
                      Xuint32  Pattern,
                      Xuint8   Subtest
                      )
```

Performs a destructive 32-bit wide memory test.

Parameters:

Addr is a pointer to the region of memory to be tested.

Words is the length of the block.

Pattern is the constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

Subtest is the test selected. See [xutil.h](#) for possible values.

Returns:

- XST_MEMTEST_FAILED is returned for a failure
- XST_SUCCESS is returned for a pass

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

```
XStatus          ( Xuint8 *
XUtil_MemoryTest8      Addr,
                      Xuint32 Words,
                      Xuint8  Pattern,
                      Xuint8  Subtest
                      )
```

Performs a destructive 8-bit wide memory test.

Parameters:

Addr is a pointer to the region of memory to be tested.

Words is the length of the block.

Pattern is the constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

Subtest is the test selected. See [xutil.h](#) for possible values.

Returns:

- o XST_MEMTEST_FAILED is returned for a failure
- o XST_SUCCESS is returned for a pass

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundry of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

wdttb/v1_00_b/src/xwdttb_i.h File Reference

Detailed Description

This file contains data which is shared between files and internal to the **XWdtTb** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	02/06/02	First release
1.00b	rpm	04/26/02	Moved register definitions to xwdttb_1.h

```
#include "xwdttb\_1.h"
```

Variables

[XWdtTb_Config](#) [XWdtTb_ConfigTable](#) []

Variable Documentation

[XWdtTb_Config](#) [XWdtTb_ConfigTable](#) ()

[]

This table contains configuration information for each watchdog timer device in the system.

XWdtTb Struct Reference

```
#include <xwdttb.h>
```

Detailed Description

The XWdtTb driver instance data. The user is required to allocate a variable of this type for every watchdog/timer device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `wdttb/v1_00_b/src/xwdttb.h`
-

wdttb/v1_00_b/src/xwdttb.h File Reference

Detailed Description

The Xilinx watchdog timer/timebase component supports the Xilinx watchdog timer/timebase hardware. More detailed description of the driver operation for each function can be found in the [xwdttb.c](#) file.

The Xilinx watchdog timer/timebase driver supports the following features:

- Polled mode
- enabling and disabling (if allowed by the hardware) the watchdog timer
- restarting the watchdog.
- reading the timebase.

It is the responsibility of the application to provide an interrupt handler for the timebase and the watchdog and connect them to the interrupt system if interrupt driven mode is desired.

The watchdog timer/timebase component ALWAYS generates an interrupt output when:

- the watchdog expires the first time
- the timebase rolls over

and ALWAYS generates a reset output when the watchdog timer expires a second time. This is not configurable in any way from the software driver's perspective.

The Timebase is reset to 0 when the Watchdog Timer is enabled.

If the hardware interrupt signal is not connected, polled mode is the only option (using `IsWdtExpired`) for the watchdog. Reset output will occur for the second watchdog timeout regardless. Polled mode for the timebase rollover is just reading the contents of the register and seeing if the MSB has transitioned from 1 to 0.

The `IsWdtExpired` function is used for polling the watchdog timer and it is also used to check if the

watchdog was the cause of the last reset. In this situation, call Initialize then call WdtIsExpired. If the result is true watchdog timeout caused the last system reset. It is then acceptable to further initialize the component which will reset this bit.

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	02/21/02	Repartitioned driver for smaller files
1.00b	rpm	04/26/02	Made LookupConfig public and added XWdtTb_Config

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

Data Structures

```
struct XWdtTb  
struct XWdtTb\_Config
```

Functions

```
XStatus XWdtTb\_Initialize (XWdtTb *InstancePtr, Xuint16 DeviceId)  
void XWdtTb\_Start (XWdtTb *InstancePtr)  
XStatus XWdtTb\_Stop (XWdtTb *InstancePtr)  
Xboolean XWdtTb\_IsWdtExpired (XWdtTb *InstancePtr)  
void XWdtTb\_RestartWdt (XWdtTb *InstancePtr)  
Xuint32 XWdtTb\_GetTbValue (XWdtTb *InstancePtr)  
XStatus XWdtTb\_SelfTest (XWdtTb *InstancePtr)
```

Function Documentation

Xuint32 (**XWdtTb ***)
XWdtTb_GetTbValue *InstancePtr*

Returns the current contents of the timebase.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

The contents of the timebase.

Note:

None.

XStatus (**XWdtTb ***
XWdtTb_Initialize *InstancePtr*,
Xuint16 *DeviceId*
)

Initialize a specific watchdog timer/timebase instance/driver. This function must be called before other functions of the driver are called.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XWdtTb** instance. Passing in a device id associates the generic **XWdtTb** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if the configuration for device ID was not found

Note:

None.

Xboolean (**XWdtTb ***)
XWdtTb_IsWdtExpired *InstancePtr*

Check if the watchdog timer has expired. This function is used for polled mode and it is also used to check if the last reset was caused by the watchdog timer.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

XTRUE if the watchdog has expired, and XFALSE otherwise.

Note:

None.

```
void ( XWdtTb * )  
XWdtTb_RestartWdt InstancePtr
```

Restart the watchdog timer. An application needs to call this function periodically to keep the timer from asserting the reset output.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus ( XWdtTb * )  
XWdtTb_SelfTest InstancePtr
```

Run a self-test on the timebase. This test verifies that the timebase is incrementing. The watchdog timer is not tested due to the time required to wait for the watchdog timer to expire. The time consumed by this test is dependant on the system clock and the configuration of the dividers in for the input clock of the timebase.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

- XST_SUCCESS if self-test was successful
- XST_WDTTB_TIMER_FAILED if the timebase is not incrementing

Note:

None.

```
void ( XWdtTb * )  
XWdtTb_Start InstancePtr
```

Start the watchdog timer of the device.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

None.

Note:

The Timebase is reset to 0 when the Watchdog Timer is started. The Timebase is always incrementing

```
XStatus ( XWdtTb * )  
XWdtTb_Stop InstancePtr
```

Disable the watchdog timer.

It is the caller's responsibility to disconnect the interrupt handler of the watchdog timer from the interrupt source, typically an interrupt controller, and disable the interrupt in the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

- XST_SUCCESS if the watchdog was stopped successfully
- XST_NO_FEATURE if the watchdog cannot be stopped

Note:

The hardware configuration controls this functionality. If it is not allowed by the hardware the failure will be returned and the timer will continue without interruption.

wdttb/v1_00_b/src/xwdttb.c File Reference

Detailed Description

Contains the required functions of the **XWdtTb** driver component. See [xwdttb.h](#) for a description of the driver.

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	rpm	04/26/02	Made LookupConfig public

```
#include "xbasic_types.h"  
#include "xparameters.h"  
#include "xio.h"  
#include "xwdttb.h"  
#include "xwdttb_i.h"
```

Functions

XStatus XWdtTb_Initialize (**XWdtTb** *InstancePtr, **Xuint16** DeviceId)

void **XWdtTb_Start** (**XWdtTb** *InstancePtr)

XStatus XWdtTb_Stop (**XWdtTb** *InstancePtr)

Xboolean XWdtTb_IsWdtExpired (**XWdtTb** *InstancePtr)

void **XWdtTb_RestartWdt** (**XWdtTb** *InstancePtr)

Xuint32 XWdtTb_GetTbValue (**XWdtTb** *InstancePtr)

Function Documentation

```
Xuint32 ( XWdtTb * )  
XWdtTb_GetTbValue InstancePtr
```

Returns the current contents of the timebase.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

The contents of the timebase.

Note:

None.

```
XStatus ( XWdtTb *  
XWdtTb_Initialize InstancePtr,  
 Xuint16 DeviceId  
 )
```

Initialize a specific watchdog timer/timebase instance/driver. This function must be called before other functions of the driver are called.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XWdtTb** instance. Passing in a device id associates the generic **XWdtTb** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if the configuration for device ID was not found

Note:

None.

```
Xboolean ( XWdtTb * )  
XWdtTb_IsWdtExpired InstancePtr
```

Check if the watchdog timer has expired. This function is used for polled mode and it is also used to check if the last reset was caused by the watchdog timer.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

XTRUE if the watchdog has expired, and XFALSE otherwise.

Note:

None.

```
void ( XWdtTb * )  
XWdtTb_RestartWdt InstancePtr
```

Restart the watchdog timer. An application needs to call this function periodically to keep the timer from asserting the reset output.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

None.

Note:

None.

```
void ( XWdtTb * )  
XWdtTb_Start InstancePtr
```

Start the watchdog timer of the device.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

None.

Note:

The Timebase is reset to 0 when the Watchdog Timer is started. The Timebase is always incrementing

XStatus (**XWdtTb** *)
XWdtTb_Stop *InstancePtr*

Disable the watchdog timer.

It is the caller's responsibility to disconnect the interrupt handler of the watchdog timer from the interrupt source, typically an interrupt controller, and disable the interrupt in the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

- XST_SUCCESS if the watchdog was stopped successfully
- XST_NO_FEATURE if the watchdog cannot be stopped

Note:

The hardware configuration controls this functionality. If it is not allowed by the hardware the failure will be returned and the timer will continue without interruption.

wdttb/v1_00_b/src/xwdttb_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xwdttb.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	rpm	04/26/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

Defines

```
#define XWdtTb_mGetTimebaseReg(BaseAddress)  
#define XWdtTb_mEnableWdt(BaseAddress)  
#define XWdtTb_mDisableWdt(BaseAddress)  
#define XWdtTb_mRestartWdt(BaseAddress)  
#define XWdtTb_mHasReset(BaseAddress)  
#define XWdtTb_mHasExpired(BaseAddress)
```

Define Documentation

```
#define XWdtTb_mDisableWdt( BaseAddress )
```

Disable the watchdog timer.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XWdtTb_mEnableWdt( BaseAddress )
```

Enable the watchdog timer. Clear previous expirations. The timebase is reset to 0.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XWdtTb_mGetTimebaseReg( BaseAddress )
```

Get the contents of the timebase register.

Parameters:

BaseAddress is the base address of the device

Returns:

A 32-bit value representing the timebase.

Note:

None.

#define XWdtTb_mHasExpired(BaseAddress)

Check to see if the watchdog timer has expired.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if the watchdog did expire, XFALSE otherwise.

Note:

None.

#define XWdtTb_mHasReset(BaseAddress)

Check to see if the last system reset was caused by the timer expiring.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if the watchdog did cause the last reset, XFALSE otherwise.

Note:

None.

#define XWdtTb_mRestartWdt(BaseAddress)

Restart the watchdog timer.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

XWdtTb_Config Struct Reference

```
#include <xwdttb.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddr](#)

Field Documentation

[Xuint32 XWdtTb_Config::](#)

BaseAddr

Base address of the device

[Xuint16 XWdtTb_Config::](#)

DeviceId

Unique ID of device

The documentation for this struct was generated from the following file:

- `wdttb/v1_00_b/src/xwdttb.h`

wdttb/v1_00_b/src/xwdttb_g.c File Reference

Detailed Description

This file contains a table that specifies the configuration of all watchdog timer devices in the system. Each device should have an entry in the table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	01/18/01	First release
1.00b	jhl	02/21/02	Repartitioned driver for smaller files

```
#include "xwdttb.h"  
#include "xparameters.h"
```

Variables

[XWdtTb_Config](#) [XWdtTb_ConfigTable](#) [XPAR_XWDTTB_NUM_INSTANCES]

Variable Documentation

[XWdtTb_Config](#) [XWdtTb_ConfigTable](#)
[XPAR_XWDTTB_NUM_INSTANCES]

This table contains configuration information for each watchdog timer device in the system.

wdttb/v1_00_b/src/xwdttb_selftest.c File Reference

Detailed Description

Contains diagnostic self-test functions for the **XWdtTb** component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	02/06/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"  
#include "xwdttb.h"  
#include "xwdttb_i.h"
```

Functions

XStatus **XWdtTb_SelfTest** (**XWdtTb** *InstancePtr)

Function Documentation

XStatus (**XWdtTb** *
XWdtTb_SelfTest *InstancePtr*)

Run a self-test on the timebase. This test verifies that the timebase is incrementing. The watchdog timer is not tested due to the time required to wait for the watchdog timer to expire. The time consumed by this test is dependant on the system clock and the configuration of the dividers in for the input clock of the timebase.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

- XST_SUCCESS if self-test was successful
- XST_WDTTB_TIMER_FAILED if the timebase is not incrementing

Note:

None.

emc/v1_00_a/src/xemc_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of EMC devices in the system. In addition, there is a lookup function used by the driver to access its configuration information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	02/04/02	First release
1.00a	rpm	05/14/02	Made configuration table/lookup public

```
#include "xemc.h"  
#include "xparameters.h"
```

iic/v1_01_d/src/xiic_slave.c File Reference

Detailed Description

Contains slave functions for the **XIic** component. This file is necessary when slave operations, sending and receiving data as a slave on the IIC bus, are desired.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01b	jhl	3/26/02	repartitioned the driver
1.01c	ecm	12/05/02	new rev

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

packet_fifo/v2_00_a/src/xpacket_fifo_l_v2_00_a.c File Reference

Detailed Description

Contains low-level (Level 0) functions for the [XPacketFifoV200a](#) driver. See [xpacket_fifo_v2_00_a.h](#) for information about the high-level (Level 1) driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

2.00a	rpm	10/22/03	First release. Moved most of Level 1 driver functions
-------	-----	----------	---

into this layer.

2.00a	rmm	02/24/04	Added L0WriteDRE function.
-------	-----	----------	----------------------------

```
#include "xbasic_types.h"
```

```
#include "xio.h"
```

```
#include "xpacket_fifo_l_v2_00_a.h"
```

pci/v1_00_b/src/xpci_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of PCI devices in the system.

Note:

None.

```
#include "xpci.h"  
#include "xparameters.h"
```

tmrctr/v1_00_b/src/xtmrctr_1.c File Reference

Detailed Description

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.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	04/24/02	First release

```
#include "xbasic_types.h"
```

```
#include "xtmrctr_1.h"
```