SPECIFICATION

FOR

X-LINK

Note:

If this copy is no longer in use, return to sender.

APPROVAL PAGE

Name	Signature	Date

AUTHOR/S

Name	Signature	Date
Jean-Philippe Arnaud		29/09/2004

DOCUMENT HISTORY

Date	Initials	Revision	Description of change
23/09/04	JPA	1.0	New document
18/11/04	JPA	1.1	Added FPGA configuration check and version check requirements.
24/11/04	JPA	1.2	Cleaned up document.
01/12/04	JPA	1.3	Added operating modes feature.
29/04/05	JPA	1.4	Added TOC.
24/05/05	PSR	1.5	Restructured & updated
09/06/05	SM	1.6	Review for release
11/07/05	HV	1.7	Added discussion about 32-bit / 64-bit and an outline firmware design.
24/10/05	HV	1.8	General review.
08/12/05	HV	1.9	Added cancel bit to OSR. (Required for the host driver for the SMT145)

TABLE OF CONTENTS

1. APPLICABLE DOCUMENTS AND REFERENCES

1.1. APPLICABLE DOCUMENTS

1.1.1. External Documents

None identified

1.1.2. Internal documents

D0000051-impl.doc

D0000051-veri.doc

1.1.3. Project Documents

D0000051-proj.mpp

1.2. REFERENCES

1.2.1. External documents

N.A

1.2.2. Internal documents

N.A

1.2.3. Project documents

N.A

1.3. PRECEDENCE

In the event of conflict between the text of this document and the applicable documents cited herein, the text of this document takes precedence. Nothing in this document however, supersedes applicable laws and regulations unless a specific exemption has been obtained and is identified in the text of this document.

2. ACRONYMS, ABBREVIATIONS AND DEFINITIONS

2.1. ACRONYMS AND ABBREVIATIONS

TIM	Texas Instruments Module
CP	Comport
SDB	Sundance Digital Bus
MBZ	Must be zero
MBO	Must be one

2.2. DEFINITIONS

Implementation-defined	Behaviour that is predictable, but may vary from implementation to implementation. An implementation must explicitly define all such behaviour.
Interrupt	a forced change in the program counter
Word	a 32-bit object
Undefined	Behaviour that is not predictable and may change from implementation to implementation.
VUINT32	volatile unsigned int

3. SCOPE

This document specifies details for a generic interface that can be used with all Sundance communication resources.

3.1. OVERVIEW

The X-Link is being established as the standard interface to Sundance point-to-point communication devices (links). Most current links are based on a 32-bit transport mechanism, but there are opportunities to improve data transfer rates by making use of 64-bit or even 128-bit techniques.

3.2. PURPOSE

The requirement is to provide a single interface definition that can deal efficiently with all these devices in a unified way without sacrificing existing devices or breaking current programming models. This generic interface can be used with any communication device with the aim of minimizing support and maintenance load. One pair of drivers (one input, one output) will handle anything, and could be made optimal. One coherent interface means that user documentation and examples are considerably simplified.

3.3. STATEMENT OF PROBLEM

There are two ways to view a link: from the processor (the software view) or from the physical communication mechanism (the hardware view). The main concern of the software is the unit of information that can be written to the device or read from it (the granularity), while the hardware is concerned with the most efficient number of bits to transfer as a unit over the actual electrical connection. It is important to realise that there is no fundamental connection between the two.

In an ideal world, the granularity of link communications would match the addressing unit of the processor's memory; a byte addressed machine would provide byte-organised links, for example. For historical and performance reasons this obvious property has rarely existed[1] and a *de facto* standard of 32-bit links has emerged. Sundance now has a considerable investment in software that has been built with that 32-bit assumption. From the hardware perspective, there is a pressure to increase the physical granularity and thereby optimise the potential transfer rate. In poor designs this is made visible to the software. As an example, the ADI TigerSHARC links have a 128-bit granularity and this imposes constraints on the software: all transfers must be in units of four 32-bit words and all DMA link transfers must be aligned on 4-word boundaries. The software has two choices: follow the hardware and break existing code and coding practices, or solve the problem by imposing a protocol on transfers and thereby reducing bandwidth.

With X-Links we have an opportunity to recognise the independence of the hardware and software views of links, and continue to present a 32-bit interface to the software while allowing hardware to adapt to whatever the physical medium requires for optimal performance. This document describes how this can be achieved.

[1] transputers were a clear exception.

4. BACKGROUND

The current interfaces (Comport and SDB) have a number of features that make writing device drivers tricky and potentially inefficient. The SDB interface, having been designed with the hindsight offered by the comport interface, is by far the better interface of the two, but several significant improvements can be made.

4.1.1. Comport

The comport interface, inherited from TI but improved by Sundance, has a complex arrangement of 15-word FIFOs and single-word buffers. In practice, these look like 16-word FIFOs, but the hardware implementation is visible to software, for example, in the need to reset an input device twice to clear both the FIFO and the single-word buffer.

The status register provides 4-bit fields that give the number of words in an input FIFO or the number of writable spaces in an output FIFO. The immediate problem here is that there are 17 possible values: 0 (completely empty) to 16 (completely full), and the equivalent for output.

Input and output transfers have to be handled in very different ways. On input, we can get two signals: FIFO not empty (i.e., at least 1 word there) or FIFO contains at least 8 words. Transfers can proceed in 8-word lumps until there are less than 8 words to transfer. At that point there is no option but to take an interrupt for each single word. This involves changing the comport control register, usually at least twice per complete transfer.

On output things are simpler, as we can write lumps of 8 words and then write the whole of any residue in one go when the FIFO indicates it has 8 spaces.

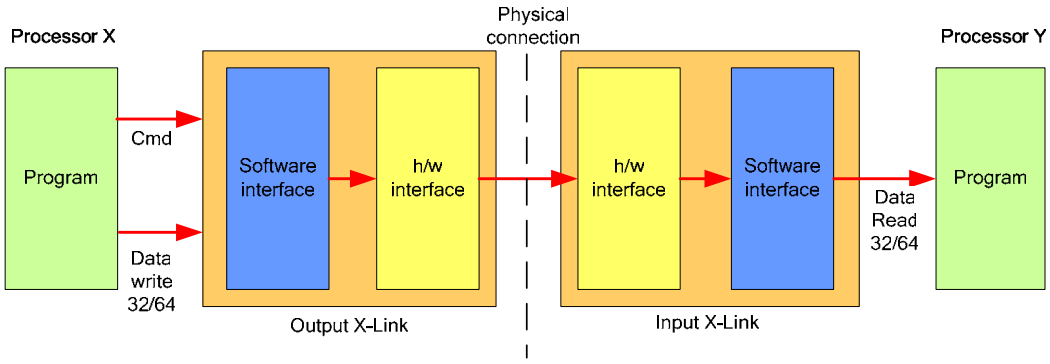
There is an added complication to this. Because the input and output sides of the comport, actually two independent devices from a software point of view, are treated as one device, there is only one control register. Input and Output threads could be operating at the same time, and so accesses to the register must be protected by disabling interrupts while they are being manipulated; an expensive operation.

4.1.2. SDB

This interface is much cleaner than the comport interface. It is possible to set the condition that sets an event based on the number of words or spaces in the FIFO. The complication here is that the conditions for input and output are packed into a single register, so the input and output drivers interact.

5. OUTLINE DESIGN

An X-Link provides a unidirectional communication path between two processors (or a processor and an external device), and may be thought of as containing a software interface and a hardware interface. The software interface provides a standard set of resources that insulate users from the actual communication mechanisms allowing most X-Links to be driven by a single driver. The hardware interface is responsible for transferring data between the output and input X-Links over a physical connection.



5.1. OPERATIONAL OVERVIEW

A user program can write data to an output X-Link in units of either 32 or 64 bits[2] using the processor's normal store instructions. The software interface is then responsible for packaging this data in a form suitable for the hardware interface to process and send across the physical link connection. For example, an X-Link driving a comport connection will transmit in units of 32-bit words: one word will be sent for every 32-bit write from the user program and two words will be sent for every 64-bit write. When the connection is Rocket I/O, the hardware interface will wait until it has gathered 64-bits before sending them across the link. On the other end, the input X-Link's hardware interface will pass data from the physical connection to the software interface where the user program can access it with 32-bit or 64-bit load instructions.

5.2. NON-64-BIT TRANSFERS

A 64-bit physical link needs a mechanism to complete output transfers involving an odd number of 32-bit words, as these would otherwise leave the output X-Link waiting for another 32-bits to build a complete a 64-bit transfer. The X-Link provides two ways of completing such transfers: a flush command or a time-out.

5.2.1. Flush Command

The user program can explicitly issue a flush command to an output X-Link. This will cause the following actions:

- If the FIFO does not contain a single 32-bit value, nothing happens and the flush command is ignored;

[2] In fact, X-Links can be capable of working in any multiple of 32-bit words, but we shall simplify this document by only considering 32-bit or 64-bit objects.

- If the interface has exactly one 32-bit word of data not yet transmitted, it will act as though it had received a dummy 32-bit write. This allows it to pass 64-bits to the hardware interface along with a signal that the data value has been padded. The hardware interface is now responsible for transmitting both the 64-bit value and the padding signal so that the receiving hardware interface can know to provide only 32-bits to its software interface. This mechanism depends on the communication link, and will be discussed later. For the time being it can be ignored and we simply assume that whatever is written to an output X-Link can be read by the associated input X-Link (or device).

5.2.2. Time-out

Each output X-Link includes a counter that can be used to generate a time-out when a single 32-bit word is waiting to be transmitted. The size of this time-out is completely implementation-defined, but would normally be long enough to allow software to provide another 32-bit write (perhaps following an interrupt marking the completion of a DMA transfer) and short enough to prevent excessive waiting by the receiver. Currently, a value of about 1μs is thought to be appropriate. The time-out is reset every time a value is written to the device. When a time-out occurs, the X-Link will behave as though it had received a flush command (see above). Each output device can be commanded to disable the time-out.

5.3. SOFTWARE MECHANISM

5.3.1. Output Examples

The design of the X-Link means that we can discuss sending data without needing to consider what happens at the receiving end of the communication.

5.3.1.1. Example 1

Processor X needs to transmit 10 words from a buffer that is aligned on a 64-bit boundary.

The driver can transfer all the data by repeatedly reading 64 bits from the source buffer and writing them to the output X-Link.

5.3.1.2. Example 2

Processor X needs to transmit 10 words from a buffer not aligned on a 64-bit boundary.

The driver could do this by a sequence of 32-bit reads and writes, but it is likely to be more efficient to use 64-bit operations when possible. The first word is not aligned, so a 64-bit read is not possible, and X has to move it with a 32-bit read and write. The next 8 words *are* on a 64-bit boundary and so can be moved using four 64-bit loads and stores. This leaves a final 32-bit word which is moved using a 32-bit read and write.

5.3.1.3. Example 3

Processor X needs to transmit 11 words from a buffer aligned on a 64-bit boundary.

The first 10 words can be moved using five 64-bit reads and writes. This leaves a single word that must be moved with a 32-bit read and write.

5.3.1.4. Example 4

Processor X needs to transmit 11 words from a buffer that is not aligned on a 64-bit boundary. The first word must be moved using a 32-bit read and write, while the remaining words are aligned on a 64-bit boundary and can be moved with 64-bit operations.

5.3.2. Input Examples

The design of the X-Link means that we can discuss receiving data without needing to consider what happens at the transmitting end of the communication.

5.3.2.1. Example 1

Processor Y needs to receive 10 words to a buffer that is aligned on a 64-bit boundary.
The driver can transfer all the data by repeatedly reading 64 bits from the input X-Link and writing them to the output buffer.

5.3.2.2. Example 2

Processor Y needs to receive 10 words to a buffer not aligned on a 64-bit boundary.
The driver could do this by a sequence of 32-bit reads and writes, but it is likely to be more efficient to use 64-bit operations when possible. The first word is not aligned, so a 64-bit write is not possible, and Y has to move it with a 32-bit read and write. The next 8 words *are* going to a 64-bit boundary and so can be moved using four 64-bit loads and stores. This leaves a final 32-bit word which is moved using a 32-bit operation.

5.3.2.3. Example 3

Processor Y needs to receive 11 words to a buffer aligned on a 64-bit boundary.
The first 10 words can be moved using five 64-bit operations, leaving a single word that must be moved with 32-bit operations.

5.3.2.4. Example 4

Processor Y needs to receive 11 words to a buffer that is not aligned on a 64-bit boundary.
The first word must be moved using a 32-bit read and write, leaving the remaining words to be transmitted to a buffer that is aligned on a 64-bit boundary.

5.4. HARDWARE MECHANISM

Now that we have seen how the software interfaces deal with data, we need to consider how the hardware interfaces actually move the bits. This is hardware-dependent, but we shall give several examples that show how this can be done with various typical interfaces.

5.4.1. Comport example

A comport hardware interface deals with 32-bits at a time (in fact it transmits a 32-bit word as four 8-bit bytes, but we can ignore this here).
The comport hardware interface is presented with either 32-bits or 64-bits of data. If the software interface provides 32-bits, the data is transmitted directly. If the s/w interface provides 64-bits of data, then two 32-bit values are transmitted sequentially. There is no need to transmit any extra information, as there can never be a pending 32-bit write. Because of this, an X-Link implemented comport can communicate directly with older comport devices.

5.4.2. Rocket I/O Example

The Rocket I/O hardware interface can transmit data in units of 64 bits, allowing the software interface to send pairs of 32-bit values. The input hardware interface will receive the 64-bits and pass them to the software interface, which will indicate that its FIFO contains two more 32-bit words available for reading.
When a flush command is executed by the transmitter and a padded 64-bit value is sent over the data channel, a flag value will be sent on a separate control channel. The receiving hardware interface will see the flag and send to the software interface only 32 of the 64 bits it has received. This will result in a single 32-bit word being inserted in the input FIFO.

5.4.3. Hypothetical 64-bit Device Example

In this example we assume that we have a 64-bit transport mechanism between an X-Link transmitter and an X-Link receiver that can transmit data only; a side channel to transmit the flag is not available. Here the data have to be moved using a protocol such as the following:
Collect a packet of 64-bit words. The size of a packet is either some implementation-defined maximum or the number of 64-bit words received before a flush command is given.

Transmit a 64-bit control word giving the number of actual 32-bit words in the packet. This value will include the first word of a transmission when an odd number of data words are being sent. Transmit the remaining even number of 32-bit words in pairs. The input X-Link's hardware interface will be expecting this protocol and will act accordingly.

5.4.4. Extreme Case

In the previous example we were able to preserve the transparency of the data by including extra information in the raw transmission. If for some reason that is not possible, when accessing a 64-bit device not under the control of an X-Link receiver, for example, we will have no option but to inhibit the time-out and ensure that we only ever send multiples of 64 bits.

6. X-LINK DESIGN

6.1. REGISTER ACCESS

The X-Link specification deals explicitly with a potential problem in accessing device registers. In a multi-threaded environment (which includes environments with a background task and interrupt handlers), changing bits within a register needs to be done indivisibly. If the only way to change the bits is by a read/modify/write sequence, software will be forced to disable interrupts before the sequence and re-enable them afterwards. This is tricky and time-consuming. Instead, the X-Link provides sets of three registers:

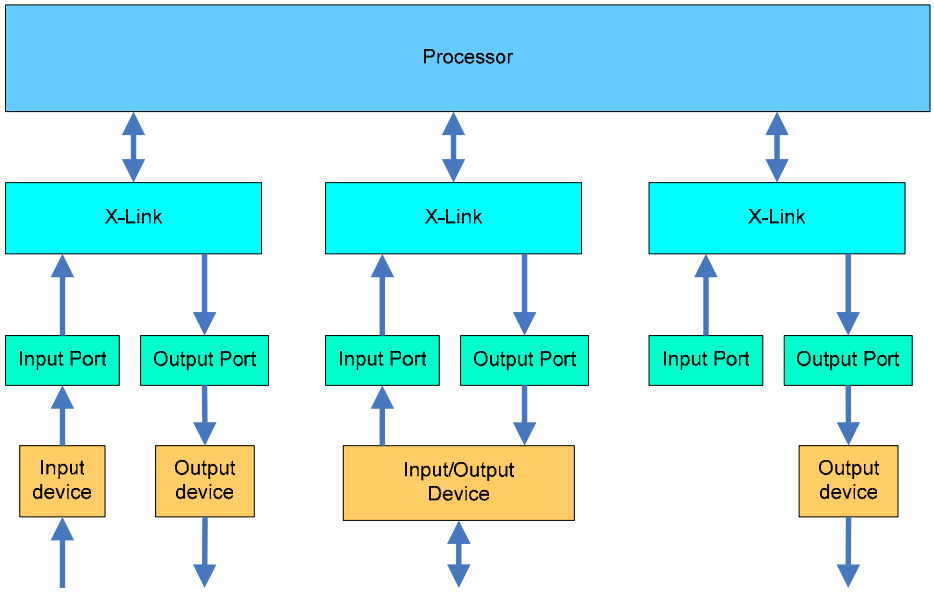
- a read-only register that gives the current state of the bits;
- a write-only register to set the bits: writing a one will set the corresponding bit, writing a zero has no effect.
- a write-only register to clear the bits: writing a one will clear the corresponding bit, writing a zero has no effect.

6.2. INTERFACE

The X-Link is a generic 32-bit interface that can be used with any communication resource to send data to and receive data from external devices.

6.3. PORTS & DEVICES

Each X-Link is made up from two components: an *input port* and an *output port;* the ports are connected to actual *devices*. Most X-Links will have both ports connected, but it is possible to have X-Links with only an input or an output port connected. The input and output ports are completely independent; even if the underlying hardware devices combine both input and output, the X-Link must present them as independent ports. X-Links *can* support devices where the input and output sides require knowledge of each other, but they would require specialised drivers and their use is strongly deprecated.

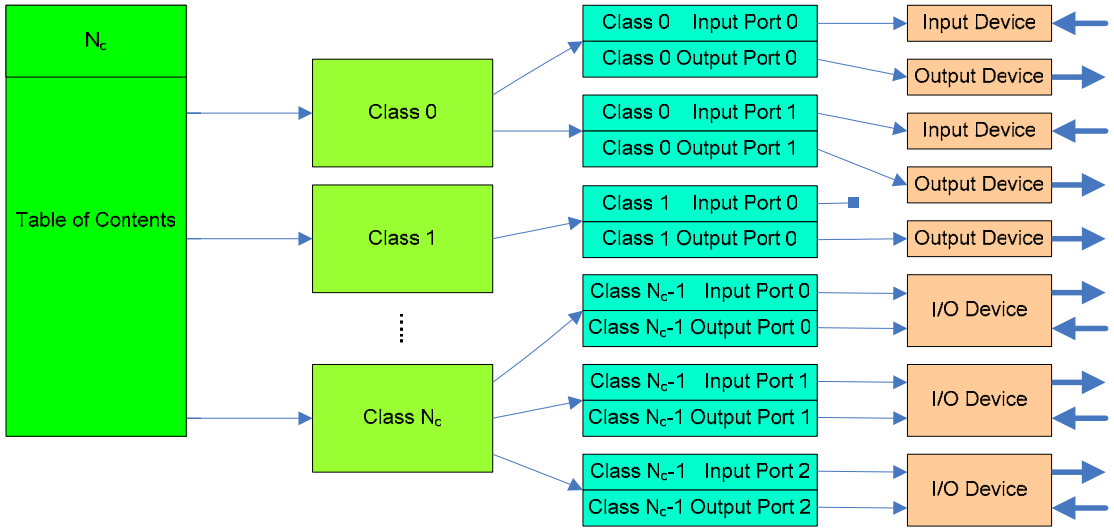


6.4. CLASSES

X-Links are grouped together into *classes*. All the X-Links in a class have identical properties. There will be a class of X-Links for Rocket I/O, a class for SDBs, and so on. There is an absolute maximum of 32 X-Links within a class, although most implementations with actually have far fewer, and current designs impose a limit of 16; there is no limit on the number of classes. Associated with each class are registers used to control *events* generated by devices of the class. Events are used to generate interrupts and synchronise DMA, and will be discussed later.

6.5. TABLE OF CONTENTS

The classes within a particular implementation are identified by a *table of contents (TOC)*. This gives the number of supported classes and for each class an indication of the type of device it supports and the location of the class information. The TOC appears at the fixed byte offset of 0x00100000 from the start of the FPGA; the offsets of no other X-Link information may be fixed. Device drivers are required to locate the information they need dynamically (usually during initialisation) starting from the TOC.



6.6. TRANSFER MODES

X-Link ports can operate in two modes: *FIFO mode* and *memory mode*. These modes are selected by the user, but the mode of each port following reset must be defined by the implementation. X-Link ports do not need to support both modes. All transfer operations are in units of 32 bits; there is no support for byte transfers.

6.6.1. FIFO Mode

A FIFO mode port will transfer data between the external device and a FIFO within the port; it is up to the user to transfer data between the processor's memory and the FIFO. Data transfers will start automatically: data will be accepted from an input device as long as the input FIFO has space, and any data in the FIFO will be sent to a ready output device. A register in the port gives the actual size of the FIFO, in units of 32-bit words. This is constrained to be at least 8 words. For performance reasons, a FIFO may appear as an array in the FPGA, however, each word in the array is mapped onto the top of the FIFO. A status bit indicates if the port can address the FIFO as an array. FIFO mode is likely to be the most efficient way for DSPs to access devices.

6.6.2. Memory Mode

A memory mode port is given an area of the processor's physical memory by the user. The memory is defined by its start address and length in words. Transfers between this memory and the device are initiated by sending the port an explicit "execute" command. Memory mode is likely to be the most efficient way for host PCs to access devices.

6.7. EVENTS

An X-Link port needs to be able to signal that an output transfer can proceed or that data are available for input. This is done by the port raising an *event* when a user-specified *trigger condition* is satisfied. The event can then be used by a program in one of three ways:

1. as a condition to generate an interrupt;
2. as a synchronization signal to a DMA device; or
3. as a flag than can be tested in a polling loop.

The event state is visible in the port's status register.

6.7.1. Trigger conditions

Each port contains a *trigger register,* and the unsigned value this contains determines when the port raises its event.

- A FIFO-mode input port will raise its event when the number of words in the input FIFO is greater than or equal to the trigger value.
- A FIFO-mode output port will raise its even when the number of free spaces in the output FIFO is greater than or equal to the trigger value.
- A memory-mode input port will raise its event (and stop transferring) when the number of words transferred to memory equals the trigger value.
- A memory-mode output port will raise its event (and stop transferring) when the number of words transferred from memory equals the trigger value.

When an event is raised, a port can optionally cause a pulse to be sent to an external interrupt line.

Following reset, the trigger register shall be set to 1. The effects of setting the trigger register to zero are not defined.

6.7.2. Interrupt Lines

While polling a port's event state is useful in certain situations, normal device operation usually requires interrupts or DMA synchronization, both of which involve the processor's external interrupt lines. The X-Link provides a mechanism to connect events to these lines.

6.7.2.1. Selecting Interrupt Lines

Each class of X-Link has a set of 16 interrupt line registers. Some or all of these will correspond to the DSP's interrupt lines and some may not be implemented; which lines are connected is implementation-defined.

A set of interrupt line registers includes two bits for each X-Link: one for the output port and one for the input port. When set, the bit connects the port's event to the corresponding external interrupt line. Note that each port has an option bit that can explicitly disable its event from affecting these lines independently of the setting of the interrupt line registers.

6.7.2.2. Pulsing Interrupt Lines

When connected to a port's event, an interrupt line will be pulsed (raised and then lowered) when an enabled event changes from 0 to 1. The interrupt line will not be pulsed again until one of the following has occurred:

- the processor has read the number of words specified in the trigger register from the port's input FIFO; or
- the processor has written the number of words specified in the trigger register to the port's input FIFO; or
- a FIFO-mode port's trigger value is written; or
- a new memory-mode transfer is initiated on the port; or
- the port's event is disconnected then connected again.

6.7.2.3. Event State

When an X-Link has interrupted the DSP, the interrupt handler needs to determine which port (or ports) is responsible. To do this, it looks at the event state register in the interrupt line registers corresponding to the line that has caused the interrupt. A set bit indicates that the corresponding port is interrupting[3] and requesting service.

[3] For this bit to result in an interrupt, the appropriate processor interrupt mechanism must be enabled.

7. IMPLEMENTATION

7.1. REGISTERS

All X-Link registers are 64 bits long, although software may choose to read only the low-order 32 bits. The registers are described by the following structure:

```
typedef struct {
   VUINT32     lo;       // low-order 32 bits
   VUINT32     hi;       // high-order 32 bits
} XLINK_REG;
```

In 32-bit implementations the high-order bits shall read as 0.

7.2. THE TABLE OF CONTENTS

The TOC is described by the following structures:

```
typedef struct {
   XLINK_REG   type;      // class identifier
   XLINK_REG   offset;    // table of devices
} XLINK_CLASS;

typedef struct {
   XLINK_REG   count;       // number of classes
   XLINK_CLASS classes[1];  // really class[classes]
} XLINK_TOC;
```

Offset		Description
TOC offset	+0x00	The number of types of communication resource classes.
	+0x08	Identification code for class 0.
	+0x10	Offset of class 0.
	+0x18	Identification code for class 1.
	+0x20	Offset of class 1.
	...	...

7.2.1. Address of the TOC

The TOC is located at byte offset 0x01000000 from the start of the FPGA.

7.2.2. Class identification

Type number	Description
0	Unknown device
1	Generic X-Link
2	Comport X-Link
3	SDB16 X-Link
4	SDB32 X-Link
5	RSL X-Link
6..127	Reserved
128	Test Device
129…	Reserved for allocation to Sundance Customers

7.3. THE TABLE OF DEVICES

A class offset in the TOC identifies a *Table of Devices* (TOD) for each class. Each TOD is described by the following structures:

```
typedef struct {
   XLINK_REG   get_offset;  // offset of X-Link input port
   XLINK_REG   put_offset;  // offset of X-Link output port
} XLINK_DEVS;

typedef struct {
   XLINK_REG   devices;     // number of devices
   XLINK_REG   events;      // class events offset
   XLINK_REG   reserved1;
   XLINK_REG   reserved2;
   XLINK_REG   reserved3;
   XLINK_DEVS  device[1];   // really device[devices]
} XLINK_TOD;
```

Offset		Comment
TOD offset	+0x00	Number of X-Links (pairs of ports) in the class
	+0x08	Offset of the class event block
	+0x10	Reserved
	+0x18	Reserved
	+0x20	Reserved
	+0x28	X-Link 0 input offset
	+0x30	X-Link 0 output offset
	+0x38	X-Link 1 input offset
	+0x40	X-Link 1 output offset
	...	...

7.4. X-LINK PORTS

Each X-Link port is described by the following structure:

```
typedef struct {
   XLINK_REG   status;        // port status (read-only)
   XLINK_REG   osr;           // option set register (write-only)
   XLINK_REG   ocr;           // option reset register (write-only)
   XLINK_REG   level;         // FIFO level (read-only)
   XLINK_REG   trigger;       // trigger condition
   XLINK_REG   size;          // size of the FIFO (read-only)
   XLINK_REG   fifo_mem;      // FIFO offset or memory address
   XLINK_REG   extras;        // number of extra registers (read-only)
   XLINK_REG   extra[1];      // really extra[extras]
} XLINK_PORT;
```

Register	Byte offset	Description
status	**0x00**	Device status (read-only)
osr	**0x08**	Option set register (write-only)
ocr	**0x10**	Option clear register (write-only)
level	**0x18**	For input, the number of words available to be read; for output, the number of spaces available to be written (read-only).
trigger	**0x20**	The number of words or spaces in the FIFO needed to set the device's condition. The effects of setting this register to zero are undefined.
size	**0x28**	The size of the FIFO in words (read-only).
memory	**0x30**	FIFO MODE: The offset of the first word of the FIFO from the FPGA base (read-only). MEMORY MODE: The user-supplied address of the memory area to be used (read-write).
extras	**0x38**	The number of registers following (read-only).
extra	**0x40…**	Implementation-defined registers that are not considered part of the standard interface and which should not need to be used during normal operation. Note that there may be cases when these registers are used during device initialisation (e.g., SDB direction if no automatic solution can be found). These registers could also provide hardware diagnostic information.

7.4.1. status: Status Register

This read-only register gathers information about the state of a port.

Bit	Use	Description
0	**Event**	0: the port's event is clear 1: the port's event is set
1	**Event enable**	0: the port's event is disabled 1: the port's event is enabled
2	**FIFO full**	0: the FIFO is not full 1: the FIFO is full
3	**FIFO empty**	0: the FIFO is not empty 1: the FIFO is empty
7	**Error**	0: no error has been detected 1: an error has been detected. More information may be provided in the extra registers. A device reset clears this bit.
8	**Mode**	0: FIFO mode 1: Memory mode
9	**FIFO access**	0: Only the first word of the FIFO can be addressed. 1: The FIFO can be addressed as an array.
10	**Not connected**	0: unknown connection status 1: the port is not connected to a device
11	**Time-out**	0: time-out feature disabled 1: time-out feature enabled

Notes:

- Event enable (bit 1) controls the connection of the event to the external interrupt lines; it does not affect the state of bit 0 (Event). See 6.7.2.1.
- You can not read from an empty input FIFO nor write to a full output FIFO.

7.4.2. osr: Option Set Register

This write-only register is used to set various options for a port. Writing one bit will set the option; zero bits are ignored. Bits that are not defined must be zero.

Bit	Use	Description
0	**Execute transfer**	Start a memory transfer using parameters previously set in "trigger" and "memory"; the bit is ignored in FIFO mode.
1	**Event enable**	Enable the port's event.
4	**Flush command**	Flush any word remaining in the transmitter FIFO.
7	**Reset**	Empty the FIFO, clear any error indication, and reset the device logic. All other osr bits are ignored when this bit is 1.
8	**Memory Mode**	Enable memory mode.
11	**Time-out**	Enable time-out feature. Words are flushed automatically after the time-out expires. The bit will be set following a reset.
16	**Cancel transfer**	Only relevant in memory mode. Cancels the current transfer and signals the port's event to acknowledge the cancel operation.

7.4.3. ocr: Option Clear Register

This write-only register is used to clear various options for a port. Writing a one bit will clear the option; zero bits are ignored. Bits that are not defined must be zero.

Bit	Use	Description
1	**Event enable**	Disable the port's event
8	**Memory Mode**	Disable Memory mode and select FIFO mode.
11	**Time-out**	Disable the time-out feature.

7.4.4. level: FIFO level register

This register indicates how many 32-bit words can be written to an output port or read from an input port.

Bits	Use	Description

31..0	**Level**	The number of words that can be written to or read from the FIFO.
63..32	**-**	MBZ

7.4.5. trigger: Trigger condition register

This register defines the length of a transfer. In FIFO mode it is the trigger value used to set the port's event. In memory mode it gives the number of words to be transferred when the execute bit is written to the **osr** register.

Bits	Use	Description
31..0	**trigger**	The length of a transfer in 32-bit words (see 6.7.1.)
63..32	**-**	MBZ

7.4.6. size: FIFO size register

This register gives the actual size of the FIFO in words.

Bits	Use	Description
31..0	**Size**	The number of 32-bit words the FIFO can store. This value will be zero for devices that only support memory mode transfers.
63..32	**-**	MBZ

7.4.7. memory: Port memory register

This register locates the address of the first word of the port's memory—the port's FIFO when in FIFO mode, or the user's memory area when in memory mode.

Bits	Field	Description
31..0	**Memory**	**FIFO mode**: the offset from the start of the FPGA of the first word of the port's FIFO (read-only). **memory mode**: the address of a memory buffer (read-write).
63..32	**-**	MBZ

7.4.8. extras: Number of extra registers

Bits	Use	Description
31..0	**Extras**	The number of extra registers. The value must be at least 2.
63..32	**-**	MBZ

7.4.9. extras[…]: Extra registers

The extra registers provide implementation-defined information. The first two registers will always be the Interface Identification Value and the Error Status. The number and position of any other registers are implementation-defined.

7.4.9.1. extras[0]: Interface Identification Value

This register gathers information allowing user to identify the interface.

Bits	Field	Description
0..7	**Type**	Interface type.
8..15	**Company**	Company identifier. 0: Sundance 1: Unallocated – available for testing 2..255: available for allocation to Sundance customers
16..63	**Version**	Company-specific information.

7.4.9.2. extras[1]: Error status[4]

Bit	Use	Description
0	**FIFO full**	Attempt to write to a full FIFO
1	**FIFO empty**	Attempt to read from an empty FIFO

[4] Not yet implemented

7.5. EVENT CONTROL

The event control registers are described by the following structures:

```
typedef struct {
   XLINK_REG   esr;          // event state register. read-only
   XLINK_REG   ecr;          // event connect register. write-only
   XLINK_REG   edr;          // event disconnect register. write-only
} XLINK_EREG;

typedef struct {
   XLINK_EREG line[16];      // only line[4..7] are used on the C6000
} XLINK_EVENTS;
```

The events field in the TOD for a class gives the offset from the base of the FPGA of a XLINK_EVENTS structure. This provides sixteen lines that correspond to interrupt lines on the target processor. An implementation may define that only a restricted number of these lines can be used. On C6000 TIMs, only line[4], line[5], line[6], and line[7] can be used, and these correspond to the processor's EXTINT4, EXTINT5, EXTINT6, and EXTINT7 interrupt lines.

Each line contains three 64-bit registers, controlling the ports connected to the associated interrupt line. The bit allocations are as follows:

63	**62**	**…**	**…**	**35**	**34**	**33**	**32**
Port 16 Output	Port 16 Input			Port 30 Output	Port 30 Input	Port 31 Output	Port 31 Input
31	**30**	**…**	**…**	**3**	**2**	**1**	**0**
Port 0 Output	Port 0 Input			Port 14 Output	Port 14 Input	Port 15 Output	Port 15 Input

The top 32-bits must be zero for 32-bit implementations.

Each port in the class is either connected to an interrupt line or disconnected from an interrupt line. All ports are disconnected on reset.

Writing to the *Event Connect Register* (ecr) will connect the interrupt line to all ports corresponding to ones in the written value; writing zero bits has no effect.

Writing to the *Event Disconnect Register* (edr) will disconnect the interrupt line from all ports corresponding to ones in the written value; writing zero bits has no effect.

The value in the Event State Register (esr) has a one bit for each connected port that has raised its enabled event. If the processor has interrupts enabled appropriately, these bits show the ports responsible for an interrupt on the particular line. The bit is cleared by any of the following:

1. the processor reads the number of words in the trigger register from an input FIFO; or
2. the processor writes the number of words in the trigger register to an input FIFO; or
3. a FIFO-mode port's trigger value is written; or
4. a new memory-mode transfer is initiated; or

5. the port's event is disconnected.

7.6. ESTIMATED RESOURCE USAGE

About 160 x 1-bit registers are needed to implement an X-Link.

About 128 x 1-bit registers are needed to implement an event management interface.

There is no easy way to estimate the logic usage.

7.7. VARIANTS OF X-LINK

This document specifies the basic X-Link interface. The number and type of extra registers implemented in X-Link is not fixed by this document. Therefore, it would be possible to add other registers to the list provided here. The interface resulting will be a variant of the X-Link. A new specification will be written for each variant of the X-Link.

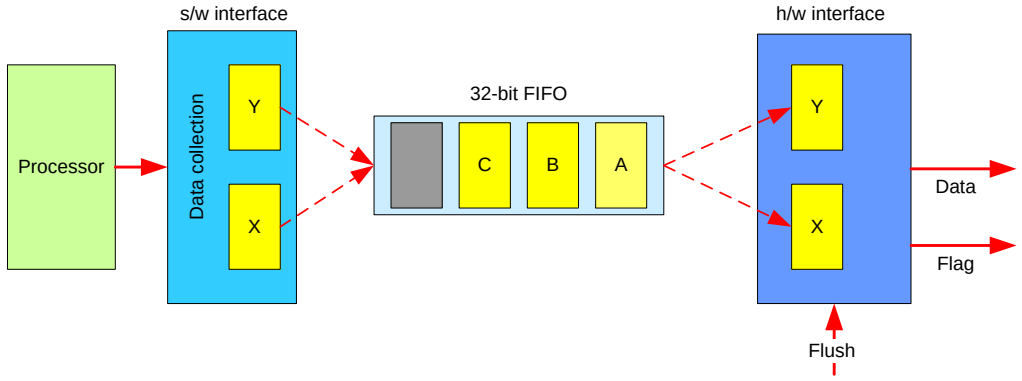
8. OUTLINE FIRMWARE DESIGN

This section is for the firmware designers to discuss any implementation related issues.

8.1. OUTPUT X-LINK

The conceptual design of an X-Link is shown in the diagram; any X-Link must perform "as if" this is how it works. The processor writes 32-bit or 64-bit words to the software interface. The s/w interface writes these words, in sequence, into the 32-bit FIFO as shown.

Word X is the low-order 32 bits of the processor bus and Y is the high-order 32 bits. For example, if the processor performs a 32-bit write operation, word X is stored into the FIFO. In the case of a 64-bit processor write, X and then Y are stored into the FIFO.



Consider the processor writing three 32-bit words to the device. The accesses might be as follows:

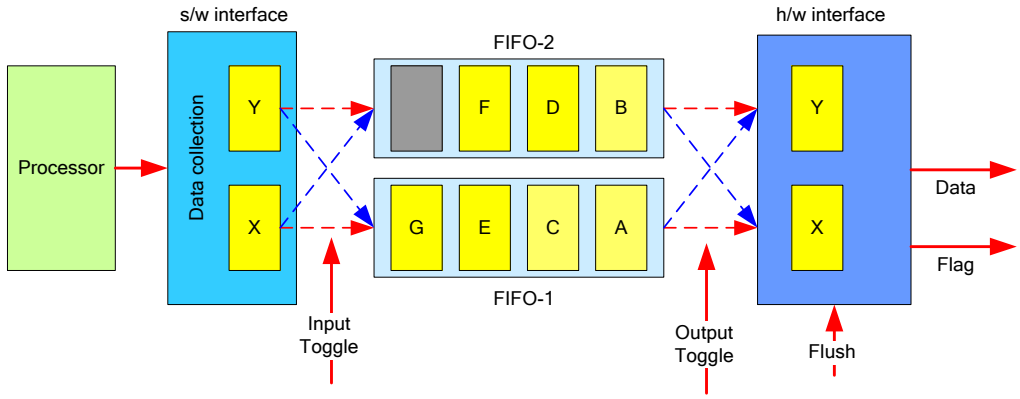
Access type	High-order bits (Y)	Low-order bits (X)
32-bit write		A
64-bit write	C	B

In the case shown, we assume that we have a 64-bit physical transfer mechanism. The hardware interface will try to take two 32-bit words from the FIFO at a time. These words are combined into a 64-bit value and transmitted. If there is only a single word in the FIFO, nothing happens until a flush is issued (by program or by time-out). Then, the h/w interface will pad the high 32-bit part to create a 64-bit word that is transmitted along with a flag indicating that only half of the 64 bits are valid. It is up to the receiving side to interpret the flag, and process the correct amount of data.

8.2. CLOSER TO THE TRUTH

The conceptual single 32-bit FIFO described above cannot be implemented in the FPGA. When the processor makes a 64-bit write, we would need to store two 32-bit values in the 32-bit FIFO within the time of the processor's write access. This would require that the FPGA clock frequency be doubled, and this is physically impossible.

We therefore suggest the scheme shown below. Two separate 32-bit FIFOs are used and data words are distributed between them. The level of the X-Link FIFO is the sum of the levels of the two 32-bit FIFOs.



The software's X and Y registers are connected to the two FIFOs by a switch that can toggle between connecting X to FIFO-1 and Y to FIFO-2 and connecting X to FIFO-2 and Y to FIFO-1.

On a 64-bit write, both values X & Y are moved into their selected FIFOs; the state of the connection is not changed.

On a 32-bit write, X is moved into its selected FIFO and the selection is toggled.

For example, imagine that words A to G have been written as shown:

Access type	**High 32-bit word (Y)**	**Low 32-bit word (X)**	**Input toggle**
64-bit write	B	A	X → FIFO1 Y → FIFO2
64-bit write	D	C	X → FIFO1 Y → FIFO2
32-bit write		E	X → FIFO1
64-bit write	G	F	X → FIFO2 Y → FIFO1

An alternative way the words might have been written is shown.

Access type	High 32-bit word (Y)	Low 32-bit word (X)	Input toggle
32-bit write		A	X → FIFO1
64-bit write	C	B	X → FIFO2 Y → FIFO1
64-bit write	E	D	X → FIFO2 Y → FIFO1
64-bit write	G	F	X → FIFO2 Y → FIFO1

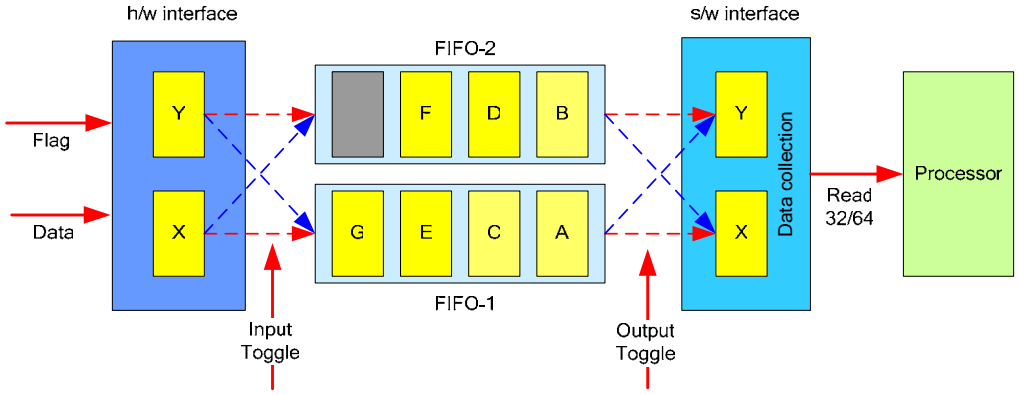
Note that these two ways of writing the same data leave the FIFOs in identical states.

The hardware interface reads words from the FIFOs in an equivalent way, with 32-bit reads changing the "Output Toggle".

Hardware interfaces such as the RSL will read 64 bits at a time from the FIFOs. The hardware interface will wait until two words are available in the FIFO, or there is only one 32-bit word and it gets a flush command. If the hardware interface is transmitting 32 bits at a time, as in the case of the comport, then the hardware interface can to take a single 32-bit word at a time from the FIFO and always ignore flush commands.

8.3. INPUT X-LINK

This section deals with the input X-Link. The figure shows the data and flag signals transmitted to the hardware interface. The hardware interface uses the flag signal to select how much of the transmitted data is valid. In a similar arrangement to the output X-Link, the hardware interface will either write a 32-bit word, or two 32-bit words to the FIFOs. If the flag signal indicates that only one 32-bit word is valid, word X will be written to the FIFOs, and the state of the "Input toggle" signal is changed. When the flag signal indicates that two words are valid, both words X and Y are written to the FIFOs, and the state of "Input toggle" doesn't change.



When the processor makes a 32-bit read access, the software interface will take a single 32-bit word from the FIFOs, and change the state of the "Output toggle" signal. In the case of a 64-bit read operation, the software interface takes two 32-bit words from the FIFO and leaves the state of the "Output toggle" unchanged.

9. NOTES

9.1. NAME

The name of the interface is **X-Link**, and it should be rendered in this style in all documents.