The IPbus Protocol

→ *An IP-based control protocol for ATCA/μTCA*

Version 1.4 - draft 1

18th Oct, 2011

Robert Frazier, Greg Iles, Dave Newbold, Andrew Rose, Dave Sankey

(Authors from v1.2 and previous: Jeremiah Mans, Erich Frahm, Eric Hazen)

Introduction

This document describes a simple, reliable, IP-based protocol for controlling hardware devices. It assumes the existence of a virtual bus with 32-bit word addressing (*i.e.* allowing up to $\mathbf{2^{34}}$ bytes to be addressed) and 32-bit data transfer. The choice of 32-bit widths is fixed in this protocol, though the target host is free to ignore address or data lines if desired.

Terminology

- ***IPbus transaction***

  ⇒ An individual IPbus request or response, e.g. a block read request.

- ***IPbus packet***

  ⇒ One or more individual *IPbus transactions* that are concatenated together to form the payload of the *transport protocol*.

- ***IPbus client***

  ⇒ The software client that generates IPbus transaction requests to control an *IPbus host* device.

- ***IPbus host***

  ⇒ The (hardware) device that responds to – and is controlled by – IPbus transaction requests from an *IPbus client*.

- ***Transport protocol***

  ⇒ The protocol responsible for transporting the IPbus packet to/from the client/host, *e.g.* the User Datagram Protocol (UDP).

Protocol Overview

The various different types of IPbus transactions are described in the *Transaction Types* section of this document. IPbus transactions can be concatenated together as necessary, in order to improve the efficiency of transport across the network if so desired.

For typical use-cases, with the hardware being controlled on an exclusive private network with simple topology, the recommended transport is UDP. Therefore, each transaction or set of transactions must fit into a single UDP packet. Note that the maximum size of a standard Ethernet packet (without using jumbo frames) is 1500 bytes; with an IP header of 20 bytes and a UDP header of 8 bytes, this gives the maximum IPbus packet size of 368 32-bit words, or 1472 bytes. Longer block transfers must be split at the software level into individual packets. If jumbo frames are in use, the IPbus packet size can be much greater – somewhere in excess of 2200 32-bit words.

An IPbus packet consists of a set of transactions. There is no overall header, however each individual IPbus transaction within the packet has its own header. The number of transactions in a given UDP packet must be deduced from the length of the packet and its content.

Each IPbus transaction carries a standard 32-bit header that describes the content of that particular transaction request/response. This header consists of an IPbus *Protocol Version*, a *Words* field to aid with sizing of variable-length transactions, a *Transaction ID* for keeping track of each individual transaction, a *Type ID* to describe the type of transaction request/response, and an *Info Code* that describes the direction of the transaction and its error status. Full details of the header format are described in the *IPbus Header* section further down.

In this version of the protocol, proper support for usage of jumbo Ethernet frames has been introduced. This necessitated a new header format in order to include a wider “WORDS” field. The header has also been simplified, with the fields byte-aligned, to make it much more user-friendly. Also note that the transaction type IDs have changed from the previous version of the protocol, with the type IDs now being in a more logical and consistent ordering. Other than these rearrangements and reorderings, the protocol itself is unchanged from v1.3.

IPbus Header

Each IPbus v1.4 transaction must start with a 32-bit header of the following format:

31 28	**27 16**	**15 8**	**7 4**	**3 0**
Protocol Version (4 bits)	**Words** (12 bits)	**Transaction ID** (8 bits)	**Type ID** (4 bits)	**Info Code** (4 bits)

Note that this header is **not** backwards compatible with previous versions of this protocol (v1.3) – the header format has changed. However, the Protocol Version field is guaranteed consistent across all past and future header definitions, allowing software to track these differences. For this version of the protocol (v1.4), the Protocol Version field should always be set to 2.

The definition of the above fields is as follows:

- ***Protocol Version*** (four bits at 31→28)

  ⇒ Protocol version field – should be set to **2** for this version of the protocol.

- ***Words*** (twelve bits at 27→16)

  ⇒ Number of 32-bit words within the addressable memory space of the bus itself that are *interacted with*, *i.e.* written/altered/read in some way.

  ⇒ Defines read/write depth of block reads/writes

- ***Transaction ID*** (eight bits at 15→8)

  ⇒ Transaction identification number, so the client/host can track each transaction if desired.

- ***Type ID*** (four bits at 7→4)

  ⇒ Defines the request/response type of the IPbus transaction – see the *Transaction Types* section for the different ID codes.

- ***Info Code*** (four bits at 3→0)

  ⇒ Defines the direction and error state of the transaction request/response. See the *Info Codes* section for the different codes.

Info Codes

The *Info Code* is a 4-bit field in the IPbus header that encodes the direction of the transaction and its error state. All requests (*i.e.* client to host) must have an *Info Code* of `0xf` – this is the only code allowed for requests. Requests that are successfully served by the host should have an *Info Code* of `0x0` in the response. All other values are response error codes that detail exactly how a request failed to be served by the host, or are reserved and have no specified meaning in this version of the protocol.

Info Code	Direction	Meaning
`0x0`	*Response*	*Request handled successfully by host*
`0x1`	*Response*	*Bad header*
`0x2`	*Response*	*Bus error on read*
`0x3`	*Response*	*Bus error on write*
`0x4`	*Response*	*Bus timeout on read*
`0x5`	*Response*	*Bus timeout on write*
`0x6`	*n/a*	*Rsvd*
`0x7`	*n/a*	*Rsvd*
`0x8`	*n/a*	*Rsvd*
`0x9`	*n/a*	*Rsvd*
`0xa`	*n/a*	*Rsvd*
`0xb`	*n/a*	*Rsvd*
`0xc`	*n/a*	*Rsvd*
`0xd`	*n/a*	*Rsvd*
`0xe`	*n/a*	*Rsvd*
`0xf`	*Request*	*Outbound request*

Transaction Types

Each of the possible IPbus transaction request/response types are described in the sections below. All have a specific 32-bit IPbus header, and most – but not all – have a body/payload.

Byte-order/Idle Transaction (Type ID = `0xf`)

For IPbus client efficiency in standard CPUs, it is required that the IPbus host firmware handles both possible byte-orderings (big-endian or little-endian). To achieve this, each IPbus packet should begin with a transaction of the form below. It is guaranteed that this is the only transaction request where the upper nibble of the least-significant byte is `0xf`, and thus the IPbus packet's byte-ordering can be determined by the host[1].

These byte-order transactions also operate as idle or "padding" transactions where necessary. For instance, if Ethernet is the link-layer protocol, it has a minimum payload size of 46 bytes. With IP used as the network-layer protocol, 20 bytes are used for the IP header. With UDP as the transport-layer protocol, a further 8 bytes are used for the UDP header. Thus, the minimum UDP payload size is 18 bytes, or five 32-bit words. If an IPbus packet falls short of minimum transport-protocol packet-size requirements, then the appropriate number of byte-order/idle transactions can be used as padding to make up the shortfall. Note that Ethernet should automatically append padding (zeros) to any Ethernet payload that is too small, but it may be more transparent/reliable to use the byte-order transactions as padding instead.

`Request`

	31 24	**23** 16	**15** 8	**7** 0
0	Ver. = **2**	Words = **0**	Transaction ID	Type = **0xf**	Inf. = **0xf**

`Response`

	31 24	**23** 16	**15** 8	**7** 0
0	Ver. = **2**	Words = **0**	Transaction ID	Type = **0xf**	Inf. = **0x0**

[1] An implication of the byte-order transaction is that a future protocol version `0xf` is disallowed.

Read Transaction (Type ID = 0x0)

Request

	31 – 24	23 – 16	15 – 8	7 – 0	
0	Ver. = **2**	Words = **DEPTH**	Transaction ID	Type = **0x0**	Inf. = **0xf**
1	BASE_ADDRESS				

Response

	31 – 24	23 – 16	15 – 8	7 – 0	
0	Ver. = **2**	Words = **DEPTH**	Transaction ID	Type = **0x0**	Inf. = **0x0**
1	Data from BASE_ADDRESS				
2	Data from BASE_ADDRESS + 1				
…	…				
n	Data from BASE_ADDRESS + (DEPTH – 1)				

Non-incrementing Read Transaction (Type ID = 0x2)

The packet formats for this are identical to the to the standard read transaction above, except the transaction type ID is `0x2`. The non-incrementing read is useful for reading from a FIFO.

Write Transaction (Type ID = `0x1`)

`Request`

	31 — 24	23 — 16	15 — 8	7 — 0	
0	Ver. = **2**	Words = **DEPTH**	Transaction ID	Type = **0x1**	Inf. = **0xf**
1	BASE_ADDRESS				
2	Data for BASE_ADDRESS				
3	Data for BASE_ADDRESS + 1				
…	…				
n	Data for BASE_ADDRESS + (DEPTH – 1)				

`Response`

	31 — 24	23 — 16	15 — 8	7 — 0	
0	Ver. = **2**	Words = **DEPTH**	Transaction ID	Type = **0x1**	Inf. = **0x0**

Non-incrementing Write Transaction (Type ID = `0x3`)

The packet formats for this are identical to the to the standard write transaction above, except the transaction type ID is `0x3`. The non-incrementing write is useful for writing to a FIFO.

Read/Modify/Write Bits (RMWbits) Transaction *(Type ID = 0x4)*

The RMWbits transaction is useful for setting or clearing bits. Only single-word (32-bit) RMWbits transactions are defined. The algorithm performed is the following:

$$X <= (X \,\&\, A) \mid B$$

Where ***X*** is the existing value, ***A*** is the AND term, and ***B*** is the OR term.

Using the RMWbits transaction, it is possible to perform efficient "masked writes" on a 32-bit register using just a single transaction.

Request

	31 — 24	23 — 16	15 — 8	7 — 0	
0	Ver. = **2**	Words = **1**	Transaction ID	Type = **0x4**	Inf. = **0xf**
1	BASE_ADDRESS				
2	AND term (***A***)				
3	OR term (***B***)				

Response

	31 — 24	23 — 16	15 — 8	7 — 0	
0	Ver. = **2**	Words = **1**	Transaction ID	Type = **0x4**	Inf. = **0x0**
1	Content of register after transaction				

Read/Modify/Write Sum (RMWsum) Transaction (Type ID = 0x5)

The RMWsum transaction is useful for adding (or subtracting, using two's complement) values from a register. The algorithm performed is the following:

$$X <= (X + A)$$

Where ***X*** is the existing value, and ***A*** is the ADDEND.

Only single-word (32-bit) RMWsum transactions are defined.

Request

	31 — 24	23 — 16	15 — 8	7 — 0	
0	Ver. = **2**	Words = **1**	Transaction ID	Type = **0x5**	Inf. = **0xf**
1	BASE_ADDRESS				
2	ADDEND (***A***)				

Response

	31 — 24	23 — 16	15 — 8	7 — 0	
0	Ver. = **2**	Words = **1**	Transaction ID	Type = **0x5**	Inf. = **0x0**
1	Content of register after transaction				

Get Reserved Address Information Transaction (Type ID = 0xe)

Each target is allowed to determine the location, length, and data width of its reserved address data. This information should be returned by this transaction.

***** MORE INFO NEEDED HERE *****

1. What is the purpose of the "Reserved" field in the response
2. Why do we need a "Data Width" field in the response… 32-bit, no?

Request

	31 24	23 16	15 8	7 0
0	Ver. = **2**	Words = **0**	Transaction ID	Type = **0xe**	Inf. = **0xf**

Response

	31 24	23 16	15 8	7 0
0	Ver. = **2**	Words = **0**	Transaction ID	Type = **0xe**	Inf. = **0x0**
1	Base address of reserved area			
2	Reserved Space Size		Reserved	Data Width

Reserved Addresses

For the purposes of uniformly identifying chips, developers, and firmware versions, a block of address space shall be reserved for identification strings and codes. Specification details to be added in a subsequent version of this document. ********* TO DO *********

The "Get Reserved Address Information" transaction shall be used to extract the information about the base address and data width for any given target. If the reserved size and data width are set to zero, no reserved/identification records are available.

Implementations of the Protocol

***** DELETE? UPDATE? *****

Multiple implementations of the protocol on the target side are possible, depending on the hardware in question. The choices between parallel and serial connections and between microcontroller C-code and HDL will depend on the hardware and application. The known complete or planned implementations are described here.

FPGA Implementation with HDL (HCAL)

***** DELETE? UPDATE? *****

For use in the HCAL development project, an implementation of the protocol has been developed which uses the Verilog HDL and the UDP protocol. For this implementation, the protocol is contained within an IP block which uses the built-in MAC and Gigabit serializer capabilities of the Virtex 5 FPGA and which provides a control interface to the rest of the chip using a simple interface allowing the insertion of wait-states where necessary.

The client interface is a simple asynchronous parallel bus interface. The timing and behavior is similar to a simplified VME and the signals are given the same names as in VME.

- `addr` [31:0] (from Core) : Address bus (word-based addressing)
- `data_o`[31:0] (from Core) : Write data bus
- `data_i`[31:0] (into Core) : Read data bus
- `strobe` (from Core) : indicates active transfer (active high)
- `write` (from Core) : indicates that the current transfer is a write (active high)
- `dtack` (into Core): acknowledge successful transfer (active high)
- `berr` (into Core) : failed transfer (active high)

`write` will always be set before strobe (at least one internal clock cycle before). `strobe` will be held until `dtack` or `berr` is seen. Once `strobe` is deasserted, `dtack` and `berr` should also be deasserted to allow the next cycle to occur.

The maximum theoretical byte rate from gigabit ethernet is 125 MB/s. Each transaction generates at least 8 bytes, either incoming or outgoing and neglecting the Ethernet, IP, and UDP overheads. Thus, the maximum transaction rate is 15.6 M bus cycles/sec. The FPGA client should have a cycle capability of at least 50 MHz if the maximum performance of the interface is to be achieved.