Extended Control Codes for Distributed Interrupts in SpaceWire Networks

“pole” should be “poll” – in several places.

Each interrupt request has to be sent as a separate transaction. If a routing switch receives several interrupt requests at the “same time” (i.e. within a short time interval), should they be handled in any particular order – e.g. highest (or lowest) number first? In other words, should the interrupts be prioritised?

Remote Memory Access Protocol

Section 6.2

Is a 32-bit address enough? We suggest using an option bit to allow an optional additional 32-bit address word to give either 32-bit addressing (option bit = 0, as currently specified) or 64-bit addressing (option bit = 1, two 32-bit address words).

Section 6.3

We generally prefer to authenticate requests and some indication of source or other authentication would be useful – it could be part of the Transaction identifier but 3-bytes is restrictive. A bit or bits in the options byte could be used to allow extension of this field or the optional addition of a source/authentication field.

The statement in section 6.3.3 that authorization is not required as it is “probably better to assume that the source knows what is doing” is dangerous. A corrupted source may not know what it should be doing and over-write data in an area it should not accessing – the ability to trash a different experiment’s data is not desirable.

Figure 1 would benefit from a statement that the leftmost bit is the most significant. (Use of bit numbers is ambiguous, different vendors number from different ends.)

The protocol allows data beyond that expected in the data count to be ignored but an error report is required by section 6.3.4. The “SMCS” chip, in 32-bit mode, reports received data in 32-bit chunks. N 32-bit chunks can result from receiving 4*N, 4*N+1, 4*N+2 or 4*N+3 bytes and thus it is not possible to determine whether there is excess data or not, unless there is enough to run into the next word(s). Section 6.3.4 wording may be too precise for practical implementation.

Indirect Write / Indirect Read

Aren't these the same thing? Using the example given, of a camera and mass memory module… An indirect write is sent to the camera to transfer data from the camera to the mass memory. The reverse of this is an "indirect read" sent to the camera to request data transfer from the mass memory to the camera – which appears top be the same as an indirect write sent to the mass memory to send data from the mass memory to the camera.

We do not understand why the reply from the final destination must go via the intermediate node and not direct to the originating node. The most obvious possibility is to allow the intermediate node to resend data lost or received incorrectly – but this may not be the right way to handle this situation. It is not always possible, or even desirable, to resend corrupt data (see below) and it seems to us that the decision should be made by the node requesting the transfer – the source node. Also, the intermediate node must store state in order to accept a reply from the target and pass in back to the correct source. Sources will have to interact with each other in order to avoid having too many outstanding requests requiring more state memory than is provided in the intermediate node.

We see the need for just three types of transaction – "read", "write" and, let us call it, "transfer". The action of "transfer" is similar to "indirect write". A source node sends a packet to an intermediate node containing an address in the intermediate node from which to read data. The intermediate node forms a "write" packet containing the data read and copies other fields from the incoming "transfer" packet. It thus produces a "write" packet that appears to have come directly from the source node (containing the source node's transaction ID and return path, but the intermediate node's data). The target node replies in the usual way *direct to the source node* so that retry requests are controlled by the source node. Intermediate nodes do not need to keep state to pass-on replies or to handle retries and retry policy; and no additional protocol is required between sources.

Data resend behaviour

We infer, from the existence of an address increment / not increment bit, that transfers of streams of data are possible. Streams of data (non-incrementing read or write address) are unlikely to be able to be re-sent as this requires buffering.

It may not even be possible to resend buffered data since doing so requires additional bandwidth, with unlimited resends consuming excessive bandwidth.

Control of resend is likely to have to be carefully controlled (somewhere between infinite retries, limited retries or once only transmission) and with timeout periods that may depend on the target node. Either the intermediate nodes must be controlled (there is currently no provision for this in the proposal) or resend policy should be left to the source node. We prefer the latter as it considerably simplifies intermediate nodes.