

Updated changes to BOSS

(with contributions by Jerry Hauck, Dave LaFollette and David Wooten)

(Version 0.85, revised 03/22/99)

Changes include:

For 0.85

1. Added “no_request” options for asynch and isoch (thanks, Alistair).
2. Added no_request_odd/even pair as a way to avoid limiting the fairness cycles to subaction gap times (trying to avoid using specified gap times). (Thanks, Mr. Wooten.)
3. Lowered priority for out-of-phase asynch requests below the out-of-phase “no_request”. This allows us to use the out-of-phase “no_request” to keep the fairness cycle from advancing. Useful for both border nodes and to remove the need for explicit minimum times for fairness intervals.

For 0.8

4. Hybrid of current/next and phase_0/1 for asynch arbitration. This hybrid allows pipelining of requests back to the link with minimal race conditions.
5. Various notes and improved explanations added. Some inconsistencies removed.
6. Resource required tables added.
7. Summary of BOSS arbitration added.
8. Simplification of border functionality (thanks Jerry!).

Resources required:

Tokens

The following is a list of tokens needed by the BOSS scheme and associated border functionality. Note that all tokens are transmitted on all ports pointing “away” from the current BOSS. In all cases except “grant”, the BOSS will transmit these tokens out all connected ports.

Table 1 - Tokens

Token name	Comment
Cycle_start_odd	Follows cycle start packet where the low order bit of the cycle count is set.
Cycle_start_even	Follows cycle start packet where the low order bit of the cycle count is clear.
Arb_reset_odd	Sent by BOSS when no current or next_even requests are received.
Arb_reset_even	Sent by BOSS when no current or next_odd requests are received.
Bus_reset	Forces a bus reset. The cycle start phase is unchanged (all new nodes must wait for a cycle start to arrive to determine the phase), the fairness phase will be set to “odd”.
Asynch_start	Sent by BOSS if cycle start has been received, and no isochronous requests of the current phase are received.
Grant	Sent by BOSS to the highest priority request (note that “highest priority” is context sensitive ... see below).

Isoch cycle start

An isochronous cycle is started by an explicit token (an arbitration code or control symbol of "cycle_start_even" or "cycle_start_odd" corresponding to the two different types of isoch requests), pipelining requires this. This means that a cycle start packet will look like:

...<DP><cycle start packet><DE><cycle start token(s)>...

(the cycle start token follows the packet to ease border functionality)

Arbitration reset

There need to be two types of arbitration resets to handle the queuing of fairness cycles.

Grants

There is a single grant token. I thought about having separate grant types for each request (for robustness), but the error modes were too complicated. Instead, there is a separate "asynch_start" token that is used by the BOSS to indicate then end of isochronous transmission. This is used to clear out any late transmissions of isochronous requests of the current phase.

Similarly, there is no "phase" in a grant. If a node doesn't correctly receive an arbration reset token, then it is possible that it will make the wrong type of request, but this will be momentary and will only cause a single fairness error.

Cable request types

We need the following cable request types:

(notes:

- a) asynch and iosch have independent priority lists since both requests are carried in parallel
- b) these are **cable** request types, there will be fewer **link** request types)

Table 2 - Asynch requests

Request name	Priority level	Comment
Cycle_start_req	1 (highest)	
Border_High	2	Keeps the 1394a/b boundary protocol working correctly, done at very high priority to handle legacy PHYs that have short request timeouts.
Next_odd	3 if the last arbitration reset token was arb_reset_odd, else 6	This is a queued request from the previous fairness cycle
Current	4	Used for all normal asynch requests by nodes that haven't used up their fairness budget.
No_req_even	5 if the last arbitration reset token was arb_reset_odd, else 7	Nodes that last saw an arb_reset_even send this request when they have no requests ... having a higher priority than the out-of-phase request keeps those the fairness cycle from advancing until all nodes receive the arb_reset of the current phase. There are two uses: 1) removes the need for programming a specific minimum fairness interval, and 2) allows a border node to synchronize the fairness cycles of the beta and legacy domains.
Next_even	6 if the last arbitration reset token was arb_reset_odd, else 3	For queuing requests across the next fairness cycle
No_req_odd	7 (lowest) if the last	

Request name	Priority level	Comment
	arbitration reset token was arb_reset_odd, else 5	

Table 3 - Isoch requests

Request name	Priority level	Comment
Isoch_odd,	1 (highest) if the last cycle_start was cycle_start_odd, else 2	Used if the last cycle start token was cycle_start_odd and the packet is intended to transmit in the current cycle. If the last cycle start token was cycle_start_even, this request is used if the packet is intended to transmit in the next cycle (which should be an even cycle).
Isoch_even.	2 if the last cycle_start was cycle_start_odd, else 1	Used if the last cycle start token was cycle_start_even and the packet is intended to transmit in the current cycle. If the last cycle start token was cycle_start_odd, this request is used if the packet is intended to transmit in the next cycle (which should be an odd cycle).
No_isoch_req	3 (lowest)	

Beta Bus Operation (normal BOSS mode)

All 1394beta PHYs have two modes of operation: BOSS (temporary bus master) and non-BOSS. A node becomes BOSS if is the last to transmit during a subaction: i.e., it transmitted the "ack" in a directed subaction (perhaps the "ack" to a concatenated response), the data packet in a broadcast subaction or some PHY packets. At any one time there is at most one BOSS node. During normal operation there are short periods of time when no node is BOSS. There are recovery algorithms to handle error conditions of no BOSS for too long, or multiple BOSSes.

The general rule is that if a port is not transmitting data, then it continuously transmits the highest priority request coming from any of the other ports **or** an attached link. The result is that each port not receiving data is receiving a continuously updated "snapshot" of the highest priority request sent by any nodes on that port's subtree. In the case of the BOSS, all ports are receiving requests so it should send a grant to the highest priority port (or its own link) immediately after finishing sending the ack or broadcast packet.

If a BOSS doesn't receive any requests by the time it finishes transmitting data, then it sends an arbitration reset token of the appropriate phase (see next paragraph) and a grant to its parent. The root node ends up being the default BOSS in an idle bus.

Fairness is implemented by using a two priority/two phase approach. As long as a node has its "arb_enable" bit set, it will transmit "current" requests, if its arb_enable bit is reset, it will transmit a lower priority "next_odd" or "next_even" request, depending on whether the last arbitration reset token was "arb_reset_even" or "arb_reset_odd". If it has no requests, it sends a "no_request_odd" or "no_request_even" depending on the last arbitration reset token received. If the BOSS last received an "arb_reset_even" token, it will issue grants first to "next_even" requests (left over from the last fairness cycle), then to "current" requests. If it only sees "next_odd" or "no_request_even" requests, it shall transmit an "arb_reset_odd", then a grant to one of the "next_odd" requests.

The "no_request" of the opposite phase has a higher priority than the "next" of the opposite phase to keep the fairness cycle from advancing until all the nodes have received the current phase arbitration reset.

This process repeats for the "even" fairness cycle, with the "*-odd" and "*-even" signals inverted. Note that the priority of the "next_x" requests with respect to the "current" request inverts on each cycle.

Isochronous arbitration is similar to fairness, although there is no "current" request. When a node receives a cycle_start_odd, then current BOSS shall only respond to isoch_odd requests, and isoch_odd requests have priority over isoch_even requests for non_BOSS nodes repeating requests. The cycle_start_odd token shall be sent by a cycle master immediately after it transmits a cycle start with an odd-numbered cycle count.

Once again, this process inverts for cycle_start_even.

If two operating busses are joined together, then the one that does not have the new cycle master may receive two consecutive cycle starts of the same phase. This means that there may need to be a "cycle start inconsistent" notification.

Hybrid Bus Operation

Attributes:

- Almost all complexity is hidden in a border PHY (a PHY that has at least one port that may be connected in DS mode and at least one in Beta mode, also any PHY that supports 1394-1995 or 1394a Links will also need border functionality),
- Border functionality is only active when needed (i.e., when all ports in DS mode, then only -1995 or -a functionality is used; when all ports are in beta mode **and** link is also beta-capable, then only beta functionality is used).
- Allows considerable overlap between DS-mode arbitration overhead and beta data traffic in the most likely topology (the most important "beta cloud" is where the root is located) (a "beta cloud" is a subnet of the bus that only has beta connections active)
- Works for all topologies

Summary description, operation examined from the point of view of a border PHY.

Case 1: asynch arbitration with the root on the beta side

For a border phy with the root on a beta port, we normally do **not** put DP onto the DS ports if there is traffic on the beta side, instead we allow the DS side to arbitrate in parallel with data transmission on the beta side. In the abstract, we could just accept a request from the DS side, and forward requests from a DS port to the beta cloud using the special Border_High request, which will always win over normal beta requests. Unfortunately, old Apple Firewire PHYs will timeout after sending request for 20 usec and force a bus reset, so we must make sure that all outstanding requests get some kind of response in a timely way.

1) Request received on DS port

2) Request is forwarded to beta port(s) using Border_High priority **and** a 10 usec timer is started (to make sure no node on the DS side will timeout on a request)

Note: If beta side is not busy (not sending or receiving a packet), normal beta-mode protocols guarantee that a grant or a new packet will arrive within 10 usec (this timing is needed to keep an old Firewire PHY from timing out on its request)

3) While timer < 10 μ sec

3.1) if a legacy packet arrives (one that can be sent out the DS port), then it is forwarded to the requesting DS port and the timer is canceled (the DP signal causes the request to be withdrawn; will happen with 10 usec, so Firewire PHY will be happy).

3.2) If a beta packet arrives, continue sending idle on DS port(s) and continue 10 μ sec timer started in (2) above

3.3) If the grant arrives, forward the grant to the DS cloud and cancel the timer.

- 4) if 10 usec timer expires, DP is transmitted on the DS port(s)
- 4.1) if a legacy packet arrives (one that can be sent out a DS port), then it is forwarded to the requesting DS port (the DP started in (4) is the data prefix for this packet).
- 4.2) If a beta packet arrives, continue sending DP on DS port(s)/
- 4.3) If the grant arrives, release the DP sent to the DS cloud and allow any attached nodes to re-arbitrate. Simultaneously transmit DP on the beta side to keep the root from timing out and assuming BOSS function. Start a timer.
- 4.4) While timer < subaction gap
- 4.4.1) If the timer expires, release DP from the beta side
- 4.4.2) If a request arrives from the DS side, respond with a grant, restart timer
- 4.5) While timer < state timeout
- 4.5.1) If the timer expires do a bus reset (just like 1394a)
- 4.5.2) If packet arrives from DS cloud, retransmit to beta side.

Case 2: Fairness with the root on the beta side:

To keep fairness working as expected, once a fairness cycle has started on one side, you don't want a new fairness cycle to finish on the other side. For this discussion, a fairness cycle starts on the first packet after an arbitration reset and finishes when an arbitration reset gap (for DS ports) or arbitration reset token (for beta ports) is received.

So, the requirement is to hold off the ending of a fairness cycle on one side until it can finish on the other. The basic idea is as follows:

If a fairness cycle must be extended on a DS-port, it is done by sending a DP on the DS port (a border node can detect the imminent occurrence of an arbitration reset gap on the DS side by some bus management magic, see below). If a legacy packet arrives on the beta side, it is repeated on the DS side (beta-side packets look to the DS side like a series of concatenated packets). This process is continued until an arbitration reset token is received (or generated) on the beta side.

Bus management magic: for all this to work, the arbitration reset gap on the DS side must be set to a "somewhat larger" value than normal (I think!). The border nodes would use a smaller value to guarantee that they detect when the arbitration reset gap is about to happen and take the appropriate action.

If the fairness cycle must be extended on the beta cloud, the border node sends a Border_Low request, and holds the beta side (sending DP) until an arbitration reset gap appears on the DS side.

Case 3: Isoch operation with the root (cycle master) on the beta side:

The one concern here is that the isoch cycle on the DS side must not end until the isoch cycle is **really** done. This works almost the same way that the fairness cycle is extended: the border node detects the imminent subaction gap at the end of the DS-side isoch data, and keeps it from happening by sending DP. Note that this requires the bus management magic described above to set the gap count to a slightly larger value than normal.

It **continues to be** a requirement that we send cycle starts at S100, unless we can determine via some higher level mechanism that the minimum speed link through the network is higher.

Case 4: Normal arbitration with the root on the DS side:

The major concern here is that the beta cloud must follow the arbitration acceleration exclusion rules of 1394a ... i.e., the beta cloud cannot hold the bus too long after a cycle synch indication, it must allow the DS root to send a cycle start in the proper time. To do this, the border node must attempt to gain control of the beta cloud at cycle synch time. The "Border_High" priority level will allow it to do this, but this also

requires that the border node have an active attached link. If the border node does not have an attached active link, it must send a Border_High request after every packet to pass control back to the DS port.

Case 5: Fairness with the root on the DS side:

Because of the nature of this configuration, there will always be an arbitration reset token on the beta cloud before one can appear on the DS side. This means that the only problem is to hold off the end of a new fairness cycle on the beta side if a fairness cycle has started on the DS side. This is done by the border node sending a Border_Low request whenever there is an arbitration reset token on the beta side. The border node will get a grant before the next arbitration reset token is sent on the beta side, and will send DP until it receives an arbitration reset gap on the DS side, at which point it will send the appropriate arbitration reset token itself.

Case 6: Isochronous with the root on the DS side:

There is no "cycle_start_even/odd" indicator on the DS side, it must be synthesized by one of the nodes on the beta side. Since PHYs cannot decode a cycle-start packet, this must be done by a link. To avoid too much complication in the various state machines, we would like a link to **always** send a message to its attached PHY saying something like "cycle_start_even/odd" received, and the PHY will synthesize the appropriate cycle start token.

***Exercise for the reader:** This means that it will be possible for nodes to receive multiple cycle start tokens of the same phase. We could suppress this by requiring that nodes do not repeat cycle start tokens of the same phase as the last one received. This **may** cause some problems (or may **remove** some!) when two operating busses are connected together.*

Note that there is not a concern with asynchronous packets slipping in after the cycle start is repeated into the beta cloud ... the border node always retains BOSS function during this period (see case 4).