P1394A Enhancements

1.0 Overview

Engineering never stops. Before the 1394 standard was ever brought to vote, more features and enhancements were being discussed (and implemented). In a very real sense, the IEEE 1394-1995 Standard represents not so much a finished work as a snapshot of a work in progress, encompassing the body of knowledge as we knew it in late 1994.

Silicon changes the engineering thought process. Before silicon exists, fundamental changes can be contemplated, changes which may obsolete prior approaches. After silicon, new ideas must pass the test of backward compatibility.

A handful of these backward-compatible enhancements have accumulated. All have been publicly disclosed and discussed; some are patented or patent-pending; some are straightforward; some are still contentious. Regardless of their legal status or palatability, this document will attempt to describe the enhancements in sufficient detail to facilitate technical discussion.

2.0 Connection Debouncing

Devices on a 1394 bus automatically count off during the bus initialization phase, thereby acquiring a physical node number. There may be up to 63 devices on a bus; permissible node numbers range from 0–62. A node number of 63 indicates that bus initialization is pending, and that a valid node number has not yet been assigned. Central 1394 dogma has always dictated that any connection status change—the addition or removal of a device(s) from a 1394 bus—will cause the bus to re-initialize.

2.1 Connection Status Change & 1394 Compliance

Section 4.4.2.1 of the 1394-1995 standard covers the bus reset phase of arbitration. In particular, Section 4.4.2.1.1 outlines the conditions which cause a node to initiate bus reset.

> **TransitionAll:R0b.** This is the entry point to the bus reset process if this node is initiating the process. This happens under the following conditions:
>
> 1) Serial Bus management makes a PHY_CONTROL.request.
> 2) The PHY detects a change in any port's connection status.
> 3) The PHY stays in any state other than Idle, Reset_Wait, Transmit, or Receive for longer than MAX_ARB_STATE_TIME.

So the standard states that a port connection status change shall cause the PHY to transition to Reset Start. The earliest silicon implemented exactly that and nothing more; any connection status change caused immediate transition to Reset Start. There are two problems with the straightforward implementation.

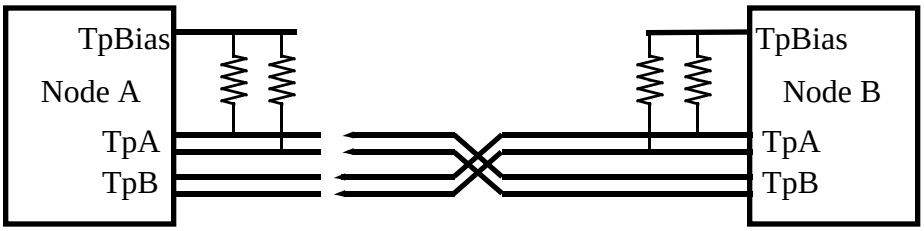


FIGURE 1. Connection Asymmetry - Bias voltage is applied to TpA, sensed on TpB. Node A will sense the new connection first.

1) The connection process is not symmetric. When PHYs A and B are connected, they will generally not recognize the connection at the same time. The earlier node will immediately go to Reset Start, but will be unable to complete bus initialization until the late node also detects the new connect. During this time interval, which may be tens of milliseconds, the early side of the bus thrashes in bus initialization states, and is unable to send or receive packets.
2) The connection process isn't clean; as contacts scrape together, the electrical connection may be made and broken many times. Bus initialization only requires ~200 µseconds; one new connection may generate a storm of connect/disconnect events.

Both of these mechanisms are of short duration—tens of milliseconds—but during this time data throughput is interrupted, which poses a serious drawback for many isochronous applications.

2.2 Contact "Bounce": Proposal for Connection Debouncing

Adding connection debouncing to 1394-1995 operation requires no change to the basic state machine. The only real change is to the definition of *connection_state_change*, the second term of the All:R0b transition shown below.

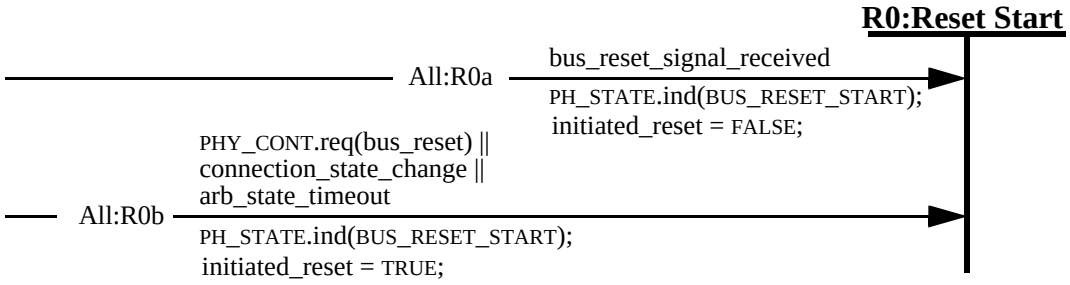


FIGURE 2. 1394-1995 Standard Pathways to Reset State

The standard requires that a connection flag for each port be latched whenever a connect status changes; this stored connect status is *old_connect(i)*. The current connect status, *connected(i)*, is continuously compared to *old_connect(i)*, to determine if there has been a change.

```
boolean connection_state_change()  {
// true if any port goes from connected to disconnected or vice-versa
int i;
state_change = FALSE;
for (i = 0; i< nport; i++)  {
   if (connected[i]  !=  old_connect[i]);  {  //did connection change?
      state_change = TRUE;
      old_connect[i]  =  ~old_connect[i];
      }
   }
return (state_change);
}
```

FIGURE 3. 1394-1995 Definition of connection_state_change

The simplest way to debounce the connection process is to add per port new connect timers. Disconnects may still be recognized immediately. This is sufficient to quell the reset storm resulting from contact scraping.

```
boolean connection_state_change() {
// true if any port goes from connected to disconnected
// true if any port goes from disconnected to connected for the timeout
boolean static connect_in_progress[NPORT];
boolean state_change = FALSE;
int i;
for (i = 0; i < NPORT; i++) {
   if (connect_in_progress[i])
      if (~connected[i])
         connect_in_progress[i] = FALSE;          // lost attempted connection
      else if (connect_timer[i] >= CONNECT_TIMEOUT) {
         connect_in_progress[i] = FALSE;
         old_connect[i] = TRUE;                   // confirmed connection
         state_change = TRUE;
         }
   else if (connected[i] && ~old_connect[i]) {  // possible new connect?
      connect_timer[i] = 0;                       // start connect timer
      connect_in_progress[i] = TRUE;
      }
   else if (~connected[i] && old_connect[i]) {  // disconnect?
      old_connect[i] = FALSE;                     // effective immediately
      state_change = TRUE;
      }
return (state_change);           // collective OR of individual port state changes
}
      connect_timer[n]
      // count up if port[n] connect_in_progress is true
      // count clears to 00 if no port connect_in_progress is true
```

FIGURE 4. Revised Definition of connection_state_change

There are several ways to implement the connection timer. The obvious approach is to have a timer for each port. However, it's a lot of bits to count a 25MHz clock up to reasonable debounce intervals, so the obvious approach is prohibitive.

Nothing is lost if the counter granularity is greater than 40 ns. An N stage counter could be used to generate a clock pulse once every $\left({}^{1}/_{24.576}\right)\cdot 2^{N}$ µseconds. Then each port could have a smaller counter which would count these clock ticks. Suppose N is 17; that yields a clock tick once every 5.33 milliseconds. If the per port timer is an additional six bits, then the timeout is 341.33 milliseconds. Since the first count could occur any time from 0–5.33 milliseconds (the granularity), the timeout is in the range 336–341.33 milliseconds.

2.3 Connection Asymmetry: Normalization Proposal

Regardless of the implementation details for the connection timer, the other problem of connection asymmetry remains. Generally two nodes being connected do not recognize the new connection at the same time; the new connect timer by itself does not solve the problem. In the case where a new PHY connects to an old PHY, the old PHY could thrash in bus initialization states until the new PHY finally recognizes the connection. Even in the more benign case of new PHY–new PHY, the connection recognition could be staggered by several milliseconds, during which time one side of the new connect hangs.

The asymmetry problem can be fixed simply:

1) During the connection timeout interval, ignore all arbitration states from the newly connected port except for bus reset (AB = 11).
2) If bus reset is received during the connection timeout interval, go to Bus Reset Start.

There are still some problem cases. Suppose a single device with an old PHY is plugged into a port on a new PHY which is in an existing 1394 bus. If the new PHY responds immediately upon seeing bus reset on the new connection, then multiple bus resets due to connector scraping are likely. This could be overcome by insisting that new PHYs in existing 1394 buses will wait for some minimum timeout period for new connections, even if bus reset is sensed on the new connection earlier.

The opposite case, where a single device is plugged into an old PHY on an existing 1394 bus, is tough to solve. The old PHY can generate a reset storm regardless; the single device should go to Bus Reset Start as soon as bus reset is detected, to insure that it doesn't add any delay to bus initialization.

Cases where two existing 1394 buses are joined are straightforward. If both PHYs of the new connection have debounce circuits, then bus initialization will proceed smoothly. If one is one old PHY, and one new, then the new PHY will stall the old PHY for the minimum timeout period. If the old PHY is in the "important" bus segment, then data traffic will be stopped for the minimum timeout period. Of course, the old PHY will generate a reset storm regardless; nothing the new PHY does can prevent some period of traffic interrup-

tion. The possible prolongation of traffic interruption when old PHY silicon is present is a reasonable price to pay for clean operation with new PHY silicon.

TABLE 1. Timeout Values for Connection Debouncing

PHY Status	Time to Initiate Bus Reset After New Connect Detection	Time to Respond to Bus Reset Detected on New Connect Port
single device phy	335–342 milliseconds	140–280 ns
networked device phy	335–342 milliseconds	79–86 milliseconds

Note: 341.33 milliseconds = (1/24.576MHz) • 2^{23} ; 85.33 milliseconds = (1/24.576MHz) • 2^{21}

The asymmetry fixes force some minor changes to the All:R0a path:

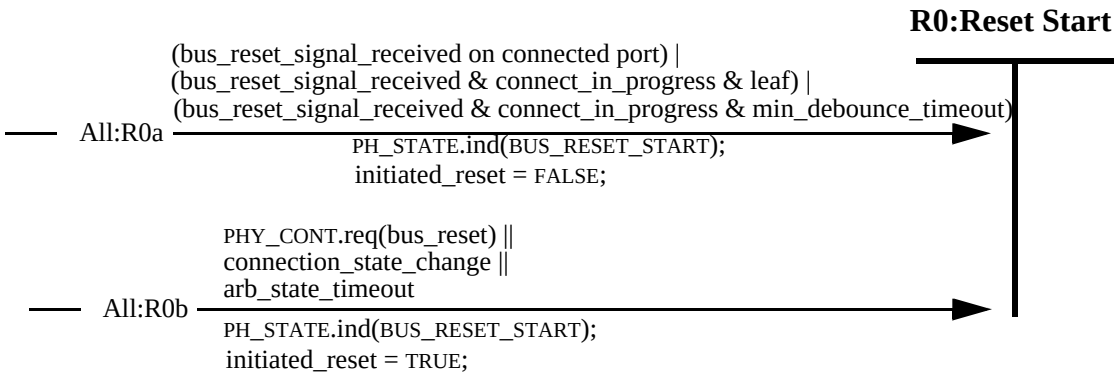


FIGURE 5. Connection Debounce Changes to R0:Reset Start Entry Paths

2.4 Associated Clarifications

A side issue is that of the “sticky bit” used to reset the gap count to its maximum value after two bus resets. This bit, called gap_count_reset_disable in Table 4-44 of the standard, is set any time the gap count register is written, and cleared by a bus reset If the bit is found to be already cleared at the start of a bus reset, then the gap count is reset to its maximum value.

Part of the rationale for this scheme was that any new connect would generate multiple bus resets, thus insuring that gap counts would fall back to the default. However software diagnostics would be able to initiate a bus reset—thus collecting a fresh batch of self-id packets—without affecting the gap count. Contact debouncing does away with multiple resets for new connects; thus a new connect might create a bus with the gap count set to too low a value, or a bus with nodes set to different gap counts. The proposed software workaround is to mandate that any time the gap count is written, software should initiate a follow-up bus reset. This allows collection of self-id packets (and therefore verification that all nodes have correctly updated their gap counts). All nodes’ gap_count_reset_disable flags are left set FALSE, and the next bus reset (perhaps from a new connect) will set gap counts to the maximum value.

Note that gap_count may be written by either of two methods. The link may directly write these parameters to its local PHY with a PHY control request (see Standard Section 4.1.1.1), Alternately a link may write these parameters with a link remote configuration request (see Standard Section 6.1.1.4). In some early PHY implementations, the link remote configuration request would not write to the local phy. There is no reason for this however; future PHY implementations should respond to PHY configuration packets, whether received or transmitted.

There is also a force_root bit. Although it is written by the same method as the gap_count, it should behave differently with bus resets. In general, it should stay set — so that the same node will be root after bus reset — but there are two exceptions.

1) If the force_root bit is set in an isolated node, it should be cleared automatically by the next bus reset. The standard specifies that isolated nodes shall not set their force_root flag (Section 8.4.2.6). This could be enforced in hardware, but then self-test diagnostics would be unable to check this bit in a normal fashion when the device is isolated. Mandating its automatic clearing upon bus reset accomplishes the same thing, and allows diagnostics programmers some latitude.
2) If arb_timer ≥ FORCE_ROOT_TIMEOUT in state T0: Tree-ID Start, then force_root should be cleared. If this timeout goes off, it is very likely that multiple nodes have their force_root bits set. If by chance the right node ended up root anyway, then software would not sense the problem. And there is a problem — bus initialization would have required an additional 83.3–167 µseconds, which could disrupt isochronous data.

Summary of Changes:

1) Gap_count should still be sticky; it should persist through one bus reset, but not two.
2) Section 8.4.6.2 permits a bus manager to initiate a bus reset as a means of verifying successful write to all nodes gap count registers. Change the permitted action to a mandatory action: After the bus manager has broadcast a PHY configuration packet which writes the gap_count, it shall initiate a bus reset. This allows the bus manager to confirm the write, and conditions the PHYs so that a later bus reset will cause the gap_count to revert to its default value.
3) The force_root bit should clear on bus reset if it was set while the node was connected to no other node. It should also clear if a force_root_timeout occurs in state T0: Tree-ID Start. Otherwise it should be unaffected by bus resets and bus initialization.
4) All future PHY implementations should allow local writes of force_root and gap_count by link remote configuration requests. This would help insure consistent bus-wide setting of gap_count and force_root fields. (See Section 5.2.2.)

3.0 Short Arbitrated Bus Reset

Bus initialization is composed of three phases: Bus Reset, Tree ID, and Self ID. The Self ID phase requires approximately one µsecond per node, roughly 70 µseconds worst-case (63 nodes). Tree ID is quite fast, probably under 10 µseconds. The longest duration phase

is Bus Reset—when the bus reset signal AB=11 propagates over the bus—which lasts ~167 µseconds. The total duration is longer than the cycle time for isochronous data, and forces the implementer to add buffer depth if uninterrupted isochronous data flow is desired. If enough time could be trimmed from bus initialization, the buffer size might be reduced.

The sole reason for the long duration of bus reset is that a transmitting node will not detect a colliding bus reset signal. The bus reset state must be long enough to out-wait the longest possible packet transmission. The long duration of the bus reset signal tends to guarantee a successful bus reset, regardless of what bus activity was in progress.

Suppose a bus is operating normally, and one of the nodes arbitrates for and wins the bus. If that node then initiates bus reset, every other node on the bus should cleanly receive the bus reset signal with little delay, since winning bus arbitration guarantees that no other node can be transmitting. In this scenario, bus reset can be of much shorter duration; 1.3 µseconds is enough, provided that PHY-to-PHY cable delays never exceed ~ 500 ns, which allows about 100 meters of cable or fiber. The worst case bus initialization time drops from ~250 µseconds to ~80 µseconds; a more common bus of 16 devices would be up and running in ~20 µseconds.

Such operation is the basis for *short arbitrated bus reset.* The intent is to provide an accelerated reset mechanism for simple connects and disconnects. Full implementation requires consideration of the implications for the bus reset states, time-outs, transitions to bus reset, a new reset_bus_request—a wealth of details. The implementation of connection debouncing is assumed as well.

3.1 Simple Bus Reset & 1394-1995

The bus reset states—Reset Start and Reset Wait—are straightforward, and well described by the Standard in Section 4.4.2.1. More is left to the imagination when it comes to the transitions into bus reset; the state machine diagrams for tree-id, self-id, and cable arbitration (Figures 4-23, 4-24, and 4-25 in the 1394 Standard) all have this caveat —*NOTE: This state machine does not include the reset conditions, since that would result in an overly-complex figure. See the All:R0a and All:R0b transition statements on page 102.* The C code for the various state actions also makes no mention of when the All:R0 transitions are permitted. There is some variation in implementations, due to differing interpretations, particularly with the All:R0a transition. Some clarification and tightening is in order, i.e.:

The All:R0a transition should be taken whenever:

1) *bus reset is received on a port whose local drivers are tri-stated, and which is receiving arbitration signals (but not receiving data-strobe signals),*
2) *bus reset is received on a port which is sending an arbitration signal which does not interfere with a bus reset signal (i.e., the port is not sending a logic 0 arbitration signal on either TpA or TpB).*

3.2 Bus Reset State Changes for Short Arbitrated Reset

The states R0: Reset Start and R1:Reset Wait are affected; both must handle short time-outs (~ 1 µsec), as well as long time-outs (~167 µsec). Long time-outs are still necessary as a fail-safe mechanism. Which timeout is used can be controlled by a new variable *go_short.* If a short reset fails by arbitration timeout in state R1: Reset Wait, then go_short should clear, and R0:Reset Start should be re-entered, this time to do a long bus reset. The Bus Reset state diagram is revised to include the short reset time-outs; the R0: Reset Start and R1: Reset Wait states are otherwise unchanged; no changes need be made to the C code for these states.

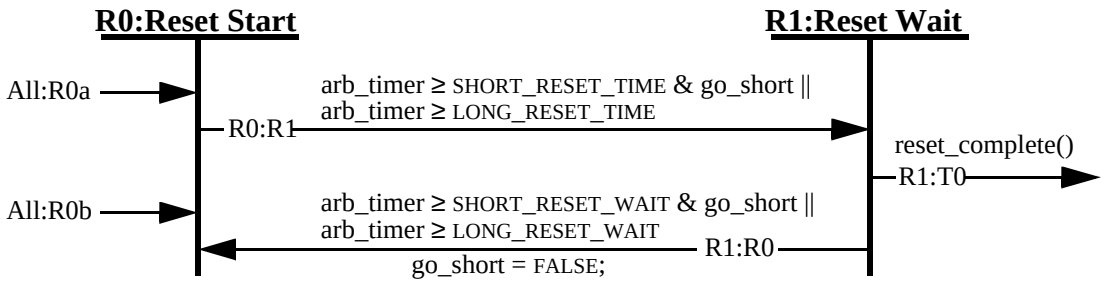


FIGURE 6. MODIFICATIONS TO R0:R1 AND R1:R0 TRANSITIONS

TABLE 2. Reset Timer Values

Timer Constants	Timeout Value	Comment
SHORT_RESET_TIME (new)	1.3 - 1.4 µs	~ 128/BASE_RATE
LONG_RESET_TIME	166.6 - 166.7 µs	~ 16384/BASE_RATE
SHORT_RESET_WAIT (new)	1.4 - 1.5 µs	~ 144/BASE_RATE
LONG_RESET_WAIT	166.8 - 166.9 µs	~ 16400/BASE_RATE

3.3 TRANSITIONS TO R0: RESET START

The bus can be in any of four phases: Reset, Tree ID, Self ID, or Arbitration. Reset was covered in the preceding section. Does short arbitrated bus reset imply any changes for Tree ID and Self ID phases? Specifically, is it worth trying a short reset in those states if a bus reset is triggered?

Tree ID is a signalling intensive phase. The TX_PARENT_NOTIFY signal (0Z) could interfere with reception of a bus reset signal, depending on details of implementation; the arbitration timeout values for Tree ID states is longer than the short reset time-outs. An attempted short reset while the bus is in Tree ID may not succeed in resetting the entire bus cleanly.

Self ID is another signalling intensive phase. There is no appreciable idle time on the bus. The TX_BUS_GRANT signal (Z0) could interfere with reception of bus reset, as could the transmission and reception of self-id packets. Again, a short reset is unlikely to succeed.

The whole rationale for short reset is that arbitration is carried out first, so that the bus is in a known receptive state, thus guaranteeing that a short reset will succeed. This also argues against trying a short reset while in Tree ID or Self ID; arbitration is not possible until the Arbitration phase is reached.

There are four general circumstances which can lead to bus resets: direct software command (setting the INITIATE_BUS_RESET bit), state time-outs, connection status changes, and reception of the bus_reset signal. Furthermore, the actions to be taken can depend on whether the PHY is part of a bus, or is single.

3.3.1 Direct Software Command

Software can set the INITIATE_BUS_RESET (IBR) bit, which always opts for an immediate long reset. There should also be a new register bit, specific for initiating short resets—INITIATE_SHORT_BUS_RESET (ISBR). For simplicity, the new bit ISBR should only initiate short resets. If ISBR is set while the PHY is in some phase other than Arbitration, then the PHY should continue with normal bus initialization until it reaches the Arbitration phase. At that time, the PHY can arbitrate for the bus and initiate short bus reset. Both IBR and ISBR bits should always clear upon entry to the R0: Bus Reset state.

What action should an unconnected PHY take if software sets ISBR? This is an arcane point, probably of more interest to manufacturing test engineers than users. But assuming the PHY has reached the Arbitration phase of bus operation, it should initiate a short bus reset, just like a connected PHY. The obvious difference is that, being unconnected, all its 1394 ports will be tri-stated, so there is no bus activity to observe. But the PHY should send the appropriate bus reset, node ID, and gap status messages to the link, as it goes through the normal bus initialization steps.

3.3.2 State Time-outs

State time-outs indicate that the whole bus arbitration mechanism has “broken.” Typically these occur as a PHY is powering up, when operation is erratic (future implementations would do well to pay special attention to the turn-on timing of the TpBias outputs as a PHY powers up). But whatever the cause, the general indication is that the bus is broken, and a long reset is called for.

3.3.3 Connection Status Changes

Short reset was invented specifically to handle new connects and disconnects. In the case of connection status changes occurring while in Arbitration phase, the short reset process should be initiated. If the PHY is not in Arbitration phase, but still in Bus Reset, Tree ID or Self ID, then there is room for discussion. Clearly some situations dictate immediate long bus reset—if disconnect is detected on a parent port, for example. In other situations, an

argument could be made for allowing bus initialization to proceed, and then initiating a short reset when Arbitration phase is reached. But note that this will cause two bus initializations, one after the other, which may be as awkward to handle as simply forcing a long reset when the connection status change is first detected.

Back to basics—short resets have been invented so that a user can plug in or remove a device from a bus running isochronous data, without unduly disrupting the data stream. If connection status changes are frequently occurring during bus initialization—more often than once a year—then something is going on beyond the simple model of a user occasionally plugging in or removing equipment. With that understanding, it is acceptable that connection changes sensed during Tree ID or Self ID result in the immediate triggering of a long reset. In a bus with infrequent bus resets, hitting the 80 (or even 250) µsecond window of bus initialization is unlikely.

Assume that the node(s) with connection status changes are in the Arbitration phase of bus operation. Make a distinction between *bus nodes* and *single nodes*. Bus nodes are part of an existing 1394 bus of two or more nodes, which are up and running in arbitration phase before the connection event. Single nodes are nodes which are not connected to any other prior to the connection event. There are four cases of interest: disconnection of a parent node, disconnection of a child node, connection of a new single node, and connection of a new bus node (i.e. connection of two operating 1394 buses together).

Parent node disconnection is the simplest. Bus arbitration is impossible since the root is no longer part of the bus fragment. Initiate long bus reset.

Child node disconnection is straightforward. As described above, the network node sensing the disconnect arbitrates for the bus, and initiates short reset. Whether one node or a subnet of several nodes was removed from the bus doesn't affect the process.

Single node connection is more interesting. Assume that a new connection is made between a single node and a bus node. To insure a successful short arbitrated reset, the two nodes have to cooperate; the actions taken by the single node are different from those of the bus node.

The bus node arbitrates for its bus. If it wins arbitration, it initiates a short bus reset.

The single node has to keep quiet. Ideally it should sit in A0: Idle for a lengthy timeout period, to allow sufficient time for the bus node to win arbitration and initiate bus reset. When bus reset is sensed on its newly connected port (from the bus node), it should transition to bus reset state, and do a short reset. The timeout period should be as long as the connection timeout period (342 milliseconds) plus a reasonable time to insure successful bus arbitration, plus some reasonable estimate to cover connection asymmetry. There is no great advantage in shaving milliseconds off—why go to all this trouble to invent a new reset mechanism, and then have it occasionally do a long reset anyway? So let the total timeout period be at least 341 (the connection timeout period) + 341 (call this the Await_Bus_Reset period) = 682 milliseconds total.

If the single node times out before bus reset is sensed, then it initiates bus reset itself. Since it has no way of knowing what state the other newly connected node is in, it initiates a long bus reset. Note that this is the pathway taken by a pair of single nodes. Each waits for the other to initiate short bus reset until finally one or the other times out and initiates long reset. Since both were single nodes, there is no bus traffic to interrupt, so there is no reason not to do a long bus reset.

Connecting two buses together is not likely to produce a short reset. The two nodes with new connections arbitrate for their respective buses. Typically one wins before the other, and initiates a short reset. If the slower node senses the short reset (it may not, depending on its activity at the time), it initiates a long reset. In the event that the slower node doesn't notice the short reset, then the early node will time out in its bus initialization attempt, and will initiate a long reset. While it is possible that the two nodes could initiate short reset at the same time, and not fall into a long reset, it is extremely unlikely.

3.3.4 Reception of Bus Reset Signal

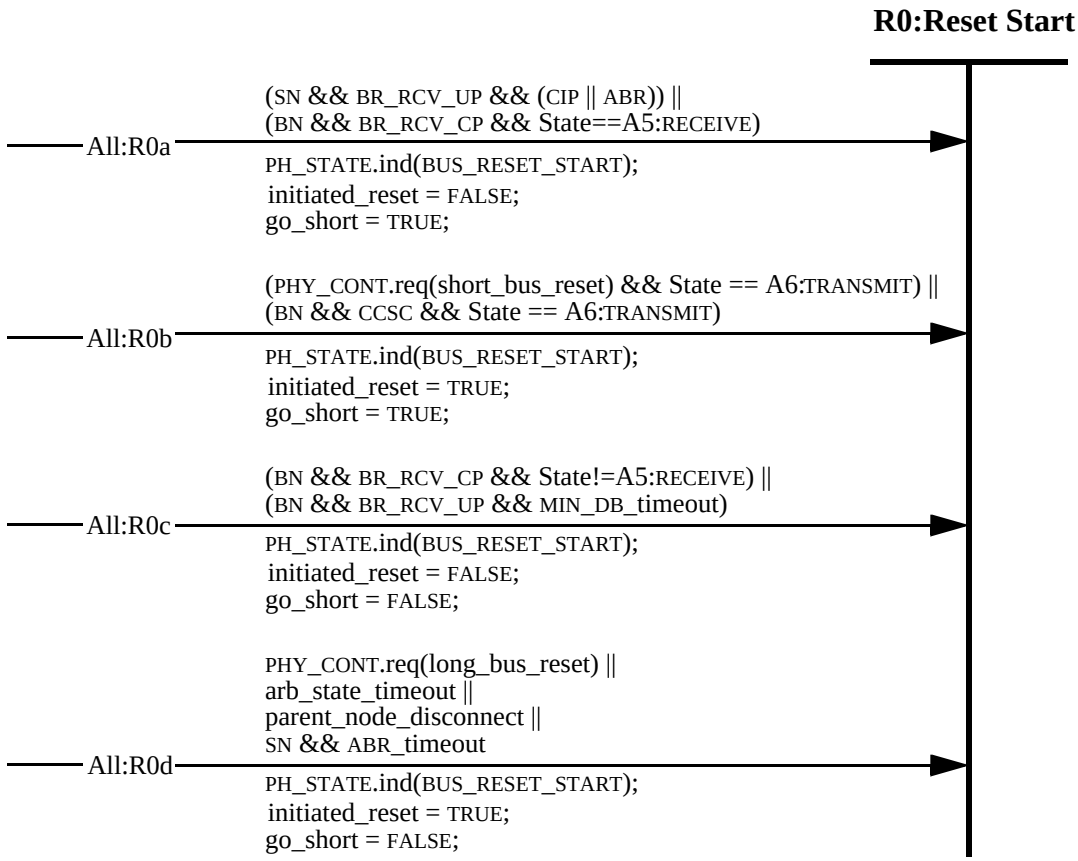
For a bus node with no new connections, the arrival of a short reset will come at a well defined time. The initiating node has arbitrated for the bus and won, and has sent out a normal duration 100 Mb/s data prefix; all other nodes should accordingly be in state A5: Receive. A bus reset signal received in any state other than A5:Receive may be assumed to be a long reset signal.

For a bus node with a new connection, arrival of a bus reset signal on the new connect port indicates: connection of an older PHY (which doesn't support short arbitrated bus reset), or timeout by a single node (682 milliseconds has elapsed since connection event), or connection of another bus (which has gone through arbitration, and is now trying to initiate a short bus reset). In all three cases, a long bus reset is required and/or inevitable.

A single node will be in a state somewhat dependent on local conditions and implementation. A fully standard compliant PHY will progress through bus initialization to state A0: Idle even when unconnected. But software could initiate bus reset, or even packet transmission on an unconnected node, so it is possible for a single node to be in almost any state. From the standpoint of getting short resets to reliably work, software diagnostics should be cautious about actions on single nodes. There are no status bits which indicate that a new connect timeout is in progress, so there is no way for software to determine when it is safe to exercise an "unconnected" phy. Implementations may differ regarding their susceptibility to interference from such activity.

Regardless of the single node's state, if a newly connected port detects bus reset, short reset should be attempted. The port will be in the connection timeout or the subsequent await_bus_reset timeout period (682 ms total) when the bus reset signal is detected.

The simple pathways to bus reset have gotten considerably more complicated. The next figure shows the various pathways, sorted by short/long and initiated_true/false. The convenience of not showing bus reset conditions from other states is rapidly losing its "convenience.



ABR = Await_Bus_Reset (timeout period of 341 ms)
BN = Bus Node (has recognized connection(s); may have new connections not yet recognized also)
BR_RCV_CP = Bus Reset Signal Received on Connected Port
BR_RCV_UP = Bus Reset Signal Received on Unrecognized Port
CIP = Connect_In_Progress (timeout period of 341 ms)
CCSC = Child Connect Status Change Flag
MIN_DB = Minimum Debounce (timeout period of 85 ms)
SN = Single Node (any new connections not yet recognized, port drivers tri-stated)

FIGURE 7. Entry Paths to R0:Reset Start

3.4 Bus Arbitration for Short Reset

Finally, the short arbitrated bus reset mechanism introduces a new flag—ISBR_request, (Initiate Short Bus Reset—which behaves like a bus request. Its behavior should be the same, whether set by direct software write, or by a child connection status change.

The fundamental purpose of this mechanism is to minimize the interference with isochronous data streams. An ISBR_request should not generate a bus request during isochronous data mode. This condition can be met simply—the new ISBR_request should have the same timing requirements as a FAIR_request. In the arbitration state diagrams, wherever a

FAIR_request is allowed, an ISBR_request should also be allowed. In the event that both a FAIR_request and a ISBR_request are active at the same time, the ISBR_request should be given preference. Unlike a FAIR_request, an ISBR_request should persist until bus arbitration is won (FAIR_requests are cleared by arbitration loss as well).

The ISBR_request, unlike a FAIR_request, should ignore the arb_enable bit; fair access is not an issue for initiating bus reset.

In the event that a CYCLE_MASTER_request and a ISBR_request are active at the same time, the CYCLE_MASTER_request should take precedence if the node is the root; otherwise the ISBR_request should take precedence. This allows a cycle start packet and isochronous cycle to occur before the bus reset. (Digression: There are numerous bus arbitration changes and enhancements that are widely discussed but undocumented. One such proposal is that CYCLE_MASTER_request may be used for purposes other than sending cycle start packets—thus the distinction here between root and non-root use of CYCLE_MASTER_request. It is possible that the root might use CYCLE_MASTER_request for a non-cycle start packet, in which case ISBR_request would be the preferred request, but there is no way to enlighten a PHY to these nuances without inventing a new request type, separate and distinct from CYCLE_MASTER_request.)

ISBR_request is cleared only upon entering R0:Reset Start. In the event that the transition to Reset Start fails to occur—arbitration is never won—there should be a timeout mechanism, which forces a long reset. The same long timeout period of 341 milliseconds would be consistent with other time-outs chosen for the connection status change state machine.

This section has generally described the implementation changes necessary to the arbitration state machine. Numerous other changes need to be rolled in as well—most for either arbitration enhancements or “Open HCI” considerations. Rather than update the state machine twice, the explicit revisions will be included in the later section on arbitration enhancements.

4.0 PHY Pinging

PHY pinging is one possible answer to the problem of optimizing the gap count setting. Numerous methods have been suggested:

1) Don’t set it; leave it at the default setting. This obviously wastes bandwidth, but would be acceptable in many applications.
2) Set it to the correct value for a “maximum” sized 1394 bus—a 16 node daisy-chain (there may be up to 63 nodes, but the longest chain of nodes on a bus must be limited to 16). This saves about 50% of the bandwidth lost by the first method, at the risk of not working properly if a user has exceeded the maximum, or if cables longer than the “suggested maximum” of 4.5 meters are used.
3) Use the information from the self-ID packets to reconstruct the topology, determine the longest daisy-chain, and set the gap count to a computed minimum value. This only works if cables truly have a maximum length. Maximum PHY delay is 144 ns; cable

delay for a 4.5 meter cable is ~23 ns, so the total delay for one “hop” is about 167 ns. However, there is no hard limit on cable length. One hundred meter cables have been discussed; the hop delay for a node with a hundred meter cable rises to roughly 650 ns. Clearly this method fails unless there is some way for a bus to be aware of long cables.

PHY pinging is a suggested mechanism for directly measuring bus delays, thereby indirectly detecting long cables. The mechanism is straightforward:

1) A diagnostic node transmits a special ping packet. The ping packet contains a target address. The node starts a timer when transmission is complete.
2) The target node receives the ping, and sends back a ping response packet.
3) The diagnostic node detects the ping response, stops the timer at start of reception, and calculates from the timer the propagation delay to the target node.

The diagnostic node can take a set of these ping measurements, and then determine the worst case end-to-end delay through the network, which enables it to set the gap count to a minimum value. There are some computational details; for instance PHY delay may cover a range, depending on local clock phase relative to the received data, so some margin should be added to the gap count for each hop. The fastest to slowest PHY propagation time for PHYs is not specified, but probably does not exceed 20 ns for any phy.

4.1 Proposal for PHY Pinging

4.1.1 Ping Packet Format

The ping packet can be a special PHY configuration packet:

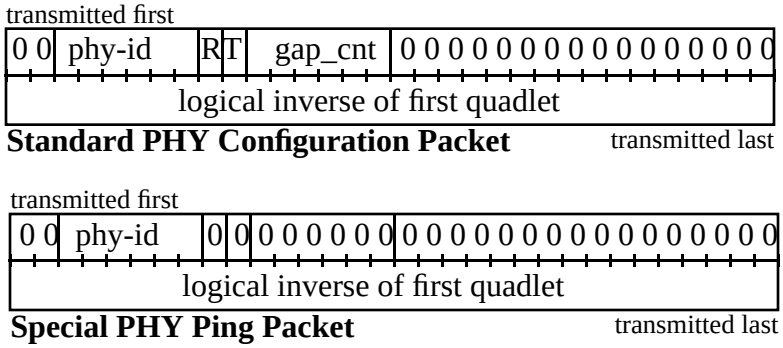


FIGURE 8. PHY Ping Packet Format

If a PHY configuration packet has both R and T bits set to 0, then receiving PHYs take no action. This permits redefinition of the special R,T=0 configuration packets as a new class of PHY packets, with the simplest one—all bits 0 except for the target phy-id—used for the ping command.

4.1.2 Ping Response Packet

For the purposes of setting the gap count, the contents of the ping response packet are unimportant; a simple ack packet would be enough. Additional utility is gained by letting a PHY return its self-id packet as a ping response. This allows diagnostic access to the node configuration information in self-id packets without initiating a bus reset. This adds some extra transitions to the PHY arbitration state machine, but doesn't actually add much silicon hardware. The only modification for the normal self-id transmit process is that the usual parent port notification may be skipped; the node may return immediately to state A0:Idle.

4.1.3 Ping Timer

Requirements for the timer are straightforward. It should start at some fixed time relative to the transmission of the ping; it should stop at some fixed time relative to the reception of the response.

The ping timer could be located in either PHY or link silicon. Since the ping response mechanism requires some modification to PHY silicon, it makes sense to put the timer in the PHY as well.

Some variation in implementation is probably tolerable. A suggested implementation is:

1) Clear and start the timer when the last bit of the ping packet is clocked out.
2) Stop the timer 60–80 ns after the first bit of the response packet is clocked in.
3) Let the timer keep count of 24.576 MHz clock periods—a count of one equals 40.7 ns.
4) An 8-bit timer will count up to 255•40.7 ns = 10.37 µs, which is almost exactly the maximum subaction gap (with gap count = 63). If the delay is longer, reliable bus operation is impossible (without modified silicon). The timer should be sticky; if it reaches its maximum count of 511 (hFF), it should stick and not roll over. A measured ping time of 511 (hFF) indicates that the bus has too much delay for reliable operation.
5) The timer count should be accessible as a read-only hardware register in the phy.

4.2 State Machine Modifications

Required state machine modifications are relatively minor. There is a new transmit action—start the ping timer if the transmitted packet is a ping command. There is an added receive state action—detection of the ping packet with target-id = local id. This must set a new ping_response flag, which causes transition back to Self-ID Transmit upon completion of all receive actions. Finally, the Self-ID Transmit actions must be modified if ping_response is set.

4.2.1 New Transmit State Actions

The transmit logic needs to include the simple parsing logic which is already used by the receive logic to detect PHY packets. This doesn't add many gates to the physical design;

the same parsing logic block can operate on either the transmit stream or the receive stream.

```
void transmit_actions() {
int i;
int bit_count = 0;
union {
   dataBit bits[64];                          //re-use this logic (from receive actions)
   struct {
      unsigned pkt_type:2;
      unsigned addr:6;
      unsigned R:1;
      unsigned T:1;
      unsigned gap_count:6;
      unsigned byte2:8;
      unsigned byte3:8;
      dataBit check_bits[32]:32;
         } phy_info;
   } phy_pkt;
boolean test_end = false;
boolean ping_timer_enable = false;            //hold count
phyData data_to_transmit;
imm_req = false;                              //clear requests
if (fair_req) {
   arb_enable = false;
   fair_req = false;
   }
isoch_req = false;
receive_port = NPORT;                         //no port has this number => PHY is transmitting
start_tx_packet(req_speed);                   //send data prefix & speed signal
while (~test_end) {
   PH_CLOCK.ind;                              //tell link to send data
   while (~PH_DATA.req & (data_to_transmit)); //wait for data from link
   switch(data_to_transmit) {
      case DATA_END:
         stop_tx_packet(DATA_END);
         test_end = true;                     //end of packet indicator
         if (bit_count == 64) {               //we have transmitted a PHY packet
            boolean good_phy_packet = true;   //check for good format
            for (i=0; i<32; i++)
               good_phy_packet =
                  (phy_pkt.bits[i] == ~phy_pkt.phy_info.check_bits[i+32) && good_phy_packet;
            if (good_phy_packet && (phy_pkt.phy_info.pkt_type == 0b00)) {
                                              //no action necessary for link-on or self-ID pkts
               if (phy_pkt.phy_info.R)        //force root, set if address match, else clear
                                              //this should work for transmit and receive state
                  force_root = (phy_pkt.phy_info.address == physical_ID);
               if (phy_pkt.phy_info.T) {
                                              //PHY gap_count set, and set reset_disable
                  gap_count = phy_pkt.phy_info.gap_count;
                  gap_count_reset_disable = true;
                  }
               if ((phy_pkt.phy_info.R == false) && (phy_pkt.phy_info.T == false)) {
                                              //new class of PHY packets transmitted
                  if (phy_pkt.phy_info.R == 0b00_0000) {
```

```
                                                    //ping packet detected
                    ping_timer_enable = true;       //clear and enable timer
                    if (phy_pkt.phy_info.address == physical_ID) ping_response = true;
                                                    //node can ping itself
                    }
                  }
                }
              }
            break;
          case DATA_PREFIX:
            stop_tx_packet(DATA_PREFIX);
            wait_time (min_packet_separation);      //hold bus for concatenated packet
            break;
          case 0, 1:                                //send data
            tx_bit(data_to_transmit);
            if (bit_count < 64)                     //accumulate first 64 bits
            phy_pkt.bits[bit_count++] = data_to_transmit;
            break;
          }
        }
      end_of_transmission = true;
      }
```

4.2.2 New Receive State Actions

First, packet reception must stop the ping timer as soon as data starts arriving:

```
...
boolean received_data;
fair_req = false;
cycle_master_req = false;
PH_DATA.ind(DATA_PREFIX);
rx_speed = start_rx_packet();                       //start up receiver and repeater
PH_DATA.ind(DATA_START(rx_speed));                  //send speed indication
ping_timer_enable = false;                          //halt ping timer
while (~test_end) {
...
```

Second, the logic must recognize the new ping command packet.

```
...
switch (phy_pkt.phy_info.pkt_type) {
  //process packets differently based on PHY packet type
  case 0b00:                                        //PHY config packet
    if (phy_pkt.phy_info.R)                         //PHY force-root, set if address match, else clear
      force_root = (phy_pkt.phy_info.address == physical_ID);
    if (phy_pkt.phy_info.T) {                       //PHY gap_count, set always and set reset_disable
      gap_count = phy_pkt.phy_info.gap_count;
      gap_count_reset_disable = TRUE;
      }
    if ((phy_pkt.phy_info.R == FALSE) && (phy_pkt.phy_info.T == FALSE) &&
      (phy_pkt.phy_info.gap_count == 0b00_0000) &&
      (phy_pkt.phy_info.address == physical_ID))
```

```
                                        //require address match & gap count field all 0's
      ping_response == true;            //new ping_response flag
    break
  case 0b01:                            //link-on packet
  ...
```

4.2.3 New Self-ID Transmit State Actions

Actually, the change is to bypass some actions normally taken in this state, if the self-ID packet is being sent as a ping response. Specifically, the final tx_ident_done handshake and speed signal transmission handshake can be omitted.

```
...
if (ping_response == FALSE) {           //bypass all this if ping_response
  for (port_number == 0; port_number < NPORT; i++) {
    if (port_number == parent_port) {
      portT(i, TX_IDENT_DONE);          //notify parent that self-ID is complete
      portTspeed (port_number, PHY_SPEED); //send speed signal (if any)
      wait_time (SPEED_SIGNAL_LENGTH);
      portTspeed(port_number, S100);    //stop sending speed signal
      ph_event.ind(SELF_ID_COMPLETE, physical_id, root);
      }
    else
      portT(port_number, IDLE);         //turn off transmitters to others
    }
  }
self_ID_complete = TRUE;                //signal completion
...
```

4.2.4 New State Machine Transitions

The new transitions are limited to the jump back to self-id transmit upon receipt of a ping command, and the jump back to A0:Idle after transmission of the ping response.

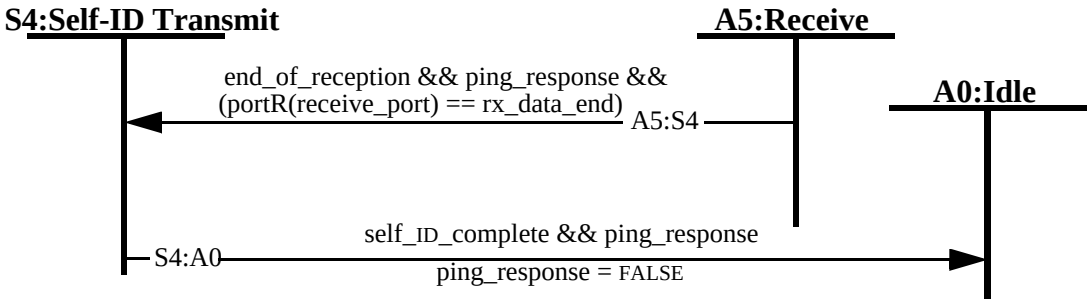


FIGURE 9. New Ping Response State Machine Transitions

5.0 Remote PHY Pinging

There are two limitations to the PHY pinging mechanism: older PHY silicon may not send a ping response, and older silicon may not implement the ping timer. The inability to send a ping response is unfortunate, but as long as there is at least one ping-able PHY on each standard-cable bus segment (a bus segment with all cables ≤ 4.5m), a combination of pinging and calculation will yield an optimal gap count.

The lack of a ping timer on a local node would make it impossible for that node to gain any timing information from the ping response (it might still be useful as a way to check the self-ID packets unobtrusively).

It is not challenging to define a mechanism by which a special node (with new PHY silicon) could be commanded by a distant node to send a ping packet, time the response, and send the timer data out. Such a mechanism would allow a software upgrade of existing 1394 devices to utilize PHY ping timing, even if the device itself has an older phy. Inclusion of one new phy in the bus (perhaps a bare phy, acting only as a repeater) would thus enable phy ping timing.

The mechanism may have a somewhat short lifespan, if future PHY designs all incorporate PHY ping timing hardware. It is nonetheless worth discussion if only for the immediate advantage of remote PHY pinging.

5.1 Proposal for Remote Pinging

The mechanism of simple PHY pinging consists of a pair of special PHY configuration packets, with the second transmitted automatically in response to the first. Remote PHY pinging is similar, except that the process grows to a cascade of four packets. The middle two packets in the sequence are simple PHY pinging.

All four packets are variants of PHY configuration packets; thus all are 64 bits transmitted at the S100 rate. Packet duration including data prefix and data end is roughly 1 µsecond. Delay between packets is primarily packet propagation time across the bus. With the most extensive bus supportable by 1394 silicon, propagation time cannot exceed 10 µseconds. Thus the duration of the entire four-packet process, from transmission of the first packet to reception of the last, is ~44 µseconds. That represents an extreme case, with unusually large delays. A more typical case would probably average 2–3 µseconds for propagation times, for a total of 8–12 µseconds used for the entire process. Since asynchronous packet duration is on the order of 44 µseconds (512 bytes payload + 24 bytes header & CRCs = 536bytes • 8 bits/byte • 10.173 ns/bit = 43.6 µseconds), the bus is inherently able to support transactions of even the worst case duration for remote pinging.

1) Bus Master node sends remote_ping_command(A, B) to node A.

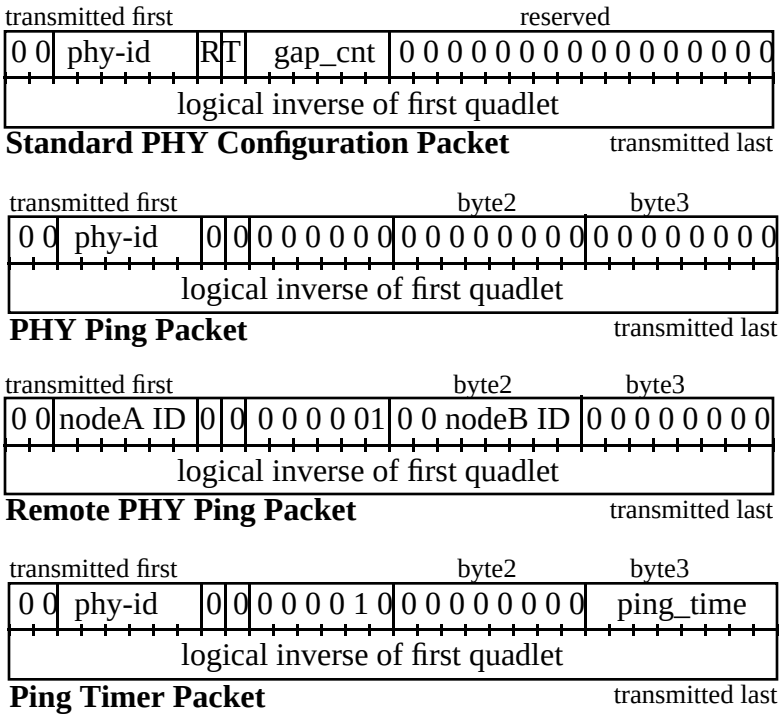
2) Node A immediately sends ping command to node B.

3) Node B immediately sends its self-ID packet.

4) Node A immediately sends a ping_time packet.

5.1.1 New Packet Definition

Two new packet formats must be defined: the remote_ping_command, and the ping_time packet. The remote_ping_command instructs node A to send out a ping targeted to node B. The ping_time packet carries the contents of a node's ping_timer register. All can be variations of PHY configuration packets.



The field formerly used solely as gap_cnt (which had meaning only if the T bit was set) now becomes a more general purpose key which indicates the type of PHY config packet, for all config packets with R, T = 0. Three cases have been defined so far: PHY Ping (key field = 00h), Remote Ping (key field = 01h), and PHY Register Data (key field = 03h).

5.2 State Machine Modifications

Transmission of the remote PHY ping packet is normal and straightforward. The transmitting node sends the command, just as it would send any PHY configuration packet.

The node whose ID matches the target ID of the remote ping command has some new receive actions: it must save the "byte2" field (formerly part of a reserved field) from the remote ping command, and it must set a ping_command bit. Note that gap_count is a misnomer for ping command, but was the original name for the field in PHY configuration packets in 1394-1995. In a ping command packet, the "gap_count" field must be 0b00_0001; it acts as a command type field. The ping_command bit is new. Its actions are similar to those of an immediate bus request; the PHY leaves A5:Receive state and quickly goes through A0:Idle to A6:Transmit.

In transmit state, the PHY sends a PHY ping packet. This is a new action; prior PHYs only transmitted packets from the link while in transmit state. However, the action is similar to transmission of a self-ID packet, so no real invention is necessary.

As explained in a preceding section, the node which receives the ping packet retransmits its own self-id packet. Upon detecting the self-id packet, the pinging node (which still has the ping_command bit set) halts its ping timer, and sends out the final time in a new ping timer packet.

5.2.1 New Receive State Actions

All the new actions are in the section dealing with PHY config packets:

```
...
switch (phy_pkt.phy_info.pkt_type) {
  //process packets differently based on PHY packet type
  case 0b00:                                    //PHY config packet
    if (phy_pkt.phy_info.R)                     //PHY force-root, set if address match, else clear
      force_root = (phy_pkt.phy_info.address == physical_ID);
    if (phy_pkt.phy_info.T) {                   //PHY gap_count, set always and set reset_disable
      gap_count = phy_pkt.phy_info.gap_count;
      gap_count_reset_disable = TRUE;
      }
    if ((phy_pkt.phy_info.R == FALSE) && (phy_pkt.phy_info.T == FALSE) &&
      (phy_pkt.phy_info.gap_count == 0b00_0000) &&
      (phy_pkt.phy_info.address == physical_ID))
                                                //require address match & gap count field all 0's
      ping_response = true;                     //ping_response flag
    break
    if ((phy_pkt.phy_info.R == FALSE) && (phy_pkt.phy_info.T == FALSE) &&
      (phy_pkt.phy_info.gap_count == 0b00_0001) &&
      (phy_pkt.phy_info.address == physical_ID))
                                                //require address match & gap count field =01
      ping_command = true;                      //new ping_command flag
  case 0b01:                                    //link-on packet
    if (phy_pkt.phy_info.address == physical-ID)
      ph_event.ind(link_on);
    break
  case 0b10:                                    //self-id packets
    if (ping_command == true) ping_timer_dump = true;
    break

...
```

5.2.2 New Transmit State Actions

The new actions cover the two new transmit tasks: transmitting a PHY ping packet, and transmitting a ping timer packet.

```
void transmit_actions() {
int i;
int bit_count = 0;
```

```
quadlet phy_tx_packet;
union {
   dataBit bits[64];                          //re-use this logic (from receive actions)
   struct {
      unsigned pkt_type:2;
      unsigned addr:6;
      unsigned R:1;
      unsigned T:1;
      unsigned gap_count:6;
      unsigned byte2:8;
      unsigned byte3:8;
      dataBit check_bits[32]:32;
         } phy_info;
   } phy_pkt;
boolean test_end = false;
boolean ping_timer_enable = false;            //hold count
phyData data_to_transmit;
imm_req = false;                              //clear requests
if (fair_req) {
   arb_enable = false;
   fair_req = false;
   }
isoch_req = false;
receive_port = NPORT;                         //no port has this number => PHY is transmitting

if ((ping_command == true) && (ping_timer_dump == false)) {
                                              //send PHY ping packet
   start_tx_packet(S100);                     //data prefix
   phy_tx_packet = 0x0000_0000 |              //format for PHY ping
      byte2 << 8;                             //byte 2 contains target ID of node to be pinged
   tx_quadlet(phy_tx_packet);                 //send packet
   tx_quadlet(~phy_tx_packet);                //send inverse
   stop_tx_packet(TX_DATA_END, S100);
   }

if ((ping_command == true) && (ping_timer_dump == true)) {
                                              //send ping timer packet
   start_tx_packet(S100);                     //data prefix
   phy_tx_packet = 0x000A_0000 |              //format for ping timer packet
      ping_timer;                             //puts ping timer value into low 8 bits
   tx_quadlet(phy_tx_packet);                 //send packet
   tx_quadlet(~phy_tx_packet);                //send inverse
   stop_tx_packet(TX_DATA_END, S100);
   ping_command = false;
   ping_timer_dump = false;
   }

if ((ping_command == false) && (ping_timer_dump == false)) {
   start_tx_packet(req_speed);                //send data prefix & speed signal
   while (~test_end) {
      PH_CLOCK.ind;//tell link to send data
      while (~PH_DATA.req & (data_to_transmit));  //wait for data from link
      switch(data_to_transmit) {
         case DATA_END:
```

```
            stop_tx_packet(DATA_END);
            test_end = true;                        //end of packet indicator
            if (bit_count == 64) {                  //we have transmitted a PHY packet
              boolean good_phy_packet = true;       //check for good format
              for (i=0; i<32; i++)
                good_phy_packet =
                  (phy_pkt.bits[i] == ~phy_pkt.phy_info.check_bits[i+32) && good_phy_packet;
              if (good_phy_packet && (phy_pkt.phy_info.pkt_type == 0b00)) {
                                                    //no action necessary for link-on or self-ID pkts
                if (phy_pkt.phy_info.R)             //force root, set if address match, else clear
                                                    //this should work for transmit and receive state
                  force_root = (phy_pkt.phy_info.address == physical_ID);
                if (phy_pkt.phy_info.T) {
                                                    //PHY gap_count set, and set reset_disable
                  gap_count = phy_pkt.phy_info.gap_count;
                  gap_count_reset_disable = true;
                  }
                if ((phy_pkt.phy_info.R == false) && (phy_pkt.phy_info.T == false)) {
                                                    //new class of PHY packets transmitted
                  if (phy_pkt.phy_info.R == 0b00_0000) {
                                                    //ping packet detected
                    ping_timer_enable = true;       //clear and enable timer
                    if (phy_pkt.phy_info.address == physical_ID) ping_response = true;
                    }                               //node can ping itself
                  }
                }
              }
            }
          break;
        case DATA_PREFIX:
          stop_tx_packet(DATA_PREFIX);
          wait_time (min_packet_separation);//hold bus for concatenated packet
          break;
        case 0, 1:   //send data
          tx_bit(data_to_transmit);
          if (bit_count < 64)//accumulate first 64 bits
          phy_pkt.bits[bit_count++] = data_to_transmit;
          break;
        }
      }
    end_of_transmission = true;
    }
```

5.2.3 New State Machine Transitions

The new transitions are very minor; the new ping_command and ping_timer_dump bits cause a transition from A0:Idle to A6:Transmit. There is already a transition between these states—for immediate request—so this just adds two more possible conditions which may cause this transition.

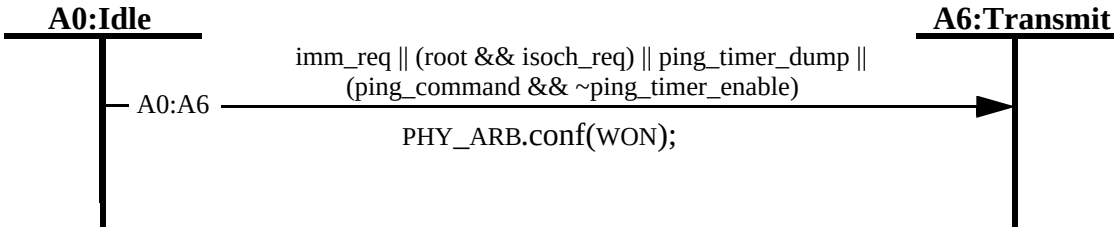


FIGURE 10. New Remote Ping State Machine Transition

6.0 Multi-Speed Packet Concatenation

Most of the work on the 1394-1995 Standard was completed before 200 Mb/second PHY silicon was available. Consequently the details on operation at higher speeds is sketchy. The standard does not explicitly forbid concatenating packets of different bit rates together, but the writers probably assumed single speed concatenation.

Multi-speed packet concatenation would allow:

1) Concatenation of a read response of arbitrary speed onto an ack.

2) Concatenation of a chain of isochronous packets, without regard to bit rate.

3) Fly-by arbitration (see later section) without regard to bit rate.

The last point is probably the most important. Fly-by arbitration is one of the bandwidth saving arbitration tricks. Without the bandwidth savings, there is little point in extending1394 to higher speeds, which has dire effects on the market potential for the technology.

6.1 1394-1995 And Multi-Speed Concatenation

There are four areas in the standard which affect concatenation: the transmit_actions for state A6:Transmit, the receive_actions for A5:Receive, the detailed operational timing parameters (data prefix, speed signal …), and finally annex J, which specifies phy-link signalling.

6.1.1 Transmit Actions

The C code for transmit_actions only performs the action *start_tx_packet(req_speed)* at the start of transmission of the first packet in a concatenated set. Thus a conforming PHY would only transmit a speed signal during the data prefix of the first packet of a concatenated chain of packets. Note that a PHY transmitting concatenated packets stays in transmit

state. It sends data prefix; it sends packet data; it sends data prefix again; it sends more packet data; etc. But it stays in transmit state the whole time, and just switches between sending data prefix and sending data. Contrast this with the C code for receive –

6.1.2 Receive Actions

The C code for receive_actions includes the action *rx_speed = start_rx_packet()*; early in the receive code. When packet reception is over, the state machine exits receive state. If a concatenated packet follows, it immediately re-enters receive state, which is designated as transition A5:A5a -

Transition A5:A5a. *If a packet ends and the received signal is rx_data_prefix (10), then there may be another packet coming, so the receive process is restarted.*

Restarting the receive process mandates that receive speed be evaluated again. Absence of a speed signal will be interpreted as a S100 packet by a 1394-1995 compliant implementation.

The C code for the transmit actions isn't really in synch with the C code for the receive actions. Clearly the receive logic expects a speed signal at the start of each packet, whether concatenated or stand-alone. But the transmit logic, as coded, will only send a speed signal at the start of a chain of concatenated packets. A 1394-1995 compliant receiver will mistakenly regard each successive packet in the chain as S100, regardless of the transmit intention.

Of course no one would implement something so obviously broken. Unfortunately, there's bound to be some variability in implementation, depending on *which* sets of logical assumptions were adopted by the various design groups.

6.1.3 Timing

The MIN_PACKET_SEPARATION time specified for concatenated packets is 340 ns. If two packets were crammed together without concatenation, but with no idle time between data_end and data_prefix, the packet separation in that case would be:

DATA_END_TIME	240	ns
+ DATA_PREFIX_TIME	40	
+ SPEED_SIGNAL_LENGTH	100	
- ARB_SPEED_SIGNAL_START	- 20	
	360	ns Total

This is almost identical to the MIN_PACKET_SEPARATION time specified, and not by accident. The MIN_PACKET_SEPARATION was intentionally made long enough to allow speed signalling between concatenated packets. No modifications or clarifications are needed for the inter-packet timing.

6.1.4 Annex J - PHY Link Signalling

Annex J defines a mechanism which allows reception of multi-speed concatenated packets. In fact, the link is generally unable to distinguish between concatenated packets and received packets which are closely packed. In all cases, the PHY asserts Ctl[0:1]=10 to indicate packet reception. In the time before data starts arriving, the data bus drives all 1's. When data starts coming in, the PHY drives the data bus to a speed_code value for one clock cycle before driving receive data across. End of packet is marked by driving Ctl[0:1]=00. The same protocol is used whether a packet is concatenated or single.

Transmitting multi-speed concatenated packets is more of a problem. No mechanism is defined to allow the link to indicate to the PHY the speed of the "next" concatenated transmit packet. It is an easy thing to invent; the more difficult part is making new PHY silicon that operates cleanly with existing link silicon.

6.2 State Machine Modifications for Multi-Speed Concatenation

The A5:Receive state actions are already consistent with multi-speed operation. Concatenated packet timing is acceptable as is. Only the transmit logic and phy-link interface require thought.

6.2.1 Modifications to Transmit Actions

The only modification necessary is to insure that a speed signal precedes each packet, including concatenated packets. This can be accomplished with a minor change to the code in the data_prefix case at the end of the transmit actions –

```
...
      case DATA_PREFIX:
         stop_tx_packet(DATA_PREFIX);
         wait_time (200 ns);                    //hold bus for concatenated packet
         start_tx_packet(req_speed);            //send data prefix & speed signal for next pkt
         break;
      case 0, 1:   //send data
         tx_bit(data_to_transmit);
         if (bit_count < 64)//accumulate first 64 bits
         phy_pkt.bits[bit_count++] = data_to_transmit;
         break;
      }
   }
end_of_transmission = true;
}
```

6.2.2 Modifications to the Phy-Link Interface

The phy-link interface operation must work for both the reception and transmission of concatenated packets. The mechanism shown in Annex J for reception is sufficient. However it does not explicitly mention packet concatenation; that is added below. Transmission requires a minor addition; the link needs a way of signalling to the PHY what speed

the next packet in the concatenated chain will be. That is easily accomplished by letting it drive the speed code for the next packet onto the data bus at the same time that it drives the bus_hold code onto the control bus.

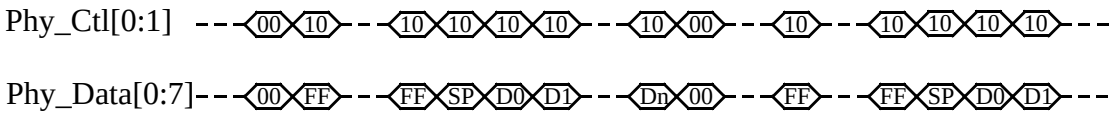


FIGURE 11. Phy-Link Interface - Concatenated Packet Reception

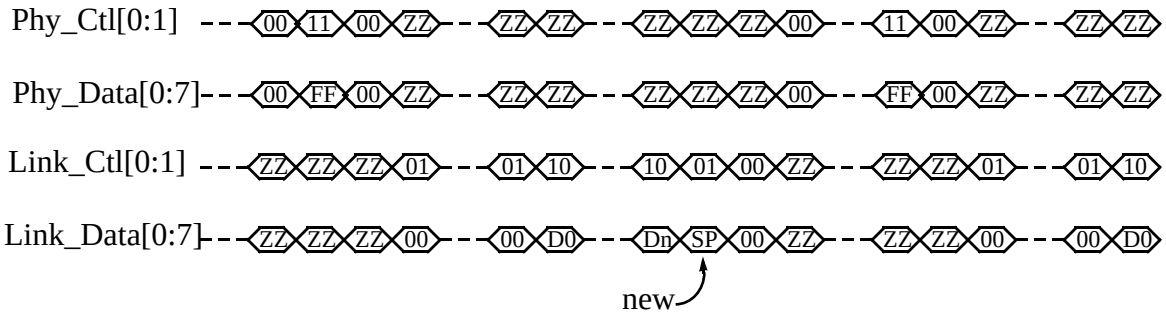


FIGURE 12. Phy-Link Interface - Concatenated Packet Transmission

Inter-operability of PHY and link silicon deserves consideration. New PHY silicon which expects the speed code to be transmitted by the link for each concatenated packet will interpret 0x00 on the data bus as indicating S100. Current link silicon (probably) assumes that each succeeding concatenated packet will be the same speed as the first packet transmitted. This can be solved by adding a control bit in a PHY register—call it multi_speed_concat_enable. If set, then the link must transmit the speed code when it indicates “holdbus” for concatenated packet. If clear, then all succeeding concatenated packets must be the same speed as the first packet transmitted.

7.0 Accelerated Arbitration

There are two arbitration acceleration “tricks”: ack-accelerated arbitration, and fly-by arbitration. The implementation strategy is best determined by considering the two mechanisms together, and modifying the arbitration state machine once.

7.1 Ack Accelerated Arbitration

The 1394-1995 standard implements a simple timing strategy for asynchronous arbitration—arbitration cannot start until the bus has been idle for a subaction_gap_detect_time.

For a small bus, a subaction gap may be one µs; a very large bus might be 5 µs; at the extreme, the largest gap time that PHY silicon can support is 10 µs.

This arbitration timing strategy was adopted at a time when the design goals for 1394 were modest. Simplicity, time-to-market, and proof-of-concept were of greater importance than getting the last bit of bandwidth. But the penalty for bus idle time becomes more onerous as the bit rates and bus sizes increase—exactly the growth path for the technology.

In a 1394 bus, the subaction_gap_detect_time is set to be greater than the worst case round-trip propagation time across the bus. This insures that when a packet is transmitted, no node will start arbitrating for the bus before the acknowledge packet has been transmitted and received; bus arbitration would interfere with packet propagation.

But in striving for simplicity, no distinction was made between acknowledge packets and other packets. Thus the bus remains idle for a subaction_gap_detect_time after transmission of an acknowledge packet. This is pure wasted bandwidth; there will be no acknowledge packet following an acknowledge packet. Arbitration could start immediately after an acknowledge packet, with no fear of interrupting bus activity.

This is an exact parallel to the style of arbitration during isochronous mode operation. Since there are no ack packets for isochronous transmissions, arbitration may start immediately after the packet.

7.2 Fly-By Arbitration

The 1394 Standard had one arbitration acceleration mechanism built in—packet concatenation. The mechanism is straightforward:

1) Node A sends Node B a read request.
2) Node B sends Node A an acknowledge packet.
3) If Node B can access the data requested by A quickly enough, it holds the bus after the ack packet, and concatenates the response packet onto the ack packet.

Whether the response packet is concatenated or not is largely a PHY matter. As illustrated in Figure 11, the link is unable to distinguish concatenated packets from closely packed packets. Fly-by arbitration is a matter of expanding the scope of the basic concatenation mechanism.

Suppose Node B has a packet ready to transmit, and is in the process of arbitrating for the bus, when an unrelated packet addressed to node B arrives, is received, and acknowledged by transmission of an ack packet. What consequences would there be if B concatenated its data packet onto the ack packet? There are some timing issues (see following section), but otherwise the bus at the link layer and above is ignorant of concatenation anyway. Providing fair arbitration rules are still followed—don’t concatenate or arbitrate if arb_enable bit is clear—everything works as before.

There is one further possible extension, but first a note about bus behavior:

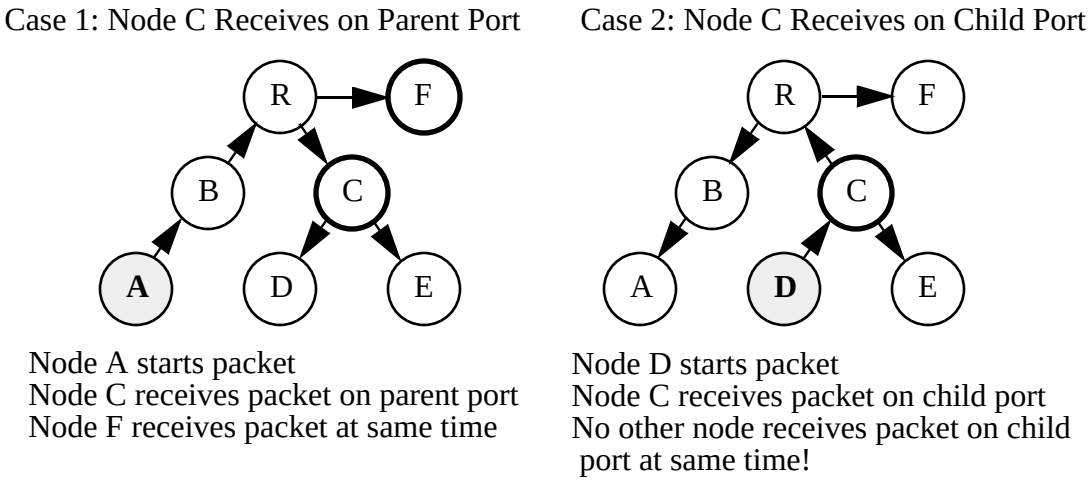


FIGURE 13. Distinction Between Parent Port and Child Port Reception

When a node receives a packet on its parent port, it is likely that other nodes are receiving the same packet at the same time. Thus any arbitration tricks used during parent port reception may be used by multiple nodes at once, which leads to packet collisions …

But when a packet is received on a child port, packet reception is unique; no other node can be receiving the packet on a child port at exactly the same time. (More precisely, when a node detects end-of-packet on a child receive port, no other node which receives the same packet on a child port will detect end-of-packet until at least one hop-delay later.)

Receiving a packet on a child port presents an opportunity for concatenation in two cases:

1) If the received packet is an acknowledge packet, then an unrelated asynchronous packet could be concatenated *on-the-fly.*
2) If the received packet is an isochronous packet (or a cycle-start packet), then an isochronous packet could be concatenated *on-the-fly.*

In either case, without fly-by arbitration, normal arbitration would follow the received packet. 1394 makes no guarantees about the order of packet transmission. As long as all nodes get to transmit asynchronous packets within a fairness interval, and isochronous packets within an isochronous cycle, the bus is well-behaved. Adding fly-by arbitration (and ack-accelerated arbitration) may change the order of packet transmission, but not the fundamental fairness of operation.

Note that the fly-by-concatenation mechanism concatenates a packet onto the end packet on the transmit ports. The original receive port just sees this extra packet as a separate packet coming back—refer to the figure below.

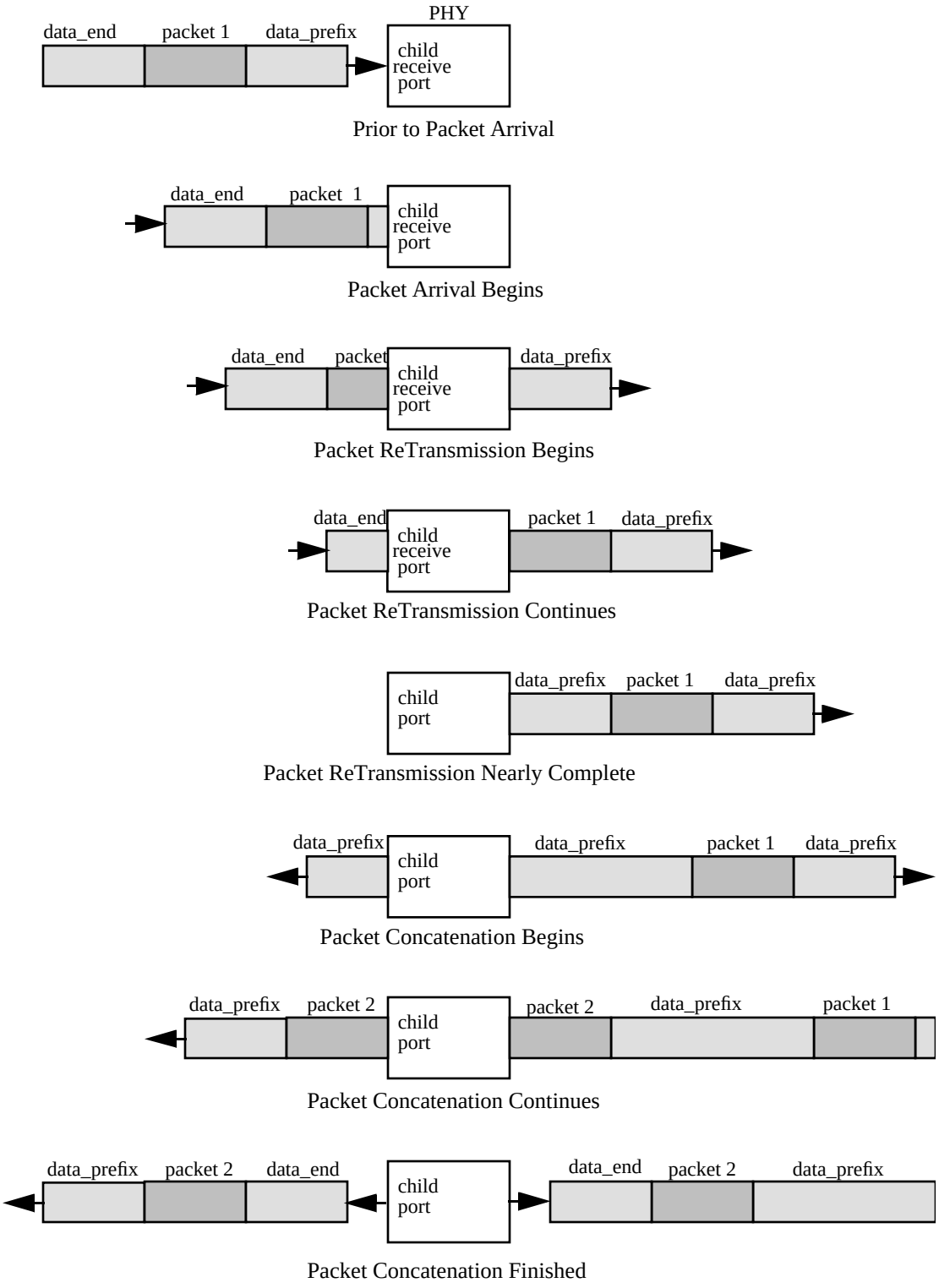


FIGURE 14. Fly-By Concatenation Process

7.3 1394-1995 Request Timing and Accelerated Arbitration

There is an inter-operability problem with ack accelerated arbitration. A 1394-1995 compliant node wishing to arbitrate for the bus will wait until after a subaction_gap_detect_time plus an additional arb_delay_time before starting arbitration. However, it will act immediately upon receiving a child port bus request. This is a problem if a compliant node is the root node. The root may be prevented from winning bus arbitration when it needs to send the cycle start packet if its child nodes are using ack-acceleration. And there are devices with compliant silicon in the market, though most of the later silicon available dodges this problem.

The problem is even more severe with fly-by arbitration. This mechanism does away with “normal” arbitration; nodes go directly from receive state to transmit state. Packets may bounce back and forth across the bus for some time without involving the root node in arbitration.

To overcome this incompatibility, the arbitration acceleration mechanisms should only be used when the cycle-start-packet is not due. If the bus is asynchronous-only, the arbitration tricks may be used freely; otherwise means must be found to sense when the cycle-start packet is due.

One approach would be to re-partition the logic between the PHY and link, so that the cycle-timer is in the phy. Bad idea. (But the problem does largely go away for integrated phy-link ICs.)

A simpler approach would be to invent a new request type for the LReq interface, a request which would be similar to a fair request, except that acceleration would be permissible. The original fair request would still be used when a cycle-start-packet is due, as well as for backward compatibility with older PHY silicon. This has the convenient advantage that only new link silicon, with the necessary linkage to the cycle-start-timer, would ever use the new accelerated fair request.

Finally, there are discussions on using the cycle-master request in a more generic fashion; nodes would also be allowed to use the cycle-master request for response packets. This would allow a node to respond to multiple read requests—from different requesters—all within one fairness interval. Suppose we call these requests priority-requests. The same acceleration considerations apply. The root node could always allow accelerated arbitration modes; other nodes would use accelerated priority arbitration only when it isn’t cycle-start-time.

TABLE 3. Summary of Bus Requests

LR[1:3]	Name	Meaning
000	ImmReq	Take control of bus immediately upon detecting idle; do not arbitrate. Used for acknowledge transfers
001	IsoReq	Arbitrate for the bus; no gaps. Used for isochronous transfers.
010	PriReq	Arbitrate after a subaction gap; ignore fair protocol. Used for cycle master request and response packet requests.

TABLE 3. Summary of Bus Requests

LR[1:3]	Name	Meaning
011	FairReq	Arbitrate after a subaction gap, following fair protocol. Used for Fair transfers. (In backplane environment, request priority field differentiates fair and urgent transfers.)
100	RdReg	Return specified register contents through status transfer.
101	WrReg	Write to specified register.
110	AccPriReq	Use like PriReq when accelerated arbitration OK (not cycle start time—unless root)
111	AccFairReq	Use like FairReq when accelerated arbitration OK (not cycle start time)

7.4 State Machine Modifications for Accelerated Arbitration

For ack acceleration, all of the modifications are of the general nature of shortcuts that allow transition out of A0:Idle at an earlier time. An AccPriReq or (AccFairReq & arb_enable) allows an early exit from A0, if the request is made before arb_timer > subaction_gap_detect_time. The root can transition directly to A6:Transmit; other nodes can go directly to A3:Request. The root is allowed to make the fast transition to A6 for PriReq as well as AccPriReq; this allows even older links to glean a little extra bandwidth. Finally, a minor change to the A0:A4 conditions clarifies the state transition taken by the root if a child request and a (PriReq or AccPriReq) go active at the same time—the root will ignore the child request and take the bus to transmit its packet.

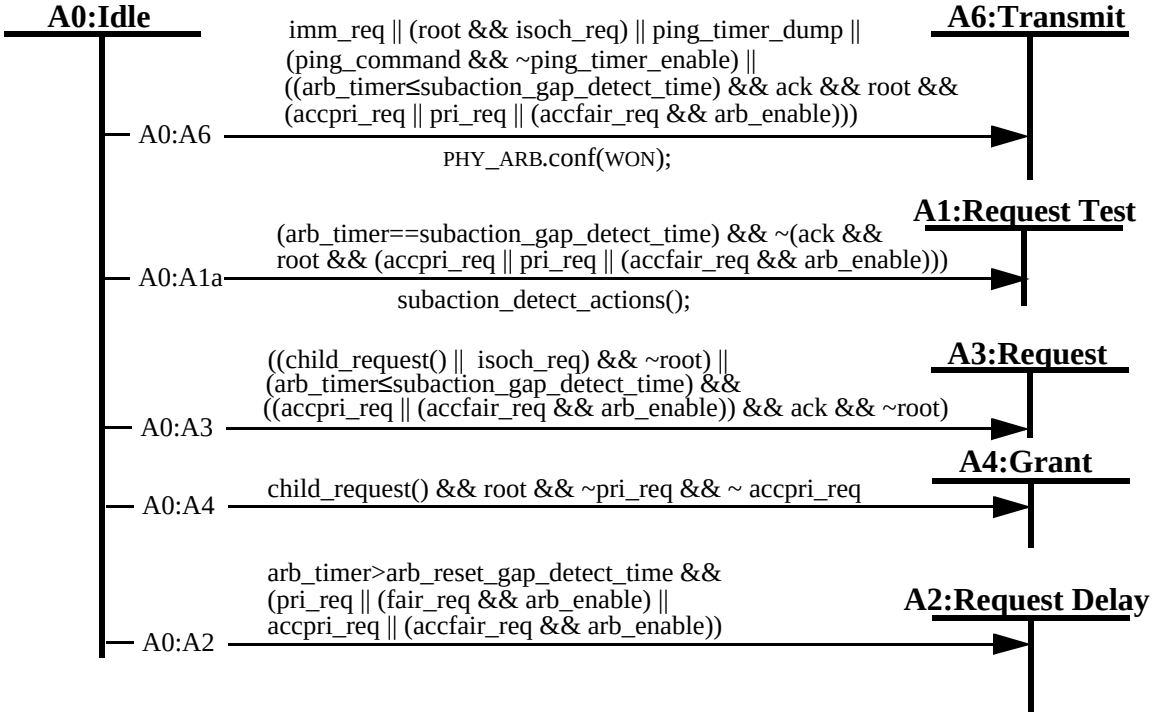


FIGURE 15. Modified A0:An State Transitions for Ack Accelerated Arbitration

State machine changes for fly-by arbitration are straightforward. There is a new state transition allowed, direct from A5:Receive to A6:Transmit, provided that the receive port is a child port, and (that an isochronous request is active, or that an accelerated request is active and that the last packet was an ack). This is all better expressed in Boolean than English. Some minor modifications are also necessary to allow accelerated bus requests to be made while in A5:Receive, and not be cleared by reception of ack packets.

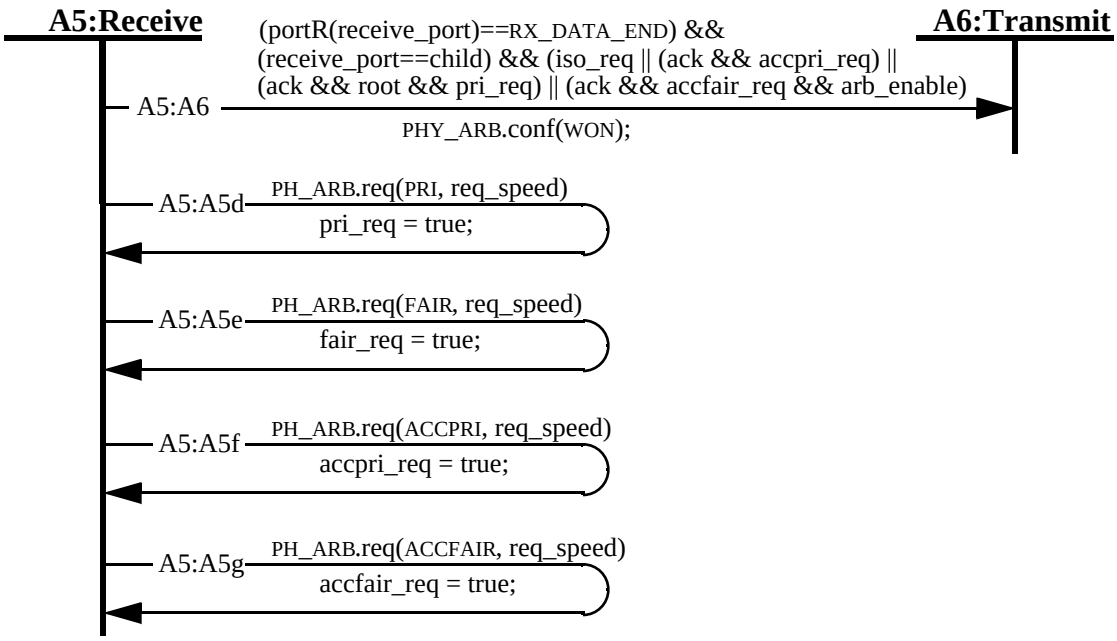


FIGURE 16. Modified State Transitions for Fly-By Arbitration

The C code for receive_actions specifies the time for clearing bus requests; some minor changes are suggested.

```
...
boolean test_end = false;
boolean received_data;
fair_req = false;                       //clear at start of receive state actions
if (~root)
   pri_req = false;                     //don't clear pri_req if root
PH_DATA.ind(DATA_PREFIX);               //send notification of receive activity
rx_speed = start_rx_packet();           //start up receiver and repeater

while (~test_end) {
   rx_bit(&received_data, &test_end);
   if (~test_end) {                     //normal data, send to link layer
      PH_DATA.ind(received_data);
      if (bit_count = 8) {              //if 8 bits and more coming, not an ack, clr reqs
         accfair_req = false;
         accpri_req = false;
         pri_req = false;
         ack = false;
```

```
        }
      if (bit_count<64)                          //accumulate first 64 bits
        phy_pkt.bits[bit_count++] = received_data;
      }
    }
ending_data = portR(receive_port);
switch (ending_data) {                           //send end of packet indicator
   case RX_DATA_PREFIX : PH_DATA.ind(DATA_PREFIX);   //concatenated packet coming
      break
   case RX_DATA_END && (iso_req || (ack && accpri_req) || (ack && pri_req && root) ||
      (ack && accfair_req && arb_enable) : start_tx_packet();
      break                                      //transitions to transmit state next
   case RX_DATA_END && ~(iso_req || (ack && accpri_req) || (ack && pri_req && root) ||
      (ack && accfair_req && arb_enable) : PH_DATA.ind(DATA_END);
      break                                      //normal end of packet
   }
stop_rx_packet (ending_data);
...
```

8.0 Token Style Arbitration

The 1394-1995 Standard actually describes two forms of bus arbitration. The arbitration mechanism used in arbitration phase is by now well known—nodes pass requests upwards, and receive grants or bus denials in return. But a second style of arbitration is involved in the self-ID process. During self-id, each node gets to transmit a packet or set of packets (if it has more than 3 ports). The opportunity to transmit circulates through the bus in a deterministic order. This self-ID arbitration behaves very much like a token-passing mechanism. There is no token, but there is a “bus grant”, which circulates around the bus. The state machine rules governing use of the grant—when it can be used to transmit, when it must be passed down—give this phase of operation the feel of token-based arbitration.

Fly-by arbitration offers considerable reduction in arbitration overhead. Unfortunately, in one of the high-end applications where it would be most attractive, it won’t work well.

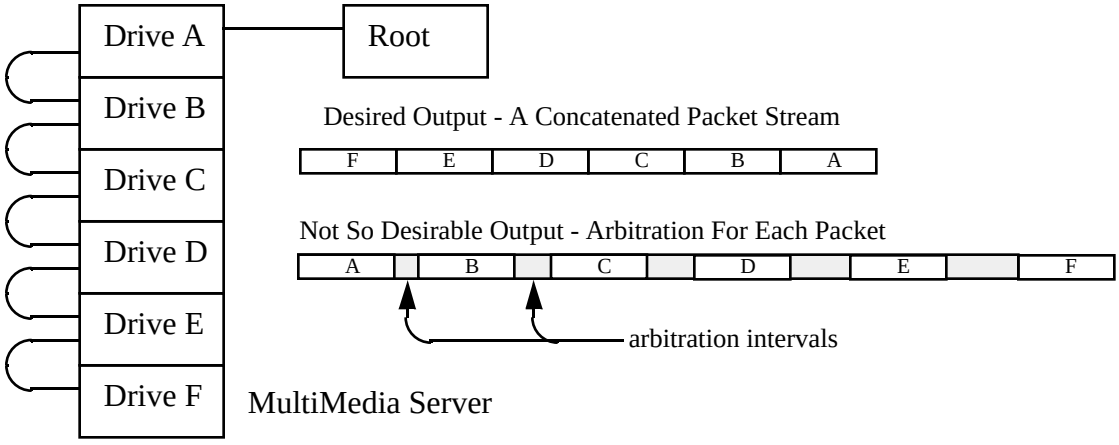
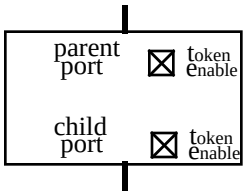


FIGURE 17. The Isochronous Fly-By Problem

For a multi-media server, an output consisting of one long concatenated packet stream would be ideal. Unfortunately, left to its natural behavior, the nodes closer to the root will tend to win bus arbitration first. The opportunity to use fly-by arbitration is lost. Some additional mechanism is needed to allow Drive F to transmit its isochronous packet first, which will enable the other drives to concatenate their packets, using fly-by arbitration. Token style arbitration is such a mechanism.

8.1 Suggested Implementation

This suggested implementation will serve as a starting point for discussion. There are many options; the implementation is simplified by limiting the operation to fly-by arbitration for isochronous transmission, and mandating that there will be no nodes in the token-capable chain that are not token-capable.



Token-Arbitration Capable Node

Give each port a token enable bit—te for short—which will control arbitration behavior during isochronous operation. The simplifying assumption is that if the parent port te bit is set, all the child te bits are automatically set. (This assumption can be discarded if the PHY has a way of detecting the cycle start message—moderately difficult for a stand-alone phy, but easy and free for an integrated phy-link IC.) The te bits generally get configured and remain unchanged for long times. In the dedicated multimedia server example, they might even be factory set, or masked into the PHY silicon.

If the te bit is set for a parent port, then the node will not arbitrate for an isochronous request, but will wait for a bus grant.

If the te bit is set for a child port, then bus grants received during isochronous operation will be passed along to those ports. If a te bit is set for a child port, but the parent te bit is clear, then the node must arbitrate for the bus after the cycle start packet is detected. (Detection could be done by the link IC, which would always respond with an iso_request.)

Note (and avoid) the undesirable situation where a parent node has its child port te bit cleared, and its child node has its parent port te bit set—the parent won't send a grant, and the child won't request—a silicon implementation of *don't ask, don't tell*.

Referring to Figure 17, there are two possibilities: the root could be part of the scheme, with a te bit set for the server port, or the root could be ignorant, and have its te bit cleared, or perhaps have no te bit whatsoever, if its silicon is unaware of token style arbitration.

In the first case, where the root is part of the scheme, the operation is as follows:

1) The root node sends the cycle start packet, which starts isochronous operation.

2) The root sends a bus grant to its lowest numbered te port, and data prefix to all others.

3) Node A receives the bus grant, and passes it on to its lowest numbered te child port (Figure 17 shows a series of two-port nodes; *lowest numbered* would apply to the more general case of nodes with three or more ports).

4) In similar fashion, the bus grant drops through to node F. Having no te child ports, node F uses the grant to transmit its isochronous packet(chain).

5) Node E receives the packet(chain) from its child port. Receipt of a packet on a child port with DATA_END termination is the transmit opportunity. If node E has isochronous packet(s) to transmit, it concatenates its traffic onto the end of the passing packet(chain). Regardless of whether node E concatenates a packet or not, the opportunity to transmit passes up the chain with the DATA_END packet termination.

6) In a similar fashion, the packet chain grows as it propagates past nodes D, C, B, and A.

If any node had two or more te child ports, it would pass the grant first to its lowest numbered child port. When a packet with DATA_END termination returns, it changes the termination to DATA_PREFIX on the parent port on the fly, and drops the grant to the next te child port. When all te child ports have had the opportunity to transmit, then the node concatenates its own traffic onto the packet chain, and terminates the chain with DATA_END.

Suppose the root does not have a te bit set for the server chain—perhaps the root silicon does not support the feature, or perhaps there are intervening nodes which don't offer token arbitration. Then node A would have its parent port te bit clear. Operation would be as follows:

1) The root node sends the cycle start packet, which starts isochronous operation.

2) All the non-token savvy nodes on the bus would start normal bus arbitration. Node A would also participate in arbitration. Node A's arbitration could be initiated by PHY recognition of the cycle start package, or it could be initiated by node A's link IC, which would recognize the cycle start message and issue an iso_request. Note that nodeA's link should always issue a bus request, even if it has no data ready for transmission itself, if there is any chance that any nodes in the token branch may have isochronous packets to send.

3) Sooner or later, node A wins arbitration. When it does, it sends data_prefix up its parent port, and drops a bus grant down its lowest numbered child te port.

4) From here, operation is the same as the prior example.

8.2 Advantages

1) The first advantage is the savings in bandwidth over standard arbitration. Assume a 6-node token chain. In non-token mode, arbitration time would be roughly 0.15µsec (2 + 4 + 6 + 8 + 10 + 12) = ~ 6 µsec. In token-mode, arbitration time is just the time for the bus grant to drop to the end of the chain, about 0.15 µsec(6) = ~ 1 µsec.

2) Whether the token chain is a straight daisy-chain, or a branched tree, the total time for the bus grant to drop is about the same. There is less penalty for a daisy-chain architecture.

3) Again referring to Figure 17, suppose Node A is the master controller for the media server. It controls the other nodes by sending them asynchronous packets, polling their status, etc. Nodes B–F respond with concatenated ack-response packets, but otherwise never autonomously arbitrate for the bus. Any external device which wishes to communicate with the media server communicates with Node A.

   Given all these limitations on operation—which may be acceptable for specialized applications—Nodes B - F would not figure into the gap count determination. Two port PHYs and daisy-chain configurations suddenly become an optimal, inexpensive solution.

8.3 State Machine Modifications for Token Style Arbitration

There are two areas of interest for state machine modifications: starting token arbitration (i.e. sending the grant), and the token arbitration process itself (propagating the grant and transmitting packets).

There are three classes of nodes in a token enabled bus: nodes in the token chain (which have their parent token-enable te bit set), the token parent (which has its child te bits set, but not its parent port te bit), and nodes outside the token chain (which have no te bits set—and may have no capability for token operation). The token parent has the sole responsibility for starting token operation.

For this discussion, assume that the PHYs have no special ability to detect cycle start packets—the link already has that capability. For the token parent node, the link has the responsibility of always issuing an isochronous request upon detection of the cycle start packet, whether it has a local packet to transmit or not; this is the action that starts token operation.

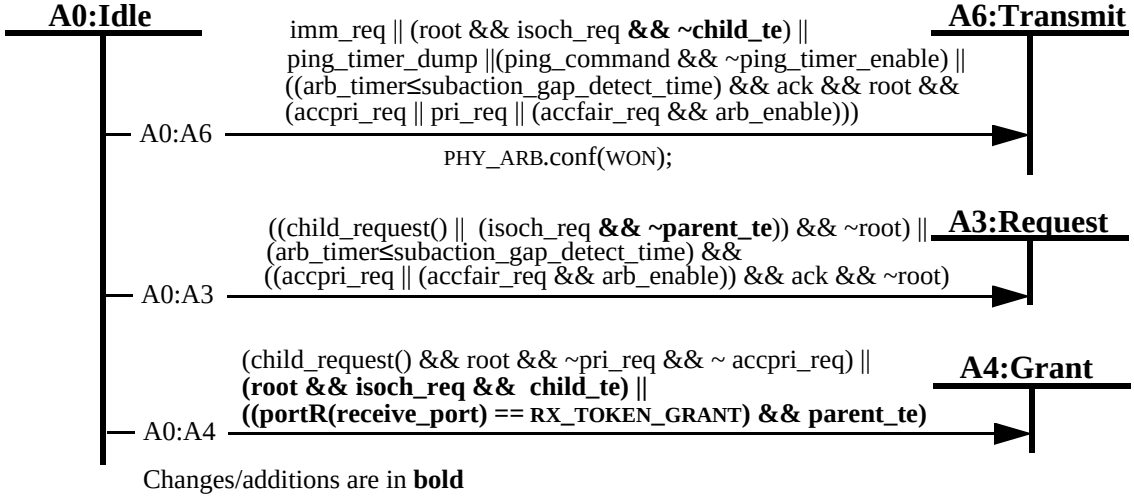


FIGURE 18. Modified A0:An State Transitions for Token Isochronous Arbitration

Note that RX_TOKEN_GRANT is the same line state as RX_SELF_ID_GRANT; it would be confusing to use the latter name in this new context.

If the token-enabled bus branch is a simple daisy chain, then the additional state machine modifications are minor; all that is required is to propagate the token_grant to the end of the chain, let the leaf node transmit, and then fly-by arbitration does the rest. If the token-enabled branch is indeed “branched”, then there must also be a token_active status bit—ta—which indicates that token operation is in progress, i.e. the token_grant has been received and passed on, but not used (to send a packet). This bit allows a node in the token chain at a branch point—having multiple child ports—to drop the grant down one child port after another until the highest numbered child chain has its transmit opportunity. The following state machine diagrams assume a branched configuration. Only the new or modified transitions are shown; the unchanged transitions are generally not shown.

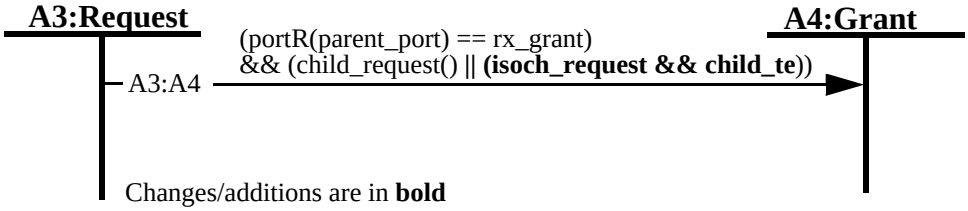


FIGURE 19. Modified A3:A4 State Transition for Token Isochronous Arbitration

If a leaf node receives the token_grant but has no isochronous packet to transmit, i.e. no active isochronous request, then it needs to refuse the grant. The conventional way to refuse a grant is to send an empty packet—data prefix followed by data end, with no data. A convenient short-cut for token-arbitration only is to respond to the token grant (AB=0Z) by driving the parent port to AB=Z0; the resulting bus state is AB=00. This seems paradoxical; AB=Z0 is normally used for TX_REQUEST. However, driving the B pair to logic 0 will not interfere with receiving data prefix, which is the eventual exit from the refused grant scenario. These actions can be included in the grant state actions. The eventual reception of data prefix gives a new transition directly to receive state.

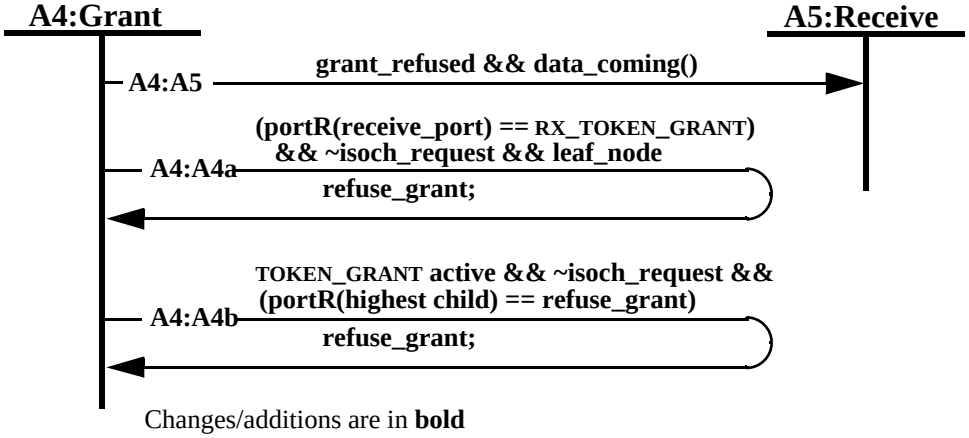


FIGURE 20. Modified A4:A5 State Transitions for Token Isochronous Arbitration

Note that in the figure above two pseudo-transitions are shown—A4:A4a and A4:A4b. These are drawn in a manner which suggests that grant actions should restart when the transitions are made, which is not true. The “transitions” should be included within the grant actions, but are brought out here for purposes of explanation. Such usage is not inconsistent with the current 1394 standard—see transitions A5:A5b and A5:A5c. Both transitions (and the similar ones added in Figure 16) are link-initiated bus requests, which do not re-start the receive actions.

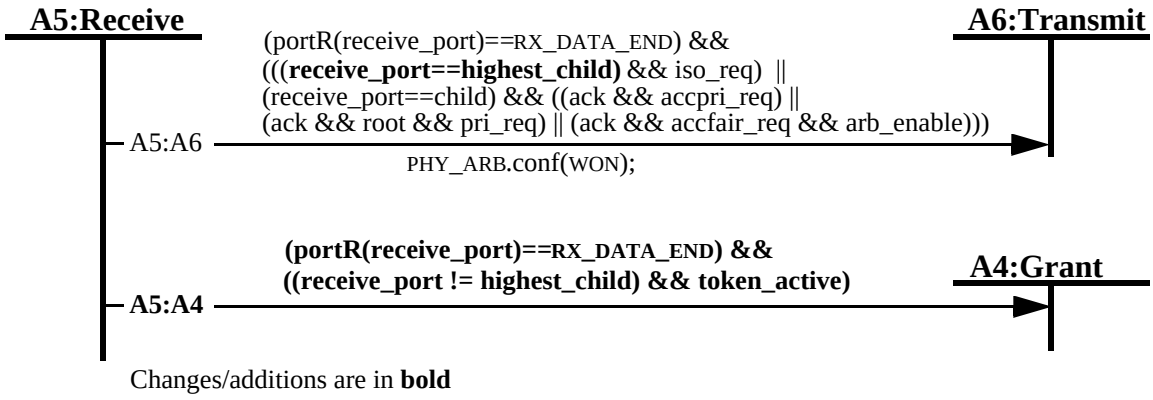


FIGURE 21. Modified A5:An State Transitions for Token Isochronous Arbitration

9.0 Per Port Software Disconnect

This long-simmering feature has been awaiting a compelling argument for adoption. The modifications to the phy to allow it to shut down a port are straightforward; the potential complications of managing a bus where devices can make local decisions about which ports to disconnect are intimidating.

The compelling argument is this: a personal computer with a 1394 hard drive might be unable to boot, due to a malfunction of an external 1394 device. If the ROM code of the PC had the ability to turn off the external 1394 ports, then at least the computer could boot to some useful level automatically, unattended. There are applications where such automatic operation is desirable or mandatory.

The counter-argument has always been that a bus where the presence of a cable does not guarantee a connection will be difficult to manage and trouble-shoot, both for human users and for software agents. A user nightmare can be avoided if:

1) A device which can perform port disconnections shall have some basic level of bus management capability.
2) No port shall be software-disconnected without some visual indication of the disconnect.

9.1 Disconnect Mechanism

For each port we define a new port disconnect bit. This bit is cleared by all resets, including bus resets. When set, it causes:

1) TpBias current for the port is shut off.

2) The port drivers are tri-stated.

3) The port line state receivers are all forced to inert “disconnect” states, with the sole exception of the common mode connection status receiver.

4) The port common mode connection status should be a readable bit. Except for being readable, it should also be forced inactive, i.e. it should appear to the phy connection state machine as a no-connect.

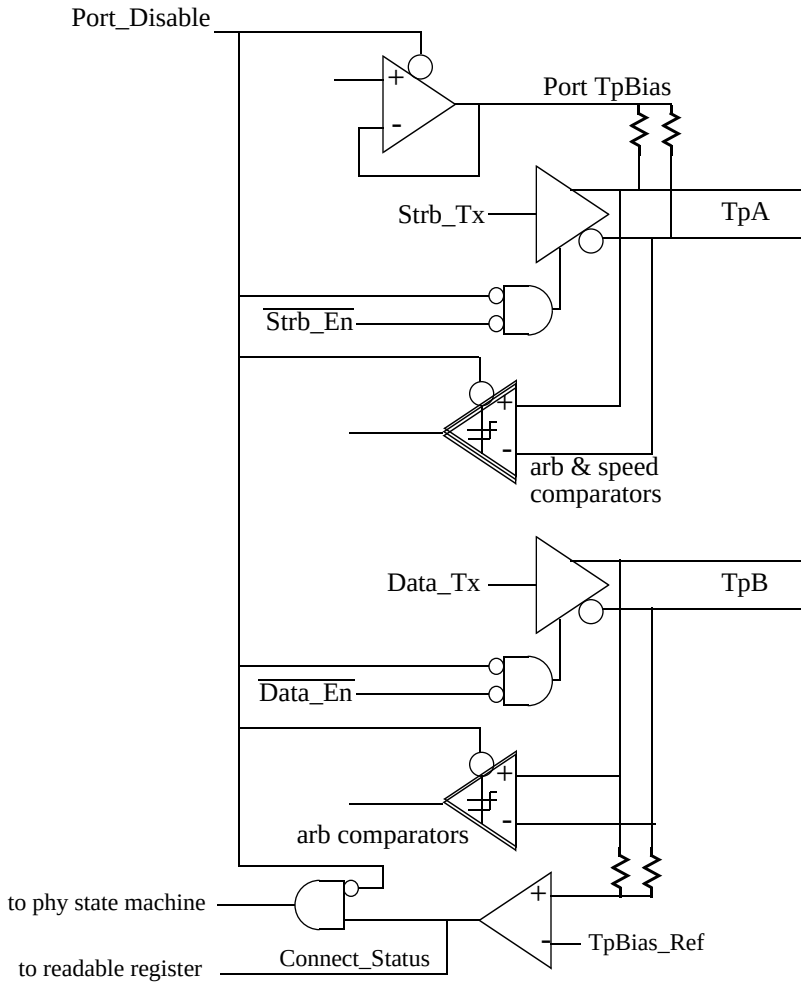


FIGURE 22. Port Disable Logic

10.0 Incremental Bus Re-Configuration

Bus Reset, long or short, is standard dogma for 1394 connection status changes. The attraction is simplicity; the drawback is the possible interruption of service while bus initialization takes place. A secondary drawback is that the physical node IDs may change with each bus re-initialization, which is an annoyance for a local bus, and a big annoyance for an extended bus (multiple buses connected by bridges).

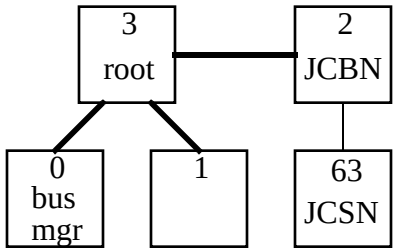
A simple observation is that the number of bus resets could be reduced by not doing one for a child port disconnect. That does put an extra burden on software; the first warning of a disconnect is a lack of response—an interruption of service. But that can also happen with short arbitrated bus resets; a disconnect will not be widely known until after one or more rounds of bus arbitration; software may detect the interruption of service before bus re-initialization in that case also.

A more *polite* mechanism would be to send a special alert message after detecting a child port disconnect. The node would arbitrate for the bus—normal fair arbitration—and send the NODE_DETACHED_ALERT upon winning arbitration. A bus topology manager, if present, could ping the sender of the alert message for confirmation. The problem remains that software may detect the loss of service before the alert message arrives. But if that can be tolerated -

Couldn’t a similar alert mechanism be used for new connections as well? In addition to the alert message, some means will be needed to set the physical address of the new node, and to keep it quiet until it has that new address. These requirements are not difficult to meet. The appeal to all this is that the two PHYs directly involved in the new connection would both have to be knowledgeable new silicon, but no other PHY on the bus would have to understand the new protocol. (An annoyance with the short arbitrated reset mechanism is that every node on the bus has to be short reset savvy; one old long reset PHY will force the entire bus into a long reset.)

10.1 New Connect Re-configuration Changes

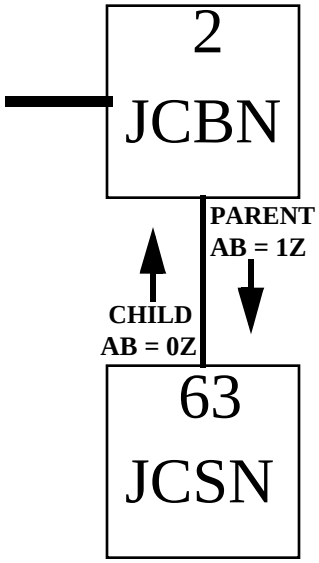
Start with the 1394 bus shown on the right. There is an existing bus, consisting of nodes 0 through 3. Node 3 is the root; node 0 is the bus manager; node 2 is the node which has just detected a new connection (JCBN = Just Connected Bus Node). The node designated JCSN (Just Connected Single Node) is the new node, which does not yet have a valid physical ID (63 is the chip reset default, but it is an invalid ID, indicating to the link that initialization is pending, preventing the node from requesting the bus).



1394 Bus With New Connection

Once node 2 detects the new connection, it sends the standard PARENT (officially known as TX_CHILD_NOTIFY) signal to its newly connected port. This is the same signal used in tree-ID. Node 2 should also begin sending its speed signal at this time.

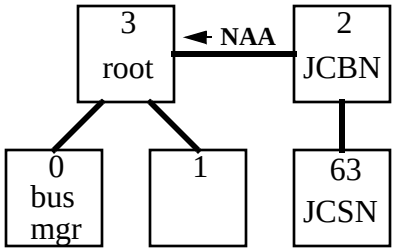
Notice that if the newly connected single node (JCSN) were actually another bus node, then both nodes would drive AB = 1Z onto the cable. Because of the AB swap in 1394 cables, both sides would see AB = 11, which would trigger a bus reset. Bus reset is the desired outcome when two buses are connected together—so far so good.



Detail of New Connect Handshake

Node 63, the JCSN responds by sending back the CHILD (officially the TX_PARENT_NOTIFY) signal. This is also the same signal used in tree-ID. The JCSN should also send its speed signal at this time. Sending the CHILD signal signifies that the JCSN has sensed the PARENT signal (and speed signal).

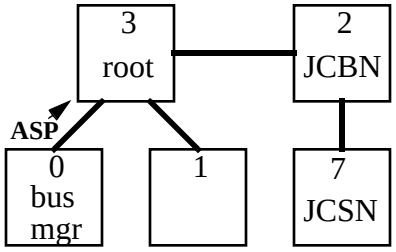
Once node 2 detects the CHILD signal (and accompanying speed signal), it drops the PARENT signal. The JCSN node responds by dropping its CHILD signal, and node 2 begins arbitrating for the bus, using normal fair arbitration. When it eventually wins arbitration, it sends out a NODE_ADDED_ALERT (NAA).



NODE_ADDED_ALERT Transmission

Despite the return to "normal" bus states, node 63 is prevented from transmitting; its node ID is still invalid. However, it can now receive packets.

The bus manager, if present, returns an ADDRESS_SET_PACKET (ASP). (If node 63, the JCSN fails to receive this packet, given a lengthy timeout period, it will initiate a bus reset.) This packet is received by the JCSN, and writes directly to the physical ID register. Exactly what node ID is used is arbitrary, so long as the bus manager insures that it is an unused address.



ADDRESS_SET_PACKET Action

A refinement can allow this mechanism to function correctly even if additional nodes are added before the first ADDRESS_SET_PACKET is transmitted. Let all nodes capable of sending a NODE_ADDED_ALERT (NAA) also be capable of detecting one. If an NAA is detected while arbitrating for the bus to send one's own NAA, then set a flag NAP (node addition in progress) which will prevent further arbitration for NAA. When and if a subsequent ADDRESS_SET_PACKET is detected, the flag is cleared, and NAA arbitration may be resumed. Also note that when NAP is set, the node must block packet transmission to its newly connected port.

The biggest drawback to this mechanism is that the bus manager must be ever vigilant, on the watch for alert packets which may arrive at any time.

Another drawback is that the regularity and predictability of node IDs has been lost; a list of self-id packets is no longer sufficient to reconstruct the topology. If the bus manager fails to keep track of the changes as they occur, or if the data is inadvertently lost, it would be necessary to do a bus reset to retrieve the topology information. This could be fixed as well, though the "fix" has the disadvantage that all PHYs on the bus would need the new feature, not just the ones involved in the new connection. But for the sake of furthering the discussion…

The problem is that the node IDs are now random. There is no way to parse the ping response packets and determine where nodes are in the hierarchy. A brute force way around that is to enlarge the ping command; call the new version CPP, for

CHILD_PARENT_PING. The target node of a CPP command responds by sending its self ID packet, and includes the termination signal TX_IDENT_DONE AB=1Z. Its parent node detects the self ID packet and TX_IDENT_DONE, and then transmit its own self ID packet (without TX_IDENT_DONE). Thus the new child_parent_ping operation yields a pair of self-id packets; the second self-id packet is guaranteed to come from the parent of the node which sent the first self-id packet.

10.2 New Packet Definition

A number of new packets have been referred to by name only. They can all be implemented in the form of PHY configuration packets, as shown below. A complete listing of all new PHY configuration packets is also included in the Appendix to this document.

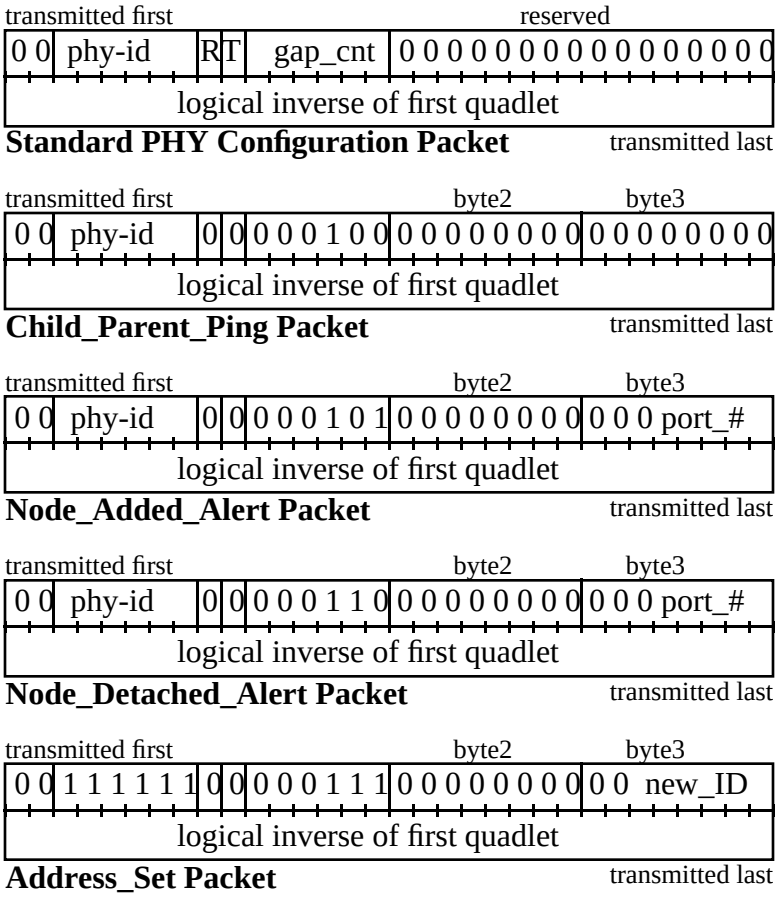


FIGURE 23. New PHY Packets for Incremental Bus Re-configuration

10.3 State Machine Additions for Incremental Bus Re-Configuration

The state machines for incremental bus re-configuration are mostly independent of the main PHY state machine. With the exception of transmitting the node added alert and node

detached alert, the necessary actions occur in the background as the phy performs its usual activities.

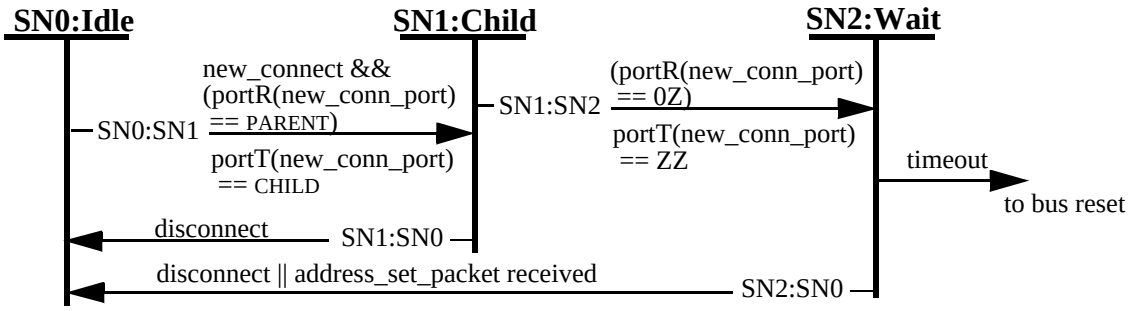
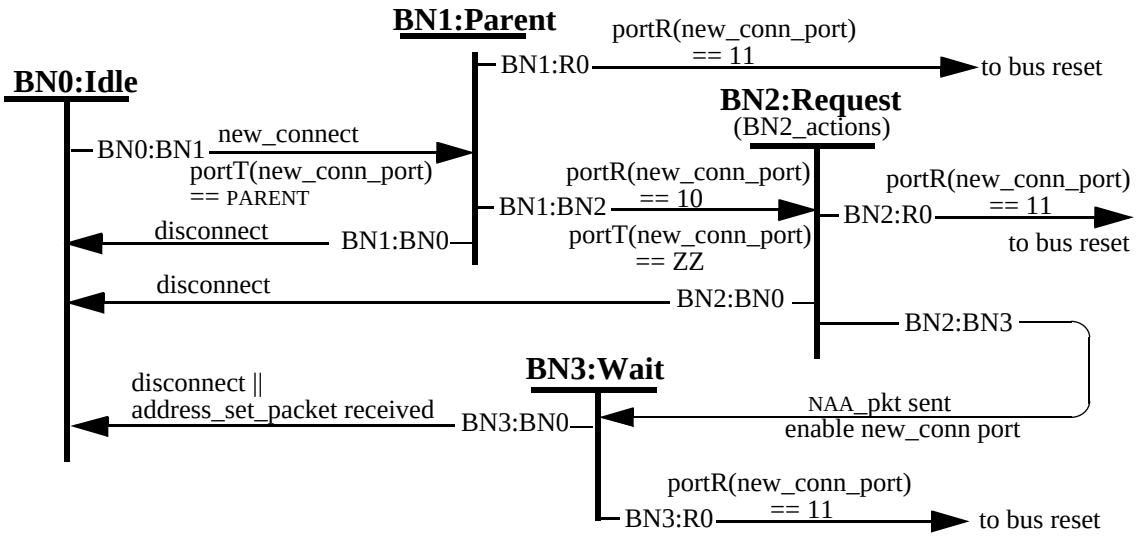


FIGURE 24. Single Node Incremental Re-Configuration State Machine

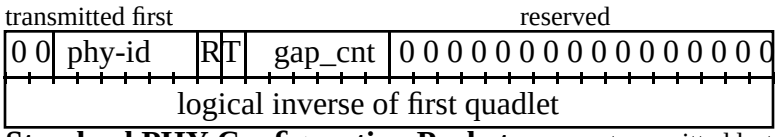


BN2 Action: if ~NAP & (portR(new_conn_port) == ZZ) NAA_BUS_REQUEST = TRUE;
(NAP is set by detection of NAA packet, and cleared by detection of address_set_packet)

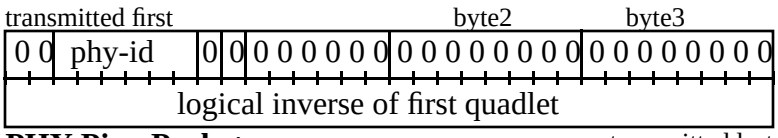
FIGURE 25. Bus Node Incremental Re-Configuration State Machine

11.0 Appendix

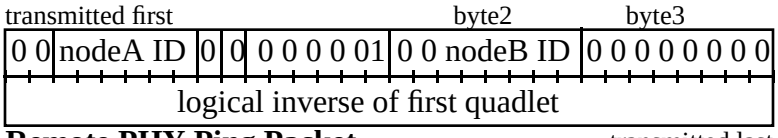
11.1 New PHY Configuration Packet Formats



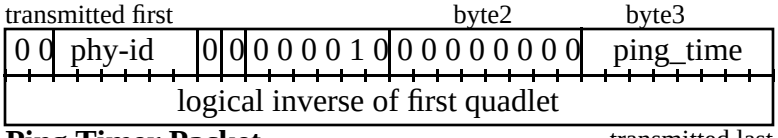
Standard PHY Configuration Packet transmitted last



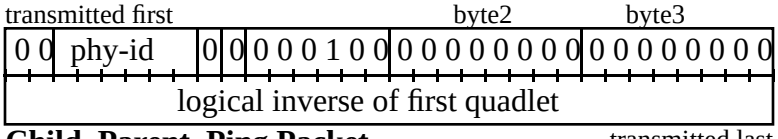
PHY Ping Packet transmitted last



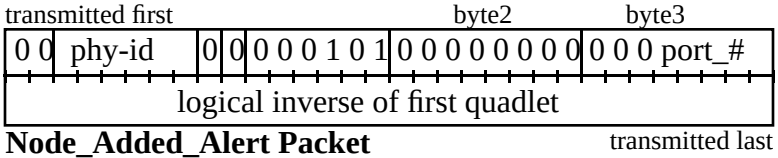
Remote PHY Ping Packet transmitted last



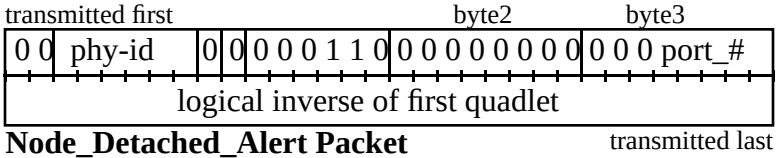
Ping Timer Packet transmitted last



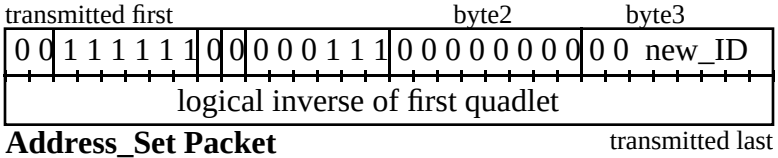
Child_Parent_Ping Packet transmitted last



Node_Added_Alert Packet transmitted last



Node_Detached_Alert Packet transmitted last



Address_Set Packet transmitted last

11.2 PHY Registers

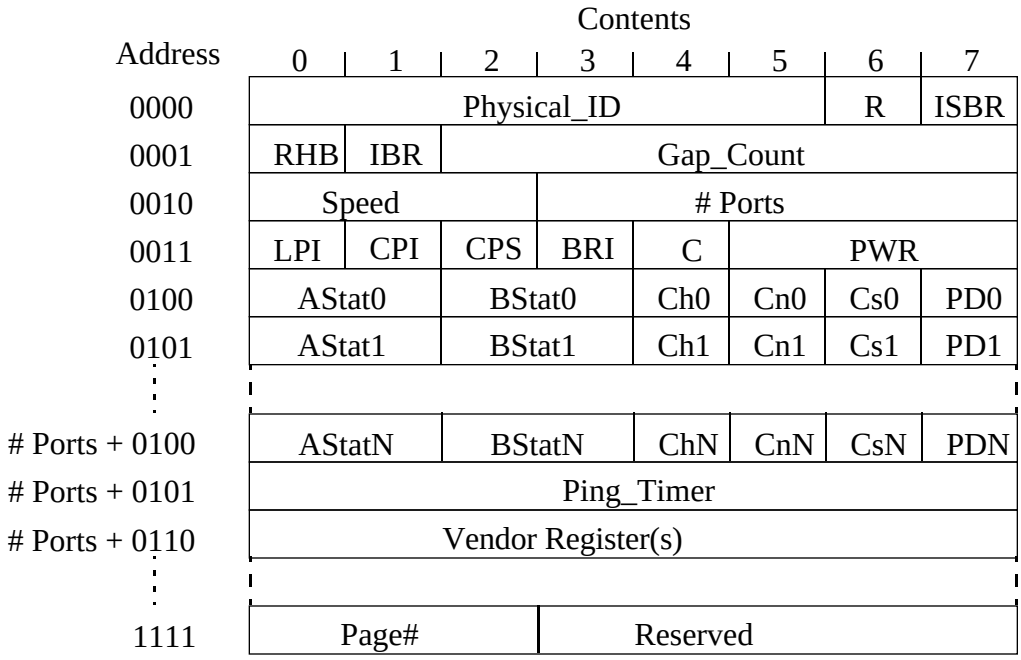


FIGURE 26. Phy Register Map

TABLE 4.

Field	Size	Type	Description
Physical_ID	6	r	The address of this node determined during self-ID
R	1	r	Indicates node is root
ISBR	1	rw	Initiate short bus reset
RHB	1	rw	Root holdoff bit
IBR	1	rw	Initiate bus reset
Gap_Count	6	rw	Arbitration timer setting
Speed	3	r	Top speed this phy can handle
# Ports	5	r	Number of ports on this phy
LPI	1	r	Loop detect interrupt
CPI	1	r	Cable power fail interrupt
CPS	1	r	Cable power status
BRI	1	r	Last bus reset was initiated by this phy
C	1	r	Contender bit (external PHY pin)
PWR	3	r	Power_class (external PHY pins)
AStat(n)	2	r	TPA line state on port n 11 = Z; 01 = 1; 10 = 0; 00 = invalid
BStat(n)	2	r	TPB line state on port n (same code as AStat)

TABLE 4.

Field	Size	Type	Description
Ch(n)	1	r	If Ch=1, port is child; else parent port.
Cn(n)	1	r	If Cn=1, port is connected & enabled
Cs(n)	1	r	If Cs=1, bias voltage is detected (possible connection)
PD(n)	1	rw	PD=1 disables the port
Ping_Timer	8	r	Round trip delay for last ping transaction
Vendor Registers	8	-	Implementation dependent
Page #	3	rw	Multiple pages needed for wide phys write page# to access desired page write not allowed if page# not valid