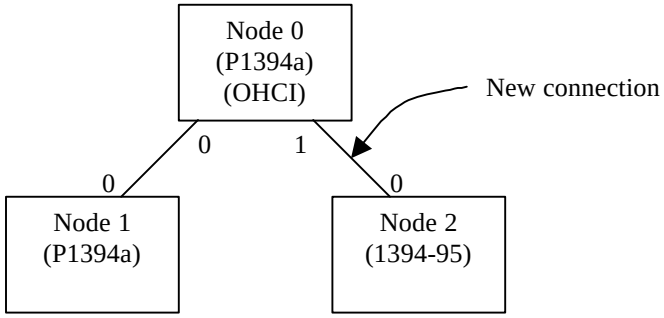
Reset/resume livelock

Colin Whitby-Strevens with lots of help from Mike Lee
Zayante Inc
02 May 1999

1 Introduction

We've been informally asked to investigate a problem which has been noticed when a -95 node is connected to a network of P1394a nodes.



Notation in the text:- Node/Port (i.e. 0/1 is node 0, port 1)

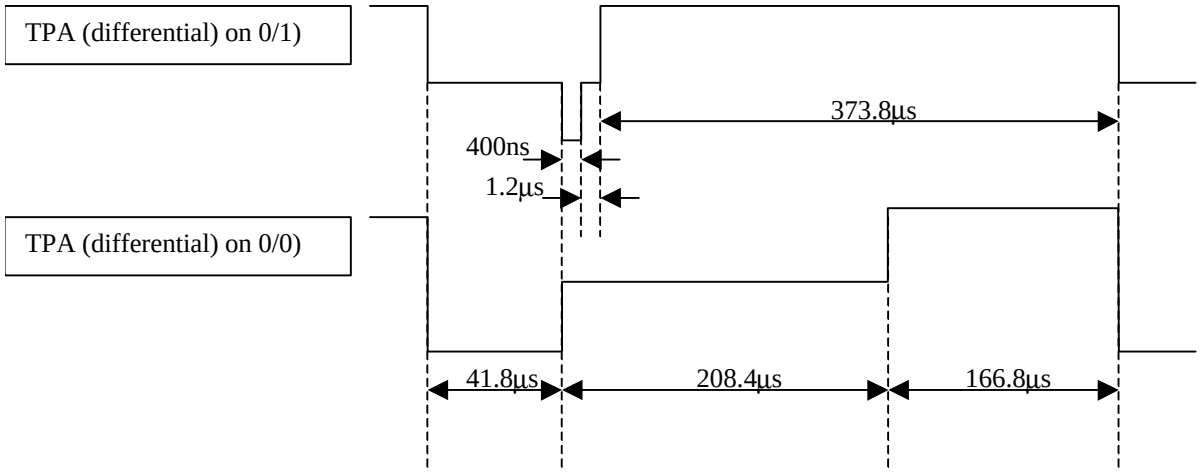
The symptoms reported are that occasionally, when the new connection is made, the software on Node 0 sees a reset and an isolated node, followed by another reset and inconsistent self-ID. The effect has been noticed with a variety of different manufacturers devices (different PHYs) for both the P1394a PHYs and the -95 PHY. The effect is also reported when Node 2 is connected but powered off and then powered on.

2 Investigation

We have investigated the above configuration, and have observed failure modes with about 5% probability (we did not count). We observed the "isolated" node effect, but we did not have time to adjust test software to look at this further. Rather, we decided to investigate further with "naked PHYs", thus eliminating software influences.

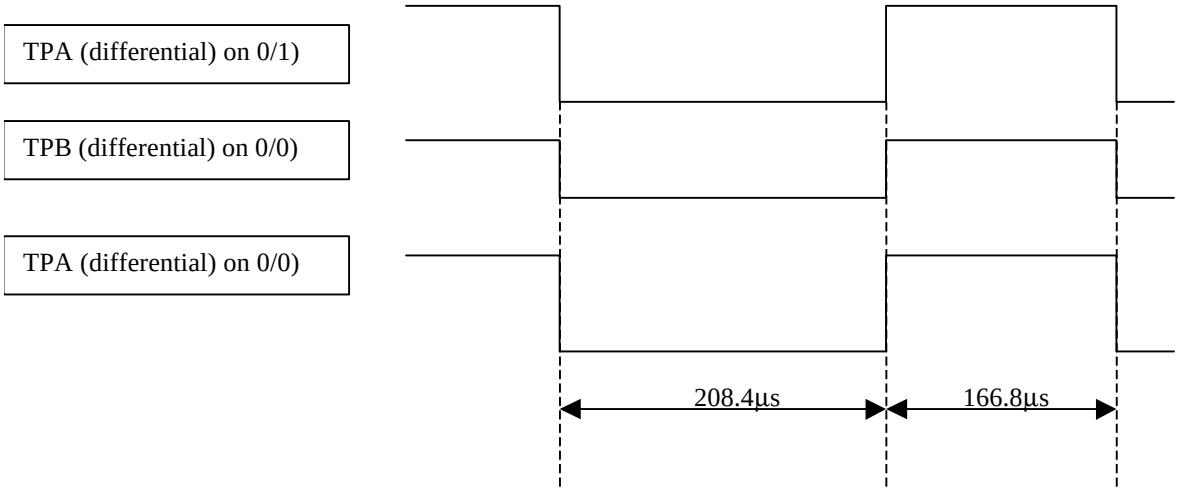
We only saw problems when making the physical connection of node 2, we did not see any problems when power cycling node 2. We may have been lucky. The connection did the expected thing (three self-IDs) about 19 times out of 20. The effects were apparently independent of cable length (we tried both very long and very short cables) and which actual ports we used.

The failure mode effects we observed were a continuous reset livelock. We saw two different versions.



Note, in this version we've lost 0.2ns in the measurements! It was also very difficult to see for the various "coincident" edges which actually came first, or from which end they were being driven.

The second version was simpler



This is as far as we got in the lab work before midnight struck and turned the coach back into a pumpkin. There's lots more we could look at.

3 Clues

3.1 Clue 1 - the numbers

Where do the numbers come from? 166.8µs is RESET_TIME (or possibly CONFIG_TIMEOUT), 208.4µs is MAX_ARB_STATE_TIME on the devices we were using, and we hypothesise that the 41.8µs is a wrap-around of MAX_ARB_STATE_TIME from the beginning of the pulse with a duration of RESET_TIME (i.e. MAX_ARB_STATE_TIME - RESET_TIME)

3.2 Clue 2 - some possible mis-behaviours

We think that the effects may be observed because one or more ports is "stuck" in resume actions at:-

```
while (((connect_timer < BIAS_HANDSHAKE) && !bias[i]) ||
bus_initialize_active)
; // Wait for peer PHY to generate TpBias
```

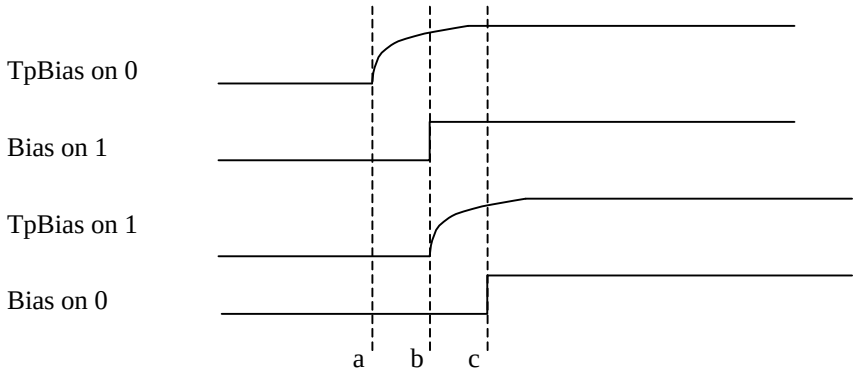
because `bus_initialize_active` is TRUE. In this state:-

1. The port can send and sense the reset arbitration state (possibly leading to reset_detected() going true)
2. The port is not examined when reset_complete() is evaluated.

If the peer port is the -95 PHY, on seeing incoming bias, it will start its tree_ID actions and get as far as T2: Parent_handshake. The local port, not being active, does not respond, and an arb_state_timeout occurs - leading to re-entry to R0 reset on both nodes.

This leaves open (for the moment) the question of why bus_initialize_active was true in the first place.

This can happen as well on the 1394a-1394a connection. Consider the sequence of events for resuming bias:-



If node 1 transitions to R0 between time b and time c, then it will generate BUS_RESET on the connection, as the qualification is:-

```
else if (bias[i] && resume[i])
portT(i, BUS_RESET); // Also propagate on resuming ports
```

and both terms will be true.

One suspicion is that some implementations of node 0 may indeed sense this before the bias handshake complete and bias is set true. In any case, they will sense the incoming reset as soon as bias is set true.

Thus, in `resume_actions()` on node 0, there is a race condition in the statement:-

```
while (((connect_timer < BIAS_HANDSHAKE) && !bias[i]) ||
                                        bus_initialize_active)
      ; // Wait for peer PHY to generate TpBias
```

`bus_initialize_active` will go at the same time as true `bias[i]` (possibly even earlier), and so the port state machine for node 0 will "hang". Node 1, meanwhile, will happily proceed past this statement, and, as `bus_initialize_active` is true, activate the port, proceed to Tree-ID, and then generate an arb state timeout as described above.

3.3 Clue 3 - blip in bias

This effect has been recently discussed. We've not (had the time to) confirm that this indeed is happening. But the effect of a blip in bias on either port 0/0 or 1/0 would be to put the connection into suspend, and, after some time, generate a bus reset on both nodes. The self_ID as reported on this bus reset would be an isolated node.

The blip in bias could be caused by a common mode ground bounce when the new connection is made. The boards we used did not have the diode protection scheme as recommended in Clause 7.3 and Annex C, and we understand that this also applies to the systems used in the original observations. Another possibility is that it is caused by failing to ignore the output of the bias comparator during speed signalling during Tree-ID.

3.4 Clue 4 - unwanted P4:P2 transition

Another hypothesis is that the port is seeing a false RX_SUSPEND (aka RX_ROOT_CONTENTION) during Tree-ID (see the long discussion on unwanted P4:P2 transitions).

3.5 Clue 5 - suspend initiator and target out of sync

If port 0/0 is a suspend initiator for lack of bias, then it waits a CONNTECT_TIMEOUT + BIAS_HANDSHAKE before withdrawing bias. Only when this happens will the peer port see loss of bias and move from the active state. During this time the ports are out of sync, and, if a reset occurs on node 1 the tree-ID hang-up can occur. Note, this has been corrected in 2.x.

3.6 Clue 6 - single connect timer

Note that there is a single connect timer for `connection_status()` and `resume_actions()` and `suspend_initiator_actions()`. So there is the possibility of the connect timer being reset during one of the several waits, and also there is the possibility of the connect timeout in `suspend_initiator_actions` popping at precisely the same moment as the connect timeout in connection_status (for a different port).

4 A scenario

We think that all these clues can add up to a scenario for the observed effects. Note, we are not claiming that this is what is happening - and we invite further detective work.

The sequence of events is along the following lines.

The connection 0/1 - 2/0 starts a connect timer in node 0, but also accidentally sets off a blip in bias which causes the 0/0 port to become a suspend initiator. The immediate bus reset on node 0 reports an isolated node. The bias filter on 0/1 detects bias and sets the bias variable to true.

Time passes

When the connect timer pops for connect_timeout, the 0/0 suspend initiator starts another timeout waiting for bias to go away (its still there). Meanwhile 0/1 looks connected and starts a resume_actions. However, suspend_in_progress is still true, so 0/1 waits for this to go false.

When the second 0/0 suspend initator timeout expires, it drives bias low and waits a bias handshake inside `activate_connect_detect`. During this time, 1/0 sees loss of bias and becomes a suspend initiator, and does its own bus reset, and then starts its own timeouts.

When 0/0 suspend actions finally compete, the port becomes suspended and `suspend_in_progress` becomes false.

So now resume actions for 0/1 can progress. First it generates bias and resumes 0/0. It waits for incoming bias or `bus_initialize_active` to go false. At that moment, bias is true, so it falls through, and waits for `bus_initialize_active to` go true. Node 2 will see the connection (incoming bias for a -95 node) and generate a long reset.

This will be detected on node 0 by `reset_detect()`.

Meanwhile, over on port 0/0, resume actions has started. It generates bias, redundantly sets `resume[1]` to true, and waits for incoming bias or `bus_initialize_active`. Which will happen first??? In general, the `bus_initialize_active` will happen first, as this depends on the response time of node 2 to generate a bus reset compared with the response time of node 1 to generate bias and the filter on node 0 to see it.

So 0/0 gets stuck at `bus_initialize_active`. In addition, the reset signal is propagated out on this port. So 1/0 will see the reset, and might also get stuck in `bus_initialize_active`, or might fall through into active.

If it falls through into active, then it will get as far as T2:parent handshake, timeout (as 0/0 is not active and so is not co-operating), and generate a reset. This will come soon after the reset wait time on node 2, and will be propagated to node 2. But the connection 0/1 - 2/0 will also have to wait for some time in root_contention, and this reset will probably hit at that time or soon after (preventing Tree_ID and self_ID from finishing). (I have not thought through the fine details of this, but I think it gives rise to the second set of timings given above).

5 Discussion

5.1 Blip in bias

This has been discussed elsewhere, and I'm not going to contribute here, beyond saying that I think it would be good to be resilient to it (the bus should recover).

5.2 Resuming a connection

I think that this is the heart of the problem. It seems to me that `resume_actions()` should really be split into two. The first half gets the connection going (bias in both directions), but, whilst it is so doing, there should be no reset signalling on the connection (can't be totally achieved, as -95 nodes do this - but at least the incoming reset can be ignored).

The second half brings the connection up at a safe time - i.e. at the start of a reset.

My proposal for doing this (and I welcome alternatives) is to have yet another per-port flag, which I'll call `reset_signals_OK[i]`. This is initially set to false, and is set to false on any exit from P4:active (possibly could be set false as we enter active).

This signal will be used instead of `resume[i]` in `reset_detected()` and `reset_start_actions()`. (The latter is to prevent the danger of propagating a reset on a resuming report until the bias handshake is fully complete).

It will also be tested in reset_complete() (i.e. the test is

```
if ((active[i] || reset_signals_OK[i]) &&
            (portR(i) != IDLE) && (portR(i) != RX_PARENT_NOTIFY))
return(FALSE);
```

(I think this is another failure mode - if the port is resuming, and about to enter into tree-ID, then exit from R1 should be deferred if the peer port is still generating reset).

The code for `resume_actions()` should be modified as follows:-

```
void resume_actions(int i) {
while (suspend_in_progress()) // Let any other suspensions complete
      ; // (we may resume those ports later)
connect_timer = 0;
connect_detect_valid[i] = FALSE; // Bias renders connect detect circuit
useless
tpBias(i, 1); // Generate TpBias
if (!resume[i] && !boundary_node) {
      for (j = 0; j < NPORT; j++)
      if (!active[j] && !disabled[j] && connected[j])
      resume[j] = TRUE; // Resume all suspended ports
} else
      resume[i] = TRUE; // Guarantee resume_in_progress() returns TRUE
while (((connect_timer < BIAS_HANDSHAKE) && !bias[i]) ||
bus_initialize_active)
      ; // Wait for peer PHY to generate TpBias and enough time for it to see
Bias
      // and for any on-going bus initialization to complete
resume_fault[i] = ~bias[i]; // Resume attempt failed if TpBias is absent
if (!resume_fault[i]) { // Bias present, connection restored to active state
      reset_signals_OK = TRUE; // Now safe to engage in reset signalling
      if ((int_enable[i] || resume_int) && !port_event) {
      port_event = TRUE;
      PH_EVENT.indication((lctrl && LPS) ? INTERRUPT : LINK_ON);
etc
```

5.3 Mura-San's comments

I think that much of the above is very closely related to Keiji Mura's comments (email of 30th August 1998), which I reproduce below for reference. I think that our two proposed solutions are very similar.

```
Can PHY exit properly from bus-reset when IBR set in the Resume state?
```

```
In P1394a draft2.0,Table7-25,reset_start_actions

else if (bias[i] && resume[i])
portT(i, BUS_RESET); // Also propagate on resuming ports

and

In draft2.0,Table7-25 (or 98-019r0.pdf,Table7-32),resume_actions
tpBias(i, 1); // Generate TpBias
if (resume[i] == 0 && !boundary_node)
   for (j = 0; j++; j < NPORT)
       if (!active[j] && !disabled[j] && connected[j])
          resume[j] = TRUE; // Resume all other suspended ports
else
   resume[i] = TRUE; // Guarantee resume_in_progress() returns TRUE
while (((connect_timer < BIAS_HANDSHAKE) && !bias[i]) ||
bus_initialize_active)
; // Wait for peer PHY to generate TpBias


In the case that PHY-A and PHY-B exists on the bus.
1:
If PHY-A sets IBR when bias[i] && resume[i]=TRUE before or during above
"while" state,PHY-A is waiting in the above "while" state until
!bus_initialize_active.
2:
According to reset_start_actions,PHY-A transmits BUS_RESET for
RESET_TIME,because bias[i] && resume[i]=TRUE.
3:
Resumed peer PHY-B receives the BUS_RESET and transmits BUS_RESET for
RESET_TIME.
4:
After transmission of BUS_RESET signal,PHY-A transitions to R1 state and soon
transitions to T0 state,because !active[i] and reset_complete=TRUE and PHY
doesn't detects reset in R0 and R1 states. Normally,if active[i]=true,PHY-A
waits in R1 until PHY-B transmits IDLE or RX_PARENT_NOTIFY after PHY-B exits
R0 state.
5:
PHY-B is transmitting BUS_RESET in R0 and PHY-A detects BUS_RESET signal in
T0,so PHY-A transitions to R0 state and start bus-reset again.
6:
PHY-B also transitions to T0 thorough R1 and receives BUS_RESET of PHY-A. PHY-
B acts as PHY-A acts at 1~5.
7:
bus_initilaize_active will not be set to FALSE ,so both PHYs can't set
active[i]=TRUE because PHY continues to wait in the above "while" state.
Consequently,PHY-A and PHY-B will continue to repeat R0->R1->T0->R0->...


<Proposal>
Resuming port should not participate the bus configuration until above "while"
sentence is finished. The following 3 modifications are required.

(1)
Add to Table 7-17
boolean busreset_mask[NPORT] ;            // busreset output and detect mask
signal when port resuming

Power reset value is FALSE.
```

(2)
Table 7-25 Bus reset actions and conditions

```
boolean reset_detect(){
....
  else if (active[i]){
      ....
  }
  else if (resume[i] and !busreset_mask[i]) {  <-modified part
            rest_time = (boundary_node) ? RESET_TIME : SHORT_RESET_TIME;
          return(TRUE);
  }
...
.....
}
```

(3)Draft2.0,Table7-25 (or 98-019r0.pdf,Table7-32),resume_actions

```
resume_actions(int i){
      busreset_mask[i] = TRUE ;           // busreset output and detect
disable until bus configuration complete <- modified part
      while(suspend_in_progress()) //
      ......

      .....
      else
            resume[i] = TRUE; //
      // wait for peer PHY to generate TpBias AND bus configuration complete!

      while ((( connect_timer < BIAS_HANDSHAKE ) && !bias[i] ||
bus_initialize_active) ;
      busreset_mask[i] = FALSE;           // bus configuration complete! this
port participate next bus configuration! <-modified part
    ...
}
```

Thank you for attention to the long-mail.
Any kinds of comments will be greatly appreciated.

K.Miura

****
Keiji Miura   E-Mail: kmiura@lsi.nec.co.jp
System ASIC Division   NEC