

Upstarts Draft 0.15 Review

Jerry Hauck

Zayante, Inc.

12/30/98

1 Editorial

- P126, L14: "signalling" -> "signaling"
- P126, L15: "This is used then the port ..." -> "This is used _when_ the port ..."
- P126, L18: "imple" -> "imply"
- P126, L38: "all speeds of packet" -> " all speeds of _packets_"
- P127, L6: "functions and variables" -> "functions, variables, and constants"
- P127, L9: "Functions and variables" -> "Functions, variables, and constants"
- P127, L51: Strike dangling "to connect."
- P128, L1: "Functions and variables" -> "Functions, variables, and constants"
- P128, L43: Move "16384" from Comment to Maximum column
- P130, P1:P5 Arc: "fault]" should be "fault"
- P130, P5:P0 Arc: Mislabeled ... "P5:P1" should be "P5:P0"
- P133, L12: "active{NPORT}" -> "active[_NPORT]"
- P133, L14: Strike declaration of Beta_mode, already declared on L12.
- P133, L31: "filter_bias" should be "bias_filter"
- P134, L17: "port was suspended 1394a" -> "port was _a_ suspended 1394a"
- P134, L36: activate_connect_detect is forward referenced.
- P137, L18: "{" should be "}"
- P137, L31-32: A closing brace "}" for the if clause started on L14 is required.
- P139, L18: "// now must to try" -> "// now must try"
- P140, L16: "{(" should be "{{"
- P140, L23: "alows" -> "allows"
- P141, L33: "ut" -> "us"
- General: Code is not consistent regarding declaration of loop variables. Some routines declare 'i' as an int, others have no declaration. For other loop variables such as count, declarations usually are present.

2 Missing/Undeclared Variables, Functions, Constants

2.1 Variables Missing from Table 11-2

Note: I'm not sure what criteria is used in determining which variables get described and which don't. So I'm just listing below the global variables which were declared/used but not listed in the table.

connect_timer
connect_detect
bias
suspend
resume
fault
sending_tone
received_speed
speed_ack
listening_for_speed

tried_bias
bias_delay_timer
toning
t_send_speed
sending_tone
disable_notify
signaled
t_speed
t_ack
bias_detect

2.2 Functions Missing from Table 11-2

Note: Again, these are the functions (non void routines) coded but not shown in the table.

suspend_in_progress()
resume_in_progress()
set_beta()

2.3 Constants Missing from Table 11-3

Note: These are constants I couldn't find in the draft anywhere.

BASE_RATE
BIAS_FILTER_TIME
CONNECT_TIMEOUT
Receive_OK_HANDSHAKE
RESET_DETECT

2.4 Variable Declarations missing from Table 11-5, Global Section

bias_detect

2.5 Variable Declarations missing from Table 11-5, Per Port Section

local_plug_present
signal_detect
toning
t_send_speed
sending_tone
disable_notify
signaled
t_speed
t_ack

2.6 Variable Declarations missing from connect_status

know_still_connected

3 Questions

- 1) P127, L23-26: Does cable_speed get reported in a register? Is this intended to be a sideband signal externally provided by the cable plug assembly or possibly strapped when the system is built?? I imagine it is "implementation dependant"?
- 2) P133, L8-16: Why are some binary variables declared as "int" and others as "boolean"? Am I missing something subtle?
- 3) P134, L37-38: Why is it necessary to wait for "a sufficiently long period" at the conclusion of a power-on reset? Assuming the power-on reset itself lasts a long period, why wait longer??
- 4) P134, L28-31: Should listening_for_speed be added to the power_reset sequence?

5) P139, L46: Why is the wait for ACTIVE_SAMPLE_INTERVAL required?

4 Potential Issues/Changes

4.1 Bias Filtering Code

4.1.1 Comment

P133, L21-27: The comment confuses me. I don't believe the code is optional anymore, particularly since the sending_tone qualification is done to make sure we ignore any bias indication while toning. I believe the comment should be removed. Any optionality should be explicitly coded as done elsewhere by testing Beta_mode_only. Since the remainder of the code ignores bias[i] on any port which is Beta_mode_only, the for loop could simply exit to the next iteration whenever the Beta_mode_only[i] is TRUE.

4.1.2 Sending_tone question

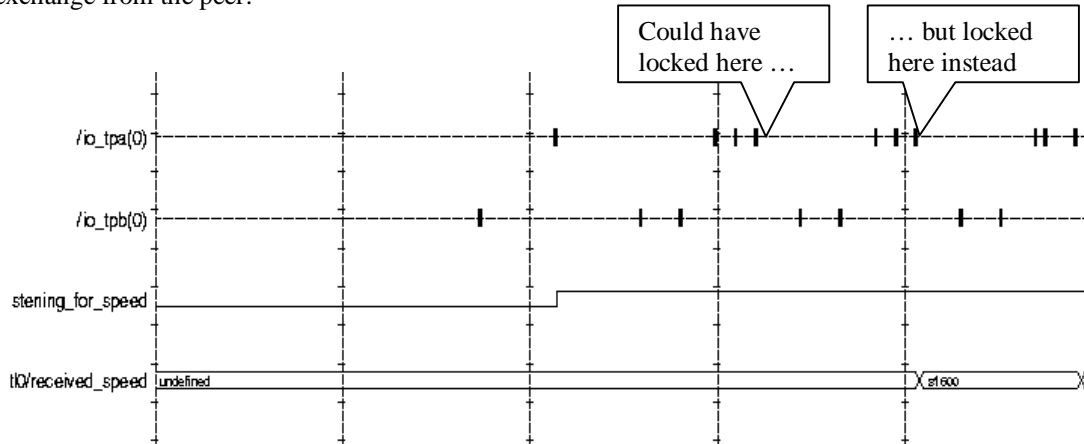
Why does bias_status return when sending_tone is TRUE? It seems this should be done on a port by port basis; i.e. sending_tone[i] should be tested. Was it intentional to abandon all bias processing if any given port was toning?? Also, if we are toning on a given port, should the bias_filter bit or the time interval be reset???

4.1.3 No initial condition for bias_filter and bias

The code doesn't seem to initialize bias[i] or bias_filter[i] with a power-on reset.

4.2 Receive Speed Indication

The following simulation output shows how the Draft 0.15 code is unable to lock onto the first speed signal exchange from the peer.



The inefficiency comes from the timeouts used to detect gaps and the subsequent time spent waiting for the start bit. Two cases are worth considering.

Case #1: Longest silent period between speed tones

Within a DISCONNECTED_TONE_INTERVAL, we have 16 TONE_INTERVALs which I'll number 0 through 15. Note that speed tones can only be sent in the first half of a DISCONNECTED_TONE_INTERVAL, i.e. intervals 0-7. Furthermore, interval 0 contains the start bit.

A TONE_INTERVAL consists of four periods of time of duration TONE_DURATION which I'll number as 0 through 3. Period 0 is the only one in which an actual tone may be sent. Periods 1 through 3 make up the SPEED_TONE_BIT_INTERVAL that separates consecutive bits in a speed signaling sequence.

The longest idle run between speed tones occurs when we send a start bit in interval 0 and a single speed bit in interval 7. The start bit ends at the end of period 0 in interval 0. The speed bit begins at the start of period 0 in interval 7. The total time between these events is 7 complete intervals less one tone period. Said differently, the silence period lasts for a total of 27 TONE_DURATIONS. Whatever code looks for the “real” gap after the speed tones must see silence for a period of time longer than 27 TONE_DURATIONS.

Case #2: Longest silent period between start bits

This is a bit more obvious. The beginning of start bits are separated by 16 TONE_INTERVALS or 64 TONE_DURATIONS. Therefore, the silent period between the end of a start bit and the beginning of a subsequent start bit is 63 TONE_DURATIONS. In order for the receive_speed_indication routine to always catch the first speed exchange, it should wait for a start bit for more than 63 TONE_DURATIONS since the last tone was heard.

Looking at Draft 0.15, we see that receive_speed_indication first looks for a tone at line 12 on page 136. This while loop finds the gap and will sample for a tone a maximum of 7 times (assuming silence). Since each pass of the loop consumes a single TONE_INTERVAL, the while loop executes for a maximum duration of 7 TONE_INTERVALS or 28 TONE_DURATIONS. However, note that the actual tone samples are taken at the top of each iteration through the loop. So the 7 samples are really spaced apart by 6 TONE_INTERVALS or 24 TONE_DURATIONS. Consequently, a gap can be falsely detected (remember Case #1 showed above that a valid sequence may have a gap of 27 TONE_DURATIONS).

The while loop on line 21 looks for the first start bit after the gap has been found. The loop executes for a maximum of 32 times, performing one tone sample per iteration and consuming one TONE_DURATION. Thus, 32 samples are taken 31 TONE_DURATIONS apart.

In the case of the very first speed exchange, the two while loops combine to search for the next start bit for up to $28+31 = 59$ TONE_DURATIONS after the solitary start bit occurred. Consequently, for the simulation example shown, receive_speed_indication times out before the 2nd start bit is found.

My proposed fix is to have both loops run with single iteration durations of TONE_DURATION (as opposed to TONE_INTERVAL). The first loop looks for a valid gap by making 29 samples (spaced 28 TONE_DURATIONS apart). This meets the 27 TONE_DURATION criteria of Case #1 and adds one to make up for PPM and sd_detected response differences.

The second loop awaits the next start bit by making 36 samples (spaced 35 TONE_DURATIONS apart). Combined with the first loop, the code makes 65 samples over a 64 TONE_DURATION period. This satisfies the 63 TONE_DURATION criteria of Case #2 and adds one to make up for PPM and sd_detected response differences.

Specific changes:

- Line 9 becomes “count = (TONE_GAP-1)*4; // the maximum gap (specified in samples separated by TONE_DURATION) that can occur within a valid speed signal including possible future codings)”
- Line 13 becomes “count = (TONE_GAP-1)*4”
- Line 15 is removed
- Line 17 becomes “count = (TONE_INTERVAL - TONE_GAP + 1)*4; // the maximum time (specified in samples separated by TONE_DURATION) for a start bit to appear after a previous gap was detected

4.3 Set Beta

4.3.1 Extra speed signal at end of handshake

The next simulation output shows an unnecessary speed signal at the end of a beta speed negotiation handshake.

The proposed fix is to make the speed transmitter autonomous just like the speed receiver. Then, the set_beta routine simply set both in motion and monitor for events signaled by both. This gives the lowest latency response and minimizes the speed negotiation handshake.

Specific suggested changes follow. Please forgive my weak C coding skills.

- Remove the send_speed routine altogether
- Replace the toner() routine with the following:

```
void toner() { // continuously running
    boolean t_send_speed, t_ack;
    int t_speed;
    while (toning || send_speed) {
        t_send_speed = send_speed; // Save consistent copies of speed indications
        t_ack = we_agree; // which can be asynchronously updated
        t_speed = port_speed;
        send_tone(); // Send the start bit
        wait(SPEEDTONE_BIT_INTERVAL);
        if (t_send_speed) {
            if (t_ack) send_tone() else wait(TONE_DURATION);
            for (i = 0; i++; i < 3) { // send three bits or spaces
                wait(SPEEDTONE_BIT_INTERVAL);
                if ((t_speed & (1 << i)) != 0) send_tone() else wait(TONE_DURATION);
            }
            signal(EVT_SENT_SPEED);
            if (t_ack) signal(EVT_SENT_ACK);
            wait(DISCONNECTED_TONE_INTERVAL -
                4*SPEEDTONE_BIT_INTERVAL - 5*TONE_DURATION);
        } else wait(DISCONNECTED_TONE_INTERVAL - TONE_DURATION - SPEEDTONE_BIT_INTERVAL);
    }
}
```

- P136, L31: Immediately after the line in which received_speed is updated, add “signal(EVT_RECEIVED_SPEED);”

- Replace the set_beta() routine with the following:

```

boolean set_beta() { // set beta mode and exchange speed signals to establish operating
                    // speed, returns FALSE if the speed negotiation failed
    int speed_agreed, we_agree, sent_ack, count, event;
    port_speed = (cable_speed < max_port_speed) ? cable_speed : max_port_speed;
                    // our starting point
    if (port_speed < min_port_speed) return(FALSE);
    count = 10; // five tries
    speed_agreed = we_agree = sent_ack = FALSE;
    send_speed = TRUE; // Start the autonomous speed sender going with the
                    // updated port_speed and the ack bit initialized
                    // to FALSE
    listening_for_speed = TRUE; // set the autonomous speed listener going;
    while (((count > 0) & !speed_agreed) | (speed_agreed & !sent_ack)) {
        event = wait_event(EVT_SENT_SPEED | EVT_RECEIVED_SPEED | EVT_SENT_ACK);
        if (event & EVT_RECEIVED_SPEED) {
            if (received_speed != 0) { // note, may not be a new indication
                if (received_speed < min_port_speed) {
                    listening_for_speed = FALSE;
                    send_speed = FALSE;
                    return(FALSE);
                }
            }
            if (received_speed == port_speed) {
                we_agree = TRUE;
                speed_agreed = speed_ack; // all agreed when we have the
                // acknowledge from the other end
            } else {
                if (received_speed < port_speed) {
                    // if the received speed is unacceptable,
                    // then we could add code here to give up altogether
                    port_speed = received_speed;
                    we_agree = TRUE;
                    count = 10; // and try again with a lower speed
                } else count++; // received speed > port_speed, so allow another go,
                // but not too many times
            }
        }
        if (event & EVT_SENT_SPEED) count = count - 2;
        if (event & EVT_SENT_ACK) sent_ack = TRUE;
    }
    listening_for_speed = FALSE; // turn off the autonomous listner (may take some time)
    send_speed = FALSE; // turn off sending of speed signal (may take some time)
    if (speed_agreed) {
        toning = FALSE;
        Beta_mode = TRUE;
        connected = TRUE;
        receive_OK = TRUE;
    }
    return (speed_agreed);
}

```

The following simulation output shows a normal handshake once the above changes are made.



One possible improvement is still noted. Node C (which is toning into Node's B tpa) had plenty of time to recognize Node B's S800 speed signal before sending it's first speed exchange. However, the connection status code as written looks for a tone before enabling the speed receiver. In the above simulation, the

receiver isn't enabled until after the start bit of the first S800 signal is seen. Consequently, it takes Node C one more loop to lock onto the next start bit, see the S800 code, and change it's own speed to S800.