

10. Beta mode port specification

What's changed?

0.071 to 0.08:

1) Mapped control state 1101 (previously spare) to SPEEDa and 0100 to SPEEDb- i.e. SPEEDb now has a positive and negative disparity variants. This allows S800 and faster packets to have no DATA_PREFIX, just a SPEED signal, and still have the start of packet disparity indicated.

2) Shorter packet format defined for S100-S400 packets in a beta only environment - this addition is provisional.

3) Embedded control symbols are now padded by repeating same control symbol, unlike data which is padded by following SPEEDa symbols. This is to allow same c-code for packet prefix control requests and embedded control requests, and to prevent confusion when a control symbol is embedded in a null packet (i.e. packet >port speed, represented by a stream of data Prefix- don't want any SPEED symbols occurring which could be mistaken for a new packet).

4) Removed state machines as these did not seem necessary.

5) added some outline text for loopback and other test modes

6) Changed speed signal format to allow faster decode

7) changed data padding format to byte padding, data byte first followed by padding bytes

0.08 to present draft

1) Added scrambler disable test mode

2) added request type signaling to support BOSS scheme

3) major revisions to c-code

4) added synchronization procedures

5) removed all reference to embedded arbitration signals

significant changes since April meeting in Newport Beach:

1) removed all service primitives

2) clarified packet formats

3) added scrambler and descrambler update functions

4) clarified disparity calculation in event of errors

changes since June meeting in St Petersburg - as indicated by change bars....

changes August 98

1) removed disparity alternative for SPEEDb speed code since with changes to packet format no packet format exists without a data prefix at start.

2) changed definition of packet format to include compulsory data prefix for all packet types.

3) Extensive updating of c-code to conform to acceleration/arbitration state machine interface.

4) Changed definition of receiver such that all components of received packets are placed in a FIFO i.e. bportR is a FIFO.

This section specifies the functions and operation of the P1394B beta mode port. These functions include the scrambling and coding layer for the physical layer. The standard IEEE Std. 1394-1995 Data/Strobe mode of symbol coding is not covered.

10.1 Overview

The relationship and interfaces between the beta mode port and other P1394B functions and layers is illustrated in figure 10-1.

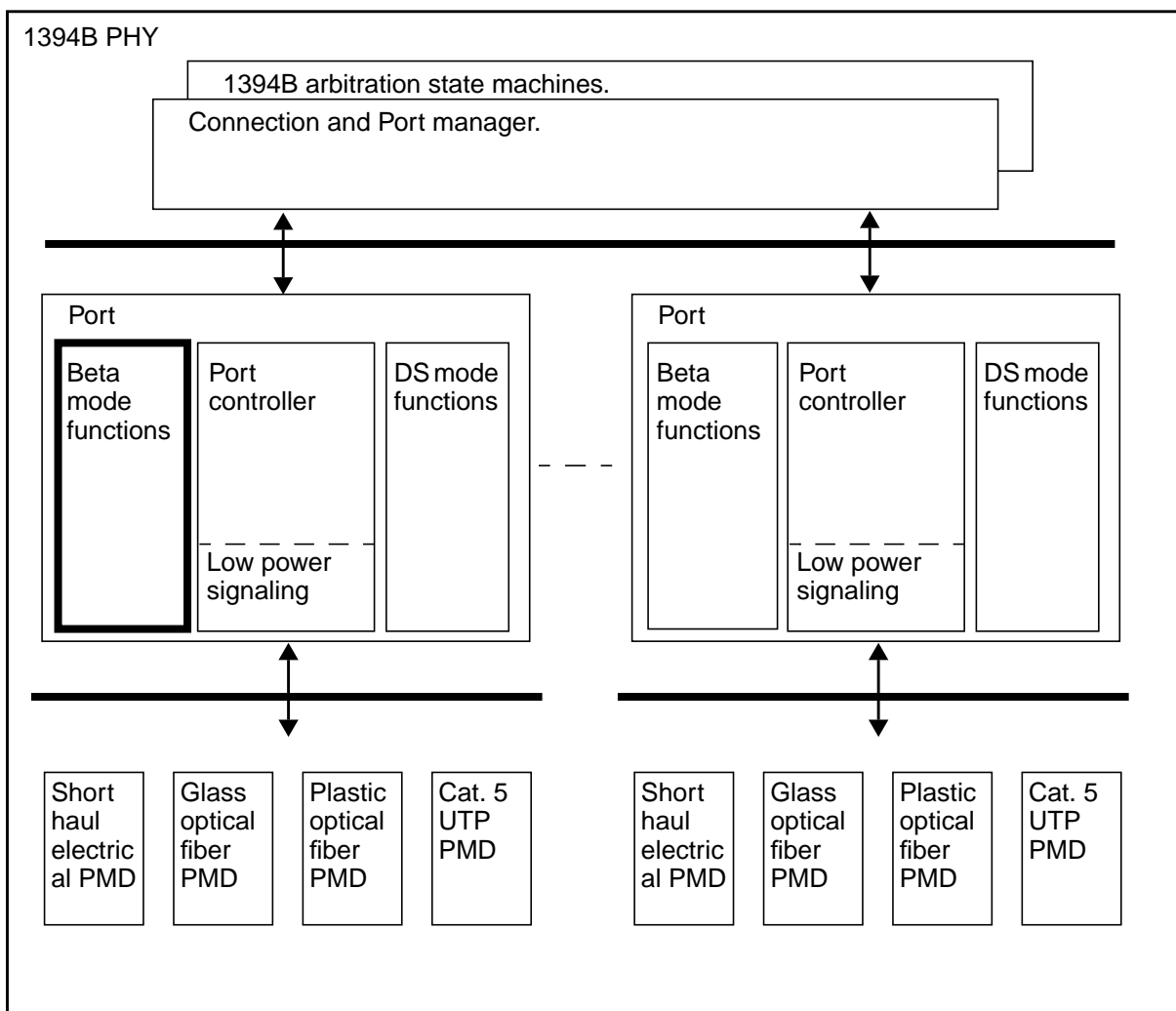


Figure 10-1—Beta mode port interfaces

1 **10.2 Definitions**
2
3

4 [tbd]
5

6 **10.3 Port functions**
7
8

9 When operating in beta mode a P1394B port scrambles and block codes all data and control signals prior to transmitting
10 them to the PMD layer. This section defines the scrambling and coding functions of a P1394B beta mode port, along with
11 decoding and descrambling.
12

13 **10.3.1 Port functions overview**
14
15

16 The port provides transmit and receive functions for forming data as it passes between the port manager and the PMD.
17 In the transmitting direction, data and control signals are both scrambled and mapped to 10-bit characters which are
18 passed to the PMD layer, as illustrated in figure 10-2. Transitions between data and control signals occur at the boundaries
19 of these 10-bit characters. Two distinct codetables are defined for mapping signals to 10-bit characters. These two codeta-
20 bles are also used for decoding signals received from the PMD layer.
21

22 Data are scrambled and then coded using an 8B/10B block code. The rate at which bits are passed to the PMD layer is
23 therefore 1.25 times the data rate.
24

25 Control information contained within packet delimiters is mapped to a four bit control symbol and then also scrambled.
26 The scrambled control symbol is coded using 10-bit characters that are unique with respect to the 10-bit characters used
27 for data. The time taken for a single 4-bit control symbol to be transmitted is therefore equivalent to the time taken for a
28 single byte of data to be transmitted.
29

30 Other control information is signaled as request types. These are mapped to six active bits of a data byte and then scram-
31 bled and coded in a similar way to data. The two inactive bits are set to zero so that the request types only use a subset of
32 the data 8B10B characters.
33

34 Similar, but reciprocal, functions are provided by the port in the receiving direction.
35

36 **10.3.2 Naming conventions**
37
38

39 P1394b adopts the following definitions and conventions in defining data scrambling and coding:
40
41

42 An unscrambled and uncoded data byte contains 8 information bits (1 byte):

43 H,G,F,E,D,C,B,A;
44

45 Bits H-through-A are the most- through least-significant bits of the byte
46

47 A scrambled and uncoded data byte has 8 information bits (1 byte):

48 H',G',F',E',D',C',B',A';
49

50 Bits H'-through-A' are the most- through least-significant bits of the byte
51

52 The scrambled and coded data character has 10 information bits:

53 a, b, c, d, e, i, f, g, h, and j (note: not strictly alphabetical ordering)
54

55 Bits are transmitted as listed above (bits a-through-j are transmitted first-through-last)
56

57 P1394b adopts the following definitions and conventions in defining control scrambling and coding:
58
59

60 An unscrambled and uncoded control symbol contains 4 information bits:

61 S,R,Q,P;
62

63 Bits S-through-P are the most- through least-significant bits
64
65
66

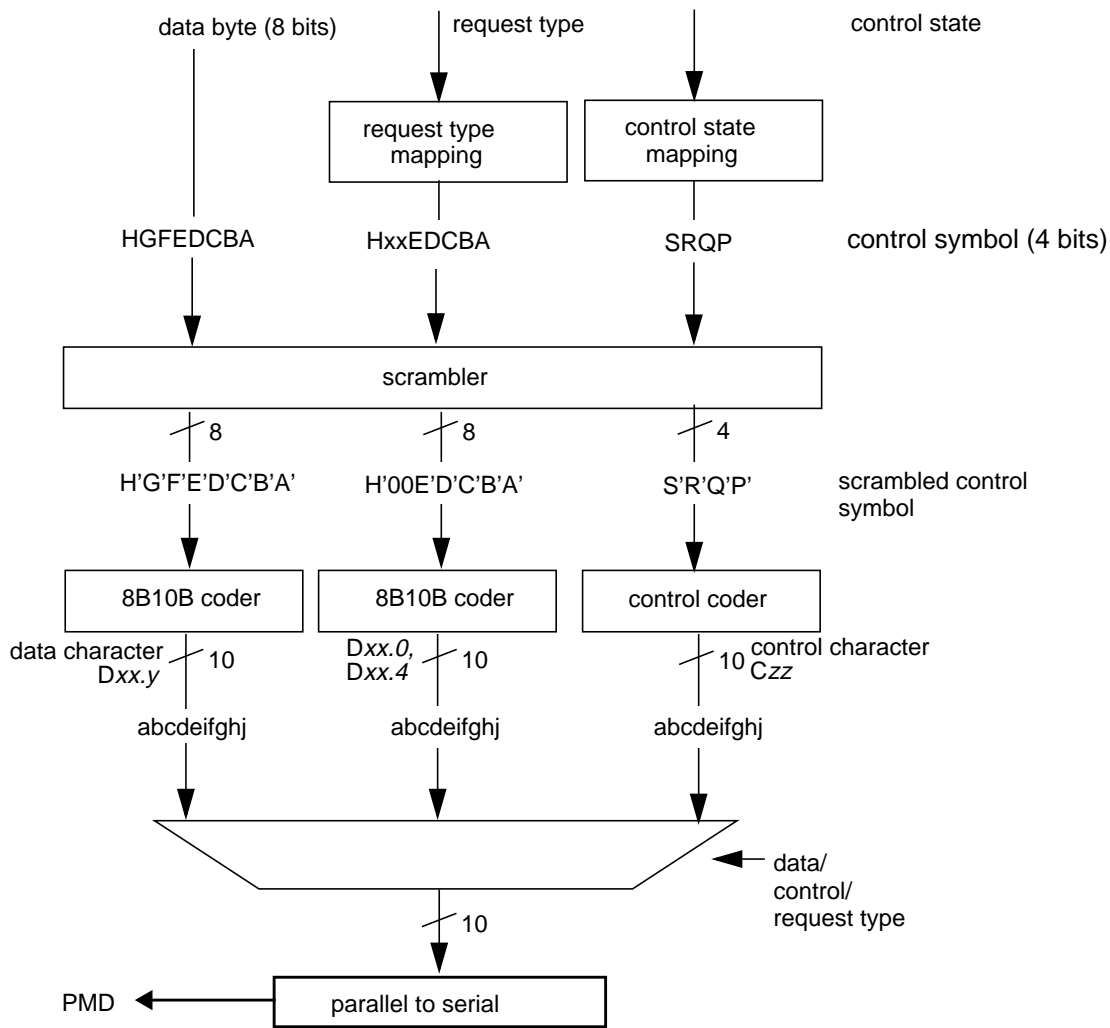


Figure 10-2—Scrambling and coding functions (Note: this figure is informative only)

A scrambled and uncoded control symbol has 4 information bits:

S',R',Q',P';

Bits S'-through-P' are the most- through least-significant bits

The scrambled and coded control character has 10 information bits:

a, b, c, d, e, i, f, g, h, and j (note: not strictly alphabetical ordering)

Bits are transmitted as listed above (bits a-through-j are transmitted first-through-last)

10.3.3 Running disparity

Throughout this clause reference is made to the running disparity of a bitstream. The running disparity is defined as the total number of 1 symbols in a bitstream minus the total number of zero symbols in that bitstream. This quantity is sometimes referred to as the running digital sum, and the abbreviations rd and rds are used interchangeably to denote the running disparity.

For the purpose of this standard, the rd shall be initially set to a value of -1. Subsequent updating of the running disparity is made at the end of 10-bit characters. In the absence of errors the rd will always take a value of +1 or -1 at the end of a 10-bit character. Should errors cause the absolute value of the rd to exceed 1 at the end of a 10-bit character, the absolute value of the rd shall be rounded down to 1 while the sign is preserved.

10.3.4 Control State Mapping

P1394b control states shall be mapped to four bit control symbols according to table 10-1.

Table 10-1—Control State Mapping

Control state	Control symbol SRQP	
	rd>0	rd<0
ARBRST_GRANT	0000	
GRANT	0001	
spare	0010	
spare	0011	
spare	0100	
SPEEDa	1101	
SPEEDb	1000	
DATA_PREFIX	1001	0101
DATA_END	1010	0110
DATA_END_ERR (Errored packet)	0111	1011
spare	1100	
ARBRST	1110	
BUS_RESET	1111	

NOTE—Pairs of control symbols representing positive and negative rds variants of the same control state (e.g. DATA_END+ and DATA_END-) have identical values for bits P and Q and complementary values for bits R and S.

10.3.5 Request Type Mapping

Asynchronous and isochronous requests are combined and mapped to a single request type symbol according to the following tables. P1394b asynchronous request types shall be mapped to a symbol according to table 10-2.

Table 10-2—Asynchronous request type mapping

Request type	symbol component bits CBA
reserved	000
NONE	001
NEXT_SLOW	010
NEXT_FAST	011
CURR_SLOW	100
CURR_FAST	101
LEGACY	110
CYCLE_START	111

P1394b isochronous request types shall be mapped to a symbol according to table 10-4.

Table 10-3—Isochronous request type mapping

Request type	symbol component bits HGFED
reserved - see Note 2	0xx00
NONE	0xx01
NEXT_SLOW	0xx10
NEXT_FAST	0xx11
CURR_SLOW	1xx00
CURR_FAST	1xx01
LEGACY	1xx10
reserved	1xx11

NOTE 1—‘x’ indicates ‘Don’t Care’. These two bits are never used to carry request type information. However, they have been included in the definition of the request type mapping, as the eight bit symbol is used as an input to the data scrambler and coder.

NOTE 2—The component bits 0xx00 must not be used for isochronous request types as the resulting symbol would be indistinguishable from a legacy request type. HGFED=0xx00 indicates that a request type is a legacy request.

P1394b legacy request types shall be mapped to a symbol according to table 10-4.

Table 10-4—Legacy request type mapping

Request type	symbol HGFEDCBA
REQGNT	0xx00001
CHILD_NOTIFY	0xx00010
PARENT_NOTIFY	0xx00011
DISABLE_NOTIFY	0xx00100
SUSPEND	0xx00101
IDLE	0xx00111
TRAINING	0xx00000
OPERATION	0xx00110

NOTE 1—‘x’ indicates ‘Don’t Care’. These two bits are never used to carry request type information. However, they have been included in the definition of the request type mapping, as the eight bit symbol is used as an input to the data scrambler and coder.

10.3.5.1 Request type mapping example (informative)

The following example illustrates the mapping of asynchronous and isochronous requests as indicated to the port through the bportT parameter.

```
bportT.tag == REQUEST
request_field = bportT.req
request_field.isoch == NEXT_SLOW
request_field.asynch == CYCLE_START
symbol HGFEDCBA = 0xx10111
```

10.3.6 Scrambling

P1394b employs a side stream scrambler. The same scrambler is used to scramble both data and control information. The scrambler uses the generating polynomial:

1 $G(x) = x^{11} + x^9 + 1$
2

3 The scrambler shall generate a continuous stream of output bits at the same rate as the port operating speed during both
4 packet and control transmission. This document represents these output bits as a sequence of bits Scr(k) where k=0, 1, 2,
5 3....., such that Scr(k+1) is the output bit immediately following Scr(k).
6

7
8 The scrambler output at any point in time is defined as:
9

10 $Scr(k) = Scr(k-9) \text{ XOR } Scr(k-11)$
11
12
13

14 **10.3.6.1 Data scrambling**

15
16
17 During packet transmission the data are scrambled using the output of the scrambler; the scrambler generates eight bits
18 for each byte of data to be transmitted. The scrambled data stream is the result of an exclusive OR operation on the data
19 stream and the scrambler output. Any data byte immediately following a data byte (i.e. with no intermediate padding sym-
20 bols) shall be scrambled according to the rule:
21
22

23
24 $[H',G',F',E',D',C',B',A'] = \text{XOR}([H,G,F,E,D,C,B,A], [Scr(k:k+7)])$
25

26 where Scr(k-1) is the scrambler output bit used to scramble the lsb of the immediately preceding data byte. For example,
27 during a sequence of data, the scrambled data bytes would be calculated as follows:
28

29
30 first scrambled data byte = $\text{XOR}([H,G,F,E,D,C,B,A], [Scr(k:k+7)])$

31 second scrambled data byte = $\text{XOR}([H,G,F,E,D,C,B,A], [Scr(k+8:k+15)])$ etc.
32

33 Any data byte immediately following a control symbol shall be scrambled according to the rule:

34
35 $[H',G',F',E',D',C',B',A'] = \text{XOR}([H,G,F,E,D,C,B,A], [Scr(k:k+7)])$
36

37 where Scr(k-2) is the scrambler output bit used to scramble the lsb of control symbol immediately preceding the sequence
38 of data bytes. i.e. the most significant bit of the data byte, H, is exclusive OR'ed with the Scr(k) and the least significant
39 bit of the data byte, A, is exclusive OR'ed with the Scr(k+7).
40

41
42 NOTE—The preceding control symbol may be part of the packet prefix or a padding symbol.
43

44 The scrambling procedure for data bytes is illustrated in figure 10-3.
45

46 **10.3.6.2 Request type scrambling**

47
48 Request types are scrambled in the same way as data bytes. However, bits F' and G' of the scrambled request type are
49 subsequently set to zero. This has the effect of limiting the set of data characters used for the coding of the request types
50 to a subset containing only Dx.0 and Dx.4 characters. This subset of data characters has the property that all characters
51 within the set are separated from each other by a Hamming distance of two. Scrambling and coding request types in this
52 way ensures that a single error in a request type character will always be detected by a receiver.
53
54

55 Any request type immediately following another request type (i.e. within a stream of request types) shall be scrambled
56 according to the rule:
57

58
59 $[H',G',F',E',D',C',B',A'] = (\text{XOR}([H,G,F,E,D,C,B,A], [Scr(k:k+7)])) \text{ AND } (10011111)$
60

61 where Scr(k-1) is the scrambler output bit used to scramble the lsb of the immediately preceding request type.
62

63 Any request type immediately following a control symbol shall be scrambled according to the rule:
64
65
66

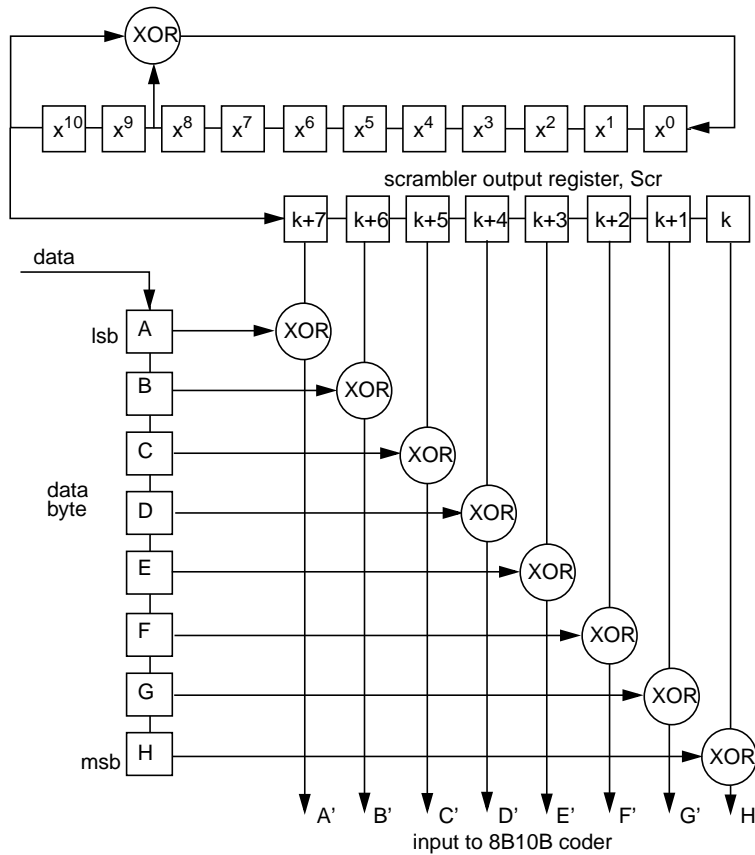


Figure 10-3—Scrambler schematic (data)

$$[H', G', F', E', D', C', B', A'] = (\text{XOR}([H, G, F, E, D, C, B, A], [\text{Scr}(k:k+7)])) \text{ AND } (10011111)$$

where $\text{Scr}(k-2)$ is the scrambler output bit used to scramble the lsb of the control symbol immediately preceding the request type i.e. the most significant bit of the request type, H, is exclusive OR'ed with the $\text{Scr}(k)$ and the least significant bit of the request type, A, is exclusive OR'ed with the $\text{Scr}(k+7)$.

The scrambling procedure for request types is illustrated in figure 10-4.

10.3.6.3 Control symbol scrambling

The scrambled control symbol shall be the result of an exclusive OR operation on the control symbol and alternate bits of the scrambler generating polynomial. When a control symbol immediately follows a data byte, the control symbol shall be scrambled according to the rule:

$$[S', R', Q', P'] = \text{XOR}([S, R, Q, P], [\text{Scr}(k), \text{Scr}(k+2), \text{Scr}(k+4), \text{Scr}(k+6)])$$

where $\text{Scr}(k-1)$ is the scrambler output bit used to scramble the lsb of the data byte immediately preceding the data byte.

When a control symbol immediately follows another control symbol, the control symbol shall be scrambled according to the rule:

$$[S', R', Q', P'] = \text{XOR}([S, R, Q, P], [\text{Scr}(k), \text{Scr}(k+2), \text{Scr}(k+4), \text{Scr}(k+6)])$$

where $\text{Scr}(k-2)$ is the scrambler output bit used to scramble the lsb of the preceding control symbol.

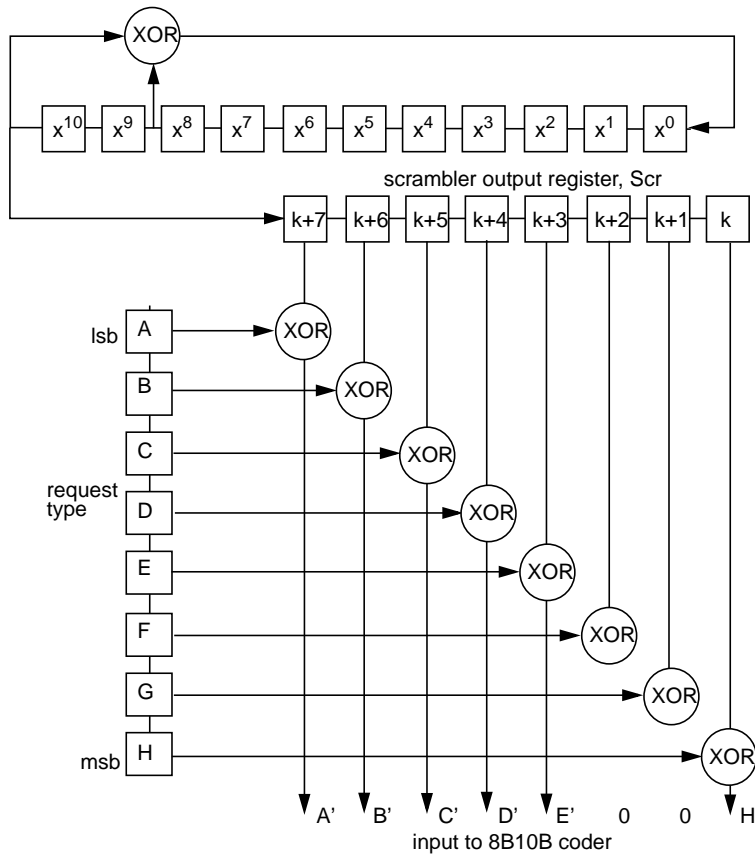


Figure 10-4—Scrambler schematic (request type)

The scrambling procedure for control symbols is illustrated in figure 10-5.

10.3.7 Coding

The symbol coding function defines and implements the character coding scheme employed by P1394b. Uncoded data bytes, request types and control symbols are encoded into 10-bit coded characters for transmission by the PMD. Similarly, coded characters are received from the physical layer, decoded, and passed to port manager as uncoded bytes, request types or control symbols. There is a small degree of error detection built into this coding method but the main advantage is the lack of DC frequency content in the electrical signals produced.

1394b uses two distinct codes. The first code maps data and request types to a set of 10 bit characters. The second code maps control symbols to a second, smaller and distinct set of 10 bit characters.

10.3.7.1 8B/10B character coding for data and request types

To provide a DC balanced code with minimal run lengths, P1394b adopts the 8B/10B Transmission Code of Widmer and Franaszek¹ for coding data and request types.

This code provides for 256 input symbols and produces sequences of 10-bit coded characters with guaranteed AC transitions and no DC frequency component. The following subsections describe the 8B/10B code features and documents the method of code construction as well as the resulting code tables.

¹ A. X. Widmer and P. A. Franaszek, *IBM J. Res. Develop.*, Vol. 27, No. 5, p.440, September 1983

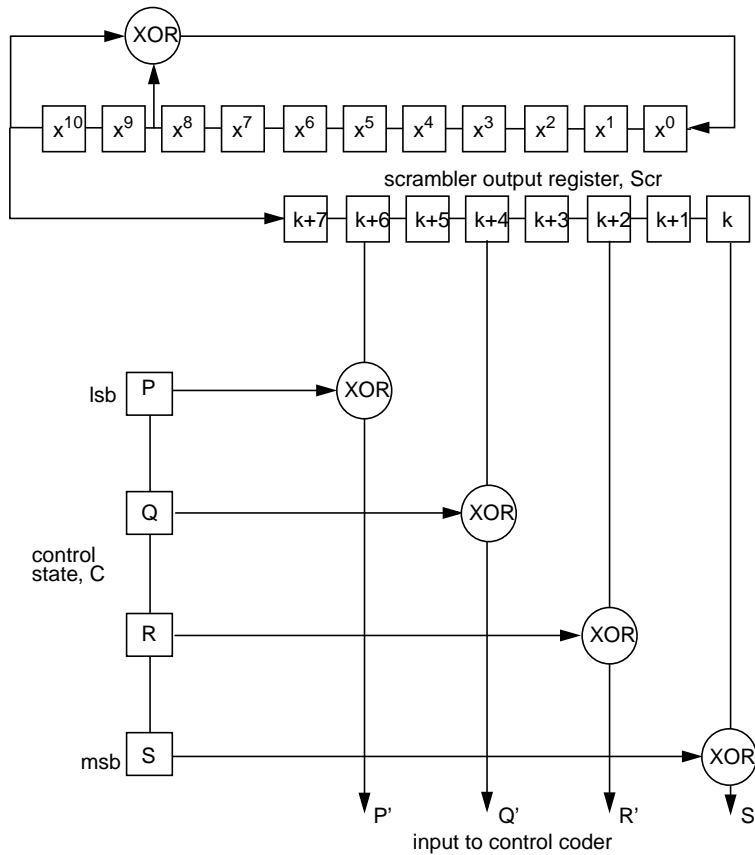


Figure 10-5—Scrambler schematic (control)

P1394b adopts the following definitions and conventions in defining the 8B/10B coding rules:

Each valid coded character has a name

D $_{xx.y}$ for data characters

K $_{xx.y}$ for special characters

xx = the 5 bit input value, base 10, for bits ABCDE

y = the 3 bit input value, base 10, for bits FGH

In the following tables,

rd represents the running disparity value from previous bit transmissions

rd1 represents the running disparity value after these bits have been transmitted

10.3.7.1.1 8B/10B properties - run length and DC balance

A sequence of valid 8B/10B coded characters has a maximum run length of 5 bits (i.e., 5 consecutive ones or 5 consecutive zeros before a mandatory bit transition). Consequently, ample transitions are provided to aid in clock extraction by the physical layer receiving PLL's.

1 Each valid 8B/10B coded character also has either zero digit disparity (i.e., same number of 1's and 0's in the character)
2 or has 2disparity. To maintain DC balance, the coding protocol demands that each character have opposite disparity from
3 the preceding character or zero disparity. Consequently, each non-zero disparity code character has an alternative coding
4 with reversed disparity. The coder produces the correct alternating disparities by means of a running disparity counter
5 which controls which alternate output character is produced.
6

7
8 The transmitter initializes the running disparity to -1 after power-on reset or after exiting specific special diagnostic
9 modes. Subsequent to this, the mechanism of alternating output characters ensures that the running disparity at the end of
10 any code character is either -1 or +1. More generally, the running disparity after any bit in a sequence of valid 10-bit char-
11 acters is constrained to be between -3 and +3 inclusive.
12

13 **10.3.7.1.2 8B/10B code construction**

14
15
16 The 8B/10B coding is defined in two stages. The first stage codes the first 5 bits of the uncoded input byte into a 6 bit
17 code. This 5B/6B coder produces the 6 bit result depending upon the input bit values, the state of the control bit, and the
18 running disparity value. The second stage codes the last 3 bits of the uncoded byte into a 4 bit code using a 3B/4B coder.
19 The running disparity value, as modified by the first coding stage, is also used for coding decisions. Table 10-5 and
20 table 10-6 give the 5B/6B and 3B/4B coding tables, respectively, that are used for coding Dxx.y characters.
21
22
23
24
25
26
27
28
29
30
31
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

Table 10-5—5B/6B Coding

inputs		abcdei outputs		rd1	inputs		abcdei outputs		rd1
Symbol	A'B'C'D'E'	rd>0	rd<0		Symbol	A'B'C'D'E'	rd>0	rd<0	
D0	00000	011000	100111	-rd	D16	00001	100100	011011	-rd
D1	10000	100010	011101	-rd	D17	10001	100011		rd
D2	01000	010010	101101		D18	01001	010011		
D3	11000	110001		rd	D19	11001	110010		
D4	00100	001010	110101	-rd	D20	00101	001011		
D5	10100	101001		rd	D21	10101	101010		
D6	01100	011001		rd	D22	01101	011010		
D7	11100	000111	111000		D23	11101	000101	111010	-rd
D8	00010	000110	111001	-rd	D24	00011	001100	110011	rd
D9	10010	100101		rd	D25	10011	100110		
D10	01010	010101		rd	D26	01011	010110		
D11	11010	110100			D27	11011	001001	110110	-rd
D12	00110	001101		rd	D28	00111	001110		
D13	10110	101100			D29	10111	010001	101110	-rd
D14	01110	011100		-rd	D30	01111	100001	011110	
D15	11110	101000	010111		D31	11111	010100	101011	

Table 10-6—3B/4B Coding

Inputs		fghj outputs		rd1
Symbol	F'G'H'	rd>0	rd<0	
Dx.0	000	0100	1011	-rd
Dx.1	100	1001		rd
Dx.2	010	0101		
Dx.3	110	0011	1100	rd
Dx.4	001	0010	1101	-rd
Dx.5	101	1010		rd
Dx.6	011	0110		
Dx.P7	111	0001	1110	-rd
Dx.A7	111	1000	0111	

Notes:
A7 replaces P7 if the following is true:
(rd<0) ? (e==1 && i==1) : (e==0 && i==0)

10.3.7.1.3 8B/10B valid data characters

Table 7-3 lists all of the valid data characters generated by passing the 256 possible input data bytes through the 5B/6B and 3B/4B coders.

Table 10-7—Valid Data (K=0) Characters

Valid Data Characters (Page 1 of 3)							
input		abcdei fghj output		input		abcdei fghj output	
Name	H'G'F'E'D'C'B'A'	rd<0	rd>0	Name	H'G'F'E'D'C'B'A'	rd<0	rd>0
D0.0	000 00000	100111 0100	011000 1011	D16.1	001 10000	011011 1001	100100 1001
D1.0	000 00001	011101 0100	100010 1011	D17.1	001 10001	100011 1001	100011 1001
D2.0	000 00010	101101 0100	010010 1011	D18.1	001 10010	010011 1001	010011 1001
D3.0	000 00011	110001 1011	110001 0100	D19.1	001 10011	110010 1001	110010 1001
D4.0	000 00100	110101 0100	001010 1011	D20.1	001 10100	001011 1001	001011 1001
D5.0	000 00101	101001 1011	101001 0100	D21.1	001 10101	101010 1001	101010 1001
D6.0	000 00110	011001 1011	011001 0100	D22.1	001 10110	011010 1001	011010 1001
D7.0	000 00111	111000 1011	000111 0100	D23.1	001 10111	111010 1001	000101 1001
D8.0	000 01000	111001 0100	000110 1011	D24.1	001 11000	110011 1001	001100 1001
D9.0	000 01001	100101 1011	100101 0100	D25.1	001 11001	100110 1001	100110 1001
D10.0	000 01010	010101 1011	010101 0100	D26.1	001 11010	010110 1001	010110 1001
D11.0	000 01011	110100 1011	110100 0100	D27.1	001 11011	110110 1001	001001 1001
D12.0	000 01100	001101 1011	001101 0100	D28.1	001 11100	001110 1001	001110 1001
D13.0	000 01101	101100 1011	101100 0100	D29.1	001 11101	101110 1001	010001 1001
D14.0	000 01110	011100 1011	011100 0100	D30.1	001 11110	011110 1001	100001 1001
D15.0	000 01111	010111 0100	101000 1011	D31.1	001 11111	101011 1001	010100 1001
D16.0	000 10000	011011 0100	100100 1011	D0.2	010 00000	100111 0101	011000 0101
D17.0	000 10001	100011 1011	100011 0100	D1.2	010 00001	011101 0101	100010 0101
D18.0	000 10010	010011 1011	010011 0100	D2.2	010 00010	101101 0101	010010 0101
D19.0	000 10011	110010 1011	110010 0100	D3.2	010 00011	110001 0101	110001 0101
D20.0	000 10100	001011 1011	001011 0100	D4.2	010 00100	110101 0101	001010 0101
D21.0	000 10101	101010 1011	101010 0100	D5.2	010 00101	101001 0101	101001 0101
D22.0	000 10110	011010 1011	011010 0100	D6.2	010 00110	011001 0101	011001 0101
D23.0	000 10111	111010 0100	000101 1011	D7.2	010 00111	111000 0101	000111 0101
D24.0	000 11000	110011 0100	001100 1011	D8.2	010 01000	111001 0101	000110 0101
D25.0	000 11001	100110 1011	100110 0100	D9.2	010 01001	100101 0101	100101 0101
D26.0	000 11010	010110 1011	010110 0100	D10.2	010 01010	010101 0101	010101 0101
D27.0	000 11011	110110 0100	001001 1011	D11.2	010 01011	110100 0101	110100 0101
D28.0	000 11100	001110 1011	001110 0100	D12.2	010 01100	001101 0101	001101 0101
D29.0	000 11101	101110 0100	010001 1011	D13.2	010 01101	101100 0101	101100 0101
D30.0	000 11110	011110 0100	100001 1011	D14.2	010 01110	011100 0101	011100 0101
D31.0	000 11111	101011 0100	010100 1011	D15.2	010 01111	010111 0101	101000 0101
D0.1	001 00000	100111 1001	011000 1001	D16.2	010 10000	011011 0101	100100 0101
D1.1	001 00001	011101 1001	100010 1001	D17.2	010 10001	100011 0101	100011 0101
D2.1	001 00010	101101 1001	010010 1001	D18.2	010 10010	010011 0101	010011 0101
D3.1	001 00011	110001 1001	110001 1001	D19.2	010 10011	110010 0101	110010 0101
D4.1	001 00100	110101 1001	001010 1001	D20.2	010 10100	001011 0101	001011 0101
D5.1	001 00101	101001 1001	101001 1001	D21.2	010 10101	101010 0101	101010 0101
D6.1	001 00110	011001 1001	011001 1001	D22.2	010 10110	011010 0101	011010 0101
D7.1	001 00111	111000 1001	000111 1001	D23.2	010 10111	111010 0101	000101 0101
D8.1	001 01000	111001 1001	000110 1001	D24.2	010 11000	110011 0101	001100 0101
D9.1	001 01001	100101 1001	100101 1001	D25.2	010 11001	100110 0101	100110 0101
D10.1	001 01010	010101 1001	010101 1001	D26.2	010 11010	010110 0101	010110 0101
D11.1	001 01011	110100 1001	110100 1001	D27.2	010 11011	110110 0101	001001 0101
D12.1	001 01100	001101 1001	001101 1001	D28.2	010 11100	001110 0101	001110 0101
D13.1	001 01101	101100 1001	101100 1001	D29.2	010 11101	101110 0101	010001 0101
D14.1	001 01110	011100 1001	011100 1001	D30.2	010 11110	011110 0101	100001 0101
D15.1	001 01111	010111 1001	101000 1001	D31.2	010 11111	101011 0101	010100 0101

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

Valid Data Characters (Page 2 of 3)							
input		abcdei fghj output		input		abcdei fghj output	
Name	H'G'F'E'D'C'B'A'	rd<0	rd>0	Name	H'G'F'E'D'C'B'A'	rd<0	rd>0
D0.3	011 00000	100111 0011	011000 1100	D16.4	100 10000	011011 0010	100100 1101
D1.3	011 00001	011101 0011	100010 1100	D17.4	100 10001	100011 1101	100011 0010
D2.3	011 00010	101101 0011	010010 1100	D18.4	100 10010	010011 1101	010011 0010
D3.3	011 00011	110001 1100	110001 0011	D19.4	100 10011	110010 1101	110010 0010
D4.3	011 00100	110101 0011	001010 1100	D20.4	100 10100	001011 1101	001011 0010
D5.3	011 00101	101001 1100	101001 0011	D21.4	100 10101	101010 1101	101010 0010
D6.3	011 00110	011001 1100	011001 0011	D22.4	100 10110	011010 1101	011010 0010
D7.3	011 00111	111000 1100	000111 0011	D23.4	100 10111	111010 0010	000101 1101
D8.3	011 01000	111001 0011	000110 1100	D24.4	100 11000	110011 0010	001100 1101
D9.3	011 01001	100101 1100	100101 0011	D25.4	100 11001	100110 1101	100110 0010
D10.3	011 01010	010101 1100	010101 0011	D26.4	100 11010	010110 1101	010110 0010
D11.3	011 01011	110100 1100	110100 0011	D27.4	100 11011	110110 0010	001001 1101
D12.3	011 01100	001101 1100	001101 0011	D28.4	100 11100	001110 1101	001110 0010
D13.3	011 01101	101100 1100	101100 0011	D29.4	100 11101	101110 0010	010001 1101
D14.3	011 01110	011100 1100	011100 0011	D30.4	100 11110	011110 0010	100001 1101
D15.3	011 01111	010111 0011	101000 1100	D31.4	100 11111	101011 0010	010100 1101
D16.3	011 10000	011011 0011	100100 1100	D0.5	101 00000	100111 1010	011000 1010
D17.3	011 10001	100011 1100	100011 0011	D1.5	101 00001	011101 1010	100010 1010
D18.3	011 10010	010011 1100	010011 0011	D2.5	101 00010	101101 1010	010010 1010
D19.3	011 10011	110010 1100	110010 0011	D3.5	101 00011	110001 1010	110001 1010
D20.3	011 10100	001011 1100	001011 0011	D4.5	101 00100	110101 1010	001010 1010
D21.3	011 10101	101010 1100	101010 0011	D5.5	101 00101	101001 1010	101001 1010
D22.3	011 10110	011010 1100	011010 0011	D6.5	101 00110	011001 1010	011001 1010
D23.3	011 10111	111010 0011	000101 1100	D7.5	101 00111	111000 1010	000111 1010
D24.3	011 11000	110011 0011	001100 1100	D8.5	101 01000	111001 1010	000110 1010
D25.3	011 11001	100110 1100	100110 0011	D9.5	101 01001	100101 1010	100101 1010
D26.3	011 11010	010110 1100	010110 0011	D10.5	101 01010	010101 1010	010101 1010
D27.3	011 11011	110110 0011	001001 1100	D11.5	101 01011	110100 1010	110100 1010
D28.3	011 11100	001110 1100	001110 0011	D12.5	101 01100	001101 1010	001101 1010
D29.3	011 11101	101110 0011	010001 1100	D13.5	101 01101	101100 1010	101100 1010
D30.3	011 11110	011110 0011	100001 1100	D14.5	101 01110	011100 1010	011100 1010
D31.3	011 11111	101011 0011	010100 1100	D15.5	101 01111	010111 1010	101000 1010
D0.4	100 00000	100111 0010	011000 1101	D16.5	101 10000	011011 1010	100100 1010
D1.4	100 00001	011101 0010	100010 1101	D17.5	101 10001	100011 1010	100011 1010
D2.4	100 00010	101101 0010	010010 1101	D18.5	101 10010	010011 1010	010011 1010
D3.4	100 00011	110001 1101	110001 0010	D19.5	101 10011	110010 1010	110010 1010
D4.4	100 00100	110101 0010	001010 1101	D20.5	101 10100	001011 1010	001011 1010
D5.4	100 00101	101001 1101	101001 0010	D21.5	101 10101	101010 1010	101010 1010
D6.4	100 00110	011001 1101	011001 0010	D22.5	101 10110	011010 1010	011010 1010
D7.4	100 00111	111000 1101	000111 0010	D23.5	101 10111	111010 1010	000101 1010
D8.4	100 01000	111001 0010	000110 1101	D24.5	101 11000	110011 1010	001100 1010
D9.4	100 01001	100101 1101	100101 0010	D25.5	101 11001	100110 1010	100110 1010
D10.4	100 01010	010101 1101	010101 0010	D26.5	101 11010	010110 1010	010110 1010
D11.4	100 01011	110100 1101	110100 0010	D27.5	101 11011	110110 1010	001001 1010
D12.4	100 01100	001101 1101	001101 0010	D28.5	101 11100	001110 1010	001110 1010
D13.4	100 01101	101100 1101	101100 0010	D29.5	101 11101	101110 1010	010001 1010
D14.4	100 01110	011100 1101	011100 0010	D30.5	101 11110	011110 1010	100001 1010
D15.4	100 01111	010111 0010	101000 1101	D31.5	101 11111	101011 1010	010100 1010

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

Valid Data Characters (Page 3 of 3)							
input		abcdei fghj output		input		abcdei fghj output	
Name	H'G'F'E'D'C'B'A'	rd<0	rd>0	Name	H'G'F'E'D'C'B'A'	rd<0	rd>0
D0.6	110 00000	100111 0110	011000 0110	D0.7	111 00000	100111 0001	011000 1110
D1.6	110 00001	011101 0110	100010 0110	D1.7	111 00001	011101 0001	100010 1110
D2.6	110 00010	101101 0110	010010 0110	D2.7	111 00010	101101 0001	010010 1110
D3.6	110 00011	110001 0110	110001 0110	D3.7	111 00011	110001 1110	110001 0001
D4.6	110 00100	110101 0110	001010 0110	D4.7	111 00100	110101 0001	001010 1110
D5.6	110 00101	101001 0110	101001 0110	D5.7	111 00101	101001 1110	101001 0001
D6.6	110 00110	011001 0110	011001 0110	D6.7	111 00110	011001 1110	011001 0001
D7.6	110 00111	111000 0110	000111 0110	D7.7	111 00111	111000 1110	000111 0001
D8.6	110 01000	111001 0110	000110 0110	D8.7	111 01000	111001 0001	000110 1110
D9.6	110 01001	100101 0110	100101 0110	D9.7	111 01001	100101 1110	100101 0001
D10.6	110 01010	010101 0110	010101 0110	D10.7	111 01010	010101 1110	010101 0001
D11.6	110 01011	110100 0110	110100 0110	D11.7	111 01011	110100 1110	110100 1000
D12.6	110 01100	001101 0110	001101 0110	D12.7	111 01100	001101 1110	001101 0001
D13.6	110 01101	101100 0110	101100 0110	D13.7	111 01101	101100 1110	101100 1000
D14.6	110 01110	011100 0110	011100 0110	D14.7	111 01110	011100 1110	011100 1000
D15.6	110 01111	010111 0110	101000 0110	D15.7	111 01111	010111 0001	101000 1110
D16.6	110 10000	011011 0110	100100 0110	D16.7	111 10000	011011 0001	100100 1110
D17.6	110 10001	100011 0110	100011 0110	D17.7	111 10001	100011 0111	100011 0001
D18.6	110 10010	010011 0110	010011 0110	D18.7	111 10010	010011 0111	010011 0001
D19.6	110 10011	110010 0110	110010 0110	D19.7	111 10011	110010 1110	110010 0001
D20.6	110 10100	001011 0110	001011 0110	D20.7	111 10100	001011 0111	001011 0001
D21.6	110 10101	101010 0110	101010 0110	D21.7	111 10101	101010 1110	101010 0001
D22.6	110 10110	011010 0110	011010 0110	D22.7	111 10110	011010 1110	011010 0001
D23.6	110 10111	111010 0110	000101 0110	D23.7	111 10111	111010 0001	000101 1110
D24.6	110 11000	110011 0110	001100 0110	D24.7	111 11000	110011 0001	001100 1110
D25.6	110 11001	100110 0110	100110 0110	D25.7	111 11001	100110 1110	100110 0001
D26.6	110 11010	010110 0110	010110 0110	D26.7	111 11010	010110 1110	010110 0001
D27.6	110 11011	110110 0110	001001 0110	D27.7	111 11011	110110 0001	001001 1110
D28.6	110 11100	001110 0110	001110 0110	D28.7	111 11100	001110 1110	001110 0001
D29.6	110 11101	101110 0110	010001 0110	D29.7	111 11101	101110 0001	010001 1110
D30.6	110 11110	011110 0110	100001 0110	D30.7	111 11110	011110 0001	100001 1110
D31.6	110 11111	101011 0110	010100 0110	D31.7	111 11111	101011 0001	010100 1110

10.3.7.1.4 8B/10B valid special characters

Special characters are defined as extra code points beyond the 256 needed to encode a byte of data. These are named Kxx.y. P1394b uses one of these special characters, K28.5, during synchronization and training. This character is defined in table 10-8.

Table 10-8—Special character (K28.5)

Control Character Name	abcdei fghj values	
	rd<0	rd>0
K28.5	001111 1010	110000 0101

1 The K28.5 special character contains a comma sequence. A “comma” indicates the proper byte boundaries and can be
 2 used for instantaneous acquisition or verification of byte synchronization. A comma sequence must be singular and must
 3 occur with a uniform alignment relative to the byte boundaries. In the absence of errors, the comma must not occur in any
 4 other bit positions, neither within characters nor through overlap between characters. The comma sequence for the
 5 8B/10B data code is the seven bit sequence “0011111” or “1100000”.
 6

7
 8 Note: this sequence is a comma for the P1394B 8B10B data code, but is not a comma for the P1394b control code.
 9

10 The K28.5 special character is used during the 1394B training procedure. While either a training request or an operation
 11 request is being transmitted, a K28.5 symbol is periodically inserted into the character stream so that the receiving port
 12 may acquire byte synchronization. Specifically, whenever a training request or operation request is transmitted, the port
 13 replaces any occurrence of a D28.0 character in the transmitted character stream with a K28.5 character.
 14
 15

16 **10.3.7.2 Control Coding**

17
 18 In addition to the Dxx.y and Kxx.y characters defined in 10.3.7.1 , P1394b also use a set of 10-bit characters for coding
 19 control symbols. Control coding also produces sequences of 10-bit coded characters with guaranteed AC transitions and
 20 no DC frequency component. The following subsections describe the control code features and document the code table.
 21
 22

23 **10.3.7.2.1 Valid Control Code characters**

24
 25 P1394b adopts the following definitions and conventions in defining the control code:
 26

27 Each valid control code character has a name

28 Czz

29 zz = the 4 bit input value, base 10, for bits P'Q'R'S'

30
 31 In the following table,

32 rd represents the running disparity value from previous bit transmissions

33 rd1 represents the running disparity value after these bits have been transmitted
 34
 35
 36
 37
 38
 39
 40
 41
 42
 43
 44
 45
 46
 47
 48
 49
 50
 51
 52
 53
 54
 55
 56
 57
 58
 59
 60
 61
 62
 63
 64
 65
 66

Table 10-9—Control Coding

Control character name	S'R'Q'P'	abcdeifghj outputs		rd1
		rd>0	rd<0	
C0	0000	0000011111		rd
C1	0001	0000101111		rd
C2	0010	0000111110		rd
C3	0011	0001001111		rd
C4	0100	0010001111		rd
C5	0101	1100000111		rd
C6	0110	0100001111		rd
C7	0111	1000001111		rd
C8	1000	0111110000		rd
C9	1001	1011110000		rd
C10	1010	0011111000		rd
C11	1011	1101110000		rd
C12	1100	1110110000		rd
C13	1101	1111000001		rd
C14	1110	1111010000		rd
C15	1111	1111100000		rd

10.3.7.2.2 Control code properties - run length and DC balance

A sequence of valid 10-bit control code characters has a maximum run length of 10 bits (i.e., 10 consecutive ones or 10 consecutive zeros before a mandatory bit transition). Consequently, ample transitions are provided to aid in clock extraction by the physical layer receiving PLL's.

Each valid control code character also has zero digit disparity (i.e., same number of 1's and 0's in the character). Consequently, control coding maintains the dc balance of the transmitted signal. When the transmitter has been initialized according to 10.3.7.1.1 the running disparity at the end of any control code character is either -1 or +1. More generally, the running disparity after any bit in a sequence of valid 10-bit control code characters is constrained to be between -6 and +6 inclusive.

10.3.7.2.3 Control code properties - error detection

A valid control code character is separated from all valid Dxx.y characters by Hamming distance 2. Consequently, a single bit error in any position will not change a 10-bit control character representing control information into a 10-bit character representing data.

10.3.7.2.4 Control code properties - comma characters (informative)

A "comma" indicates proper character and byte boundaries and can be used for instantaneous acquisition or verification of character and byte synchronization. To be useful, the comma sequence must be singular and must occur with a uniform alignment relative to the byte boundaries. In the absence of errors, the comma must not occur in any other bit positions, neither within characters nor through overlap between characters, in sequences of control (Czz) characters.

The following control code characters are commas with respect to the set of sixteen control code characters:

- C4 = 0010001111
- C11 = 1101110000

The occurrence of the C4 and C11 comma characters in a sequence of control code characters may be used by a receiver to determine the correct timing of the transitions between 10 bit characters (i.e. character synchronization). However, this is not a requirement of this standard.

10.3.8 Character transmission

Data and control characters shall be transmitted serially with the most significant bit (i.e. bit ‘a’) being transmitted first.

10.3.9 Decoding

10.3.9.1 Bit and character synchronization

Before control and data characters can be correctly decoded, a receiver must determine the correct time to sample bits received from the PMD. This is typically achieved through the use of a phase locked loop. A receiver must also determine the correct timing of the transitions between 10-bit characters. This is typically achieved by examining a received sequence of bits and detecting the occurrence of a comma character.

NOTE—The training request and operation request signals contain periodic occurrences of a K28.5 comma character that may be used to acquire character synchronization. See TBD.

10.3.9.2 Character decoding and error detection

8B/10B decoding is performed using table 7-5, table 7-6 and table 7-7. Based on the receiver’s running disparity, the appropriate columns are searched for the received character. If found, the character is considered valid. If the character is not found in the appropriate columns, the character is considered to be invalid. Regardless of the received character’s validity, the receiver’s running disparity is updated according to the received character’s disparity.

A received character not found in the appropriate table and determined to be invalid may not actually contain transmission errors. Table 10-10 shows an example of an earlier undetected error disrupting the receiver’s running disparity and causing a code violation in the current character.

Table 10-10—Delayed error detection example

Time Sequence for Error	RD	Previous Character	RD	Previous Character	RD	Current Character	RD
Transmitted character stream	rd<0	D21.1	rd<0	D10.2	rd<0	D23.5	rd1>0
Transmitted bit stream	rd<0	101010 1001	rd<0	010101 0101	rd<0	111010 1010	rd1>0
Bit stream after error	rd<0	101010 1011	rd>0	010101 0101	rd>0	111010 1010	rd1>0
Decoded character stream	rd<0	D21.0	rd>0	D10.2	rd>0	code violation	rd1>0

TBD—This needs to be reviewed for correctness and presented in a clearer fashion.

10.3.9.3 Special character decoding

Whenever a K28.5 character is received it shall be decoded according to table 10-11. The result of decoding a K28.5 character is the same as the result obtained from decoding a D28.0 character.

Table 10-11—Special character decoding

abcdei fghj input			output
Name	rd<0	rd>0	H'G'F'E'D'C'B'A'
D28.0	001110 1011	001110 0100	000 11100
K28.5	001111 1010	110000 0101	000 11100

10.3.10 Descrambling

Control and data symbols shall be descrambled using the reverse of the scrambling procedures as described in 10.3.6. For successful operation, the state of a receiver's descrambler should be synchronized with the state of the scrambler in the remote transmitter. This requires that the receiver learn the state of the remote transmitter's scrambler during port training. A receiver shall synchronize the state of its descrambler with the state of the remote transmitter's scrambler while receiving sequences of valid TRAINING request or OPERATION request symbols.

10.4 Beta mode port operation

10.4.1 Transmit operations

10.4.1.1 Control transmission

For each beta port control state the port shall first determine the control symbol according to table 10-1. This control value shall be scrambled according to 10.3.6.3 and coded according to 10.3.7.2 and the resulting control characters shall be transmitted according to 10.3.8 .

10.4.1.2 Request type transmission

For each beta port request type the port shall first determine the request type symbol according to table 10-4. This control value shall be scrambled according to 10.3.6.2 and coded according to 10.3.7.1 and the resulting data characters shall be transmitted according to 10.3.8 .

Whenever the port is transmitting either a TRAINING request or OPERATION request signal, and the data character to be transmitted is a D28.0 character, the port shall replace this character with a K28.5 special character.

NOTE—The insertion of K28.5 characters enables a remote receiver to establish byte synchronization with the transmitting port.

10.4.1.3 Packet transmission

Packet transmission begins with either a speed signal or, in the case of an S100 packet, the first data byte. During packet transmission the port may still be required to transmit some control states, for example to complete a packet prefix. These control states and the packet data are referred to collectively as the packet payload. When the packet speed is less than the port operating speed, other characters are transmitted during the packet along with the payload characters. These characters are referred to as padding characters.

NOTE—In contrast to 1394a DS ports, the port transmission speed remains constant for all packet speeds.

10.4.1.3.1 Speed signaling

A beta mode port is required to transport packets at all 1394 rates up to and including the operating speed of that particular port. The port signals the speed of a packet by transmitting a sequence of SPEEDa and SPEEDb control symbols during the packet prefix. The combination of SPEEDa and SPEEDb control symbols transmitted indicates the packet

speed relative to the port operating speed. When required to transmit a speed signal, the port shall generate a sequence of SPEED control symbols appropriate for the packet speed according to table 10-12. Each SPEED symbol shall be scrambled according to 10.3.6.3 and coded according to 10.3.7.2 .

Table 10-12—Speed signal formats

Port operating speed	Packet speed					
	S100	S200	S400	S800	S1600	S3200
S100	-					
S200	-	S_b				
S400	-	$S_a S_b$	S_b			
S800	-	$S_a S_a S_b S_a$	$S_a S_b$	S_b		
S1600	-	$S_a S_a S_a S_b S_a S_a S_a$	$S_a S_a S_b S_a$	$S_a S_b$	S_b	
S3200	-	$S_a S_a S_a S_a S_b S_a S_a S_a S_a S_a S_a S_a S_a S_a S_a S_a$	$S_a S_a S_b S_a S_a S_a S_a$	$S_a S_b S_a$	$S_a S_b$	S_b
speed signal duration (nsecs)	0	40	20	10	5	2.5

- 1) S_a represents the SPEED_a control symbol.
- 2) S_b represents the SPEED_b control symbol.
- 3) For any particular packet speed, the speed signal has constant duration at all port speeds. At all packet speeds other than S100, the number of speed symbols used at a particular port operating speed is equal to the number of symbols transmitted per byte of packet data at that port operating speed.

10.4.1.3.1.1 General packet format

In general a beta mode port shall generate packets with the format shown below:

.....<data prefix symbols><speed symbols><data prefix symbols><padded data>
[packet end symbols]<arbitration token symbols>.....

The <> delimiters indicate elements of the packet that may be optional at some (or all) combination of operating speed and packet speed, as described below. The [] delimiters indicate mandatory elements of the packet. The nomenclature [a|b] indicates a mandatory element that may be either a or b. The nomenclature [symbol]ⁿ indicates that the specified symbol shall be repeated n times.

A packet end symbol shall be any of DATA_END, DATA_END_ERR, DATA_PREFIX or DATA_PREFIX_ERR. An arbitration token symbol shall be any of ARBRST, ARBRST_GRANT, GRANT or a packet end symbol.

10.4.1.3.1.2 Packets at speeds greater than the port operating speed

When the packet speed exceeds the operating speed of a port, the port shall not generate a speed signal and shall send DATA_PREFIX continuously during the packet, until the packet end_symbols are to be sent. The port shall then send the packet end symbols.

10.4.1.3.1.3 S100 packet speeds

No speed signal is sent for an S100 packet at any operating speed. An S100 packet is always preceded by a number of DATA_PREFIX symbols which indicate the polarity of the running disparity at the start of the packet. A valid S100 packet is ended by either a number of DATA_END symbols or, in the case of concatenated packets, a number of DATA_PREFIX symbols. An S100 packet that is known to be corrupt is ended by either a number of DATA_END_ERR symbols or, in the case of concatenated packets, a number of DATA_PREFIX_ERR symbols. The packet format for a S100 packet forwarded on any beta mode port shall be:

.....[DATA_PREFIX symbols][padded data][packet end symbol]^m[arbitration token symbol]ⁿ.....

1 where m is equal to the ratio of the port operating speed and the packet speed. n is chosen such that the duration of the
2 packet end symbols and arbitration token symbols shall be greater than or equal to the minimum duration of DATA_END
3 required on a DS port, DATA_END_TIME, as specified in the 1394a standard i.e. 240nsecs, and also equal to the time
4 required to transmit an integer number of bytes on a port operating at the packet speed i.e. $n = i \times m$ for integer values of i.
5

6
7 The duration of the data prefix symbols shall be greater than or equal to the minimum duration of DATA_PREFIX
8 required on a DS port, MIN_DATA_PREFIX, as specified in the 1394a standard i.e. 140nsecs.
9

10 NOTE—Editorial: 1394a specifies MIN_DATA_PREFIX=140nsecs but recommends 180nsecs is used for compatibility - which should
11 1394b specify?
12

13 NOTE—These requirements ensure that if the packet is subsequently forwarded to a DS port, the DS port will be able to generate a
14 minimum length DATA_PREFIX and DATA_END without buffering data. The requirements also ensure that if the packet is
15 subsequently forwarded to a beta mode port operating at a lower speed, that port is able to maintain the duration of the packet end
16 symbols.
17

18 19 **10.4.1.3.1.4 S200 & S400 packets** 20

21
22 The packet format for S200 and S400 packets forwarded on any beta mode port operating at a speed equal to or greater
23 than the packet speed shall be:
24

25<data prefix symbols>[speed signal][data prefix symbols]^l[padded data][packet end symbol]^m[arbitra-
26 tion token symbol]ⁿ.....
27

28
29 where m is the ratio of the port operating speed and the packet speed. n is chosen such that the duration of the packet end
30 symbols and arbitration token symbols shall be greater than or equal to the minimum duration of DATA_END required on
31 a DS port, DATA_END_TIME, as specified in the 1394a standard i.e. 240nsecs, and also equal to the time required to
32 transmit an integer number of bytes on a port operating at the packet speed i.e. $n = i \times m$ for integer values of i.
33

34 The time between completion of transmission of the SPEEDb symbol of the speed signal and completion of transmission
35 of the first byte of data shall be greater than or equal to MIN_DATA_PREFIX, as specified by 1394a. In addition, the
36 duration of the DATA_PREFIX symbols between the speed signal and first byte of data shall be equal to the time required
37 to transmit an integer number of bytes on a port operating at the packet speed, i.e. $l = i \times m$ for integer values of i.
38

39
40 NOTE—These requirements ensure that if the packet is subsequently forwarded to a DS port, the DS port will be able to generate a
41 minimum length DATA_PREFIX without buffering data. The requirements also ensure that if the packet is subsequently forwarded to a
42 beta mode port operating at a lower speed, that port is able to transmit a speed signal followed by an integer number of DATA_PREFIX
43 symbols (when required) before the first data byte and is able to maintain the duration of the packet end symbols.
44

45 46 **10.4.1.3.1.5 S100, S200 & S400 packets - beta mode only transmission** 47

48
49 In the special case when a port has determined that all ports between a packet's source and destination are operating in
50 beta mode (for example when all nodes on a bus are known to be operating in beta mode), packets at speeds up to and
51 including S400 may be transmitted using a shorter packet format. If used, the format used for a packet forwarded on any
52 beta mode port operating at a speed equal to or greater than the packet speed shall be:
53

54[speed signal][data prefix symbols]^m[padded data][packet end symbols]^m[arbitration token sym-
55 bols]^m.....
56

57
58 where m is equal to the ratio of the port operating speed and the packet speed.
59

60
61 NOTE—The requirements ensure that if the packet is subsequently forwarded to a beta mode port operating at a lower speed, that port
62 is able to transmit a speed signal and packet end symbols without buffering data.
63
64
65
66

mitted according to table 10-14. The port shall generate the appropriate number of SPEEDa control symbols. These shall be scrambled according to 10.3.6.3 and coded according to 10.3.7.2, and the resulting control characters shall be transmitted contiguously following the data character.

Table 10-14—Data padding formats

Port operating speed	Packet speed					
	S100	S200	S400	S800	S1600	S3200
S100	D					
S200	DP	D				
S400	DPPP	DP	D			
S800	DPPPPPPP	DPPP	DP	D		
S1600	DPPPPPPPPPPPPPPPP	DPPPPPPP	DPPP	DP	D	
S3200	DPP	DPPPPPPPPPPPPPPPP	DPPPPPPP	DPPP	DP	D

- 1) For any particular packet speed, a padded sequence has constant duration at all port speeds.
- 2) D represents a payload data byte.
- 3) P represents a SPEEDa control symbol.
- 4) Table entries represent the information stream prior to scrambling and coding.

If the packet speed is less than the port speed then any any payload control symbols are also padded. During packet transmission each control state shall be mapped to a control symbol according to table 10-1, scrambled according to 10.3.6.3 and coded according to 10.3.7.2. The resulting character shall be transmitted repeatedly such that the length of the repeated control sequence is equal to the length of a padded data sequence at the requested packet speed. The port compares the value of the packet speed parameter with the port operating speed and determines the number of control symbols to be transmitted according to table 10-15. The port shall generate the appropriate number of control symbols. These shall be scrambled according to 10.3.6.3 and coded according to 10.3.7.2, and the resulting control characters shall be transmitted contiguously following the payload control character.

Table 10-15—Control padding formats

Port operating speed	Packet speed					
	S100	S200	S400	S800	S1600	S3200
S100	C					
S200	CC	C				
S400	CCCC	CC	C			
S800	CCCCCCCC	CCCC	CC	C		
S1600	CCCCCCCCCCCCCCCCCC	CCCCCCCC	CCCC	CC	C	
S3200	CC	CCCCCCCCCCCCCCCCCC	CCCCCCCC	CCCC	CC	C

- 1) For any particular packet speed, a padded sequence has constant duration at all port speeds.
- 2) C represents a payload control symbol.
- 3) Table entries represent the information stream prior to scrambling and coding.

10.4.2 Packet transmission examples

These examples of packet formats are given for illustrative purposes. The following nomenclature is used:

Bn represents the nth data byte of a packet.

DP represents a DATA_PREFIX symbol.

DE represents a DATA_END symbol.

1 ARB indicates an arbitration token (ARBRST or ARBRST_GRANT or GRANT or DATA_END)

2
3 P represents a padding symbol (SPEEDa).

4
5 X^y is used to indicate a sequence of y consecutive X symbols e.g. $DE^{\{24,28\}}$ indicates a sequence of either 24 or 28
6 DATA_END symbols.
7

8 9 **10.4.2.1 S100 packet forwarded on an S800 port:**

10
11 $DP^{>13} B_1 P P P P P P P B_2 P P P P P P P B_3 \dots B_n P P P P P P P DE^8 ARB^{\{16,24,32,40,\dots\}}$

12 13 14 **10.4.2.2 S200 packet forwarded on an S800 port:**

15
16 $DP \dots DP S_a S_a S_b S_a DP^{\{12,16,20,24,28,\dots\}} B_1 P P P B_2 P P P B_3 \dots B_n P P P DE^4 ARB^{\{20,24,28,32,\dots\}}$

17 18 19 **10.4.2.3 S800 packet forwarded on S800 port:**

20
21 $S_b DP B_1 B_2 B_3 B_4 \dots B_n DE ARB$

22 23 24 **10.4.2.4 S800 packet forwarded on S1600 port:**

25
26 $S_a S_b DP DP B_1 P B_2 P B_3 P \dots B_n P DE DE ARB ARB$

27 28 29 **10.4.5 Receive operations**

30 31 **10.4.5.1 Port training**

32
33 When a beta mode port is initially activated, and also whenever loss of synchronization occurs, the port must initiate a
34 training procedure. During this procedure, the port transmits a training request signal to the attached port. This signal
35 causes the attached port to also begin the training procedure, and to transmit the training request signal. While receiving
36 the training request signal, the port synchronizes itself with the received codeword stream. The port must acquire bit syn-
37 chronization and also determine the boundary between 10-bit codewords (codeword synchronization). The port also syn-
38 chronizes its descrambler with the scrambler of the attached transmitter.
39
40

41 42 **10.4.5.1.1 Loss of synchronization detection procedure**

43
44 The port determines that it has lost synchronization with the received codeword stream by monitoring the validity of
45 received codewords. The port shall maintain a count of the number of invalid codewords received. Whenever an invalid
46 codeword is received, this count shall be incremented, to a maximum value of four. Whenever two consecutive valid code-
47 words are received, the count shall be decremented (to a minimum value of zero). If the count reaches four then the port
48 shall attempt to resynchronize.
49
50

51
52 Whilst in the Receive state (i.e. during normal operation) the port receiver shall consider the control state TRAINING
53 request to be an INVALID signal. This provides a means for a connected transmitting port to force the receiving port to
54 enter the synchronization procedure.
55
56

57 58 **10.4.5.1.2 Resynchronization procedure**

59
60 When the port is initially activated and whenever the port determines that synchronization has been lost, it shall transmit
61 a TRAINING request signal and attempt to resynchronize with the incoming codeword stream. The training request signal
62 is treated in the same way as an INVALID symbol by the remote port, and therefore causes the remote port to enter the
63 synchronization procedure.
64
65
66

1 The port shall initially attempt to acquire character synchronization. In order to verify that character synchronization is
2 complete, the port shall receive a valid comma character (K28.5) preceded by at least two valid request type characters
3 (Dx.0 or Dx.4).
4

5
6 NOTE—When a port first begins resynchronization, it may not necessarily be receiving training signals from the remote port. The
7 requirement for a K28.5 to be preceded by two valid request type characters ensures that incorrect character synchronization cannot
8 occur when signals other than the TRAINING request and OPERATION request are received.
9

10 Once character synchronization is complete, the port shall train its descrambler. The port shall verify that scrambler train-
11 ing has been successful by checking for at least SYNC_CHECK consecutive occurrences of a TRAINING request or
12 SYNC_CHECK consecutive occurrences of a OPERATION request.
13

14
15 If during this check a request type is received that indicates an isochronous request type of NEXT_FAST and an asyn-
16 chronous request type of CYCLE_START, or if a request type is received that indicates an isochronous request type of
17 NEXT_FAST and an asynchronous request type of NONE, then a polarity inversion may have occurred in the cable plant,
18 and the port shall subsequently invert all received characters and restart this check.
19

20
21 NOTE—It is not necessary for the port to repeat the character synchronization and descrambler training if the polarity is found to be
22 inverted.
23

24 Upon achieving synchronization, the port shall begin to transmit an operation request signal. The port shall continue to
25 transmit an operation request signal until it has received and successfully decoded at least 2*SYNC_CHECK consecutive
26 operation request symbols since the beginning of sending an operation request signal, or it is deactivated by the connec-
27 tion management function.
28
29

30 **10.4.5.2 Control reception**

31
32 When any valid control character is received, the port shall determine the received control symbol by decoding the control
33 character according to 10.3.9 and descrambling the result of the decoding operation according to 10.3.10 . In order to be
34 a valid control character, a received character shall appear in table 10-9. For each received control symbol the port shall
35 determine the received control state according to table 10-1 and indicate this to the port manager.
36
37

38 If an invalid character is received the port shall maintain the previous control state indication.
39

40 **10.4.5.3 Packet prefix reception**

41
42 All packets received by a port are preceeded by a packet prefix consisting of a number of DATA_PREFIX and possibly a
43 number of SPEED symbols. The DATA_PREFIX symbol indicates the state of the remote source port's disparity at the
44 start of the packet. Since the receiving port is required to check running disparity during packet reception, it is necessary
45 for the receiving port to have knowledge of the running disparity both at the start and at the end of a packet. The receiver
46 running disparity should be updated at the start of each received packet as information held over from previous packets
47 may be incorrect due to errors in the received signals.
48
49

50
51 The port shall detect the first occurrence of a DATA_PREFIX before a packet and from this shall determine the running
52 disparity of the character stream at the start of the packet. The port shall check that the running disparity indication is
53 consistent for all DATA_PREFIX symbols received prior to the packet data. If the port detects any inconsistent disparity
54 indications then it shall consider the immediately following packet to be corrupt.
55
56
57
58
59
60
61
62
63
64
65
66

10.4.5.3.1 Speed signal determination

A Beta mode port shall receive a speed signal and indicate the speed of the subsequent packet to the port manager. This indication shall not be generated before a SPEEDb code or a data byte has been received. Valid speed signals shall be used to determine the speed of the subsequent packet according to table 10-16.

Table 10-16—Speed signal decoding

Received speed signal	Port operating speed					
	S100	S200	S400	S800	S1600	S3200
none	S100	S100	S100	S100	S100	S100
S _b	(S100)	S200	S400	S800	S1600	S3200
S _a S _b	-	(S100)	S200	S400	S800	S1600
S _a S _a S _b S _a	-		(S100)	S200	S400	S800
S _a S _a S _a S _b S _a S _a S _a	-			(S100)	S200	S400
S _a S _a S _a S _a S _b S _a S _a S _a S _a S _a S _a S _a	-				(S100)	S200
S _a S _a S _a S _a S _a S _b S _a S _a S _a S _a S _a S _a S _a S _a						(S100)
S _a S _a S _a S _a S _a S _a S _a S _a S _a S _a S _a S _a S _a S _a						(S100)

NOTE—The (S100) entries indicate a speed signal that should only be received when all ports between a packet’s source and destination are operating in beta mode.

A speed signal shall only be considered valid if it meets the following criteria:

- a) All SPEEDa and SPEEDb codes are received consecutively without interruption.
- b) The pattern of SPEEDa and SPEEDb codes is found in table 10-16.

If an invalid speed signal is received the port shall not pass a packet speed indication to the port manager and shall consider the immediately following packet to be corrupt.

10.4.5.4 Payload reception

All characters, whether control or data, received following a speed signal or the first byte of data are considered part of the packet payload, until the packet end is received. The packet speed may be used to determine the padding format of these characters. During packet reception the port shall discard all padding characters and only indicate payload characters to the port manager.

The port shall check that the format of the packet data sequence (padded or not) agrees with that indicated by table 10-14 for the packet speed. If no speed signal is received prior to the packet, then the packet speed should be S100, and the port shall check that the padding format is correct for an S100 packet.

If at any point during payload reception the padding format does not agree with that specified for the packet speed, then an error has occurred and the port shall consider that packet to be corrupt.

During packet reception the port shall determine whether a received payload character is a valid data character or a valid control character. In order to be a valid data character, a received character shall appear in table 10-7 and have a disparity that conforms to the coding rules described in 10.3.7 i.e. if the current disparity of the received stream is positive, a character shall have negative or zero disparity in order to be valid; if the current disparity of the received stream is negative, a character shall have positive or zero disparity in order to be valid. In order to be a valid control character, a received character shall appear in table 10-9.

The port shall decode all valid data payload characters according to 10.3.9 and descramble the result of the decoding operation according to 10.3.10 . The port shall update the receive disparity according to the disparity of the payload character.

1 NOTE—The receive disparity is only updated when a character is received that is a payload character. Padding characters are not used
2 to update the receive disparity. The disparity is updated regardless of the validity of the payload character.
3

4 If a valid control payload character is received, the port shall determine the received control state by decoding the control
5 character according to 10.3.9 and descrambling the result of the decoding operation according to 10.3.10 . The port shall
6 consider the first control character in a padded sequence to be the payload character. The port shall determine the format
7 of the padded sequence for the packet speed according to table 10-15.
8

9
10 If during packet reception the port receives a payload character that is neither a valid data character nor a valid control
11 character, it shall indicate to the port manager that a data byte of value [0000 0000] was received and consider that packet
12 to be corrupt..
13

14
15 *NOTE—A packet is only considered as errored if the invalid character is in a payload position. Errors in padding positions are ignored.*
16

17 Data reception ends when a DATA_END, DATA_END_ERR or DATA_PREFIX or DATA_PREFIX_ERR is received. The
18 port shall then perform further checks to determine whether the received packet was corrupt.
19

20
21 A packet shall be considered as containing errors if a DATA_END_ERR or DATA_PREFIX_ERR control state is
22 received.
23

24
25 *NOTE—The DATA_END_ERR and DATA_PREFIX_ERR signals indicate that a previous port has determined this packet to be*
26 *corrupt. This may occur even if the packet appears to be correct to the currently receiving port, and under all circumstances the packet*
27 *should be treated as corrupt if followed by a DATA_END_ERR or DATA_PREFIX_ERR signal.*
28

29 The port shall also attempt to determine whether the received packet contained errors by performing all of the following
30 checks:
31

- 32 a) A packet shall be considered as containing errors if the disparity of the received payload data character stream
33 does not match that indicated by the first DATA_PREFIX, DATA_END state to follow the packet. e.g. a packet is
34 considered errored if the disparity of the received payload data character stream is positive when a control value
35 of [0110] is received (DATA_END indicating negative disparity).
36
- 37 b) A packet shall be considered as containing errors if the number of data payload characters received is not equal to
38 an integer multiple of four (a valid packet may contain zero data characters).
39

40 *NOTE—In the event of sufficient errors, control characters at the periphery of packets or embedded in packets may appear as valid*
41 *data characters at the receiving port. This may be detected by checking that a packet consists of an integer number of quadlets.*
42

43 If during packet reception or at the end of a packet the port determines that a packet contains errors then it shall indicate
44 a control state of DATA_END_ERR to the port manager in place of the first occurrence of DATA_END at the end of a
45 packet, or indicate a DATA_PREFIX_ERR to the port manager in place of the first occurrence of DATA_PREFIX at the
46 end of the packet.
47
48

49
50 *NOTE—Subsequent indications should return to indicating the actual received state*
51

52 **10.4.5.5 Error reporting** 53

54
55 Whenever the port receives an INVALID codeword, it shall increment the value of the port_error_reg register, to a maxi-
56 mum value of 255. The port_error_reg register shall be an 8 bit read-only PHY register which shall be reset on comple-
57 tion of port training. The port takes no action based on the value of this register. A codeword shall be considered
58 INVALID if:
59

- 60 a) it does not belong to the set of data codewords nor the set of control codewords, or
61
- 62 b) it has incorrect disparity, or
63
- 64 c) it is a valid data codeword, but not Dx.0 or Dx.4, and occurs outside the bounds of a packet, or
65
- 66 d) it is a control codeword that is not received in the correct context, or

- 1 e) it is a valid payload character but is not received in the correct payload position within a padded packet, or
2 f) it is a valid padding character but is not received in a correct padding position within a padded packet.
3

4 NOTE—The register may be read by higher level applications, which may take action on the basis of the register values. For example,
5 a higher level application might disable a port that reports a large error rate. However, the specification of such functionality is beyond
6 the scope of this standard.
7

8 9 **10.5 Test modes**

10 A beta port is required to provide the following functions for the purpose of testing correct operation of the port and
11 attached PMD, and to enable testing of attached nodes.
12

13 **10.5.1 Transmit scrambler disabled mode**

14 During normal operation, the relationship between data and control states and actual characters transmitted is difficult to
15 predict, due to the scrambling function in the transmit path. This test mode allows an application to deterministically
16 transmit specific patterns of characters by disabling the scrambler.
17

18 Whenever the `disable_scrambler` register has a value of 1, the port shall inhibit the operation of the scrambler, such that
19 scrambled data, control states and request types are equal in value to unscrambled data, control states and request types.
20 Otherwise the scrambler shall be enabled.
21

22 NOTE—A port will not successfully complete training while the transmit scrambler is disabled, since no K28.5 comma characters will
23 be transmitted. However, once training is complete, the transmit scrambler can be disabled without causing loss of synchronization,
24 since the transmitted characters are still valid. Data passed from the receiving PHY to its LINK layer will not be valid if the transmit
25 scrambler is disabled and the receiver scrambler is enabled.
26

27 28 **10.6 State machines and C code**

29 **10.6.1 Transmit state machine**

30 **10.6.1.1 Transmit state machine notes**

31 **State PTX0: Inactive.** The port is not transmitting.
32

33 **Transition PTX0:PTX1.** The beta mode port is activated by the connection management function setting `bport_active` to
34 be true. The first action after being activated is for the port to attempt to synchronize with the connected port.
35

36 **State PTX1: Sync lost.** While in this state the port transmits a training request signal to the connected port.
37

38 **Transition PTX1:PTX2.** This transition is made when the receiver signals that it has acquired synchronization with the
39 connected port.
40

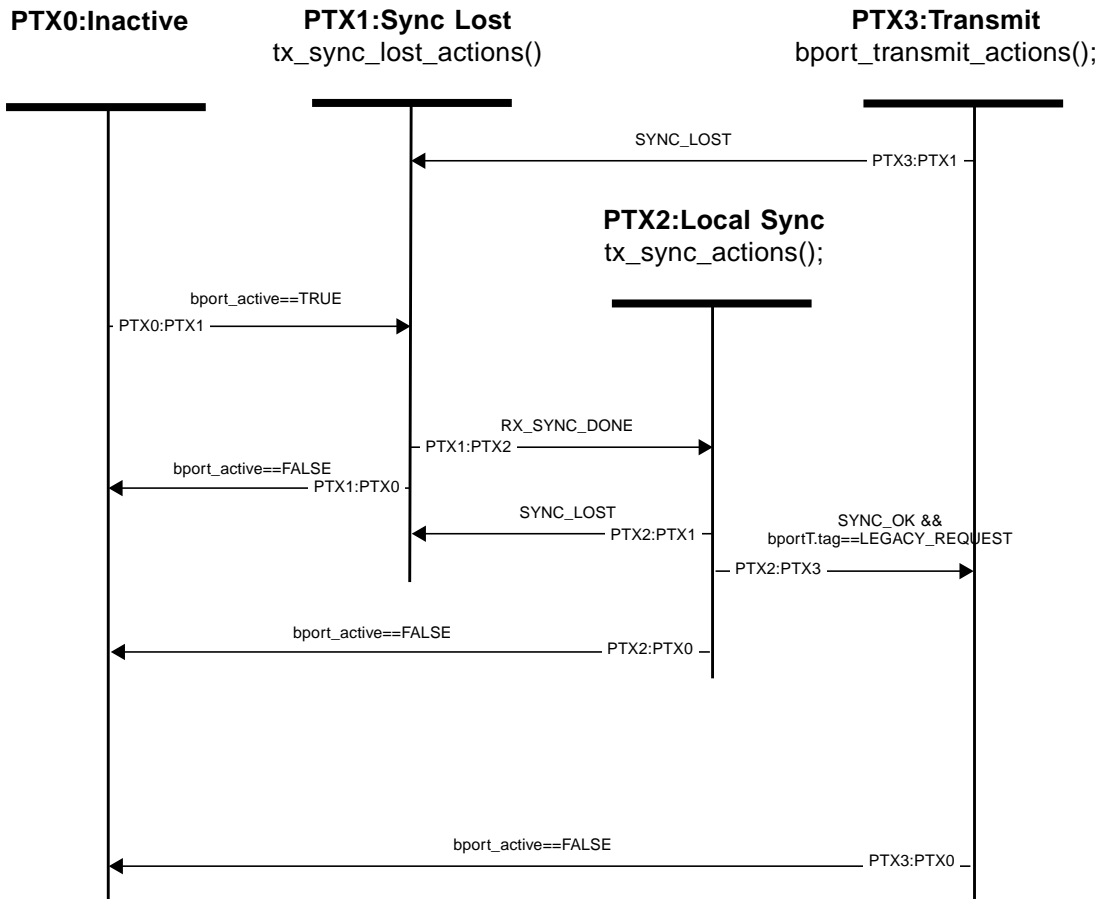
41 **Transition PTX1:PTX0.** If the connection management function sets `bport_active` false, then the port returns to the Inac-
42 tive state. This may happen if the connection management function decides that training has failed by timing out.
43

44 **State PTX2: Local Sync.** The port receiver is synchronized with the connected port, and the port transmits an operation
45 request signal to indicate that it wishes to commence normal operation.
46

47 **Transition PTX2:PTX0.** The connection management function may still cause the port to transition to the Inactive state
48 from the Local Sync state by setting `bport_active` false.
49

50 **Transition PTX2:PTX1.** If the port receiver signals that it has lost synchronization with the connected port, the transmit-
51 ter returns to the Sync Lost state.
52

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66



Transition PTX2:PTX3. However, if the port receiver determines that the connected port has also acquired synchronization, then the transmitter transition to the Transmit state.

State PTX3: Transmit. This is the normal operating state for the transmitter.

Transition PTX3:PTX1. If synchronization is lost the transmitter returns to the Sync Lost state.

Transition PTX3:PTX0. The port returns to the inactive state if the connection management function sets `bport_active` false.

10.4.4 Port transmit actions and conditions

```

enum portState {PTX0, PTX1, PTX2, PTX3, PRX0, PRX1, PRX2, PRX3, PRX4};
enum bportSymbol bportT; //variable that indicates signal to be sent on port
enum bportSymbol localT; //variable that indicates signal to be sent on port
enum bportSymbol bportR[FIFO_DEPTH]; //variable that indicates signal received on port
boolean bport_active; //set by connection management
enum speedCode port_speed; //operating speed of this port, set by connection management function.

enum portState rx_state; //indicates state of port receive function
enum speedCode rx_speed; //speed of received packet
int port_speed; //operating speed of port
int phy_speed; //operating speed of PHY

int data_byte; //8 bit data value
    
```

```

1 void bport_transmit_actions() {
2   int pkt_speed;           //speed of each packet.
3   int i,j;
4
5   pkt_speed=port_speed;
6
7   while(~SYNC_LOST && bport_active) {
8     if(bportT.tag==SPEED) { //currently will send a speed signal even for S100 - needs arb. st. machine to not send speed signal for
9                           //S100 if non-beta nodes on bus i.e. long packet format.
10      pkt_speed=convertspeed(bportT.speed);
11      j=pkt_speed;
12      localT.tag=CTRL;
13      for(i=pkt_speed;i<=port_speed;i=i+pkt_speed) {
14        if(j=port_speed) {
15          localT.ctrl=SPEEDb;
16          tx_character(localT);
17        }
18        else {
19          localT.ctrl=SPEEDa;
20          tx_character(localT);
21        }
22        j=j*2;
23      }
24    }
25    else if(bportT.tag==DATA) {
26      tx_character(bportT);
27      localT.tag=CTRL;
28      localT.ctrl=SPEEDa;
29      for(i=pkt_speed;i<port_speed;i=i+pkt_speed) {
30        tx_character(localT);           //send padding if required
31      }
32    }
33    else if(bportT.tag==REQUEST) {
34      tx_character(bportT);
35      pkt_speed=port_speed; //occurence of a request indicates end of all packet components, so set pkt_speed to default
36    }
37    else if(bportT.tag==CTRL) {
38      for (i=pkt_speed; i<=port_speed; i=i+pkt_speed) tx_character(bportT); //stretch control to length of a padded byte
39    }
40    else tx_character(bport T); //send legacy requests as and when instructed
41  }
42
43 void tx_sync_lost_actions() {
44
45 while(bport_active && ~RX_SYNC_DONE) {
46   localT.tag=REQUEST;
47   localT.req=TRAINING;
48   tx_character(localT);
49 }
50 }
51
52
53 void tx_sync_actions() {
54
55 while(bport_active && ~(SYNC_OK && bportT.tag==LEGACY_REQUEST)) {
56   localT.tag=REQUEST;
57   localT.req=OPERATION;
58   tx_character(localT);
59 }
60 }
61
62
63
64
65
66

```

```

1  int convert_speed(speedCode speed) {
2  int speed_integer;
3  //simple routine to convert the speed code to an integer value that can be used for integer arithmetic.
4
5  switch (speed) {
6      case S100: speed_integer=100; break;
7      case S200: speed_integer=200; break;
8      case S400: speed_integer=400; break;
9      case S800: speed_integer=800; break;
10     case S1600: speed_integer=1600; break;
11     case S3200: speed_integer=3200; break;
12     }
13 return (speed_integer);
14 }
15
16 void tx_character(bportSymbol tx) {
17 int tx_rds;           // running transmit disparity
18 int control_sym;     //4 bit representation of control state
19 int req_type;        //8 bit request type symbol
20 int scram;           //scrambler state
21 int scram_control;   //scrambled control symbol
22 int scram_data;      //scrambled data byte
23 int scram_req_type;  //scrambled request type
24 int control_sym_table; //table mapping control states to control symbol values
25 int control_codetable[16]; //table mapping scrambled control values to control characters
26 int data_codetable[256,2]; //table of data characters
27 int tx_rds;           //disparity of transmitted character stream
28 int tx_rds_table[256,2]; //table of disparity values
29 int character;        //10 bit character
30
31 if (tx.tag==DATA) {
32     scram_data=tx.data^scram;           //scramble the data byte
33     character=data_codetable[scram_data, tx_rds]; //lookup character and update disparity
34     tx_rds=tx_rds_table[scram_data, tx_rds];
35 }
36 else if(tx.tag==CTRL) {
37     control_sym=control_sym_table[tx.ctrl];
38     scram_control=control_sym^(scram&0x80)>>4 || (scram&0x20)>>3 || (scram&0x8)>>2 || (scram&0x2)>>1;
39     //check scrambler bits****
40     character=control_codetable[scram_control];
41 }
42 else {
43     if(tx.tag==REQUEST) req_type=req_symbol_map(tx.req);
44     else if(tx.tag==LEGACY_REQUEST) req_type=legacy_req_symbol_map(tx.sig);
45     else { //shouldn't happen!
46         scram_req_type=(req_type^scram) & 0x9F;
47         if(tx.tag==LEGACY_REQUEST && (tx.req==TRAINING || tx.req==OPERATION) && scram_req_type==0x1C)
48             character=K28.5_table[rds];
49         else character=data_codetable[scram_req_type, tx_rds]; //lookup character and update disparity
50         tx_rds=tx_rds_table[scram_req_type, tx_rds];
51     }
52 }
53 update_scrambler();
54
55 for(i=0;i<10;i++) {
56     PMD_DATA.request(character & 0x80);
57     character <<1;
58     for (i=0;i<phy_speed/port_speed;i++) wait_event(PHY_BIT_CLOCK.indication); //wait for next port bit time
59 }
60 }
61
62
63
64
65
66

```

```

1  int req_symbol_map(beta_request_code txrequest) {
2  int async_part, isoch_part;
3
4  switch(txrequest.async) {
5      case NONE: async_part=1; break;
6      case NEXT_SLOW: async_part=2; break;
7      case NEXT_FAST: async_part=3; break;
8      case CURR_SLOW: async_part=4; break;
9      case CURR_FAST: async_part=5; break;
10     case CYCLE_START: async_part=6; break;
11     }
12  switch(txrequest.isoch) {
13     case NONE: isoch_part=1; break;
14     case NEXT_SLOW: isoch_part=2; break;
15     case NEXT_FAST: isoch_part=3; break;
16     case CURR_SLOW: isoch_part=4; break;
17     case CURR_FAST: isoch_part=5; break;
18     }
19  return(3<<isoch_part | async_part);
20  }
21
22  int legacy_req_symbol_map(legacyReqType txrequest) {
23
24  switch(txrequest) {
25     case REQNT: return(1); break;
26     case CHILD_NOTIFY: return(2); break;
27     case PARENT_NOTIFY: return(3); break;
28     case DISABLE: return(4); break;
29     case SUSPEND: return(5); break;
30     case IDLE: return(7); break;
31     case TRAINING: return(0); break;
32     case OPERATION: return(6); break;
33     }
34  }
35
36  static int scram_old; // should be initialized to any value other than zero on power up
37  void update_scrambler() {
38
39  int i;
40  int scram;
41  int scram_new;
42  scram_new=0;
43
44  for (i=0;i<8;i++) {
45     scram_new <<1;
46     scram_new=scram_new|((scram_old & 0x700)^(scram & 0x100));
47     }
48  scram_old=scram_new;
49  scram=scram_old & 0xFF;
50  }
51
52

```

10.4.5 Receive state machine

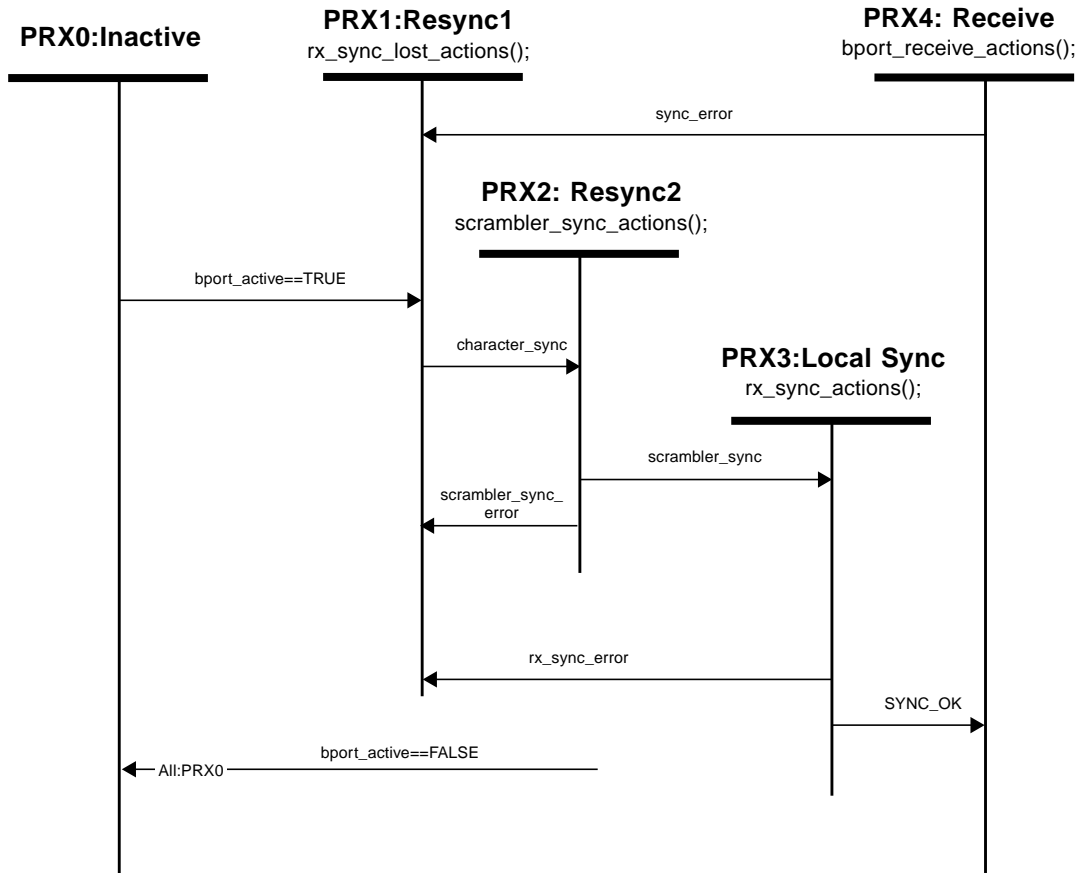
10.4.6 Receive state machine notes

State PRX0: Inactive. The port is not required to receive or decode any signals.

Transition PRX0:PRX1. When the connection management function sets bport_active true, the port shall enter the Resyn1 state and attempt to acquire bit and character synchronization.

State PRX1: Resync1. While in this state the receiver is attempting to acquire bit and character synchronization. The receiver also requests that the transmitter sends a TRAINING request signal.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66



Transition PRX1:PRX2. Character synchronization is considered complete when a comma character (K28.5) has been received, immediately preceded by at least two consecutive valid request type characters (Dx.0 or Dx.4).

State PRX2: Resync2. In this state the receiver trains its descrambler using the scrambler samples contained in the incoming character stream.

Transition PRX2:PRX1. If any invalid character is received, the receiver returns to the start of the synchronization procedure.

Transition PRX2:PRX3. After the descrambler is trained and least SYNC_CHECK consecutive training or operation requests have been received, the receiver is synchronized.

State PRX3: Local Sync. Once it is synchronized, the receiver requests that the transmitter sends a OPERATION request signal and waits for the attached port to indicate that it also is synchronized.

Transition PRX3:PRX1. If any invalid character, or any unexpected signal is received then the receiver returns to the start of the synchronization procedure.

Transition PRX3:PRX4. Once the receiver has received an operation request from the attached transmitter, it begins normal operation.

1 **State PRX4: Receiver.** This is the normal operating state.
2

3 **Transition PRX4:PRX1.** The receiver monitors the received character stream and the synchronization loss procedure
4 determines when it is necessary for the receiver to resynchronize.
5
6

9 10.4.8 Receive actions and conditions

```
11 int char_disparity;      //disparity of most recently received character
12 int disparity_indicator; //disparity indicated by a packet delimiter (e.g. Data prefix)
13 int pkt_err;           //indicates an error while receiving a packet
14 enum bportSymbol localR; //record of decoded received symbol
15 enum bportSymbol last_localR; //record of last localR
16 boolean rx_comma;      //true when the most recently received character was a K28.5
17 boolean pkt;           //when true, port is receiving a packet
18 boolean pkt_prefix;    //when true, port is receiving packet prefix
19 boolean fill_fifo;     //when true, received symbols are buffered in a FIFO
20 boolean train_descrambler; //when true receiver should train scrambler using samples from received signals
21 int rx_rds;            //disparity of received character stream
22 int SYNC_CHECK=17;    //
23 int DESCRAM_TRAIN_CYCLES=22;
24 int POLARITY;         //when 1 polarity of received character stream is deliberately inverted by port. Initialised as 0.
25 int port_error_reg;   //record of number of INVALID codewords received
26 int descram;          //state of receive descrambler
27
```

```

1 void bport_receive_actions() {
2 // Equivalent mostly to 1394-1995 decode_bit routine.
3
4 int pad_count; //keeps track of padding
5 boolean sync_error; //true when port has lost sync., as detected by port_error_monitor()
6 boolean speed_sig_done; //true when all of a valid speed signal has been received
7
8 rx_state=PRX4;
9 while(~sync_error) {
10 sync_error=rx_character();
11 if(localR.tag==DATA || localR.tag==REQUEST || localR.tag==INVALID) rx_rds=char_disparity;
12 if (bspeed_filter()) {
13 bportR[fifo_wr_ptr].tag=SPEED;
14 bportR[fifo_wr_ptr].speed=localR.speed;
15 fifo_wr_ptr=(fifo_wr_ptr==FIFO_DEPTH-1? 0:fifo_wr_ptr++);
16 rx_speed=localR.speed;
17 signal(SPEED_SIGNAL_RECEIVED);
18 pkt_prefix=TRUE;
19 fill_fifo=TRUE;
20 }
21 else {
22 if (localR.tag==DATA) {
23 if(pad_count!=0) pkt_error=TRUE; //note check is not made on first data character
24 pkt=TRUE;
25 pkt_prefix=FALSE;
26 fill_fifo=TRUE;
27 signal(DATA_STARTED);
28 bportR[fifo_wr_ptr]=localR;
29 fifo_wr_ptr=(fifo_wr_ptr==FIFO_DEPTH-1? 0:fifo_wr_ptr++);
30 }
31 else if(localR.tag==CTRL) {
32 if(pkt || pkt_prefix) {
33 if(localR.ctrl==DATA_END && pad_count==0) {
34 if(rx_rds != disparity_indicator || pkt_error==TRUE) {
35 bportR[fifo_wr_ptr].tag=CTRL;
36 bportR[fifo_wr_ptr].ctrl=DATA_END_ERR;
37 rx_rds=disparity_indicator;
38 pkt_error=FALSE;
39 }
40 else bportR[fifo_wr_ptr]=localR;
41 }
42 if(localR.ctrl==DATA_PREFIX && pad_count==0) {
43 if(pkt && (rx_rds != disparity_indicator || pkt_error==TRUE)) {
44 bportR[fifo_wr_ptr].tag=CTRL;
45 bportR[fifo_wr_ptr].ctrl=DATA_PREFIX_ERR;
46 rx_rds=disparity_indicator;
47 pkt_error=FALSE;
48 }
49 else bportR[fifo_wr_ptr]=localR;
50 }
51 else if((localR.ctrl==DATA_END_ERR || localR.ctrl==DATA_PREFIX_ERR) && pad_count=0)
52 bportR[fifo_wr_ptr]=localR;
53 else if(localR.ctrl != SPEEDa) pkt_error=TRUE;
54 }
55 else{
56 if(localR.ctrl==DATA_PREFIX ) { //data prefix preceding packet
57 pkt_prefix=TRUE;
58 rx_rds=disparity_indicator;
59 }
60 if(localR.ctrl != SPEEDa && localR.ctrl != SPEEDb) bportR[fifo_wr_ptr].tag=localR;
61 }
62 }
63 else if(localR.tag==REQUEST || localR.tag==LEGACY_REQUEST) {
64 pad_count=0;
65 fill_fifo=FALSE;
66 bportR[fifo_wr_ptr]=localR;

```

```
1         }
2     else if(localR.tag==INVALID) {
3         pkt=FALSE;
4     }
5     if (fill_fifo) fifo_wr_ptr=(fifo_wr_ptr==FIFO_DEPTH-1? 0:fifo_wr_ptr++);
6     if (pkt || pkt_prefix) {
7         pad_count++;
8         if (pad_count==port_speed/rx_speed) pad_count=0;
9     }
10    }
11 }
12 }
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
```

```

1  boolean rx_character() {
2
3  int char_disparity;
4  int i;
5  int rx_bit;
6  int rx_character;
7  int rx_scram_data;
8  int rx_scram_control;
9  int rx_scram_type;           //scrambled request type value
10 int rx_control_sym;
11 int rx_type;                 //request type value
12 int rx_control_codetable;    //table mapping control codewords to scrambled control values
13 int rx_control_map;         //table mapping control values to control states
14 int disparity_map;          //table mapping received control values to a disparity indicator
15 int request_type_map;       //table mapping request type values to request types
16 int rx_data_codetable;      //table mapping data codeword to data value
17
18 last_localR=localR;
19 rx_comma=FALSE;
20 char_disparity=0;
21 rx_character=0;
22 for (i=0;i<10;i++) {
23     wait_event(PMD_DATA.indication);
24     rx_character << 1;
25     rx_character=rx_character | rx_bit;
26     char_disparity=block_disparity + (rx_bit-0.5);
27 }
28 if(char_disparity>0) char_disparity=1;
29 else if(char_disparity<0) char_disparity=-1;
30
31 if(ismember(character, CONTROL_CHARACTERS)) {
32     rx_scram_control=rx_control_codetable[character];
33     rx_control_sym=rx_scram_control^((descram&0x80)>>4 | (descram&0x20)>>3 | (descram&0x8)>>2 | (descram&0x2)>>1);
34     rx_control_map(rx_control_sym);
35 }
36 else if(((ismember(character, POS_RDS_Dx.0_Dx.4) && rx_rds>0) || (ismember(character, NEG_RDS_Dx.0_Dx.4) && rx_rds<0)) &&
37 ~pkt && ~pkt_prefix){
38     rx_scram_type= rx_data_codetable[character];
39     rx_type=(rx_scram_type ^ descram) && 0x9F;
40     request_type_map(rx_type);
41 }
42 else if(((ismember(character, POS_RDS_DATA_CHARACTERS) && rx_rds>0) || (ismember(character,
43 NEG_RDS_DATA_CHARACTERS) && rx_rds<0)) && (pkt || pkt_prefix) ) {
44     localR.tag=DATA;
45     rx_scram_data= rx_data_codetable[character];
46     localR.data=rx_scram_data ^ descram;
47 }
48 else if((character==POS_RDS_K28.5 && ~pkt && rx_rds>0) || (character==NEG_RDS_K28.5 && ~pkt && rx_rds<0) ) {
49     rx_comma=TRUE;
50     rx_scram_type= rx_data_codetable[D28.0];
51     rx_type=(rx_scram_type ^ descram) & 0x9F;
52     request_type_map(rx_type);
53 }
54 else localR.tag=INVALID;
55
56
57
58 update_descrambler();
59 return(port_error_monitor());           //check that synchronization is OK
60 }
61
62
63
64
65
66

```

```

1  static int invalid_count;
2  static int valid_count;
3
4  boolean port_error_monitor() {
5  //called by rx_character()
6
7  if(localR.tag==INVALID) {
8      port_error_reg++; //increment error counter
9      invalid_count=(invalid_count==4? 4:invalid_count++);
10     valid_count=0;
11 }
12 else if(localR.tag==LEGACY_REQUEST && localR.sig==TRAINING && rx_state == PRX4) {
13     invalid_count=(invalid_count==4? 4:invalid_count++);
14     valid_count=0;
15 }
16 else {
17     if(invalid_count>0) {
18         valid_count=valid_count++;
19         if(valid_count>1) {
20             invalid_count=invalid_count-1; // If second consecutive valid then decrement invalid counter
21             valid_count=0;
22         }
23     }
24 }
25 if (invalid_count==4) return (TRUE); // Sync lost on reaching threshold of four, causing receiver to enter
26 // rx sync lost state.
27 else return(FALSE);
28 }
29
30
31 void rx_control_map(int control_sym) {
32
33 switch(control_sym) {
34     case 0000: localR.tag=CTRL; localR.ctrl=ARBRST_GRANT; break;
35     case 0001: localR.tag=CTRL; localR.ctrl=GRANT; break;
36     case 0010: localR.tag=INVALID; break;
37     case 0011: localR.tag=INVALID; break;
38     case 1101: localR.tag=CTRL; localR.ctrl=SPEEDa; break;
39     case 1000: localR.tag=CTRL; localR.ctrl=SPEEDb; break;
40     case 0100: localR.tag=CTRL; localR.ctrl=SPEEDb; break;
41     case 1001: localR.tag=CTRL; localR.ctrl=DATA_PREFIX; disparity_indicator=1; break;
42     case 0101: localR.tag=CTRL; localR.ctrl=DATA_PREFIX; disparity_indicator=-1; break;
43     case 1010: localR.tag=CTRL; localR.ctrl=DATA_END; disparity_indicator=1; break;
44     case 0110: localR.tag=CTRL; localR.ctrl=DATA_END; disparity_indicator=-1; break;
45     case 0111: localR.tag=CTRL; localR.ctrl=DATA_END_ERR; disparity_indicator=1; break;
46     case 1011: localR.tag=CTRL; localR.ctrl=DATA_END_ERR; disparity_indicator=-1; break;
47     case 1100: localR.tag=INVALID; break;
48     case 1110: localR.tag=CTRL; localR.ctrl=ARBRST; break;
49     case 1111: localR.tag=CTRL; localR.ctrl=BUS_RESET; break;
50 }
51 }
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

```

```

1 void request_type_map(int type) {
2 int async_part, isoch_part;
3 beta_request_code rxrequest;
4
5 async_part=type & 0x7;
6 isoch_part=type & 0x98;
7 if(isoch_part==0) { //must be a legacy request type
8     localR.tag=LEGACY_REQUEST;
9     switch(async_part) {
10        case 000: localR.sig=TRAINING; break;
11        case 001: localR.sig=REQGNT; break;
12        case 010: localR.sig=CHILD_NOTIFY; break;
13        case 011: localR.sig=PARENT_NOTIFY; break;
14        case 100: localR.sig=DISABLE_NOTIFY; break;
15        case 101: localR.sig=SUSPEND; break;
16        case 110: localR.sig=OPERATION; break;
17        case 111: localR.sig=IDLE; break;
18    }
19 }
20 else {
21     switch(async_part) {
22        case 000: localR.tag=INVALID; break;
23        case 001: localR.tag=REQUEST; rxrequest.async=NONE; break;
24        case 010: localR.tag=REQUEST; rxrequest.async=NEXT_SLOW; break;
25        case 011: localR.tag=REQUEST; rxrequest.async=NEXT_FAST; break;
26        case 100: localR.tag=REQUEST; rxrequest.async=CURR_SLOW; break;
27        case 101: localR.tag=REQUEST; rxrequest.async=CURR_FAST; break;
28        case 110: localR.tag=REQUEST; rxrequest.async=CYCLE_START; break;
29        case 111: localR.tag=INVALID; break;
30    }
31     switch(isoch_part) {
32        case 000: localR.tag=INVALID; break;
33        case 001: localR.tag=REQUEST; rxrequest.isoch=NONE; break;
34        case 010: localR.tag=REQUEST; rxrequest.isoch=NEXT_SLOW; break;
35        case 011: localR.tag=REQUEST; rxrequest.isoch=NEXT_FAST; break;
36        case 100: localR.tag=REQUEST; rxrequest.isoch=CURR_SLOW; break;
37        case 101: localR.tag=REQUEST; rxrequest.isoch=CURR_FAST; break;
38        case 110: localR.tag=INVALID; break;
39        case 111: localR.tag=INVALID; break;
40    }
41     localR.req=rxrequest;
42 }
43
44 static int speed_ratio;
45 static int speed_sig_length;
46 static boolean speed_decode;
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

```

```

1  boolean bspeed_filter() {
2
3  if(localR.tag=CTRL && localR.ctrl==SPEEDa) {
4      if(~speed_decode) speed_ratio=2*speed_ratio;    //for every SPEEDa increment the speed ratio...
5      speed_sig_length++;
6  }
7  else if(localR.tag==CTRL && localR.ctrl==SPEEDb) {
8      if((actual_speed=port_speed/speed_ratio) >=100) {
9          localR.speed=actual_speed;    //actual speed is a function of the port speed
10         speed_sig_length++;
11         speed_decode=TRUE;
12         return(TRUE);
13     }
14 }
15 else {
16     speed_ratio=1;
17     speed_sig_length=0;
18     speed_decode=FALSE;
19 }
20 return(FALSE);
21 }
22
23 static int descram_old;    //should be initialised to any value other than 0 on power up.
24
25 void update_descrambler() {
26
27 int i;
28 int descram_new;
29 descram_new=0;
30 if(train_descrambler) descram_old=(descram_old & 0xF7F) | (rx_scram_type & 0x80);
31 for (i=0;i<8;i++) {
32     descram_new <<1;
33     descram_new=descram_new| ((descram_old & 0x700)^(descram & 0x100));
34 }
35 descram_old=descram_new;
36 descram=descram_old & 0xFF;
37 }
38
39 void rx_sync_lost_actions() {
40
41 signal(SYNC_LOST);    //signal transmit channel to send training request.
42 train_character_sync();    //acquire character synchronization - method is beyond scope of this standard.
43
44 }
45
46
47
48 boolean character_sync() {
49 int char_check;
50
51 repeat {
52     rx_character();
53     if(localR.tag==REQUEST || localR.tag==LEGACY_REQUEST) char_check=(char_check==3? 3:char_check++);
54     else char_check=0;
55     if (rx_comma && char_check==3) return (true);
56 }
57 }
58
59
60
61
62
63
64
65
66

```

```

1 void scrambler_sync_actions() {
2 int i;
3 int sync_counter;
4 boolean scrambler_sync;
5 boolean scrambler_sync_error;
6
7
8 sync_counter=0;
9 scrambler_sync=FALSE;
10 scrambler_sync_error=FALSE;
11
12 train_descrambler=TRUE;
13 for (i=0;i<DESCRAM_TRAIN_CYCLES;i++) rx_character;
14 train_descrambler=FALSE;
15
16 //Following routine checks for a valid sequence of either training
17 //or operation request to occur before moving on.
18 while(~scrambler_sync_error && ~scrambler_sync) {
19 rx_character();
20 if(localR.tag==LEGACY_REQUEST && localR.sig==TRAINING) {
21 if (last_localR==localR) { //i.e. part of a consecutive stream
22 sync_counter++;
23 }
24 else sync_counter=0; // i.e. isolated occurrence
25 }
26 else if(localR.tag==LEGACY_REQUEST && localR.sig==OPERATION) {
27 if (last_localR==localR) { //i.e. part of a consecutive stream
28 sync_counter++;
29 }
30 else sync_counter=0; // i.e. isolated occurrence
31 }
32 else if(rx_type== 0xx11111 || rx_type==0xx11001) {
33 sync_counter=0;
34 POLARITY=~POLARITY;
35 }
36 else scrambler_sync_error=TRUE; //If any other request type is received then restart training
37
38 //If there have been SYNC_CHECK consecutive occurrences of TRAINING(request) received,
39 // or if there have been SYNC_CHECK consecutive occurrences of TRAINING(operation)
40 //received, then receiver is considered to be synchronized.
41
42 if(sync_counter==SYNC_CHECK) {
43 scrambler_sync=TRUE; //condition for transition to rx_sync_done state.
44 sync_counter=0;
45 }
46 }
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

```

```

1 void rx_sync_actions() {
2 int sync_counter;
3 boolean remote_sync_lost;
4 boolean rx_sync_error;
5
6 remote_sync_lost=TRUE;
7 rx_sync_error=FALSE;
8 signal(RX_SYNC_DONE); //causes transmitter to send OPERATION request and indicates to receiver that lock has been achieved
9
10 //Following routine checks for a sequence of valid OPERATION request
11 //symbols. This indicates that the connected port is also synchronized and must
12 //occur before receiver transitions to receive state.
13 sync_counter=0;
14 while(~rx_sync_error && remote_sync_lost) {
15 rx_character();
16 if(localR.tag==LEGACY_REQUEST && localR.sig==TRAINING) sync_counter=0;
17 else if(localR.tag==LEGACY_REQUEST && localR.sig==OPERATION) {
18 if (last_localR==localR)sync_counter++;
19 else sync_counter=0;
20 }
21 else rx_sync_error=TRUE;
22 if(sync_counter==SYNC_CHECK) remote_sync_lost=FALSE;
23 }
24 //Once connected port is seen to be synchronized, allow some extra time for remote port to receive
25 //sufficient operation requests.
26 if (~rx_sync_error) {
27 sync_counter=0;
28 for(sync_counter==0; sync_counter<SYNC_CHECK; sync_counter++) rx_character();
29 signal(SYNC_OK); //This causes transmitter to resume normal operation and receiver transitions to receive state.
30 port_error_reg=0; //reset port error register on exiting training
31 }
32 }
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66