

10. Beta mode port specification

What's changed?

changes February 1999:

- 1) *corrected update_scrambler c-code, and added disable_scrambler test mode.*
- 2) *changes to bport_receive_actions to reflect control signals (e.g. ARB_RESET) being buffered so as not to be dropped. Also added some comments!*
- 3) *changed (provisionally) transmit_actions so that ARB_RESET and ARB_RESET_GRANT are always length of an S100 byte. GRANT is length of byte at packet speed (or port speed if isolated from packet).*
- 4) *changed scrambler diagrams to clarify operation.*

changes from 0.16 include:

- 1) *error handling: DATA_END_ERR and DATA_PREFIX_ERR removed, port no longer indicates errors detected in packet (either disparity, code or format errors). DATA_END and DATA_PREFIX rds variants still used to reset rx_rds.*
- 2) *Revised mapping of control symbols and request types to account for new BOSS proposals and standby scheme.*
- 3) *Packet format changed to include S100 speed signal for packets originating from a 1394b Link. Legacy and beta format definitions revised. Other changes & additions to packet formatting.*
- 4) *Removed redundant information re control code comma properties.*
- 5) *Extensive c-code changes*
- 6) *Resynchronization procedure enhanced to clarify port operation immediately after resync has occurred, and to include interaction between port and arb state machine on entry and exit to resync process..*
- 7) *changed terminology for request types: legacy request is no config request, other requests now arbitration requests.*
- 8) *Made all receive actions conditional upon bport_active.*
- 9) *Clarified disparity update rules.*

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.

10.2 Definitions

[tbd]

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

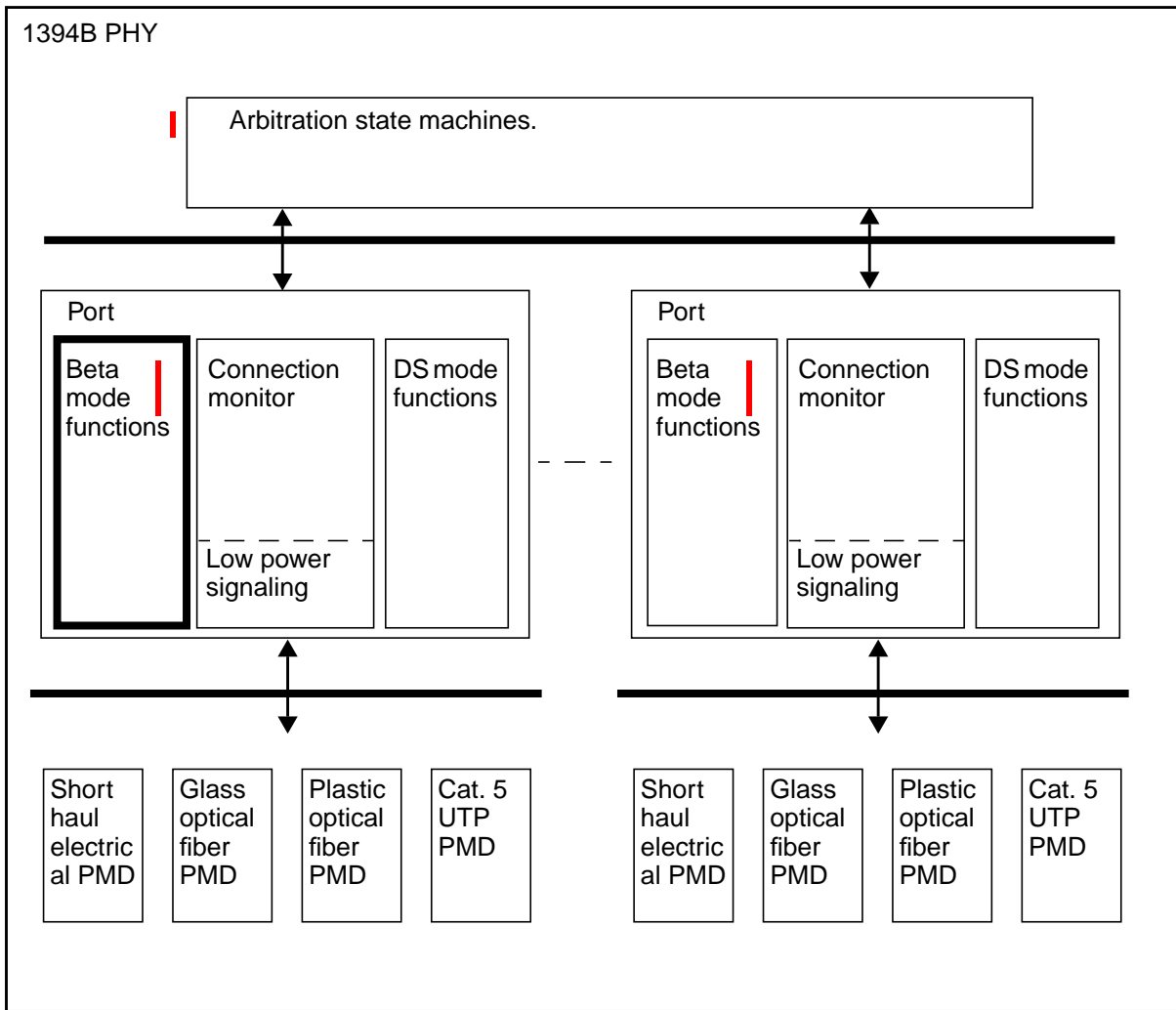


Figure 10-1—Beta mode port interfaces

10.3 Port functions

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

10.3.1 Port functions overview

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

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

Some control information, including packet delimiters and arbitration tokens, is mapped to a four bit control symbol and then also scrambled. The scrambled control symbol is coded using 10-bit characters that are unique with respect to the 10-bit characters used for data. The time taken for a single 4-bit control symbol to be transmitted is therefore equivalent to the time taken for a single byte of data to be transmitted.

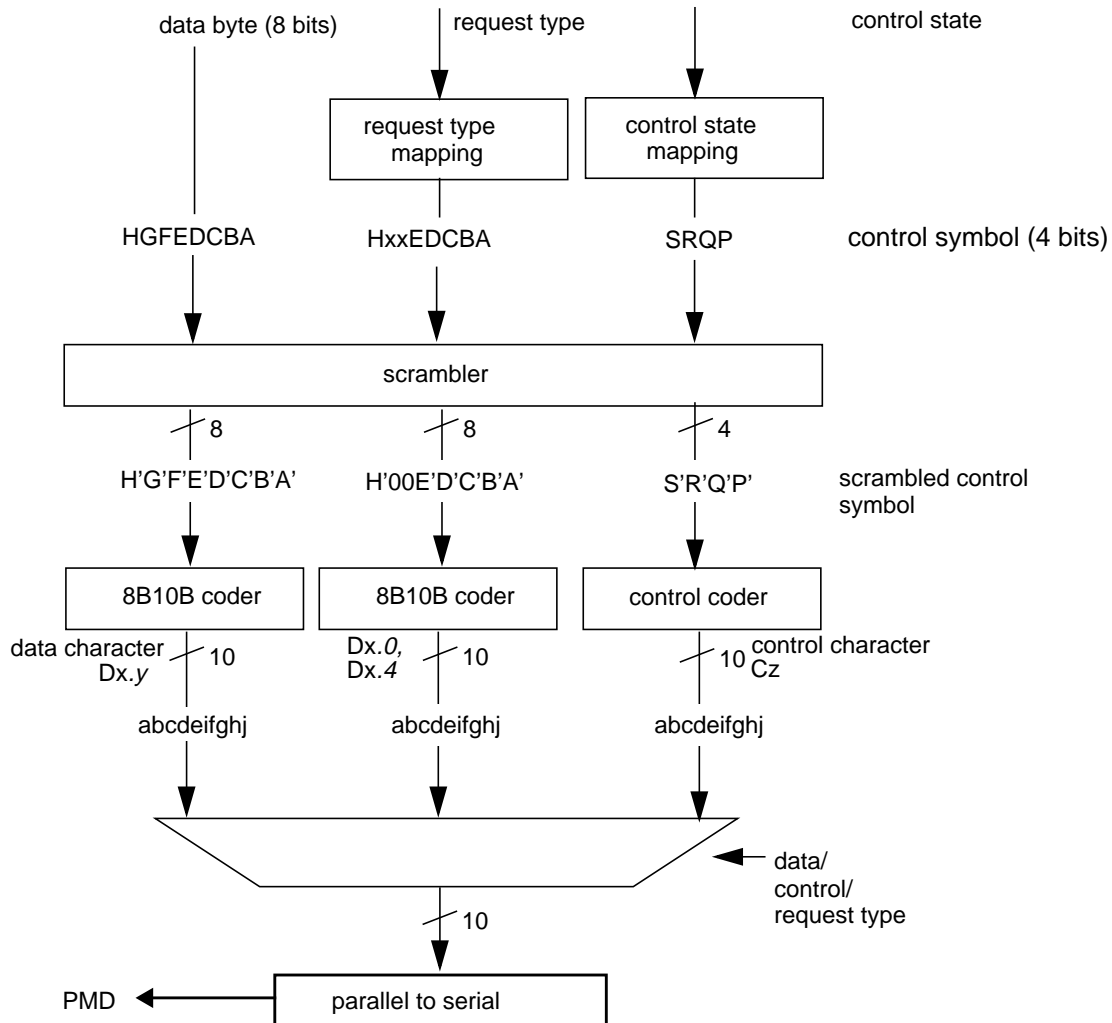


Figure 10-2—Scrambling and coding functions (Note: this figure is informative only)

Other control information is signaled as request types. These are mapped to six active bits of a data byte and then scrambled 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 the data 8B10B characters.

Similar, but reciprocal, functions are provided by the port in the receiving direction. Data characters are identified as representing data or request types according to the context in which they are received i.e. whether they are received within a packet or not.

10.3.2 Naming conventions

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

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

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

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

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

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

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

The scrambled and coded data 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)

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

An unscrambled and uncoded control symbol contains 4 information bits:

S,R,Q,P;

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

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. In the absence of errors the rd will always take a value of +1 or -1 at the end of a 10-bit character.

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
ASYNC_START	0000	
GRANT	0001	
SPEEDb	0010	
SPEEDa	0011	
CYCLE_START_ODD	0100	
ARBRST_ODD	0111	
CYCLE_START_EVEN	1000	
DATA_PREFIX	1001	0101
DATA_END	1010	0110
ARBRST_EVEN	1011	
spare	1100	
spare	1101	
SPEEDc	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 types

Two types of request are defined. An arbitration request combines requests for both isochronous and asynchronous packet transmissions into a single symbol that, after scrambling, is represented by a single 10 bit character. A configuration request contains information pertaining to the configuration of the bus.

10.3.5.1 Arbitration request mapping

Asynchronous and isochronous requests are combined and mapped to a single arbitration 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_EVEN	001
NONE_ODD	010
NEXT_EVEN	011
CURRENT	100
NEXT_ODD	101
BORDER_HIGH	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
ISOCH_ODD	0xx10
ISOCH_EVEN	0xx11
reserved	1xx00
reserved	1xx01
reserved	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 configuration request type. HGFED=0xx00 indicates that a request type is a configuration request.

10.3.5.2 Configuration requests

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

Table 10-4—Configuration request type mapping

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

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.

10.3.6 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:

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

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

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

Scr(k) = Scr(k-9) XOR Scr(k-11)

10.3.6.1 Data scrambling

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

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

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

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

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

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

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

where Scr(k-2) is the scrambler output bit used to scramble the lsb of control symbol immediately preceding the sequence 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 bit of the data byte, A, is exclusive OR'ed with the Scr(k+7).

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

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

10.3.6.2 Request type scrambling

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

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

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

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

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

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

where 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 Scr(k) and the least significant bit of the request type, A, is exclusive OR'ed with the Scr(k+7).

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

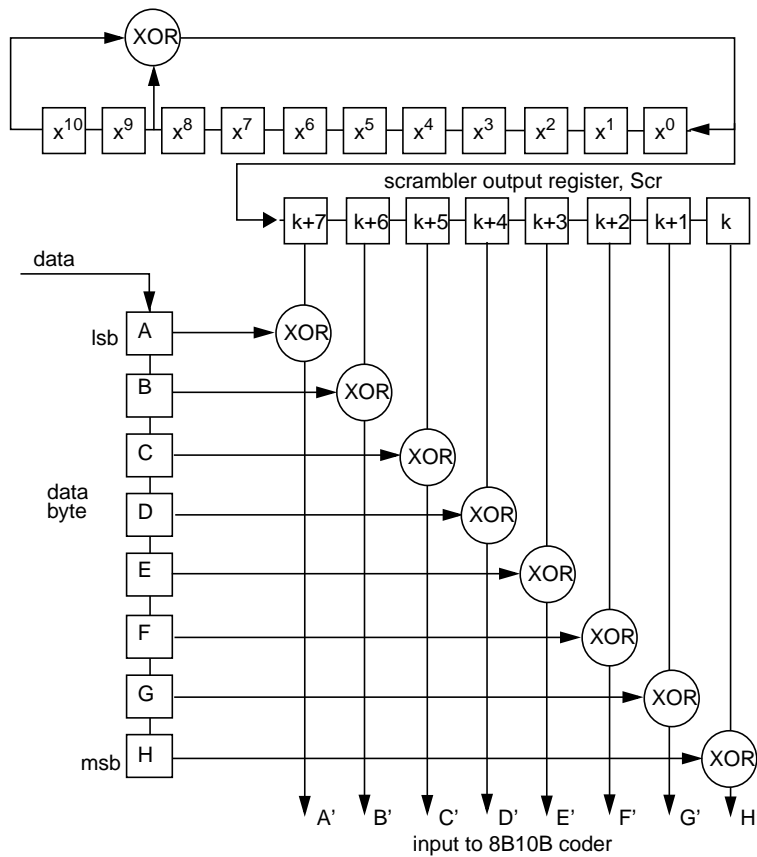


Figure 10-3—Scrambler schematic (data) - informative only

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.

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

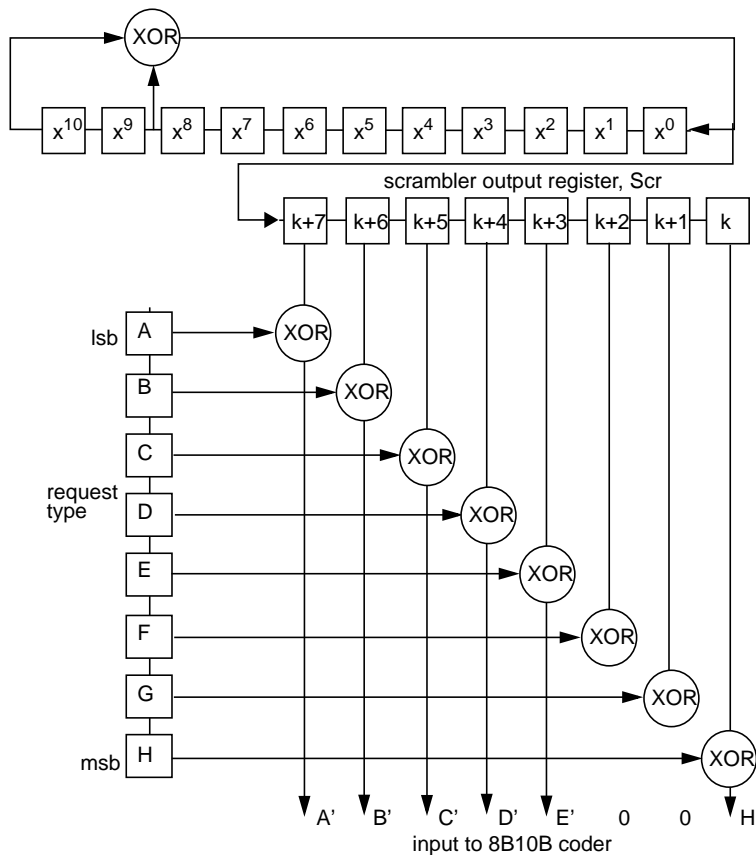


Figure 10-4—Scrambler schematic (request type) - informative only

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 arbitration state machine 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 requests to a set of 10 bit characters which are referred to as data characters. The second code maps control symbols to a second, smaller and distinct set of 10 bit characters which are referred to as control 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.

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

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

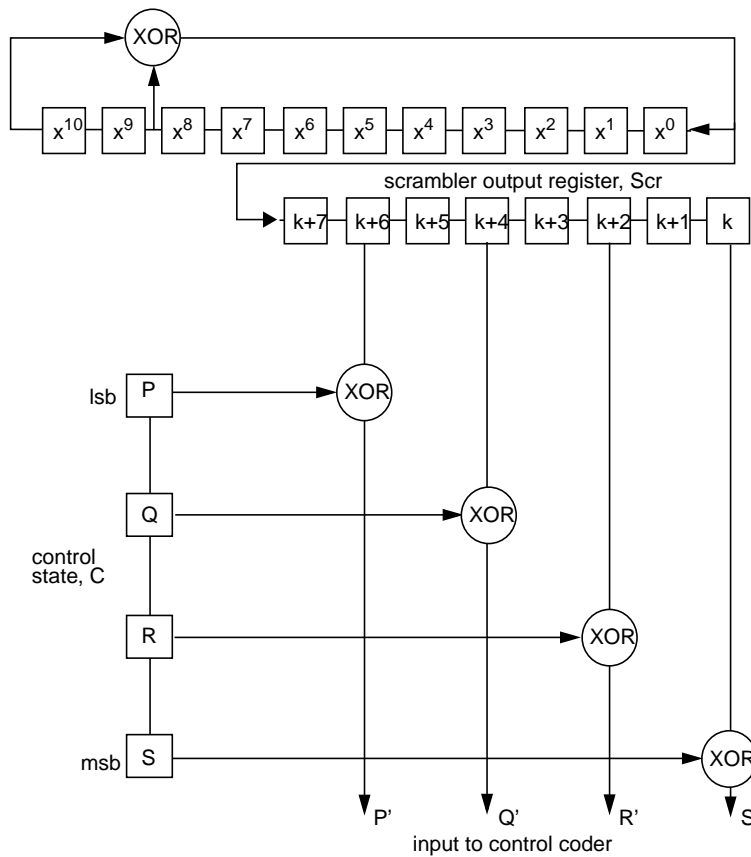


Figure 10-5—Scrambler schematic (control) - informative only

Each valid coded character has a name

$D_{x.y}$ for data characters

x = the 5 bit input value, base 10, for bits A'B'C'D'E'

y = the 3 bit input value, base 10, for bits F'G'H'

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.

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

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

10.3.7.1.2 8B/10B code construction

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 code. This 5B/6B coder produces the 6 bit result depending upon the input bit values and the 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. The running disparity value, as modified by the first coding stage, is also used for coding decisions. Table 10-5 and table 10-6 give the 5B/6B and 3B/4B coding tables, respectively, that are used for coding Dx.y characters.

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		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
D9	10010	100101		rd	D25	10011	100110		rd	
D10	01010	010101			D26	01011	010110			
D11	11010	110100			D27	11011	001001	110110	-rd	
D12	00110	001101		rd	D28	00111	001110		rd	
D13	10110	101100			D29	10111	010001	101110	-rd	
D14	01110	011100			D30	01111	100001	011110		
D15	11110	101000	010111	-rd	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 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
	i	data_table[i][0]	data_table[i][1]		i	data_table[i][0]	data_table[i][1]
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	001110 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
	i	data_table[i][0]	data_table[i][1]		i	data_table[i][0]	data_table[i][1]
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

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
	i	data_table[i][0]	data_table[i][1]		i	data_table[i][0]	data_table[i][1]
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

The data code also provides a number of special characters beyond the 256 needed to encode a byte of data. These are named Kx.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

The K28.5 special character contains a comma sequence. A “comma” indicates the proper byte boundaries and can be used for instantaneous acquisition or verification of byte synchronization. A 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. The comma sequence for the 8B/10B data code is the seven bit sequence “0011111” or “1100000”.

NOTE—This sequence is a comma for the P1394B 8B10B data code, but is not a comma for the P1394b control code.

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

10.3.7.2 Control Coding

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

10.3.7.2.1 Valid Control Code characters

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

Each valid control code character has a name

Cz

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

In the following table,

rd represents the running disparity value from previous bit transmissions

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

Table 10-9—Control Coding

Control character name	S'R'Q'P'	abcdeifghj outputs		rd1
		rd><0	rd>0	
	i	control_table[i]		
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 Dx.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.

10.3.8 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 clause 10.6.2.1.2.

10.3.9.2 Data and control character decoding and error detection

Data and control character decoding is performed using table 10-7 and table 10-9. Based on the receiver’s running disparity, the appropriate columns in these tables 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.

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 Receiver running disparity

The port shall not update the receiver disparity when a Cz character is received. Whenever any other character is received, including an invalid character that is not found in table 10-7, table 10-8 or table 10-9, the receiver disparity shall be updated according to the following rules:

- a) The receiver disparity is updated at the end of each character sub-block, where character sub-blocks are the 6 most significant bits and four least significant bits of the character, i.e. bits a,b,c,d,e,i and bits f,g,h,j.

- b) The receiver disparity is positive at the end of any sub-block if that sub-block contains more 1's than 0's. The receiver disparity is also positive at the end of the 6 bit sub-block 000111 and at the end of the 4 bit sub-block 0011.
- c) The receiver disparity is negative at the end of any sub-block if that sub-block contains more 0's than 1's. The receiver disparity is also negative at the end of the 6 bit sub-block 111000 and at the end of the 4 bit sub-block 1100.
- d) Otherwise the receiver disparity at the end of any sub-block is the same as at the start of that sub-block.

NOTE 1—All sub-blocks with equal number of 1's and 0's leave the receiver disparity unchanged apart from the particular cases of 6 bit sub-blocks 000111, 111000 and 4 bit sub-blocks 0011, 1100.

NOTE 2—All Cz characters have equal number of 1's and 0's and therefore do not change the disparity of the received signal. However, the rules above should not be used to update the receiver disparity when a Cz character is received.

10.3.11 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 port's 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 Packet formats

Editorial - This clause should probably be moved to the arbitration state machine chapter.

The following sections define the formats to be used for packets originating at the local node. Rules for formatting packets being forwarded between ports are defined elsewhere.

10.4.1 General packet format

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

.....<data prefix symbols><speed signal><data prefix symbols><padded data>
[data 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 [symbol]ⁿ indicates that the specified symbol shall be repeated n times.

The packet prefix shall be the sequence of data prefix symbols and the speed signal. The packet end shall consist of data end symbols and arbitration tokens. A data end symbol shall be any of DATA_END or DATA_PREFIX. An arbitration token symbol shall be any of ARBRST_ODD, ARBRST_EVEN, ASYNC_START, GRANT, CYCLE_START_EVEN, CYCLE_START_ODD, or a data end symbol.

10.4.2 Legacy and beta packet formats

For packets at speeds less than S800, two packet formats are defined. The legacy format is used whenever it is known that a packet is to be forwarded through a DS port at some point on the bus (not necessarily at the local PHY). The beta format may be used whenever it is known that a packet does not need to be forwarded through a DS port in order to reach its destination. The beta format is always used for packet speeds of S800 and above.

NOTE—The manner in which a port determines that all ports between a packet's source and destination are operating in beta mode is beyond the scope of this standard.

The legacy packet format includes a speed signal for all S200 and S400 packets. At S100 a speed signal is included in the legacy packet format whenever the packet originates from the local node and the local node does not have a 1394a LINK layer attached.

10.4.3 Legacy format with speed signal

Whenever a legacy format S200 or S400 packet is originated by the local node a speed signal shall be transmitted during the packet prefix. A speed signal shall also be transmitted during the packet prefix of a legacy format S100 packet originating at the local node when that node does not have a 1394a LINK layer. If a legacy format S100 packet is originated by the local node and it has a 1394a LINK layer, then no speed signal shall be generated when that packet is transmitted.

A legacy format packet prefix always contains a number of DATA_PREFIX symbols which indicate the polarity of the running disparity at the start of the packet. A valid legacy format packet is ended by either a number of DATA_END symbols or, in the case of concatenated packets, a number of DATA_PREFIX symbols.

NOTE—Editorial: is concatenation allowed when a 1394b PHY has a 1394a LINK?

The legacy format with speed signal shall be:

$$\text{.....<data prefix symbols>[speed signal][data prefix symbols]^p[\text{padded data}][\text{data end symbol}]^m[\text{arbitration token symbol}]^n\text{.....}$$

where m is the ratio of the port operating speed and the S100 packet speed. n shall be equal to twice the ratio of the port operating speed and the S100 packet speed i.e. $n = 2m$.

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

NOTE—Editorial: should length of DATA_PREFIX also be an integer multiple of S100 character period?

NOTE—These requirements ensure that the duration of the packet prefix is greater than the minimum duration of DATA_PREFIX required on a DS port, MIN_DATA_PREFIX, as specified in the 1394a standard i.e. 140nsecs, and that the duration of the packet end is equal to the minimum duration of DATA_END_TIME required on a DS port, as specified in the 1394a standard i.e. 240nsecs. If the packet is subsequently forwarded to a DS port, the DS port will be able to generate a minimum length DATA_PREFIX and DATA_END without excessive buffering of data. The requirements also ensure that if the packet is subsequently forwarded to a beta mode port operating at a lower speed, that port is able to maintain the duration of the packet prefix and packet end.

10.4.4 Legacy format for S100 packets without speed signal

The packet format for a S100 packet originated by the local node when the local node has a 1394a link layer shall be:

$$\text{.....[data prefix symbols]^p[\text{padded data}][\text{packet end symbol}]^m[\text{arbitration token symbol}]^n\text{.....}$$

where m is equal to the ratio of the port operating speed and the S100 packet speed, p is an integer multiple of m that is at least twice m i.e. $p = i \times m$, for $i=2,3,4,5,\dots$, and n shall be equal to twice the ratio of the port operating speed and the S100 packet speed i.e. $n = 2m$.

NOTE—Editorial: can we restrict DATA_PREFIX duration to be exactly 2m ?

NOTE—These requirements ensure that the duration of the data prefix symbols is greater than the minimum duration of DATA_PREFIX required on a DS port, MIN_DATA_PREFIX, as specified in the 1394a standard i.e. 140nsecs, and that the duration of the packet end symbols and arbitration token symbols is equal to the minimum duration of DATA_END_TIME required on a DS port, as specified in the 1394a standard i.e. 240nsecs. If the packet is subsequently forwarded to a DS port, the DS port will be able to generate a minimum length DATA_PREFIX and DATA_END without excessive buffering of data.

NOTE—Editorial: 1394a specifies MIN_DATA_PREFIX=140nsecs but recommends 180nsecs is used for compatibility - which should 1394b specify? p=2m guarantees 160ns of DATA_PREFIX.

10.4.5 Beta format for all packet speeds

Packets transmitted using the beta format will never be forwarded on a DS port. The packet format rules for such packets are therefore different than for legacy format packets. The beta packet format shall be:

.....[speed signal][data prefix symbol]^m[padded data][packet end symbols]^m[arbitration token symbols]^m.....

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

NOTE—Packet concatenation does not occur at these speeds and so the data prefix variants of the packet end symbols are not required. The requirements ensure that if the packet is subsequently forwarded to a beta mode port operating at a lower speed, that port is able to transmit a speed signal and packet end symbols without buffering data.

10.4.6 Packet transmission examples (informative only)

NOTE—Editorial: these examples have not been checked since the packet format rules were changed, and may therefore be wrong.

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

B_n represents the nth data byte of a packet.

DP represents a DATA_PREFIX symbol.

DE represents a DATA_END symbol.

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

P represents a padding symbol (SPEEDa).

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 DATA_END symbols.

10.4.6.1 Legacy format S100 packet forwarded on an S800 port:

DP^{>13} B₁ PPPPPPP B₂ PPPPPPP B₃.....B_n PPPPPPP DE⁸ARB^{16,24,32,40....}

10.4.6.2 Legacy format S200 packet forwarded on an S800 port:

DP.....DP S_a S_a S_b S_a DP^{12,16,20,24,28..} B₁ PPP B₂ PPP B₃.....B_n PPP DE⁴ARB^{20,24,28,32....}

10.4.6.3 Beta format S800 packet forwarded on S800 port:

S_b DP B₁ B₂ B₃ B₄.....B_n DE ARB

10.4.6.4 Beta format S800 packet forwarded on S1600 port:

S_a S_b DP DP B₁ P B₂ P B₃ P.....B_n P DE DE ARB ARB

10.4.7 Minimum packet spacing

Whenever a node that does not have a 1394a LINK originates a packet it shall ensure that the packet is separated from the previous packet transmitted by that node by at least MIN_PKT_SPACING.

NOTE—Editorial: Is it necessary for this requirement to be conditional on not having a 1394a LINK?

NOTE—A node is not required to maintain MIN_PKT_SPACING between a sequence of packets that it receives from other nodes on the bus.

10.5 Packet forwarding

Editorial - This clause should probably be moved to the arbitration state machine chapter.

Packets received on a port are forwarded on all other ports that are capable of transmitting a packet at the necessary speed, and are compatible with the packet format. Beta format packets are not forwarded on DS ports.

10.5.1 Packets at speeds greater than the port operating speed

When the packet is received whose speed exceeds the operating speed of a port, a speed signal shall not be transmitted on that port. DATA_PREFIX shall be transmitted continuously on that port during the packet, until the packet end symbols are to be sent. The port shall then send the packet end symbols.

10.5.2 Packet forwarding: beta port to beta port

When a correctly formatted packet that has been received on a beta port is subsequently retransmitted on another beta port the duration of the packet prefix, data and packet end shall be the same for the retransmitted packet as for the received packet. The 1394b packet formatting rules ensure that the durations of packet prefix, data and packet end are equal to an integer number of character periods at the operating speed of any port on which the packet might be forwarded.

If an incorrectly formatted packet is received.....[TBD]

10.5.3 Packet forwarding: DS port to beta port

When a packet is received on a DS port and subsequently retransmitted on a beta port it may not be possible to maintain the exact duration of the received packet's packet prefix, data and packet end for the retransmitted packet, due to the quantisation effect of the 8B10B coding. For example, a period of DATA_END lasting 260nsecs that is received on the DS port may be represented by 3 characters lasting a total of only 240nsecs on the beta port.

Furthermore, if a series of packets is received on a DS port, it is necessary for the period of idle between these packets to be represented on the beta port, even when this period may be less than the character period of the beta port (e.g. when the duration of idle is 40nsecs and the beta port is operating at S100 with a character period of 80nsecs). In this situation and others it will be necessary for the duration of some packet components to be reduced on the beta port in order to allow for the period of idle to be signaled and to compensate for the 8B10B quantisation effect. The packet format requirements specified in this section accommodate this by allowing a shorter packet end duration.

The format for packets received on a DS port and retransmitted on a beta port shall be:

.....<data prefix symbols><speed signal>[data prefix symbols]^p[padded data][data end symbol]^m[arbitration token symbol]ⁿ....

where m is the ratio of the port operating speed and the packet speed. n shall be equal to or twice the ratio of the port operating speed and the S100 packet speed i.e. n = m or 2m.

If a speed signal is detected on the receiving DS port, a speed signal shall be transmitted during the packet prefix on the beta port. In this case, the time between completion of transmission of the SPEEDb symbol of the speed signal and completion of transmission of the first byte of data shall be greater than or equal to MIN_DATA_PREFIX, as specified by 1394a. In addition, the duration of the DATA_PREFIX symbols between the speed signal and first byte of data shall be equal to the time required to transmit an integer number of bytes on a port operating at the packet speed, i.e. $p = i \times m$ for integer values of i .

If a speed signal is not detected on the receiving port then a speed signal shall not be transmitted on the beta port. In this case the duration of the data prefix symbols shall be greater than or equal to MIN_DATA_PREFIX as specified by 1394a.

NOTE—This requirement still ensures that the packet prefix and packet end consist of a minimum of two control characters.

10.5.4 Packet forwarding: beta port to DS port

Correctly formatted packets received at a border node beta port and transmitted on a DS port must be retimed using the event information conveyed in the packet delimiters received on the beta port. The border node is not required to reproduce the packet prefix and packet end signals for the same duration as received on the beta port. Rather, the packet prefix and packet end durations must be regenerated according to the requirements of 1394a. Additionally, the border node should ensure that a period of idle satisfying the MIN_IDLE_TIME requirement of 1394a is transmitted between packets on the DS port.

Incorrectly formatted packets received at a border node beta port...[TBD].

10.6 Beta mode port operation

10.6.1 Transmit operations

10.6.1.1 Control transmission

When requested to transmit a control state, the port may be required to stretch the duration of that control signal by transmitting a number of control characters that represent that control state. This ensures that when the control state is subsequently forwarded through ports operating at a lower speed, those ports are able to transmit at least one control character representing that control state. This stretching is controlled by an effective speed parameter that is passed to the port along with the control state. The number of characters transmitted for each control state is equal to the ratio of the port operating speed and this speed parameter, as shown in table 10-12.

Table 10-12—Control stretching formats

Port operating speed	effective speed					
	S100	S200	S400	S800	S1600	S3200
S100	C					
S200	CC	C				
S400	CCCC	CC	C			
S800	CCCCCCCC	CCCC	CC	C		
S1600	CCCCCCCCCCCCCCCC	CCCCCCCC	CCCC	CC	C	
S3200	CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC	CCCCCCCCCCCCCCCC	CCCCCCCC	CCCC	CC	C

NOTE 1—For any particular effective speed, a stretched sequence has constant duration at all port speeds, equal to the time taken to transmit a single character on a port operating at the effective speed.

NOTE 2—C represents a control symbol.

NOTE 3—Table entries represent the information stream prior to scrambling and coding.

NOTE 3—For any particular packet speed, the speed signal has constant duration at all port speeds. At all packet speeds 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.

10.6.1.5 Payload transmission

If the packet speed is less than the port operating speed then data payload is padded with SPEEDa symbols. Each data byte shall be scrambled according to 10.3.6.1 and the scrambled data shall be coded according to 10.3.7.1. The resulting character shall be transmitted immediately before any SPEEDa control symbols. The port compares the value of the packet speed parameter with the port operating speed and determines the number of SPEEDa control symbols to be transmitted 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	DPPPPPPPPPPPPPP	DPPPPPPP	DPPP	DP	D	
S3200	DPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP	DPPPPPPPPPPPPPP	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 payload control symbols are stretched as described in clause 10.6.1.1.

10.6.2 Receive operations

10.6.2.1 Port training

When a beta mode port is initially activated, and also whenever loss of synchronization occurs, the port must initiate a training procedure. During this procedure, the port transmits a training signal to the remote port. This signal causes the remote port to also begin the training procedure, and to transmit a training signal. While receiving a training signal, the port synchronizes itself with the received character stream. The port must acquire bit synchronization and also determine the boundary between 10-bit characters (character synchronization). The port also synchronizes its descrambler with the scrambler of the attached transmitter.

10.6.2.1.1 Loss of synchronization detection procedure

The port determines that it has lost synchronization with the received character stream by monitoring the validity of received characters. The port shall maintain a count of the number of invalid characters received. Whenever an invalid character is received, this count shall be incremented, to a maximum value of four. Whenever two consecutive valid characters are received, the count shall be decremented (to a minimum value of zero). If the count reaches four then the port shall attempt to resynchronize.

For the purposes of loss of synchronization detection, an invalid character shall be any character that does not appear in the appropriate disparity column of table 10-7, table 10-8 or table 10-9. In addition, whilst in the Receive state (i.e. during normal operation) the port receiver shall consider the TRAINING configuration request to be an invalid signal. This provides a means for a remote port to force the receiving port to enter the synchronization procedure.

If the port detects loss of synchronisation during packet reception, it shall make a single indication to the arbitration state machine that a DATA_END control symbol has been received (regardless of the actual received information) and for the remainder of the resynchronisation process it shall indicate to the arbitration state machine that an arbitration request type of (isoch_NONE, async_NONE_EVEN) has been received.

NOTE—Editorial: should it be NONE_EVEN or NONE_ODD, or does it not matter?

10.6.2.1.2 Resynchronization procedure

When the port is initially activated and whenever the port determines that synchronization has been lost, it shall transmit a TRAINING configuration request signal and attempt to resynchronize with the incoming character stream. The TRAINING configuration request signal is treated in the same way as an INVALID symbol by the remote port, and therefore causes the remote port to enter the synchronization procedure.

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

NOTE—When a port first begins resynchronization, it may not necessarily be receiving training signals from the remote port. The requirement for a K28.5 to be preceded by two valid request type characters is necessary to ensure that incorrect character synchronization cannot occur when signals other than the TRAINING or OPERATION configuration requests are received.

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

If during this check an arbitration request type is received that indicates an isochronous request type of ISOCH_EVEN and an asynchronous request type of CYCLE_START (0xx11111), or if an arbitration request type is received that indicates an isochronous request type of ISOCH_EVEN and an asynchronous request type of NONE_EVEN (0xx11001), then a polarity inversion may have occurred in the cable plant, and the port shall subsequently invert all received characters and restart this check.

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

Upon achieving synchronization, the port shall begin to transmit an OPERATION configuration request signal. The port shall continue to transmit an OPERATION configuration request signal until it has received and successfully decoded at least 2*SYNC_CHECK consecutive OPERATION configuration request symbols since the beginning of sending an OPERATION configuration request signal, or has received a control symbol, or is deactivated by the connection management function.

Before resuming normal operation, the port ensures that the remote port is aware of the context of the signals it might receive (i.e. whether Dx.y characters are part of a packet transmission or represent request types).

- a) If the port is required by the arbitration state machine to transmit an arbitration or configuration request type, it shall first transmit a single DATA_END symbol, and then resume normal operation.
- b) If the port is required by the arbitration state machine to transmit any component of a packet prefix or packet payload, the port shall first transmit a valid beta format [S100??] packet prefix and then continue to transmit DATA_PREFIX until requested to transmit a DATA_END symbol, at which point it shall resume normal operation.

- 1 c) If the port is required by the arbitration state machine to transmit any control symbol other than DATA_PREFIX
2 or a speed signal, the port shall transmit that control symbol and then resume normal operation.
3

4 NOTE—The reception of any control symbol other than DATA_PREFIX or a SPEED symbol is sufficient for a remote port to deduce
5 that Dx.y characters immediately following that control symbol represent request types rather than packet data. Similarly the reception
6 of DATA_PREFIX or SPEED symbols is sufficient for a remote port to deduce that Dx.y characters immediately following that symbol
7 represent packet data.
8

9 **10.6.2.2 Control reception**

10 When any valid control character is received, the port shall determine the received control symbol by decoding the control
11 character according to 10.3.9 and descrambling the result of the decoding operation according to clause 10.3.11. In order
12 to be a valid control character, a received character shall appear in table 10-9.
13

14 When control symbols are received within a packet (including the packet end??) the port shall determine the stretching
15 format of the control symbols from the packet speed and port operating speed according to table 10-12. For the first con-
16 trol character in each stretched sequence the port shall determine the received control state according to table 10-1 and
17 indicate this to the arbitration state machine . Other control symbols in the stretched sequence shall not be indicated to the
18 arbitration state machine (i.e. the port shall only indicate the control symbol received in the payload position).
19

20 When control symbols are received outside of a packet, the port shall determine the received control state according to
21 table 10-1. When a change on the received control state is detected (including the beginning of control state reception) the
22 port shall indicate the first occurrence of the new control state to the arbitration state machine. The port is not required, but
23 is permitted, to indicate subsequent contiguous occurrences of the same control state to the arbitration state machine.
24

25 NOTE—This requirement is intended to ensure that isolated occurrences of a control state are never deleted by the port, while allowing
26 redundant control symbols to be deleted when necessary to compensate for any difference between the local port clock and remote port
27 clock frequencies.
28

29 If an invalid character is received during control state reception the port shall maintain the previous control state indica-
30 tion.
31

32 **10.6.2.3 Request type reception**

33 Characters representing request types are distinguished from those representing packet data by the fact that they are
34 received outside of packet boundaries. In order to be a valid request character, a received character shall belong to the set
35 of Dx.0 and Dx.4 characters (x=0,1,...31) and appear in the appropriate column of table 10-7 as determined by the running
36 disparity at the start of reception of the character. The port shall decode all valid request characters according to
37 clause 10.3.9 and descramble the result of the decoding operation according to clause 10.3.11. The request type shall be
38 determined from table 10-2, table 10-3 and table 10-4.
39

40 Any difference between the frequencies of the remote port clock and local port clock will result in the local port occa-
41 sionally needing to either insert or delete request indications. When a request type is inserted it shall be equal to the pre-
42 viously indicated request type.
43

44 If an invalid character is received during request type reception, the port shall maintain the previous request type indica-
45 tion.
46

47 **10.6.2.4 Packet prefix reception**

48 All packets received by a port are preceded by a packet prefix consisting of a number of DATA_PREFIX and possibly a
49 number of SPEEDx (x=a, b or c) symbols. The DATA_PREFIX symbol indicates the state of the remote source port's dis-
50 parity at the start of the packet. Since the receiving port is required to check running disparity during packet reception, it
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

If a valid control payload character is received, the port shall determine the received control state by decoding the control character according to 10.3.9 and descrambling the result of the decoding operation according to 10.3.11. The port shall consider the first control character in a padded sequence to be the payload character. The port shall determine the format of the padded sequence for the packet speed according to table 10-12.

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 packet reception a valid data character is received at a time when the padding format for the packet speed requires a padding symbol to be received, then an error has occurred and the port shall not report that data to the arbitration state machine. If during packet reception the port receives a payload character that is neither a valid data character nor a valid control character, it shall indicate to the arbitration state machine that a data byte of value [0000 0000] was received and consider that packet to be corrupt.

NOTE—A data character is only considered as errored if the invalid character is in a payload position. Errors in padding positions are ignored.

Data reception ends when a DATA_END or DATA_PREFIX is received.

10.6.2.7 Error reporting

Whenever the port detects a coding error, it shall increment the value of the port_error_reg register, to a maximum value of 255. The port_error_reg register shall be an 8 bit read-only PHY register which shall be reset on completion of port training. The port takes no action based on the value of this register. A coding error shall be detected if a character is received which:

- a) does not belong to the set of data characters nor the set of control characters, or
- b) has incorrect disparity, or
- c) is a valid data character, but not Dx.0 or Dx.4, and occurs outside the bounds of a packet, or
- d) is a control character that is not received in the correct context, or
- e) is a valid payload character but is not received in the correct payload position within a padded packet, or
- f) is a valid padding character but is not received in a correct padding position within a padded packet.

NOTE—Editorial: not sure what d means.

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

10.7 Test modes

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

10.7.1 Transmit scrambler disabled mode

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

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

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

10.8 State machines and C code

10.8.1 Functions, variable and constants used in C code (under construction)

The functions and variables used in the C code description of the port functions are described in table 10-16.

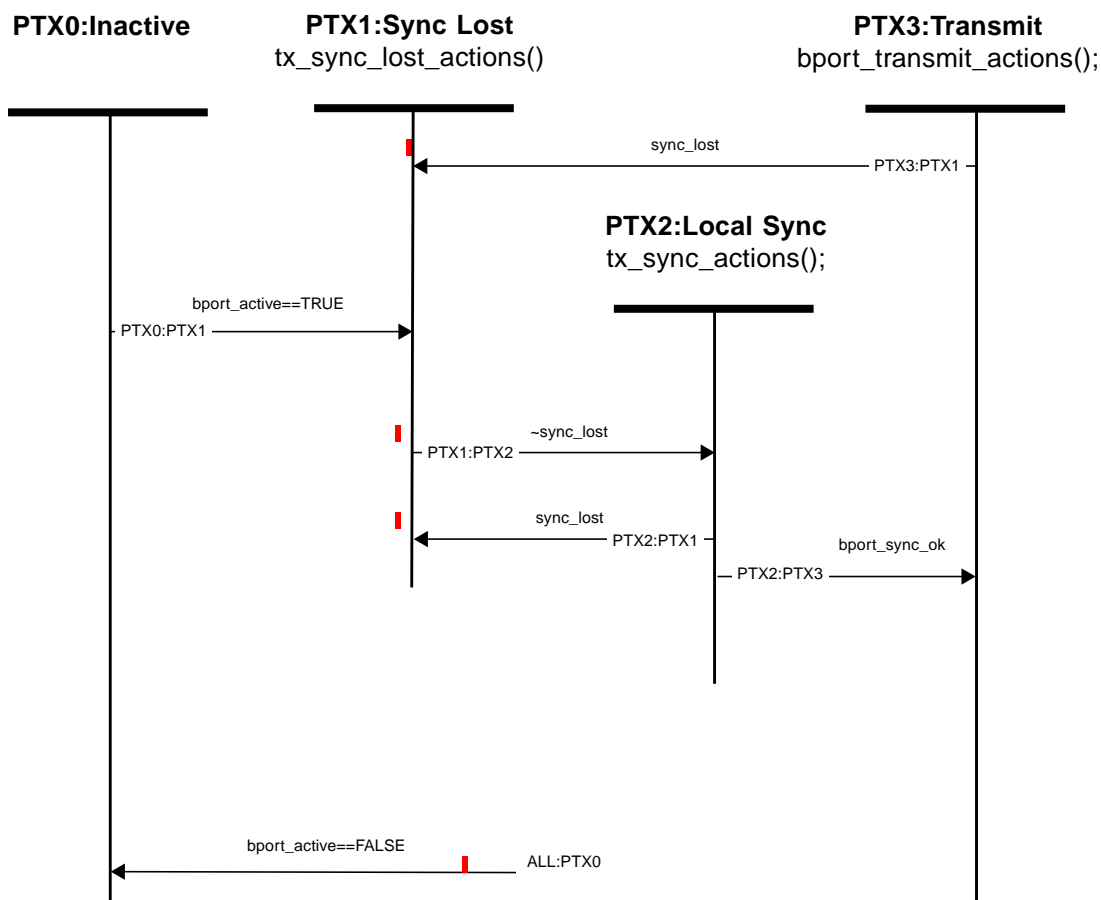
Table 10-16—C code functions and variables for transmit actions

Function or variable name	Description
bport_sync_ok	Indicates to connection management function that a port has achieved synchronisation with peer port.
bportT	Indication from arbitration state machine of required transmitted signal.
localT	Port internal description of transmitted signal.
bportR	Indication to arbitration state machine of received signal.
port_speed	Port operating speed, defined by connection management function.
phy_speed	PHY operating speed.
rx_speed	Speed of a received packet.
min_bus_speed	Speed of slowest DS or beta mode port on bus.
data_byte	Value of 8 bits of data.
disable_scrambler	This per port variable is a register in the Port register map. The port uses this register to determine whether or not to disable operation of the scrambler in the port transmit path.
tx_rds	Record of the running disparity of the transmitted signal. Takes values of 0 or 1. 0 indicates that the running disparity is positive. 1 indicates that the running disparity is negative.
tx_speed	Effective speed at which a signal is to be transmitted. Signals are only transmitted when the effective speed is less than or equal to the port speed. The effective speed determines the degree of padding or stretching that a port applies to data or control signals respectively.
tx_speed_ratio	Ratio of port_speed to tx_speed. This variable is used to control the degree of padding or stretching performed by the port.
pkt_type	Port internal record of the format (legacy or beta) of the currently transmitting packet.
tx_ctrl	This variable is the numerical representation of a control state.
tx_req_type	This variable is the numerical representation of a request type.
scram	This variable represents the 8 bits of the scrambler shift register that are used to scramble each data byte, request type or control code.
tx_scram_ctrl	This variable represents the scrambled control symbol.
tx_scram_data	This variable represents the scrambled data byte.
tx_scram_req	This variable represents the scrambled request type.
control_table	A 16 x 1 array containing the 16 Cz control characters.
data_table	A 256 x 2 array containing the two variants (positive and negative rds) of the 256 Dx.y data characters.
comma_table	A 2 x 1 array containing the two variants (positive and negative starting rds) of the K28.5 comma characters.
tx_rds_table	A 256 x 2 array containing the two variants (positive and negative starting rds) of the finishing rds of the 256 Dx.y data characters.
character_out	This variable represents the 10 bit character to be transmitted.
async_part	This variable is the numerical representation of the asynchronous request type.

Table 10-16—C code functions and variables for transmit actions

Function or variable name	Description
isoch_part	This variable is the numerical representation of the isochronous request type.
scram_new	This variable is used to temporarily store the state of the scrambler shift register during scrambler update.
scram_old	This variable is used to store the state of the scrambler shift register between updates.
control_symbol_map()	Returns a numerical representation for a control state.
req_symbol_map()	Returns a numerical representation for a request type.
config_req_map()	Returns a numerical representation for a configuration request.
update_scrambler()	Updates the state of the transmitter scrambler, performing 8 shift register operations.

10.8.2 Transmit state machine



10.8.2.1 Transmit state machine notes

State PTX0: Inactive. The port is not transmitting.

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

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

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

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

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

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

Transition PTX2:PTX1. If the port receiver signals that it has lost synchronization with the connected port, the transmitter returns to the Sync Lost state.

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 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
int phy_speed;                    //operating speed of PHY
int rx_speed;                      //speed of received packet

int min_bus_speed=0;              //minimum speed port on bus

int data_byte;                    //8 bit data value
boolean disable_scrambler;        // value of disable_scrambler register
int tx_rds;                        //disparity of transmitted character stream
                                   //takes value 0 if rds<0 and value 1 if rds>0
boolean bport_sync_ok;           //true when synchronisation is complete

int scram_old; // represents present state of scrambler
// should be initialized to any value other than zero on power up
int scram; // represents the rightmost 8 bits of scrambler state

```

```
1 void bport_transmit_actions() {
2   int tx_speed; //speed of each packet.
3   int tx_speed_ratio; // number of symbols per byte for given pkt_ speed and port_ speed
4   int i,j;
5   enum pktType pkt_type; //local record of current packet type
6
7   tx_speed=port_speed;
8   while(~sync_lost && bport_active) {
9     if(bportT.speed == DEFAULT)
10      tx_speed = port_speed; //need to define DEFAULT
11    else
12      tx_speed = bportT.speed;
13    tx_speed_ratio = 1<<(port-speed - tx_speed);
14    if(bportT.tag==SPEED ) //port always sends a speed signal if requested to do so
15      tx_speed_signal(tx_speed, bportT.pkt);
16    else if(bportT.tag==DATA) {
17      tx_character(bportT);
18      localT.tag=CTRL;
19      localT.ctrl=SPEEDa;
20      for(i=1; i<=tx_speed_ratio; i++)
21        tx_character(localT); //send padding if required
22    }
23    else {
24      tx_character(bportT);
25      localT = bportT; // remember bportT in case it goes away before stretching is done
26      for(i=1; i<=tx_speed_ratio; i++)
27        tx_character(localT); //stretch as required
28    }
29    // as of 1/3/99, leave arb code to sort out the following, and indicate required stretching through bportT.speed
30    // else if(bportT.tag==CTRL) {
31    //   if (bportT.ctrl==ARB_RESET || bportT.ctrl==ARB_RESET_GRANT)
32    //     //stretch control to length of a S100 byte
33    //     for (i=min_bus_speed; i<=port_speed; i=i+min_bus_speed) tx_character(bportT);
34    //   else
35    //     //stretch control to length of a padded byte at packet speed
36    //     for (i=tx_speed; i<=port_speed; i=i+tx_speed) tx_character(bportT);
37    //   }
38    // else tx_character(bport T); //send configuration requests as and when instructed
39  }
40
41 void tx_sync_lost_actions() {
42
43 tx_rds=0;//initialisation of rds
44 while(bport_active && sync_lost) {
45   bport_sync_ok=FALSE;
46   localT.tag=CONFIG_REQUEST;
47   localT.req=TRAINING;
48   tx_character(localT);
49 }
50 }
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
```

```
void tx_sync_actions() {
2 while(bport_active) {
3     while(sync_error) {
4         localT.tag=ARB_REQUEST;
5         localT.req=OPERATION;
6         tx_character(localT);
7     }
8
9 //if sync completes while bportT indicates packet component to be sent, send a packet prefix followed by null packet.
10 //This captures the idea but needs cleaning up - what if bportT transitions through DATA_END while S100 speed signal is being sent?
11 //Also, if sync completes during a packet in a concatenation, all concatenated packets get overwritten with data prefix.
12
13     if(bportT.tag==DATA || (bportT.tag==CTRL && bportT.ctrl==DATA_PREFIX) || bportT.tag==SPEED) {
14         tx_speed_signal(S100, BETA);
15         while(bportT.tag != CTRL || bportT.ctrl != DATA_END) {
16             localT.tag=CTRL;
17             localT.ctrl=DATA_PREFIX;
18         }
19     }
20
21 //if sync completes while bportT indicates a request to be sent, send a data end first to establish context at far end.
22 if(bportT.tag==ARB_REQUEST || bportT.tag==CONFIG_REQUEST) {
23     localT.tag=CTRL;
24     localT.ctrl=DATA_END;
25     tx_character(localT);
26 }
27 bport_sync_ok=TRUE;
28 }
29
30 //if sync completes while bportT indicates a control to be sent, go straight to normal operation.
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 void tx_character(bportSymbol tx) {
2     int i, j;
3     int tx_ctrl;           //4 bit representation of control state
4     int tx_req;           //8 bit request symbol
5     int scram;            //scrambler state
6     int tx_scram_ctrl;    //scrambled control symbol
7     int tx_scram_data;    //scrambled data byte
8     int tx_scram_req;     //scrambled request type
9     int control_table[16]; //table mapping scrambled control values to control characters (see table 10-9)
10    int data_table[256,2]; //table of data characters (see table 10-7)
11    int comma_table[2];   //table of K28.5 characters (see table 10-8)
12    int character_out;     //10 bit character
13
14    if (tx.tag==DATA) {
15        tx_scram_data=tx.data^scram;           //scramble the data byte
16        character_out=data_table[tx_scram_data, tx_rds]; //lookup character and update disparity
17    }
18    else if(tx.tag==CTRL) {
19        tx_ctrl=control_sym_map(tx.ctrl, tx_rds);
20        tx_scram_ctrl=tx_ctrl^((scram&0x80)>>4 || (scram&0x20)>>3 || (scram&0x8)>>2 || (scram&0x2)>>1);
21                                                    //check scrambler bits****
22        character_out=control_table[tx_scram_ctrl];
23    }
24    else {
25        if(tx.tag==ARB_REQUEST) tx_req=arb_req_symbol_map(tx.req);
26        else if(tx.tag==CONFIG_REQUEST) tx_req=config_req_symbol_map(tx.sig);
27        else {} //shouldn't happen!
28        tx_scram_req=(tx_req^scram) & 0x9F;
29        if(tx.tag==CONFIG_REQUEST && (tx.req==TRAINING || tx.req==OPERATION) && tx_scram_req==0x1C)
30            character_out=comma_table[rds];
31        else character_out=data_table[tx_scram_req, tx_rds]; //lookup character
32    }
33    update_scrambler();
34    if (tx.tag!=CTRL)
35        tx_rds=update_rds(character_out, tx_rds);
36
37    for(i=0;i<10;i++) {
38        PMD_DATA.request(character_out & 0x200); //send msb first
39        character_out <<1;
40        for (j=0;j<phy_speed/port_speed;j++) wait_event(PHY_BIT_CLOCK.indication); //wait for next port bit time
41    }
42 }
43
44 void tx_speed_signal(int tx_speed, pktType pkt_type) {
45     int i, j;
46
47     j=port_speed - tx_speed;
48     localT.tag=CTRL;
49     for(i=0; i<= 1<<(port-speed - tx_speed); i++) {
50         if(i==j && pkt_type==LEGACY)
51             localT.ctrl=SPEEDb; //SPEEDb denotes packet is legacy format
52         else if(i==j && pkt_type=BETA)
53             localT.ctrl=SPEEDc; //SPEEDc denotes packet is beta format
54         else
55             localT.ctrl=SPEEDa;
56         tx_character(localT);
57     }
58 }
59 }
60
61
62
63
64
65
66
    
```

```
1 int control_symbol_map(betaCtrl txctrl, int rds) {
2
3     switch(txctrl) {
4         case ASYNC_START: return(0); break;
5         case GRANT: return(1); break;
6         case SPEEDb: return(2); break;
7         case SPEEDa: return(3); break;
8         case CYCLE_START_ODD: return(4); break;
9         case ARBRST_ODD: return(7); break;
10        case CYCLE_START_EVEN: return(8); break;
11        case DATA_PREFIX: if(rds==1) return(9) else return(5); break;
12        case DATA_END: if(rds==1) return(10) else return(6); break;
13        case ARBRST_EVEN: return(11); break;
14        case SPEEDc: return(14); break;
15        case BUS_RESET: return(15); break;
16    }
17 }
18
19 int arb_req_symbol_map(beta_request_code txrequest) {
20     int async_part, isoch_part;
21
22     switch(txrequest.async) {
23         case NONE_EVEN: async_part=1; break;
24         case NONE_ODD: async_part=2; break;
25         case NEXT_EVEN: async_part=3; break;
26         case CURRENT: async_part=4; break;
27         case NEXT_ODD: async_part=5; break;
28         case BORDER_HIGH: async_part=6; break;
29         case CYCLE_START: async_part=7; break;
30     }
31     switch(txrequest.isoch) {
32         case NONE: isoch_part=1; break;
33         case ISOCH_ODD: isoch_part=2; break;
34         case ISOCH_EVEN: isoch_part=3; break;
35     }
36     return(isoch_part<<3 | async_part);
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 int config_req_symbol_map(configReqType txrequest) {
2
3     switch(txrequest) {
4         case TRAINING: return(0); break;
5         case STANDBY: return(1); break;
6         case CHILD_NOTIFY: return(2); break;
7         case PARENT_NOTIFY: return(3); break;
8         case DISABLE_NOTIFY: return(4); break;
9         case SUSPEND: return(5); break;
10        case OPERATION: return(6); break;
11    }
12 }
13
14 void update_scrambler() {
15
16     int i;
17     int scram_new; // represents next state
18     scram_new = scram_old;
19
20     for (i=0; i<8; i++) {
21         scram_new << 1;
22         scram_new = scram_new | (((scram_old & 0x400) >> 10) ^
23             ((scram_old & 0x100) >> 8));
24         scram_old = scram_new;
25     }
26
27     if (training_disable) scram=0x00; // disables scrambling operation.
28     else scram = scram_old & 0x0FF; // used for XORin with input byte
29
30 }
31
32 int update_rds(int character, int rds) {
33     int i, disparity=0;
34
35     for (i=0; i<6; i++)
36         disparity = disparity + ((character <<i) & 0x200);
37     if(disparity>3 || (character & 0x3F0)==0x070) rds = 1; // rds is positive if 6 msb's are 000111
38     else if(disparity<3 || (character & 0x3F0)==0x380) rds = 0; // rds is negative if 6 msb's are 111000
39
40     disparity=0;
41     for (i=6; i<10; i++)
42         disparity = disparity + ((character <<i) & 0x200);
43     if(disparity>2 || (character & 0xF)==0x3) rds = 1; // rds is positive if 4 lsb's are 0011
44     else if(disparity<2 || (character & 0xF)==0xC) rds = 0; // rds is negative if 4 lsb's are 1100
45
46     return (rds);
47 }
48
49
50
```

10.4.5 Receive state machine

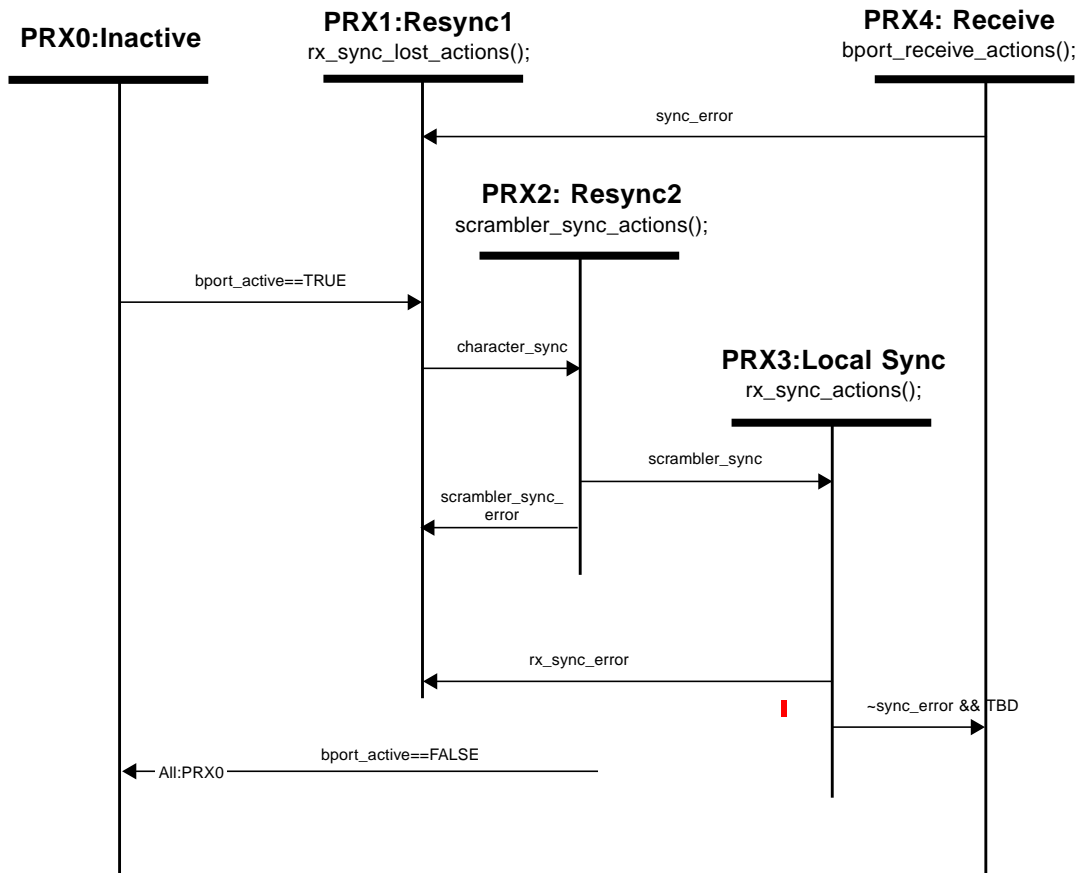
10.4.6 Receive state machine notes

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
7
8

9 10.4.8 Receive actions and conditions

```
10 int disparity_indicator; //disparity indicated by a packet delimiter (e.g. Data prefix)
11 enum bportSymbol localR; //record of decoded received symbol
12 enum bportSymbol last_localR; //record of last localR
13 boolean rx_comma; //true when the most recently received character was a K28.5
14 boolean pkt; //when true, port is receiving a packet
15 boolean pkt_prefix; //when true, port is receiving packet prefix
16
17 boolean train_descrambler; //when true receiver should train scrambler using samples from received signals
18 int SYNC_CHECK=17; //
19 int DESCRAM_TRAIN_CYCLES=22;
20 int polarity; //when 1 polarity of received character stream is deliberately inverted by port. Initialised as 0.
21 int port_error_reg; //record of number of coding errors detected
22 int descram; // rightmost 8 bits of descrambler state
23 boolean sync_error; // when true indicates that receiver error monitor has detected loss synchronisation
24 boolean sync_lost; // when true indicates that receiver is in a resynchronisation state
25 int rx_speed_ratio;
26 pktType rx_pkt_type; //type of packet format received
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 void bport_receive_actions() {
2 // Equivalent mostly to 1394-1995 decode_bit routine.
3 pkt=FALSE;
4 pkt_prefix=FALSE;
5 int pad_count=0; //keeps track of padding
6 boolean fill_fifo; //when true, received symbols are buffered in a FIFO
7
8 while(bport_active && ~sync_error) {
9 // rx_character() has been called at end of rx_sync_actions()
10 // If received character completes a speed signal, then a packet prefix is arriving.
11 // Speed signal is placed in fifo, and fifo update is turned on.
12
13     if (bspeed_filter()) {
14         bportR[fifo_wr_ptr].tag=SPEED;
15         bportR[fifo_wr_ptr].speed=rx_speed;
16         bportR[fifo_wr_ptr].pkt=rx_pkt_type;
17         fifo_wr_ptr=(fifo_wr_ptr==FIFO_DEPTH-1? 0:fifo_wr_ptr++);
18         signal(SPEED_SIGNAL_RECEIVED);
19         pkt_prefix=TRUE;//in case no data prefix was already received
20         fill_fifo=TRUE;
21     }
22     else {
23
24 // If data is received then the packet has started, and packet prefix is done.
25 // Position of data relative to padding is checked.
26 // Fifo update is turned on (if not already).
27
28         if (localR.tag==DATA) {
29             if (pad_count==0) {
30                 pkt=TRUE;
31                 pkt_prefix=FALSE;
32                 fill_fifo=TRUE;
33                 signal(DATA_STARTED);
34                 bportR[fifo_wr_ptr]=localR;
35             }
36             else port_error_reg++;//increment error counter, data in wrong position
37         }
38         else if(localR.tag==CTRL) {
39             if(localR.ctrl==DATA_PREFIX) {
40                 if(pkt) {
41                     pkt_prefix=TRUE; //concatenated packet
42                     pkt=FALSE;
43                     fifo_wr_ptr=(fifo_wr_ptr==FIFO_DEPTH-1? 0:fifo_wr_ptr++);
44                     //buffer this first occurrence of data prefix after packet
45                     fill_fifo=FALSE;//packet prefix is not buffered until speed signal is done
46                 }
47                 else if(~pkt_prefix) {
48                     pkt_prefix=TRUE;
49                     fill_fifo=FALSE; // in case buffering has been turned on for isolated control symbols
50                     //packet prefix is not buffered until speed signal is done.
51                 }
52                 bportR[fifo_wr_ptr]=localR;
53             }
54             else if(localR.ctrl==SPEEDa) {
55                 if(pkt && pad_count==0) { //Padding in wrong place
56                     port_error_reg++; //increment error counter
57                     bportR.tag=DATA; //substitute 0x00 data
58                     bportR.data=0x00;
59                 }
60             }
61         }
62         else {
63             pkt=FALSE;
64             pkt_prefix=FALSE;
65             fill_fifo=FALSE;
66             bportR[fifo_wr_ptr]=localR;
67             // Buffer first occurrence of each new control signal...

```

```

        if (last_localR != localR) fifo_wr_ptr=(fifo_wr_ptr==FIFO_DEPTH-1? 0:fifo_wr_ptr++);
    }
}

else if(localR.tag==ARB_REQUEST || localR.tag==CONFIG_REQUEST) {
    pad_count=0;
    fill_fifo=FALSE; // request types are not buffered in fifo.
    bportR[fifo_wr_ptr]=localR;
    pkt=FALSE;
}
else if(localR.tag==INVALID) {
    port_error_reg++; //increment error counter
    if(pkt && pad_count==0) { //if a packet payload character, substitute 0x00
        bportR.tag=DATA;
        bportR.data=0x00;
    }
}

if (fill_fifo)
    fifo_wr_ptr=(fifo_wr_ptr==FIFO_DEPTH-1? 0:fifo_wr_ptr++);
if (pkt || pkt_prefix) {
    pad_count++;
    if (pad_count==rx_speed_ratio) pad_count=0;
}
}
sync_error=rx_character();
}
}

```

```

1  boolean rx_character() {
2
3  int i;
4  int rx_rds;
5  int rx_bit;
6  int character_in;
7  int rx_scram_data;
8  int rx_scram_control;
9  int rx_scram_type;           //scrambled request type value
10 int rx_control;
11 int rx_type;                 //request type value
12 int rx_control_map;         //table mapping control values to control states
13 int disparity_map;         //table mapping received control values to a disparity indicator
14 int request_type_map;      //table mapping request type values to request types
15
16 last_localR=localR;
17 rx_comma=FALSE;
18 character_in=0;
19 for (i=0;i<10;i++) {
20     wait_event(PMD_DATA.indication);
21     character_in << 1;
22 }
23
24 if((rx_scram_control=rx_control_decode(character_in))>=0) {
25     rx_control=rx_scram_control^((descram&0x80)>>4 | (descram&0x20)>>3 | (descram&0x8)>>2 | (descram&0x2)>>1);
26                                     //check scrambler bits****
27     rx_control_map(rx_control);
28 }
29 else if(((rx_scram_type=rx_reqtype_decode(character_in,rx_rds)) >=0) && ~pkt && ~pkt_prefix){
30     rx_type=(rx_scram_type ^ descram) && 0x9F;
31     request_type_map(rx_type);
32 }
33 else if(((rx_scram_data=rx_data_decode(character_in,rx_rds))>=0) && (pkt || pkt_prefix) ) {
34     localR.tag=DATA;
35     localR.data=rx_scram_data ^ descram;
36 }
37 else if((rx_scram_type=rx_comma_decode(character_in, rx_rds))>=0 && ~pkt ) {
38     rx_comma=TRUE;
39     rx_type=(rx_scram_type ^ descram) & 0x9F;
40     request_type_map(rx_type);
41 }
42 else
43     localR.tag=INVALID;
44
45 update_descrambler();
46 if (localR.tag != CTRL)
47     rx_rds=update_rds(character_in, rx_rds);
48     //control characters are all neutral disparity.
49     //update_rds() should never be used for control characters.
50 if (localR.tag != CTRL && (localR.ctrl==DATA_END || localR.ctrl==DATA_PREFIX))
51     rx_rds=disparity_indicator;
52     //rds is always reset when a data prefix or data end symbol is received.
53 return(port_error_monitor());           //check that synchronization is OK
54 }
55
56
57
58
59
60
61
62
63
64
65
66

```

```

1  boolean port_error_monitor() {
2  static int invalid_count;
3  static int valid_count;
4
5  if(localR.tag==INVALID) {
6      invalid_count=(invalid_count==4? 4:invalid_count++);
7      valid_count=0;
8  }
9  else if(localR.tag==CONFIG_REQUEST && localR.sig==TRAINING && rx_state == ~sync_error) {
10     invalid_count=(invalid_count==4? 4:invalid_count++);
11     valid_count=0;
12 }
13 else {
14     if(invalid_count>0) {
15         valid_count=valid_count++;
16         if(valid_count>1) {
17             invalid_count=invalid_count-1; // If second consecutive valid then decrement invalid counter
18             valid_count=0;
19         }
20     }
21 }
22 if (invalid_count==4)
23     return (TRUE); // Sync lost on reaching threshold of four, causing receiver to enter
24 // rx sync lost state.
25 else
26     return(FALSE);
27 }
28
29 void rx_control_map(int rx_ctrl) {
30
31 switch(rx_ctrl) {
32     case 0000: localR.tag=CTRL; localR.ctrl=ASYNC_START; break;
33     case 0001: localR.tag=CTRL; localR.ctrl=GRANT; break;
34     case 0010: localR.tag=CTRL; localR.ctr=SPEEDb; break;
35     case 0011: localR.tag=CTRL; localR.ctr=SPEEDa; break;
36     case 0100: localR.tag=CTRL; localR.ctrl=CYCLE_START_ODD; break;
37     case 0101: localR.tag=CTRL; localR.ctrl=DATA_PREFIX; disparity_indicator=0;break;
38     case 0110: localR.tag=CTRL; localR.ctrl=DATA_END; disparity_indicator=0;break;
39     case 0111: localR.tag=CTRL; localR.ctrl=ARBRST_ODD; break;
40     case 1000: localR.tag=CTRL; localR.ctrl=CYCLE_START_EVEN; break;
41     case 1001: localR.tag=CTRL; localR.ctrl=DATA_PREFIX; disparity_indicator=1; break;
42     case 1010: localR.tag=CTRL; localR.ctrl=DATA_END; disparity_indicator=1; break;
43     case 1011: localR.tag=CTRL; localR.ctrl=ARBRST_EVEN; break;
44     case 1100: localR.tag=INVALID; break;
45     case 1101: localR.tag=INVALID; break;
46     case 1110: localR.tag=CTRL; localR.ctrl=SPEEDc; break;
47     case 1111: localR.tag=CTRL; localR.ctrl=BUS_RESET; break;
48 }
49 }
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 configuration request type
8     switch(async_part) {
9         case 000: localR.tag=CONFIG_REQUEST; localR.sig=TRAINING; break;
10        case 001: localR.tag=CONFIG_REQUEST; localR.sig=STANDBY; break;
11        case 010: localR.tag=CONFIG_REQUEST; localR.sig=CHILD_NOTIFY; break;
12        case 011: localR.tag=CONFIG_REQUEST; localR.sig=PARENT_NOTIFY; break;
13        case 100: localR.tag=CONFIG_REQUEST; localR.sig=DISABLE_NOTIFY; break;
14        case 101: localR.tag=CONFIG_REQUEST; localR.sig=SUSPEND; break;
15        case 110: localR.tag=CONFIG_REQUEST; localR.sig=OPERATION; break;
16        case 111: localR.tag=INVALID; break;
17    }
18 }
19 else {
20     switch(async_part) {
21         case 000: localR.tag=INVALID; break;
22         case 001: localR.tag=ARB_REQUEST; rxrequest.async=NONE_EVEN; break;
23         case 010: localR.tag=ARB_REQUEST; rxrequest.async=NONE_ODD; break;
24         case 011: localR.tag=ARB_REQUEST; rxrequest.async=NEXT_EVEN; break;
25         case 100: localR.tag=ARB_REQUEST; rxrequest.async=CURRENT; break;
26         case 101: localR.tag=ARB_REQUEST; rxrequest.async=NEXT_ODD; break;
27         case 110: localR.tag=ARB_REQUEST; rxrequest.async=BORDER_HIGH; break;
28         case 111: localR.tag=ARB_REQUEST; rxrequest.async=CYCLE_START; break;
29     }
30     switch(isoch_part) {
31         case 000: localR.tag=INVALID; break;
32         case 001: localR.tag=ARB_REQUEST; rxrequest.isoch=NONE; break;
33         case 010: localR.tag=ARB_REQUEST; rxrequest.isoch=ISOCH_ODD; break;
34         case 011: localR.tag=ARB_REQUEST; rxrequest.isoch=ISOCH_EVEN; break;
35         case 100: localR.tag=INVALID; break;
36         case 101: localR.tag=INVALID; break;
37         case 110: localR.tag=INVALID; break;
38         case 111: localR.tag=INVALID; break;
39     }
40     localR.req=rxrequest;
41 }
42
43
44
45
46 boolean bspeed_filter() {
47 static boolean speed_decode; //true while speed signal is being received and Sb or Sc has been received.
48
49 if(localR.tag=CTRL && localR.ctrl==SPEEDa)
50     if(~speed_decode)
51         rx_speed_ratio=rx_speed_ratio++; //for every SPEEDa increase the speed ratio...
52
53 else if(localR.tag==CTRL && localR.ctrl==SPEEDb) {
54     rx_speed=port_speed-rx_speed_ratio; //actual speed is a function of the port speed
55     rx_pkt_type=LEGACY;//SPEEDb indicates a legacy packet.
56     speed_decode=TRUE;
57     return(TRUE);
58 }
59 else if(localR.tag==CTRL && localR.ctrl==SPEEDc) {
60     rx_speed=port_speed-rx_speed_ratio; //actual speed is a function of the port speed
61     rx_pkt_type=BETA;//SPEEDc indicates a beta packet.
62     speed_decode=TRUE;
63     return(TRUE);
64 }
65 else {
66     rx_speed_ratio=0;

```

```

1     speed_decode=FALSE;
2     }
3     return(FALSE);
4     }
5
6     void update_descrambler() {
7     static int descram_old; // represents present state of descrambler
8     int i;
9     int descram_new; // next state
10
11     if (train_descrambler)
12         descram_old = (descram_old & 0xF7F) | (rx_scram_type & 0x080);
13     // replacing rx_scram_type bit 7 (from the right) into descram_old
14     // when training descrambler
15
16     descram_new = descram_old;
17
18     for (i=0; i<8; i++) {
19         descram_new << 1;
20         descram_new = descram_new | (((descram_old & 0x400) >> 10) ^
21             ((descram_old & 0x100) >> 8));
22         descram_old = descram_new;
23     }
24
25     descram = descram_old & 0x0FF; // used for XOR with input byte
26
27     }
28
29     void rx_sync_lost_actions() {
30
31     sync_lost=TRUE; //signal transmit channel to send training request.
32     if(pkt || pkt_prefix) { //make sure arb state machine gets a packet end indication
33         bportR.tag=CTRL;
34         bportR.ctrl=DATA_END;
35     }
36     //is a fifo pointer increment needed here??
37     rx_character();
38     bportR.tag=ARB_REQ;
39     bportR.req.async=NONE_EVEN;
40     bportR.req.isoch=NONE;
41
42     pkt=FALSE;
43     pkt_prefix=FALSE;
44     while (~character_sync() && bport_active)
45         train_character_sync(); //acquire character synchronization - method is beyond scope of this standard.
46
47     }
48
49     boolean character_sync() {
50     static int char_check;
51
52     rx_character();
53     if(localR.tag==ARB_REQUEST || localR.tag==CONFIG_REQUEST)
54         char_check=(char_check==3? 3:char_check++);
55     else
56         char_check=0;
57     if (rx_comma && char_check==3)
58         return (TRUE);
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 sync_counter=0;
8 scrambler_sync=FALSE;
9 scrambler_sync_error=FALSE;
10
11 train_descrambler=TRUE;
12 i=0;
13 while(bport_active && i<DESCRAM_TRAIN_CYCLES) {
14     rx_character();
15     i++;
16 }
17 train_descrambler=FALSE;
18                                     //Following routine checks for a valid sequence of either training
19                                     //or operation request to occur before moving on.
20 while(bport_active && ~scrambler_sync_error && ~scrambler_sync) {
21     rx_character();
22     if(localR.tag==CONFIG_REQUEST && localR.sig==TRAINING) {
23         if (last_localR==localR) //i.e. part of a consecutive stream
24             sync_counter++;
25         else
26             sync_counter=0; // i.e. isolated occurrence
27     }
28     else if(localR.tag==CONFIG_REQUEST && localR.sig==OPERATION) {
29         if (last_localR==localR) //i.e. part of a consecutive stream
30             sync_counter++;
31         else
32             sync_counter=0; // i.e. isolated occurrence
33     }
34     else if(rx_type== 0xx11111 || rx_type==0xx11001) {
35         sync_counter=0;
36         polarity = ~polarity;
37     }
38     else
39         scrambler_sync_error=TRUE; //If any other request type is received then restart training
40
41     //If there have been SYNC_CHECK consecutive occurrences of TRAINING(request) received,
42     // or if there have been SYNC_CHECK consecutive occurrences of TRAINING(operation)
43     //received, then receiver is considered to be synchronized.
44
45     if(sync_counter==SYNC_CHECK) {
46         scrambler_sync=TRUE; //condition for transition to rx_sync_done state.
47         sync_counter=0;
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 sync_lost = FALSE; //causes transmitter to send OPERATION request and indicates to receiver that lock has been achieved
7 remote_sync_lost=TRUE; //receiver does not assume peer node is synchronised until following checks are complete
8 rx_sync_error=FALSE;
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(bport_active && ~rx_sync_error && remote_sync_lost) {
15 rx_character();
16 if(localR.tag==CONFIG_REQUEST && localR.sig==TRAINING) sync_counter=0;
17 else if(localR.tag==CONFIG_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 to know local sync is done. If remote port sends a control signal then
26 //it must know local sync is done. At same time, local port may determine context of signals.
27 if (bport_active && ~rx_sync_error) {
28 sync_counter=0;
29 while(bport_active && sync_counter<SYNC_CHECK && localR.tag!=CTRL) {
30 sync_counter++;
31 rx_character();
32 }
33 port_error_reg=0; //reset port error register on exiting training
34 sync_error = FALSE; //This causes transmitter to resume normal operation and receiver transitions to receive state.
35 while(bport_active && localR.tag==CTRL && localR.ctrl==DATA_END)
36 rx_character(); //Do not report DATA_END symbols that have been sent for context info only
37 }
38 }
39
40 int rx_control_decode(int character_in) {
41 int control_table[16]; //table mapping control characters to scrambled control values - see table 10-9
42 int i;
43
44 for (i=0; i<16; i++) {
45 if (character_in==control_table[i]) return(i); // check for a Cz
46 }
47 return(-1);
48 }
49
50 int rx_reqtype_decode(int character_in, int rds) {
51 int data_table[256][2]; //table mapping data character to data value - see table 10-7
52 int i;
53
54 for (i=0; i<32; i++) {
55 if (character_in==data_table[i][rds]) return(i); //check Dx.0's
56 }
57 for (i=128; i<160; i++) {
58 if (character_in==data_table[i][rds]) return(i); // check Dx.4's
59 }
60 return(-1);
61 }
62
63
64
65
66

```

