Proposal for PCS Transmit Using 8b6T line code

100 Mb/s Long-Reach Single Pair Ethernet Task Force

IEEE 802.3dg

Philip Curran
Brian Murray

Interaction with MII

- The PCS transmit function generates code-groups of the following form

  $\{A_n, B_n, C_n, D_n, E_n, F_n\}$

  Each code-group is a 6-tuple of ternary symbols
- The index, n, may be viewed as the value of a code-group counter or byte counter
- For each n, the PCS transmit function consume 2 samples of each MII transmit signal. We identify these using indices 2n and 2n+1 like this

  TXD_{2n}

  TXD_{2n+1}
- We will implement PCS data transmission enabling as in Figure 146-4
  - We will use the names tx_enable and tx_error rather than tx_enable_mii and tx_error_mii

Errors Flagged by the MAC

- Per clause 22.2.1.6, the MAC may request that the PHY corrupt a frame
  - This is done by asserting TX_ER while TX_EN is high
- In clause 146 this situation is handled in the PCS transmit state diagram of Figure 146-5
  - A different end-of-stream delimiter identifies a frame that is to be corrupted
  - This approach was necessitated due to a shortage of special code-groups
- Previous clauses, such as clause 40, have used special symbols to propagate errors that are flagged by the MAC
  - We will use this approach
- The variable xmt_error will request the transmission of a special code-group

  $xmt\_error = (tx\_enable_{2n} \,\&\, tx\_error_{2n}) \,|\, (tx\_enable_{2n+1} \,\&\, tx\_error_{2n+1})$

  - An error that is flagged by the MAC for one nibble is expanded to a byte which is consistent with the intended use of TX_ER as described in clause 22.2.1.6

Scrambler and Generation of Bits X_n and Y_n

- The side-stream scramblers for the LEADER and FOLLOWER will use the same generator polynomials as specified in clause 40.3.1.3.1

$$g_L(x) = 1 + x^{13} + x^{33}$$

$$g_F(x) = 1 + x^{20} + x^{33}$$

- A realization of each of these scramblers using a linear feedback shift register (LFSR) is shown in Figure 40-6
- Bits X_n and Y_n are generated from the bits of the scrambler LFSR as in clause 40.3.1.3.2

$$X_n = Scr_n[4] \hat{} Scr_n[6]$$

$$Y_n = Scr_n[1] \hat{} Scr_n[5]$$

Generation of Bits $Sy_n[3:0]$

- The 4 bits $Sy_n[3:0]$ are generated from the bit $Scr_n[0]$ using the following generating polynomial

$$g(x) = x^3 \wedge x^8$$

- The equations for bits $Sy_n[3:0]$ are as follows

$$Sy_n[0] = Scr_n[0]$$

$$Sy_n[1] = g(Scr_n[0]) = Scr_n[3] \wedge Scr_n[8]$$

$$Sy_n[2] = g^2(Scr_n[0]) = Scr_n[6] \wedge Scr_n[16]$$

$$Sy_n[3] = g^3(Scr_n[0]) = Scr_n[9] \wedge Scr_n[14] \wedge Scr_n[19] \wedge Scr_n[24]$$

- This is as was specified in clause 40.3.1.3.2

Generation of Bits $Sx_n[3:0]$

- The 4 bits $Sx_n[3:0]$ are generated from the bit X_n using the same generating polynomial, $g(x)$, as was used to generate bits $Sy_n[3:0]$
- The equations for bits $Sx_n[3:0]$ are as follows

$$Sx_n[0] = X_n = Scr_n[4]\wedge Scr_n[6]$$

$$Sx_n[1] = g(X_n) = Scr_n[7]\wedge Scr_n[9]\ \wedge Scr_n[12]\wedge Scr_n[14]$$

$$Sx_n[2] = g^2(X_n) = Scr_n[10]\wedge Scr_n[12]\ \wedge Scr_n[20]\wedge Scr_n[22]$$

$$Sx_n[3] = g^3(X_n) = Scr_n[13]\wedge Scr_n[15]\wedge Scr_n[18]\wedge Scr_n[20]\wedge$$
$$Scr_n[23]\wedge Scr_n[25]\wedge Scr_n[28]\wedge Scr_n[30]$$

- Again, this is as was specified in clause 40.3.1.3.2

Generation of Bits $Sg_n[1:0]$

- The 2 bits $Sg_n[1:0]$ are generated from the bit Y_n using the same generating polynomial, $g(x)$, as was used to generate bits $Sy_n[3:0]$
- The equations for bits $Sg_n[1:0]$ are as follows

$$Sg_n[0] = Y_n = Scr_n[1]\text{^}Scr_n[5]$$

$$Sg_n[1] = g(Y_n) = Scr_n[4]\text{^}Scr_n[8]\ \text{^}Scr_n[9]\text{^}Scr_n[13]$$

- This is as was specified in clause 40.3.1.3.2 except that we only generate 2 bits

Generation of Bits $Sd_n[3:0]$

- We use similar equations to those of clause 146.3.3.4.3

$$Sd_n[3] = \begin{cases} TXD_{2n}[3] \wedge Sy_n[3], & \text{if (tx\_enable}_{2n} = \text{TRUE)} \\ 1 \wedge Sy_n[3], & \text{else if (loc\_rcvr\_status} = \text{OK)} \\ Sy_n[3], & \text{else} \end{cases}$$

$$Sd_n[2] = \begin{cases} TXD_{2n}[2] \wedge Sy_n[2], & \text{if (tx\_enable}_{2n} = \text{TRUE)} \\ 1 \wedge Sy_n[2], & \text{else if (loc\_lpi} = \text{TRUE)} \\ Sy_n[2], & \text{else} \end{cases}$$

$$Sd_n[1] = \begin{cases} TXD_{2n}[1] \wedge Sy_n[1], & \text{if (tx\_enable}_{2n} = \text{TRUE)} \\ Sy_n[1], & \text{else} \end{cases}$$

$$Sd_n[0] = \begin{cases} TXD_{2n}[0] \wedge Sy_n[0], & \text{if (tx\_enable}_{2n} = \text{TRUE)} \\ Sy_n[0], & \text{else} \end{cases}$$

- Note that we do not reverse bits $Sy_n[1]$ and $Sy_n[2]$ during IDLE as was done in clause 146
  - The rationale for this will be explained shortly

Generation of Bits $Sd_n[7:4]$

▶ We use the following equation for these bits

$$Sd_n[7:4] = \begin{cases} TXD_{2n+1}[3:0]\wedge Sx_n[3:0], & \text{if } (\text{tx\_enable}_{2n+1} = \text{TRUE}) \\ Sx_n[3:0], & \text{else} \end{cases}$$

Stream Delimiters

- The PHY generates a continuous stream of code-groups
  - Each code-group is a 6-tuple of ternary symbols
- Code-groups align with byte boundaries in the PHY
  - This is due to the use of an 8b6T line code
- MII is a nibble-oriented interface
  - There is no guarantee that byte boundaries in the PHY match those in the MAC
  - TX_EN may rise or fall on odd nibble boundaries from a PHY perspective
- Could delay TX_EN transitions to align with PHY byte boundaries
  - There is no guarantee that byte alignment between the MAC and the PHY is consistent from one system reset to the next
  - This could create latency variability
  - We will maintain the byte alignment from the MAC and pass it through the physical layer
  - This requires different stream delimiters for changes in TX_EN on odd nibble boundaries

Code-group Categories

- The variables tx_mode, tx_enable and xmt_error determine the code-group category which will be one of the following

Code-group Category	Criterion for Selecting this Category
SEND_Z	tx_mode is SEND_Z
SSD	tx_enable switches from FALSE to TRUE on a byte boundary
SSD_ODD	tx_enable switches from FALSE to TRUE on an odd nibble boundary
ESD	tx_enable switches from TRUE to FALSE on a byte boundary
ESD_ODD	tx_enable switches from TRUE to FALSE on an odd nibble boundary
XMT_ERR	An error condition is to be propagated to the link partner
DATA	Frame data is to be sent
IDLE	None of the other code-group categories have been selected

Outline of Transmission Process

- The code-group category determines which table will be used to select the non-negative disparity (NND) code-group for transmission
  - If the code-group category is SEND_Z, then the code-group is {0, 0, 0, 0, 0, 0}
  - Otherwise, the code-group is determined either by the bit $Sg_n[1]$ or by the bits $Sd_n[7:0]$, depending on the code-group category
  - NND code-groups are denoted {TA_n, TB_n, TC_n, TD_n, TE_n, TF_n}
- The NND code-groups are passed to the running disparity (RD) control function
  - This function may negate a code-group with positive disparity to bound RD
  - Negation of a code-group means negating each of the ternary symbols in the 6-tuple
  - We call the resulting code-group a balanced code-group
  - Balanced code-groups are denoted {A_n, B_n, C_n, D_n, E_n, F_n}
- Balanced code-groups are transmitted one ternary symbol at a time
  - The leftmost symbol in the code-group is transmitted first

Code-group Category Selection

	tx_enable					xmt_error		Code-group Category	Comment
	2n-3	2n-2	2n-1	2n	2n+1	n-1	n		
SEND_Z	x	x	x	x	x	x	x	SEND_Z	
not SEND_Z	0	0	0	0	0	x	x	IDLE	
	0	0	0	1	1	x	x	SSD	Ignore any error until next byte
	0	0	0	0	1	x	x	SSD_ODD	Ignore any error until next byte
	0	1	1	1	1	1	x	XMT_ERR	Delayed error from start of frame
	0	0	1	1	1	1	x	XMT_ERR	Delayed error from start of frame
	0	0	1	1	1	0	0	DATA	
	0	0	1	1	1	0	1	XMT_ERR	
	0	1	1	1	1	0	0	DATA	
	0	1	1	1	1	0	1	XMT_ERR	

- Sending a code-group from the SSD_ODD code-group category informs the receiver that it should assert RX_DV on an odd nibble boundary

Code-group Category Selection

tx_mode	tx_enable					xmt_error		Code-group Category	Comment
	2n-3	2n-2	2n-1	2n	2n+1	n-1	n		
not SEND_Z	1	1	1	1	1	x	0	DATA	
	1	1	1	1	1	x	1	XMT_ERR	
	1	1	1	1	0	x	0	DATA	Last nibble of frame padded
	1	1	1	1	0	x	1	XMT_ERR	Error in last nibble of frame
	1	1	1	0	0	x	x	ESD	
	1	1	0	0	0	x	x	ESD_ODD	
	1	0	0	0	0	x	x	IDLE	

- If the falling edge of tx_enable occurs on an odd nibble boundary, the final data nibble is padded out to a byte
  - Per the equations for $Sd_n[7:4]$, the nibble that is added is determined by $Sx_n[3:0]$
  - The code-group that follows is selected from the ESD_ODD code-group category. This informs the receiver that it should discard the last nibble of the previous byte and deassert RX_DV on an odd nibble boundary.

Start-of-Stream Code-groups

- We use bit $Sg_n[1]$ to choose between 2 code-groups to avoid unnecessary correlation artefacts in the stream
  - Each of these 2 code-groups is the element-wise negative of the other
  - The code-groups in the SSD_ODD category are the same as those in the SSD category but with $Sg_n[1]$ inverted

Code-group Category	$Sg_n[1]$	TA_n, TB_n, TC_n, TD_n, TE_n, TF_n
SSD	0	+1, +1, -1, -1, +1, -1
	1	-1, -1, +1, +1, -1, +1
SSD_ODD	0	-1, -1, +1, +1, -1, +1
	1	+1, +1, -1, -1, +1, -1

End-of-Stream Code-groups

- We use bit $Sg_n[1]$ to choose between 2 code-groups to avoid unnecessary correlation artefacts in the stream
  - Each of these 2 code-groups is the element-wise negative of the other
  - The code-groups in the ESD_ODD category are the same as those in the ESD category but with $Sg_n[1]$ inverted

Code-group Category	$Sg_n[1]$	**TA_n, TB_n, TC_n, TD_n, TE_n, TF_n**
ESD	0	+1, -1, -1, +1, +1, -1
	1	-1, +1, +1, -1, -1, +1
ESD_ODD	0	-1, +1, +1, -1, -1, +1
	1	+1, -1, -1, +1, +1, -1

Transmit Error Code-groups

- We use bit $Sg_n[1]$ to choose between 2 code-groups to avoid unnecessary correlation artefacts in the stream
  - Each of these 2 code-groups is the element-wise negative of the other

Code-group Category	$Sg_n[1]$	TA_n, TB_n, TC_n, TD_n, TE_n, TF_n
XMT_ERR	0	-1, -1, +1, +1, +1, -1
	1	+1, +1, -1, -1, -1, +1

DATA Code-groups

- When the selected code-group category is DATA we use bits $Sd_n[7:0]$ to choose the code-group
  - As there are too many code-groups to list them all here, the following file is provided

    data_code_groups_05132024.txt

  This file has 256 lines. Each line has 7 columns. The first column is the $Sd_n[7:0]$ value in binary form. The remaining 6 columns provide the ternary values for the code-group. The following table lists the first 4 code-groups from this file.

Code-group Category	$Sd_n[7:0]$	TA_n, TB_n, TC_n, TD_n, TE_n, TF_n
DATA	00000000	-1, +1, -1, +1, +1, -1
	00000001	-1, +1, +1, -1, +1, -1
	00000010	+1, -1, -1, +1, -1, +1
	00000011	+1, -1, +1, -1, -1 , +1

IDLE Signaling

- In clause 146 there is no reliable way to distinguish IDLE from DATA
  - Bits $Sy_n[1]$ and $Sy_n[2]$ are swapped during IDLE. However, a data pattern can readily reproduce this effect. The result is that there is no way to detect IDLE signaling while receiving a frame.
  - If the COMMA code-groups that mark the end of a frame are missed, the PCS receive state machine of Figure 146-9 becomes stuck in the DATA and DATA DECODE states. The situation may persist until the start of the next frame and both frames are lost.
  - By contrast, the sign reversal scheme of clause 40.3.1.3.6 ensures that IDLE signaling can be detected while receiving a frame.
- We propose to use certain spare code-groups to identify IDLE
  - We replace 16 of the code-groups that are used for DATA signaling with special code-groups that are used only during IDLE
  - Detecting any of these special code-groups while receiving a frame will be treated as a premature end condition

IDLE Code-groups

- When the selected code-group category is IDLE we use bits $Sd_n[7:0]$ to choose the code-group
  - For $Sd_n[7:0]$ values in the binary range 00000000 to 11101111 the IDLE code-groups are the same as the data code-groups. However, for $Sd_n[7:0]$ values in the binary range 11110000 to 11111111 the special IDLE code-groups listed in the following file are used

    idle_code_groups_05132024.txt

    This file has 16 lines and is organized in the same way as was used for the DATA code-groups. The following table lists the first 4 code-groups from this file.

Code-group Category	$Sd_n[7:0]$	**TA_n, TB_n, TC_n, TD_n, TE_n, TF_n**
IDLE	11110000	-1, +1, +1, +1, +1, +1
	11110001	+1, -1, +1, +1, +1, +1
	11110010	+1, +1, -1, +1, +1, +1
	11110011	+1, +1, +1, -1, +1, +1

Running Disparity Checking

- It is proposed not to support RD checking in the receiver
  - A disturbance such as an EFT event would be likely to cause an error in such an RD check
  - The RD checking process would take time to resynchronize after such an error
  - This would result in error propagation
- We do not see a benefit in including RD checking in the receiver
  - Such checking is not required to detect frame errors
  - The benefit of RD control is on the transmit side
- As we will not support RD checking in the receiver, there is no need to reset RD at the start and the end of each frame as was done in clause 146

Running Disparity Control

- The RD control function maintains a running disparity value, RD_n. This value is initialized to 0 at n=0
- A sign value SX_n is generated using the following equations

$$\mathrm{DS}_n = (TA_n + TB_n + TC_n + TDn + TEn + TF_n)$$

$$SX_n = \begin{cases} -1, & \text{if } (DS_n > 0)\ \&\ \big((RD_n > 0)\,|\,((RD_n = 0)\ \&\ (Sg_n[0] = 1))\big) \\ +1, & \text{else} \end{cases}$$

- The balanced code-group is generated by applying the computed sign to the NND code-group

$$\{A_n, B_n, C_n, D_n, E_n, F_n\} = SX_n * \{TA_n, TB_n, TC_n, TD_n, TE_n, TF_n\}$$

Here $*$ denotes element-wise multiplication by a scalar.

- The running disparity value is updated as shown below

$$\mathrm{RD}_{n+1} = \mathrm{RD}_n + (A_n + B_n + C_n + Dn + En + F_n)$$