Energy Efficient Ethernet Encodings

Yanpei Chen
Randy H. Katz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-68
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-68.html

May 19, 2009

Energy Efficient Ethernet Encodings

by Yanpei Chen

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Randy H. Katz
Research Advisor

19 May 2009

(Date)

* * * * * * *

Professor Jean Walrand
Second Reader

6 May 2009

(Date)

Abstract

The energy efficiency of network elements is becoming more prominent, with growing concern for Internet power consumption and heat dissipation in datacenters and communications closets. Previous work has looked at energy efficient wireless topologies, network nodes, routers, and protocols. In considering a fresh redesign of the Internet datacenter for energy efficiency, we believe that energy efficient encodings are worthy of study. In this work, we re-examine the choice of Ethernet encoding, develop an associated energy model, evaluate current encodings, propose new encodings, and identify the desirable features of future encodings. We found that simpler encodings are more energy efficient, with power savings of around 20% for the best encoding. Our work represents a first step in re-examining the established assumptions and practices of the PHY level of the network stack with respect to energy.

Table of Contents

1. Introduction

The energy efficiency of communication networks is receiving increasing attention. Global warming, energy costs, and heat dissipation in datacenters and communication closets makes power management an essential part of network research. Studies in 2001 found that 2% of U.S. electricity consumption can be attributed to powering our information infrastructure. This amounts to 74TWhr and $6 billion spent in 2001 [1, 2]. In contrast to the continuously increasing energy demand of the Internet, U.S. national electricity generation capacity has remained constant since 2005 [3]. Improving Internet energy efficiency will not only reduce the operating costs of Internet equipment, it will also bring tangible reductions to Internet's carbon footprint. There is considerable work on Internet energy efficiency. The network stack has already been thoroughly examined [4, 5, 6], except for link layer encodings. The dominant Internet link layer technology is Ethernet. Its encodings were traditionally considered difficult to change, and consequently, there is no systematic understanding of the problem space for energy efficiency in this area. Recent interest in "Greenfield" datacenter design opens up the question of how to design link layer encodings for energy efficiency.

We suspected that there would be opportunities to save energy, since the widely implemented encodings were developed before energy concerns became important. It is now critical to quantify the possible energy savings through energy conscious encodings, compared with savings attributed to other techniques. As a first step, any

new encodings we propose must be compatible with existing technology, in that we should require no changes to higher layers of the network stack. We focus on three encodings: 4B5B and MLT-3 for 100Mbps over UTP Ethernet cables, 8B10B for 1Gbps over optical fiber, and 4D-PAM5 for 1Gbps over UTP. Section 2 gives a brief overview of these encodings, which are the most widely deployed. Internet datacenters and high end networks are predominantly 1Gbps, and most residential networks remain 100Mbps. We do not consider 10Gbps because while it may present an even greater opportunity for energy savings, it is significantly less widely deployed in 2009 due to cost reasons. Nevertheless, we will show that the methodology and insights gained from a study at 100Mbps and 1Gbps become even more relevant for higher speed links.

To the best of our knowledge, our work is the first detailed study to re-consider the choice of Ethernet encoding with respect to energy efficiency. Our contribution is three-fold. First, we offer a new view of how to consider the impact of energy consumption on communications encodings. Second, we examine the energy consumption of Ethernet encodings for 100Mbps and 1Gbps technologies, and suggest an improved encoding for 100Mbps. Third, we offer a power model of Ethernet encodings to help guide further work in the area.

The report is structured as follows. Section 2 quickly reviews existing Ethernet encodings and the large body of prior work in Internet power management and energy efficiency. Section 3 explains our view of the encoded communication problem and outlines our power model for Ethernet encodings. Section 4 describes our proposed

encoding. Section 5 evaluates various encoding through Matlab and Verilog simulation, at the same time verifying our power model. Section 6 distills key insights from our work, and makes recommendations for future work in energy efficient Ethernet encodings and Internet energy efficiency in general. Our key result is that simpler encodings are more energy efficient, with savings of around 20% for the encoding.

2. Background and Related Work

2.1. Existing Ethernet Encodings

We look at 4B5B and MLT-3 for 100Mbps over UTP cables, 8B10B for 1Gbps over optical fiber, and 4D-PAM5 for 1Gbps over UTP. Full descriptions of the Ethernet encodings are found in the IEEE and ANSI standards [7, 8]. We will give a brief overview below, and relevant excerpts from the IEEE standards [7] are found in Appendix 1.

4B5B and MLT-3 encodings are used in 100BASE-TX for 100Mbps over UTP. Four-bit blocks of the input bit stream are mapped to five-bit output blocks to facilitate synchronization and other functions. The outgoing bit rate is 125Mbps, above the natural frequencies of the copper UTP cable. The MLT-3 encoding allows 125Mbps to be delivered at 31.25 MHz. MLT-3 output signals have three analog levels, with a peak-to-peak voltage of 2V. It delivers 125Mbps at 31.25 MHz because it cycles

through +1, 0, and -1 logic levels, with a “1” bit in the input causing a logic level transition. There are no direct transitions between the +1 and -1 logic levels, allowing a low frequency signal to be used.

8B10B encoding is used in 1000BASE-LX/SX for 1Gbps over single/multi-mode optical fiber. Eight-bit blocks of the input bit stream are mapped to ten-bit blocks to facilitate clock synchronization. A running parity check ensures the output is DC balanced. The output bit stream gets sent over optical fiber using either on-off-keying or phase modulation.

The 4D-PAM5 (4-dimensions, 5 levels pulse amplitude modulation) encoding delivers 1Gbps over UTP. Eight-bit blocks of the incoming bit stream are converted to four PAM5 signals, with a peak-to-peak voltage of 2V and sent over four twisted pairs of the UTP cable. A complex scrambling scheme ensures output DC balance and facilitates full duplex on all four twisted pairs, with each pair being 250Mbps full duplex.

The original rationale for selection of these encodings was data rate and convenience over efficiency. The problem facing the designers of 4B5B and MLT-3 encodings was how to enable 100 Mbps over an analog UTP channel that has a bandwidth of 31.25 MHz. The pre-existing UTP encoding for 10 Mbps, the Manchester Encoding, would require 200 MHz to deliver 100 Mbps. 4B5B/MLT-3 allowed the required data rate to be achieved within the bandwidth constraints. The encoding uses only one of four twisted-pair wires in the UTP cable for simplex communication, a significant strength

since signaling on all four twisted-pair wires in the UTP cable would readily permit duplex communications at 200Mbps.

Similarly, the 4D-PAM5 encoding was devised to achieve 1Gbps over UTP channel with limited analog bandwidth. The encoding has more bits per symbol (PAM5 vs. something resembling PAM3 for MLT-3), which leads to a higher bit rate when the signaling rate is a fixed number of symbols per second. It also facilitates a scrambling scheme to permit duplex communications on a single twisted-pair, thus reducing the data rate needed from each of the four twisted-pair wires.

The choice of 8B10B for single and multi-mode fiber was one of convenience. 8B10B has been an established technology before its adoption as standard for 1Gbps over fiber. It was not adopted as standard earlier because it was formerly covered by a patent from IBM. When the patent expired, 8B10B proliferated to a variety of uses, including 1Gbps over fiber, SATA 1.5Gbps and 3Gbps, InfiniBand, USB 3.0, and others. A significant strength of the encoding is bounded bit disparity in a given run of symbols, which will reduce demand for the lower bandwidths of the channel.

2.2. Network Energy Efficiency

One of the earliest works in Internet energy efficiency is [9]. Many studies have followed. There is well established research in wireless energy efficiency, motivated by the limited power budgets for wireless devices, such as those in sensor nets and ad

hoc networks [10, 11]. Studies have also looked at how channel conditions and wireless protocols affect power consumption [12]. In comparison, our work focuses on higher speed wired topologies.

Prior work has looked at power management at network nodes, such as network switches [4, 13], networked storage and disk drives [14], servers [15], and PCs [16]. Our work focuses on communication between network nodes.

Other power saving strategies focus on protocols at both the transport and network layers. There have been studies on the power consumptions of different flavors of TCP [5], TCP in wireless [17], sleep option for TCP [18], and using proxies to facilitate extensive sleep time [19]. Our work is focused on PHY and link encodings rather than new protocols.

Possible ways to save energy in the link layer includes reducing the link layer speed to facilitate energy savings during times of low traffic [6]. Current work in the IEEE 802.3 Energy Efficient Ethernet (EEE) Task Force includes diverse ideas on how to save energy in the link layer [20]. We complement the work there by offering energy efficient encodings that can be deployed in conjunction with other link layer power management techniques.

Past work in ADSL2+ power management hints at the idea of using line encodings to save energy [21]. There, it was suggested that different modulation schemes would

deliver different transmission energy. We are not aware if the idea was pursued further.

Our work on energy efficient Ethernet encodings (EEEE) is an alternative approach to saving energy for the Internet. Our focus on Ethernet is driven both by a vacuum in our understanding in the area, and that the biggest energy savings are to be had at the edge of the network than at the highly concentrated network core. We differ from previous work in our focus to investigate and quantify the energy benefits of alternative PHY encodings. Our work complements existing research. One can envision a power efficient Internet in the future, with energy efficient protocols and energy efficient nodes, energy efficient wireless for wireless nodes, and optimized wired links with EEEE for encoding the data sent.

3. Energy Conscious Encodings

3.1. A Framework for Energy Conscious Communications

The canonical digital communication problem is: given a certain bandwidth of the communication channel, an energy budget at the communication endpoints, and a target error rate, we try to maximize the data rate of communication. The bandwidth and energy budget are the resources available, the error rate is a performance bound, and the maximized data rate is the performance goal. Encoding is a tool to maximize performance using the resources available.

Traditionally, bandwidth and channel conditions have been the bottlenecks, affecting the data rate and the error rate respectively. Thus, the focus of encoding schemes has been to efficiently use the available bandwidth, and correct errors introduced by channel noise. Only in applications with limited power supply, such as mobile devices or sensor networks, has the energy budget been a concern. Now, with a rising focus on power consumption, we need an alternate view that highlights energy issues.

We formulate an alternative digital communication problem. Given a certain bandwidth, an error rate, and a data rate prescribed by Internet standards, we minimize the energy budget required. In this view, the bandwidth is the resource available, the data rate and error rate are bounds, and the minimized energy consumption is the goal.

A performance metric common to both views is the bandwidth-energy product, helpful for comparing different encodings, given the same data rate, error rate, and "all else equal". For an "efficient" encoding, the value of this product should be low. We could also look at the data rate to bandwidth-energy ratio, i.e., (data rate) / (bandwidth × energy), a re-cast of the "energy-per-bit" concept. This metric is not as helpful since the data rate is often prescribed, and different communications channels may preclude certain data rates.

For comparing the energy efficiency of two encodings, the only metric we need is the energy budget. The encodings need to have the same bandwidth, data rate, error rate,

and other performance criteria. Otherwise, we cannot compare two encodings using energy efficiency alone.

3.2. An Energy Model for Ethernet Encodings

The goal of an energy model is to understand how different parts of the communication system contribute to the energy consumed. For Ethernet encodings, we find it helpful to include only two sources of energy consumption – the encoding circuits, and the transmission energy put on the communication channel. The encoding circuits include digital encoding circuits, D/A converters, and pulse-shaping circuits. The transmission energy of the communication channel is either dissipated into the channel, or received at the destination.

There are, of course, other energy consumers in a communication system, including send/receive buffers, memory, and possibly the operating system. These sources are independent of the encodings used, and are left out of our energy model. A more fine-grained model should account for these energy consumers and quantify their contribution.

We propose that in wired Ethernet, the energy spent in encoding circuits is much larger than the transmission energy. This is a significant departure from wireless energy models, where the encoding circuit energy is routinely assumed to be negligible compared to the transmission energy, e.g., in [22, 23, 24]. The difference is

due to two reasons. First, the wireless channel is noisy and three-dimensional. Therefore, large transmission energy is required for a useful signal to noise ratio (SNR). In contrast, wired Ethernet is less noisy, and the signal is concentrated along a one-dimensional cable. Therefore, the transmission energy need not be large. Second, the data rates prescribed for 100Mbps and 1Gbps greatly exceeds the natural frequencies for UTP cables, requiring non-trivial and sometimes very complex encodings to deliver the prescribed data rate. Complex encodings means circuits with larger areas, and higher signaling speeds means greater circuit energy consumption. Therefore, we believe the encoding energy to be larger than transmission energy in wired Ethernet, and the ratio would become even more acute as data rates increase to 10Gbps and beyond.

Verifying this energy model requires getting a quantitative measure of the encoding energy and transmission energy. The transmission energy is relatively easy to estimate. For 100Mbps and 1Gbps over UTP, we can estimate the transmission energy by the line voltage and the cable insertion loss. For 1Gbps over optical fiber, we can estimate the transmission energy by the optical output of laser diodes. This estimate would exclude the power consumed in the DC bias circuits necessary for operating the laser diode; the power for these circuits should also be excluded from the encoding energy. Estimating the encoding energy is more difficult. The most direct verification would require probing the encoding circuits. We are not aware of commercial chip-sets that allow such probing. This approach may also run into difficulties in identifying and isolating the encoding circuits in a highly optimized chip-set layout. Another approach would be to design and layout a communication

chip-set from scratch, then simulate the power consumption using a CAD suite. This approach has the side benefit of generating a fine-grained power model for the chipset in a NIC card, but is probably best left to experts in circuit design. We use a third approach, which gives only a first-order estimate. We take the power consumption of typical NICs, and subtract from it the transmission power. The remainder would include the power consumption of encoding circuits and other circuits, such as buffers, or DC bias circuits in optical fiber NICs. In the absence of a more fine-grained method, this calculation gives a ball-park estimate of the encoding circuit power consumption. As will be evident later in the report, such estimates are sufficient to distinguish the relative energy efficiency of different encodings.

In addition, we can compare the relative encoding circuit energy for different encodings by looking at the size of the different encoding circuits. Larger circuits generally mean greater energy consumption. In particular, we can build circuit simulations of different encodings using a uniform technology, and compare the size of the resulting circuits. Such simulations would not give us absolute values for energy consumption. They would, however, give us an idea on the relative energy consumptions for the circuits used in different encodings.

4. An Alternative Encoding for MLT-3

We propose an alternative encoding to MLT-3 that has lower power consumption. Like MLT-3, it takes as input the result of 4B5B encoding, and outputs the voltages to be sent. It also delivers 125 Mbps at 31.25 MHz. Our encoding is a fully backwards

compatible alternative to MLT-3, in that we require no changes to 4B5B and the Mac layer above, nor the physical medium below. In Section 5, we show that our encoding can save 18% of transmission energy, and 60% of encoding circuit energy.

Our encoding is inspired by the observation that while the 4B5B output bit stream is not DC-balanced, the output of 4B5B in conjunction with MLT-3 is. If we assign voltage levels based on sequences in the 4B5B output stream, we can get a larger fraction of symbols in a low energy state, at the cost of slight DC-imbalance. If we ensure our encoding conforms to the output level transition constraints of MLT-3 and signals at the same baud rate, then we can deliver 125 Mbps at 31.25 MHz for lower transmission energy.

The new encoding uses a state machine involving the past two bits in the 4B5B output stream. The state transition diagram is shown in Fig. 1. An incoming data bit causes a state transition. For example, if the past two bits are 01, the machine is in State 01. An incoming bit with value 0 will cause a transition to State 10. In State 10, the previous two bits are the 1 bit carried over from State 01, and the new 0 bit. Given a predetermined initial state, the output is uniquely decodable.

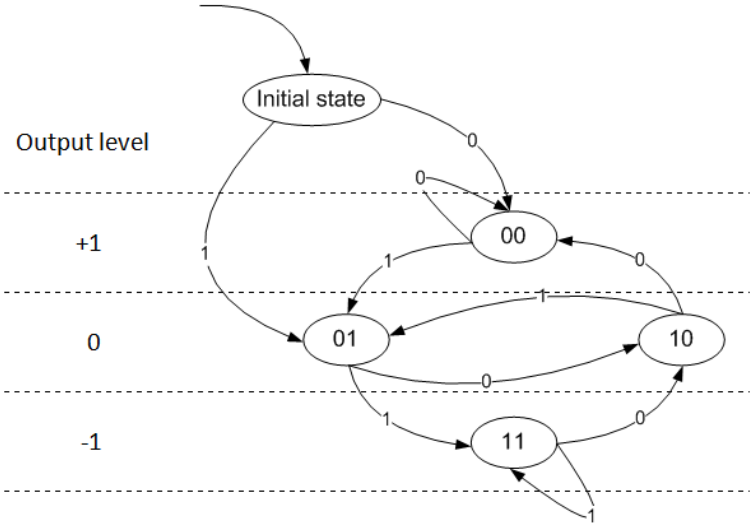

Figure 1. State transition diagram for our improved encoding.

The states are mapped to three logical output levels, the same as the three logical levels in MLT-3. The mapping is shown in Fig. 1. Like MLT-3, our encoding has no direct transitions between +1 and -1 output levels. This allows 125Mbps at 31.25MHz. If we signal at the same baud rate and use the same peak-to-peak voltage, our encoding would be fully compatible with 4B5B and existing technology. We would require both endpoints on the link use either MLT-3 or our alternative, and endpoints could potentially support both kind of ports.

As we will detail later, our alternative is significantly simpler to implement compared with MLT-3, leading to lower encoding circuit energy.

A potential limitation is that our encoding is not DC balanced. This was formerly a significant issue. When encodings were first developed for the telephone network, the

AC signal component would carry the voice, while the DC component would be used to power the telephone sets and the line repeaters. Any DC imbalance in the encoding would be treated as another DC power component, and would be unrecoverable at the receiver. Also, the DC component would not pass through any transformers used for impedance matching for the transmission line. Such concerns may still persist for DSL, but not for the vast majority of Ethernet links. Also, a DC balanced signal would allow hardware implementations to use AC coupled communication circuits, which are easier to design than their DC counterpart. DC balance may be restored through scrambling, but applying the technique may significantly add to the complexity of the circuit.

Another possible limitation is that our encoding is essentially amplitude modulation, where data is encoded in the voltage levels, in contrast to differential signaling in MLT-3, where data is encoded in the presence or absence of transitions. Differential signaling is preferred in noisy environments, because detecting a transition is easier than comparing against a threshold voltage. We believe this is not a critical issue, since wired Ethernet is a relatively non-noisy medium. Also, our encoding divides 2V peak-to-peak into three levels, with the difference between the voltage amplitude thresholds being smaller than that for 1000BASE-TX, with PAM5 dividing 2V peak-to-peak into five levels.

5. Evaluation

We built simulations of the encodings in Matlab and Verilog to evaluate their performance. The former allow us to analyze the statistical distribution of output voltage levels, and identify any obvious opportunities to save on transmission energy. The later give us an FPGA implementation of the encoding circuits, allowing us to compare the encoding circuit size and encoding circuit energy. The simulations take in a random bit stream supplied by the MAC layer and output a symbol stream sent to the media D/A converter.

For our Matlab simulations, we simulate for 100Mbps over UTP (100BASE-TX) the Physical Coding Sublayer (PCS) and the Physical Medium Dependent (PMD) sublayer, containing the 4B5B and MLT-3 encodings. For 1Gbps over fiber (1000BASE-LX/SX) and 1Gbps over UTP (1000BASE-T), our Matlab simulations include the PCS only, with the 8B10B encoding for 1Gbps over fiber, and the 4D-PAM5 encoding for 1Gbps over UTP. The output voltages from our simulation correspond to the inputs to the Media Dependent Interface (MDI), which is immediately converted to an analog signal and sent on the physical medium.

For our Verilog simulations, we also include the Physical Medium Attachment sublayer (PMA), which serializes the output bit blocks from the PCS. The PMA also performs some other functions that were not simulated, such as generating control signals and performing synchronization. We do not count the PMA towards our encoding circuit size, since it is not a part of the encoding per se.

Fig. 2 shows the relationship between different sublayers included in our simulations.

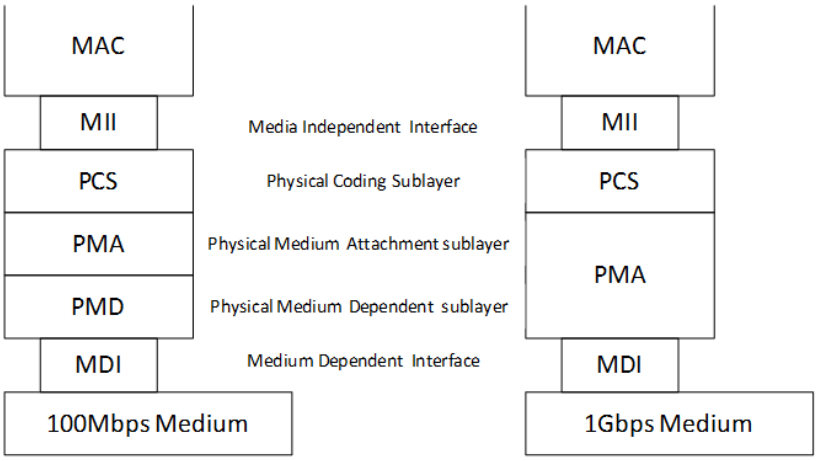

Figure 2. Simulation layers.

For transmission energy, encodings with a greater fraction of symbols in lower voltage levels would consume less transmission energy. For encoding circuit energy, encodings with smaller circuits, indicated by fewer logical units required, would consume less circuit energy. Encodings with lower energy are more preferable.

5.1. Simulation Results – Transmission Energy

We used Matlab simulations to obtain the output characteristics of the encodings and analyze the output transmission energy. The full Matlab simulation code is in Appendix 2. Table 1 shows the Matlab simulation results. For each encoding, we

compute the fraction of output symbols at each logic voltage level. To simplify analysis, we have lumped together logic levels of the same magnitude, e.g. for 4B5B and MLT-3 encoding, the +1 and -1 logic levels together account for 50% of the output symbols. The peak-to-peak analog voltage is 2V for all encodings on UTP.

TABLE I. MATLAB SIMULATION RESULTS

Encoding	Fraction of output symbols in each logical level		
	0	*±1*	*±2*
4D-PAM5	0.22	0.44	0.34
8B10B	0.5	0.5	-
4B5B MLT-3	0.5	0.5	-
4B5B and alternative to MLT3	0.59	0.41	-

For 100Mbps, logic levels ±1 correspond to analog levels ±1V. For 1Gbps, logic levels ±1 are ±0.5V in analog, and logic levels ±2 are ±1V in analog. For 8B10B on optical fiber, the logic level is translated to either on-off-keying of the laser diode, or a phase modulated optical signal.

For 4D-PAM5, we see from Table 1 that it has a large fraction of symbols in logic level 1. This means the 4D-PAM5 is somewhat optimized, in that more symbols are in logic level 1 than in logic level 2. However, there is likely to be opportunities to further optimize the code, since there are more symbols in logic level 1 than in logic level 0. Given the complexity and functional requirements of 4D-PAM5, it is not

immediately obvious what the possible optimizations are. Any efforts must consider that we are optimizing the transmission energy. Since this is small compared with encoding circuit energy, "optimizations" that result in a more complex code is likely to lead to larger circuits and an undesired increase in energy consumption.

For 8B10B encoding, the output logic levels are mapped to on-off-keying or phase modulation optical signals. The transmission energy in optical fiber is given by the optical power used to drive the fiber, and not the logic levels sent. Hence 8B10B has no need to be further optimized for transmission on optical fiber. We will mention that for phase modulation, the optical signal is always "on", with logical high and low distinguished by a different optical phase. This means that for the same peak optical power, phase modulation spends two times the transmission energy of on-off-keying. However, the energy is not wasted, since phase modulation will result in a 3dB gain in the optical signal to noise ratio. From a circuit energy perspective, phase modulation circuitry is more complex than circuits for on-off-keying. Hence it is preferential to send 8B10B using on-off-keying, but at twice the transmitted optical power to achieve the same signal to optical noise ratio as phase modulation.

For 4B5B and MLT-3 encoding, we see that half the symbols are not energized. However, our improved alternative to MLT-3 allows over half of the symbols to be in a non-energized state. It is inspired by DC-imbalance in the 4B5B output. Our improved encoding takes advantage of this fact to assign over half of the symbols to the non-energized state. However 9% of the output symbols are assigned to logic level 1, while 32% of the symbols are assigned to logic level -1, leading to a slight DC-

imbalance. On the other hand, based on simulation results, our improved encoding spends 9% less time in logic levels ±1 compared with MLT-3, leading to an 18% saving in transmission energy.

To repeat our transmission energy saving calculations, we use the results in Table 2. This shows the statistical characteristics of the 4B5B output bit stream, when a random bit stream is used as the input. The 4B5B output is then fed into either MLT-3 or our improved encoding. As shown in the table, when we encode a random bit stream into 4B5B, 61% of the output bits are “1”. Also, 9% of the output two-bit sequences are “00” and 32% of the output two-bit sequences are “11”. Thus, for our encoding, the state machine would spend 9% of the time in State “00” and 32% of the time in State “11”. Fig. 1 shows that these two states respectively correspond to analog voltage levels +1 and -1. Thus, our improved encoding spends 9% + 32% = 41% of the time in an energized state, compared with 50% for MLT-3, leading to a (41% – 50%) / 50% = 18% saving in transmission energy.

As we will show below, our improved encoding is also significantly easier to implement, leading to a reduction in the circuit encoding energy also.

TABLE II. STATISICAL ANALYSIS FOR 4B5B OUTPUT SEQUENCES

Output Sequence	Fraction of all sequences
0	0.3875
1	0.6125
Total	1.0
00	0.0937
01	0.2938
10	0.2938
11	0.3186
Total	1.0

5.2. Simulation Complexity – Encoding Circuit Energy

We used Verilog simulations to quantify the circuit implementation complexity of the encodings. This would give us an indication of the relative encoding circuit energy consumption. Larger circuits and more complex code would lead to greater encoding circuit energy consumption. We built our Verilog simulations with the Xilinx FPGA design suite. Our block designs follow the general block layout found in [25]. We used ModelSim to verify the correctness of our simulations. We ran the Xilinx synthesis tool to synthesize the design for FPGA implementation, and we looked at the synthesis report to extract the circuit size in terms registers and logical look-up tables (LUTs) used. Table 3 shows a summary of our simulated encoding circuit size. We can break down each encoding circuit into two parts – an asynchronous

translation table for mapping input symbols to output symbols, and a synchronous state machine for cycling through voltages or ensuring output DC balance.

TABLE III. VERILOG SIMULATION ENCODING CIRCUIT SIZE

Simulation Block	Registers Used	LUTs Used
4B5B translation table	0	4
MLT-3 state machine	4	6
Alt. to MLT-3 state machine	1	2
4B5B & MLT-3 total	0 + 4 = 4	4 + 6 = 10
4B5B & alt. to MLT-3 total	0 + 1 = 1	4 + 2 = 6
8B10B translation table	0	10
8B10B state machine	1	3
8B10B total	0 + 1 = 1	10 + 3 = 13
4D-PAM5 translation table	0	46
4D-PAM5 state machine	38	28
4D-PAM5 total	0 + 38 = 38	46 + 28 = 74

Compared with the state machine for MLT-3, the state machine for our improved alternative encoding uses only a third of the registers and LUTs, due to the reduced state space for our improved encoding. Fig. 3 shows the optimized state machines. Our alternative to MLT-3 is the equivalent of the state machine in Fig. 1, with the state transitions given by *input* / *output*. The MLT-3 state machine has transitions driven by *input* and states marked with *state* / *output*. Using a good first-order

approximation that circuit power is proportional to circuit area, we find that our encoding uses only a third of the encoding circuit power of MLT-3.

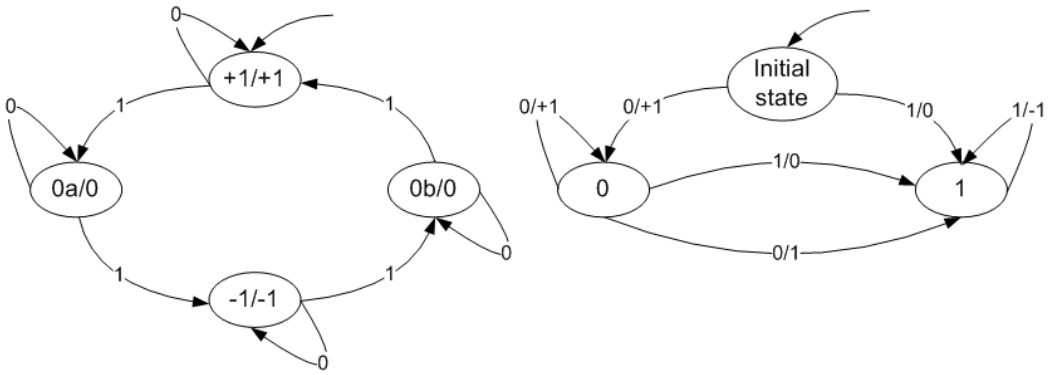

Figure 3. State machines for MLT-3 (left) and our alternative (right).

Also, the 8B10B state machine is as simple as that for our improved alternative to MLT-3, while the 4D-PAM5 state machine is an order of magnitude more complex. The 8B10B uses one memory variable of one bit to keep track of the running disparity in the output. In comparison, the state machine for 4D-PAM5 has a 32 bit scrambler, with an exponentially larger state space.

Consequently, the complexity of encoding state machines is determined by the size of the state space, given by the number of memory variables required and the bits for each variable. The translation table sizes offer another insight. For 8B10B, the translation table has 256 rows of 10 bits each compared with 16 rows of 5 bits for 4B5B. By counting bits, we would expect the 8B10B translation table to be 32 times larger than that for 4B5B, instead of 2.5 times as large. 8B10B breaks down its input-

output translation into a 5B6B translation table and a 3B4B translation table. There are two translations for each input, with the state machine keeping track of a running disparity that determines which of the two translations would be used. Thus the 8B10B translation tables would have 32 lines of 12 bits, and 8 lines of 8 bits, for 5.5 times the number of bits. Automatic optimization in Xilinx tools reduces the translation table even further, and it is 2.5 times as large as that for 4B5B.

For 4D-PAM5, the translation table has 256 rows of 12 bits each, where each row is a 4-tuple of 3-bit PAM5 symbols. Counting bits give a translation table 38.4 times larger than that for 4B5B. Instead, the table is 46 times as large, even with automatic optimization. We believe this is the overhead of
having large tables, where additional circuitry is required to facilitate lookup under the same timing constraints. Thus, we believe that encodings should avoid large translation tables whenever possible. One way to reduce the complexity of input-to-output bit translations is to adopt the strategy used by 8B10B, dividing the input bits into groups, and translating each group separately.

In short, our Verilog simulations show that to reduce encoding circuit complexity, we need to keep the state space small and avoid large translation tables — an unsurprising result.

Generally speaking, reducing circuit size and complexity have been traditional circuit design goals. Reducing state space and translation tables size are natural ways to achieve this goal, and have been routinely employed in the circuit design field. We

offer a new perspective here that the same techniques can reduce not just the cost of producing circuits, but also the circuits' operating energy costs. These traditional circuit design goals remain relevant in the context of green computing.

5.3. Verifying the Energy Model

We seek to verify our proposed energy model for Ethernet encodings, i.e., that encoding circuit energy is much larger than transmission energy. We calculate the transmission power from line voltage and insertion loss, and compare it against the total power of NIC cards to get a estimate of encoding circuit power.

We calculate the total transmission power in several steps. We take the RMS transmission voltage and DC resistance to find the RMS current, assuming sinusoidal waveforms. The power loss is given by $I^2_{RMS}R_{DC}$. The insertion loss for the cable tells us what fraction of the total transmission power is lost. We divide the power loss by this fraction to get the total transmission power. Fig. 4 illustrates our calculations.

The peak voltage for 100Mbps and 1Gbps over UTP is 1V. We take from technical specs for off-the-shelf Cat5e UTP Ethernet cables typical values for DC resistance and insertion loss. Substituting typical DC resistance of 9 Ohms at 100 meters and insertion loss of 10dB at 31.25 MHz for 100Mbps, and larger for 1Gbps, we find the transmission power to be approximately 0.062 W.

V_{peak} = Peak AC voltage on the cable

V_{RMS} = VDC equivalent of VAC = $\frac{V_{peak}}{\sqrt{2}}$ for sinusoids

I_{RMS} = DC equivalent current = $\frac{V_{RMS}}{R_{DC}}$

R_{DC} = DC resistance of the cable

P_{lost} = Transmission power lost in the cable = $I_{RMS}^2 R_{DC}$

$P_{transmit}$ = Total transmission power

i = Insertion loss in dB = $-10\log_{10}\left(\frac{P_{transmit} - P_{lost}}{P_{transmit}}\right)$

$$P_{transmit} = \frac{P_{lost}}{1 - 10^{-\frac{1}{10}i}}$$

Figure 4. Calculating Transmission Power

For encoding circuit power, we estimate by subtracting the transmission power from the total power for NICs. We obtain the power consumption of NICs from the technical specs for some off-the-shelf 1Gbps Ethernet NICs [26, 27, 28]. NIC power consumption ranges from 3.3W to 5W.

Average NIC power is typically lower than stated peak power rating. Even if we make a conservative assumption that average power is 10% of peak power, we find the transmission power is 0.062W / (10% × 3.3W) = 19% of the total power consumption, and encoding circuit power is the remaining 81%. This calculation confirms our encoding energy model, that most of the energy is consumed in the encoding circuits rather than in data transmission.

Our energy model is indirectly verified by an independent study [29]. Recall the results from Section 5.B. indicate that 4D-PAM5 is almost an order of magnitude more complex than 4B5B and MLT-3. If circuit power is directly related to circuit size, our observations suggest that 4D-PAM5 consumes almost an order of magnitude more energy. In [29], using a completely different method, the NIC power is shown to indeed grow exponentially as link speed increases from 100Mbps to 1Gbps and 10Gbps. Hence, we believe in our assumption that encoding circuit power is correlated with NIC power. At the same time, we are acutely aware that NIC power is more than encoding power, since it also contains contributions from buffers, OS interfaces, Wake-on-LAN, and other functions.

We should also mention that even though the available NIC specs offer the same data rate, their power consumption is considerably different. The NIC from 2001 consumed 5 W [27]. A 2004 model consumed 4.5 W [28]. A product from 2006 consumes only 3.3 W [26]. We believe that these numbers indicate either a growing energy consciousness in the circuit design community, or improving technology that leads to decreased transistor features and reduced circuit area, thus lowering energy consumption as a side effect. Circuits for the same encodings can consume considerably less power. We are encouraged by these results, because our energy model suggests that there is room for energy savings both in simpler encoding for simpler circuits, and more energy efficient circuits for a given encoding.

For Ethernet over optical fiber, the calculations are more straight-forward. The transmission power is simply the optical output power of the laser diode, typically 20-40mW, the same order of magnitude as the transmission power for Ethernet over UTP. The power consumption for optical fiber NICs is near identical to counterpart UTP NICs in the same product family, as suggested by [28]. Hence we can extend the above discussion to say that for optical fiber, the encoding circuit energy is also much larger than the transmission energy.

6. Future Work and Conclusion

6.1. Key Insights

We developed several key insights regarding energy efficient Ethernet encodings.

We can look at the canonical encoded communication problem from a perspective that prioritizes energy. The goal of encoding is to deliver the prescribed data rate with the least energy, rather than to deliver the maximum data rate using a given energy budget. We believe this energy conscious perspective is helpful in designing future encodings. For wired Ethernet, the encoding circuit energy is much larger than the transmission energy. This reverses the energy model for wireless encodings, where the encoding circuit energy is negligible, and the transmission energy dominates.

In the near future, the encoding circuit energy is likely to take an even larger share of the total energy for encodings, with lower transmission voltage over UTP and more complex encodings for 10Gbps and beyond.

Our proposed alternative to MLT-3 and our simulations shows that existing Ethernet encodings may not be energy efficient. In particular, we can reduce transmission energy by devising encodings for which a large fraction of encoded symbols have low energy.

Also, given that encoding circuit energy is much larger than transmission energy, we can get more energy savings by using simpler code. Simpler code means small translation tables, and state machines with a smaller state space. Our recommendation for simpler code is mirrored in suggestions for energy conscious wireless applications, where on-off-keying is also championed as the preferred encoding scheme [30].

Summarizing our insights, we believe that the ideal energy efficient Ethernet encoding for 1Gbps and beyond should have the following characteristics:

- Meets the prescribed data rate, bandwidth, and error rate constraints
- Has small state space – smaller circuits, reduced circuit power
- Has high bits per symbol – slower circuits, reduced circuit power density
- Uses low transmission voltage – reduced transmission power

6.2. Future Work

We make several recommendations for future work in energy efficient Ethernet encodings, in energy conscious encoding in general, and in energy efficient Internet.

For energy efficient Ethernet encodings, a more rigorous method is required for verifying or measuring encoding circuit energy. Our approximations, although reasonable, is not sufficiently fine grained. Also, the evaluations here should be extended to emerging encodings for 10Gbps Ethernet and even higher speeds. Furthermore, we should extend the simulations in Section 5.B to include the decoder; for some encodings, the encoder and decoder may be highly asymmetric in complexity.

Last but not least, any new encodings should ideally be backwards compatible, although Greenfield datacenters give opportunities for the entire network stack to be redesigned. For work on energy conscious communications in general, we believe our

alternative view on the encoded communication problem would be helpful. Our approach to understand the relative energy consumptions between encoding circuit power and transmission power should also be helpful.

Energy efficient Ethernet encodings is one of many ways to improve Internet energy efficiency. We hope our work invites a re-examination of the established assumptions and practices of the network stack with respect to energy.

6.3. Closing

The energy efficient Ethernet is a new study to reduce Internet energy consumption. Power considerations make it worthwhile to reconsider the choice of link layer encodings.

We evaluated existing encodings and proposed a new energy efficient encoding. Our study showed that simpler encoding is better, and encodings can be made more power efficient by being energy conscious.

Our work is a first step in this area. There are likely to be encodings even better than the alternative to MLT-3 we proposed. One colleague suggested another alternative to MLT-3 that has a slight advantage in transmission energy (22.5% saving vs. 18% saving for us) at the cost of a disadvantage in circuit complexity (same complexity as MLT-3 vs. considerable savings for us). The suggested alternative is DC balanced.

We hope our work would catalyze a thorough reexamination of Ethernet encodings, leading to potentially a flurry of proposals for energy efficient alternatives. The tradeoffs for all these encodings are worthy of further analysis and implementation.

Acknowledgements

Sincere thanks to Tracy Xiaoxiao Wang, my collaborator on an earlier version of this work.

I also thank the following people for their help, suggestions, and pointers to related work: David Tse, Wael Diab, Jean Walrand, Bruce Nordman, and Martin Graham. We are especially grateful to Ken Christensen, Andrea Baldini, and Claudio DeSanti for their invaluable feedback on an early draft of this work.

References

[1] K. Kawamoto, J. Koomey, B. Nordman, R. Brown, M. Piette, M. Ting, A. Meier, "Electricity used by office equipment and network equipment in the US: Detailed report and appendices," Technical Report LBNL- 45917, Lawrence Berkeley National Laboratory, February 2001.

[2] C. Gunaratne, K. Christensen, B. Nordman, "Managing Energy Consumption Costs in Desktop PCs and LAN Switches with Proxying, Split TCP Connections, and Scaling of Link Speed," Intl. J. of Network Management, vol. 15, pp. 297-310, September 2005.

[3] Energy Information Administration, "Annual Energy Review 2007," U.S. Department of Energy, June 2008.

[4] M. Gupta, S. Grover, and S. Singh, "A Feasibility Study for Power Management in LAN Switches," Proc. of the IEEE Intl. Conf. on Network Protocols, pp. 361-371, October 2004.

[5] H. Singh and S. Singh, "Energy consumption of TCP reno, newreno, and sack in multi-hop wireless networks," ACM SIGMETRICS Performance Evaluation Review, vol. 30, pp. 201-206, June 2002.

[6] C. Gunaratne and K. Christensen, "Ethernet Adaptive Link Rate: System Design and Performance Evaluation," Proc. of the 31st IEEE Conf. on Local Computer Networks, pp. 28-35, November 2006.

[7] IEEE LAN/MAN Standards Committee, “IEEE Std 802.3-2005 Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications,” IEEE, December 2005.

[8] National Standards Institute, “Fibre Distributed Data Interface (FDDI) - Token Ring Twisted Pair Physical Layer Medium Dependent (TP-PMD) (formerly ANSI X3.263-1995 (R2000)),” ANSI, 2000.

[9] M. Gupta, S. Sigh, “Greening of the Internet,” Proc. Of ACM SIGCOMM, pp.19-26, August 2003.

[10] L. Feeny and M. Nilsson, “Investigating the energy consumption of a wireless network interface in an ad hoc networking environment,” Proc. of the 20th INFOCOM Annual Joint Conf. of the IEEE Computer and Communications Societies, vol. 3, pp.1548-1557, April 2001.

[11] J. M. Kahn, R. H. Katz, and K. S. J. Pister, “Mobile Networking for Smart Dust,” Proc. Of 5th Intl. Conf. on Mobile Computing and Networking, pp.271-278, August 1999.

[12] M. Zorzi, M. Rossi, and G. Mazzini, “Throughput and energy performance of TCP on a wideband CDMA air interface,” J. of Wireless Communications and Mobile Computing, vol. , pp. 71-84, December 2002.

[13] “Green Computing and D-Link,” white paper, D-Link, Sepetember 2007, ftp://ftp10.dlink.com/pdfs/products/D-Link_Green_whitepaper.pdf, last accessed April 2008.

[14] S. Gurumurthi, A. Sivasubramaniam, and V. Natarajan, “Disk Drive Roadmap from the Thermal Perspective: A Case for Dynamic Thermal Management,” Proc. of the 32nd Intl. Symp. on Computer Architecture, pp. 38-49, June 2005.

[15] P. Ranganathan, P. Leech, D. Irwin, and J. Chase, “Ensemble-level power management for dense blade servers,” Proc. of the 33rd Intl. Symp. on Computer Architecture, vol. 34, pp. 66–77, October 2006.

[16] “Magic Packet Technology,” White Paper, Advanced Micro Devices, November 1995, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/20213.pdf, last accessed April 2008.

[17] B. Wang and S. Singh, “Computational Energy Cost of TCP,” Proc. Of the 23rd Annual Joint Conf. of the IEEE Computer and Communications Societies, vol. 2, pp. 785-795, March 2004.

[18] L. Irish, and K. Christensen, “A ‘Green TCP/IP’ to reduce electricity consumed by computers,” Proc. of IEEE Southeastcon 1998, pp. 302–305, April 1998.

[19] K. Christensen, “The next frontier for communications networks: Power management,” Proc. of SPIE – Performance and Control of Next-Generation Communications Networks, vol. 5244, pp. 1–4, August 2003.

[20] “802.3az Energy Efficient Ethernet Task Force,” meeting materials, IEEE, January 2008, http://www.ieee802.org/3/az/public/jan08/, last accessed April 2008.

[21] “Lower-Power Modes for ADSL2 and ADSL2+,” white paper, Texas Instruments, January 2005, http://focus.ti.com/pdfs/bcg/adsl2_low_power_wp.pdf, last accessed April 2008.

[22] S. Cui, A. J. Goldsmith, and A. Bahai, “Energy-constrained modulation optimization,” IEEE Trans. Wireless Communications, vol. 4, pp. 2349-2360, September 2005.

[23] J. Kleinschmidt, W. Borelli, and M. Pellenz, “Power Efficient Error Control for Bluetooth-based Sensor Networks,” Proc. of the 30th IEEE Conf. on Local Computer Networks, pp. 284-293, November 2005.

[24] J. Sanchez, P. Ruiz, “LEMA: Localized Energy-Efficient Multicast Algorithm based on Geographic Routing,” Proc. of the 31st IEEE Conf. on Local Computer Networks, pp. 3-12, November 2006.

[25] “Implementing FDDI Over Copper, The ANSI X3T9.5 Standard,” application note, Advanced Micro Devices, 1993, http://www.amd.com/files/connectivitysolutions/networking/archivednetworking/18258.pdf, last accessed April 2008.

[26] “Intel PRO/1000 PT Desktop Adapter,” tech. spec., Intel Corporation, 2006, http://www.intel.com/network/connectivity/products/prodbrf/pro1000_pt_desktop_adapter.pdf, last accessed April 2008.

[27] “Instant Gigabit Network Adapter,” tech. spec., Linksys, 2001, http://www.kodeks.hr/linksys/Assets/Pdf/linksys_eg1032.pdf, last accessed April 2008.

[28] “HP 3X-DEGXA-XX PCI-X/PCI Gigabit Ethernet Network Interface Controllers,” tech. spec. Hewlett-Packard Company, 2004, http://h18002.www1.hp.com/alphaserver/download/ek-degxa-in-b01-web.pdf, last accessed April 2008.

[29] Bruce Nordman, “Energy, Networks, Electronics, Data Centers,” Technical Presentation, Lawrence Berkeley National Laboratory, October 2007, unpublished.

[30] J. Ammer and J. Rabaey, “The Energy-per-Useful-Bit Metric for Evaluating and Optimizing Sensor Network Physical Layers,” Proc. Of the IEEE Intl. Workshop on Wireless Ad Hoc & Sensor Networks, vol. 2, pp. 695-700, September 2006.

Appendix 1 – Excerpts from IEEE Ethernet Specifications

The following are excerpts from IEEE 802.3 specifications for Ethernet encodings, the authoritative source for the subject. The specifications are very lengthy, so we quote only the sections most relevant to the encodings that we examined for 100Mbps and 1Gbps.

Each section describes the Physical Coding Sublayer (PCS) for the encoding, which is responsible for converting the bit stream from the MAC layer and the Media Independent Interface (MII) to a format required by the Physical Media Attachment sublayer (PMA).

The relationship between the sublayers is shown in Figure 2.

Appendix 1.a. – Excerpts for 4B5B encoding

24.2 Physical Coding Sublayer (PCS)

24.2.1 Service Interface (MII)

The PCS Service Interface allows the 100BASE-X PCS to transfer information to and from the MAC (via the Reconciliation sublayer) or other PCS client, such as a repeater. The PCS Service Interface is precisely defined as the Media Independent Interface (MII) in Clause 22.

In this clause, the setting of MII variables to TRUE or FALSE is equivalent, respectively, to “asserting” or “de-asserting” them as specified in Clause 22.

24.2.2 Functional requirements

The PCS comprises the Transmit, Receive, and Carrier Sense functions for 100BASE-T. In addition, the collisionDetect signal required by the MAC (COL on the MII) is derived from the PMA code-bit stream. The PCS shields the Reconciliation sublayer (and MAC) from the specific nature of the underlying channel. Specifically for receiving, the 100BASE-X PCS passes to the MII a sequence of data nibbles derived from incoming code-groups, each comprised of five code-bits, received from the medium. Code-group alignment and MAC packet delimiting is performed by embedding special non-data code-groups. The MII uses a nibble-wide, synchronous data path, with packet delimiting being provided by separate TX_EN and RX_DV signals. The PCS provides the functions necessary to map these two views of the exchanged data. The process is reversed for transmit.

The following provides a detailed specification of the functions performed by the PCS, which comprise five parallel processes (Transmit, Transmit Bits, Receive, Receive Bits, and Carrier Sense). Figure 24–4 includes a functional block diagram of the PCS.

The Receive Bits process accepts continuous code-bits via the PMA_UNITDATA.indicate primitive. Receive monitors these bits and generates RXD <3:0>, RX_DV and RX_ER on the MII, and the internal flag, receiving, used by the Carrier Sense and Transmit processes.

The Transmit process generates continuous code-groups based upon the TXD <3:0>, TX_EN, and TX_ER signals on the MII. These code-groups are transmitted by Transmit Bits via the PMA_UNITDATA.request primitive. The Transmit process generates the MII signal COL based on whether a reception is occurring simultaneously with transmission. Additionally, it generates the internal flag, transmitting, for use by the Carrier Sense process.

The Carrier Sense process asserts the MII signal CRS when either transmitting or receiving is TRUE. Both the Transmit and Receive processes monitor link_status via the PMA_LINK.indicate primitive, to account for potential link failure conditions.

24.2.2.1 Code-groups

The PCS maps four-bit nibbles from the MII into five-bit code-groups, and vice versa, using a 4B/5B block coding scheme. A code-group is a consecutive sequence of five code-bits interpreted and mapped by the PCS. Implicit in the definition of a code-group is an establishment of code-group boundaries by an alignment function within the PCS Receive process. It is important to note that, with the sole exception of the SSD, which is used to achieve alignment, code-groups are undetectable and have no meaning outside the 100BASE-X physical protocol data unit, called a “stream.”

The coding method used, derived from ISO/IEC 9314-1, provides

a) Adequate codes (32) to provide for all Data code-groups (16) plus necessary control code-groups;

b) Appropriate coding efficiency (4 data bits per 5 code-bits; 80%) to effect a 100 Mb/s Physical Layer interface on a 125 Mb/s physical channel as provided by FDDI PMDs; and
c) Sufficient transition density to facilitate clock recovery (when not scrambled).

Table 24–1 specifies the interpretation assigned to each five bit code-group, including the mapping to the nibble-wide (TXD or RXD) Data signals on the MII. The 32 code-groups are divided into four categories, as shown.

For clarity in the remainder of this clause, code-group names are shown between /slashes/. Code-group sequences are shown in succession, e.g., /1/2/....

The indicated code-group mapping is identical to ISO/IEC 9314-1: 1989, with four exceptions:

a) The FDDI term *symbol* is avoided in order to prevent confusion with other 100BASE-T terminology. In general, the term *code-group* is used in its place.
b) The /S/ and /Q/ code-groups are not used by 100BASE-X and are interpreted as INVALID.
c) The /R/ code-group is used in 100BASE-X as the second code-group of the End-of-Stream delimiter rather than to indicate a Reset condition.
d) The /H/ code-group is used to propagate receive errors rather than to indicate the Halt Line State.

24.2.2.1.1 Data code-groups

A Data code-group conveys one nibble of arbitrary data between the MII and the PCS. The sequence of Data code-groups is arbitrary, where any Data code-group can be followed by any other Data code-group. Data code-groups are coded and decoded but not interpreted by the PCS. Successful decoding of Data code-groups depends on proper receipt of the Start-of-Stream delimiter sequence, as defined in Table 24–1.

24.2.2.1.2 Idle code-groups

The Idle code-group (/I/) is transferred between streams. It provides a continuous fill pattern to establish and maintain clock synchronization. Idle code-groups are emitted from, and interpreted by, the PCS.

24.2.2.1.3 Control code-groups

The Control code-groups are used in pairs (/J/K/, /T/R/) to delimit MAC packets. Control code-groups are emitted from, and interpreted by, the PCS.

24.2.2.1.4 Start-of-Stream delimiter (/J/K/)

A Start-of-Stream delimiter (SSD) is used to delineate the boundary of a data transmission sequence and to authenticate carrier events. The SSD is unique in that it may be recognized independently of previously established code-group boundaries. The Receive function within the PCS uses the SSD to establish code-group boundaries. A SSD consists of the sequence /J/K/.

On transmission, the first 8 bits of the MAC preamble are replaced by the SSD, a replacement that is reversed on reception.

24.2.2.1.5 End-of-Stream delimiter (/T/R/)

An End-of-Stream delimiter (ESD) terminates all normal data transmissions. Unlike the SSD, an ESD cannot be recognized independent of previously established code-group boundaries. An ESD consists of the sequence /T/R/.

Table 24–1—4B/5B code-groups

	PCS code-group [4:0] 4 3 2 1 0	Name	MII (TXD/RXD) <3:0> 3 2 1 0	Interpretation
DATA	1 1 1 1 0	0	0 0 0 0	Data 0
	0 1 0 0 1	1	0 0 0 1	Data 1
	1 0 1 0 0	2	0 0 1 0	Data 2
	1 0 1 0 1	3	0 0 1 1	Data 3
	0 1 0 1 0	4	0 1 0 0	Data 4
	0 1 0 1 1	5	0 1 0 1	Data 5
	0 1 1 1 0	6	0 1 1 0	Data 6
	0 1 1 1 1	7	0 1 1 1	Data 7
	1 0 0 1 0	8	1 0 0 0	Data 8
	1 0 0 1 1	9	1 0 0 1	Data 9
	1 0 1 1 0	A	1 0 1 0	Data A
	1 0 1 1 1	B	1 0 1 1	Data B
	1 1 0 1 0	C	1 1 0 0	Data C
	1 1 0 1 1	D	1 1 0 1	Data D
	1 1 1 0 0	E	1 1 1 0	Data E
	1 1 1 0 1	F	1 1 1 1	Data F
	1 1 1 1 1	I	undefined	IDLE; used as inter-stream fill code
CONTROL	1 1 0 0 0	J	0 1 0 1	Start-of-Stream Delimiter, Part 1 of 2; always used in pairs with K
	1 0 0 0 1	K	0 1 0 1	Start-of-Stream Delimiter, Part 2 of 2; always used in pairs with J
	0 1 1 0 1	T	undefined	End-of-Stream Delimiter, Part 1 of 2; always used in pairs with R
	0 0 1 1 1	R	undefined	End-of-Stream Delimiter, Part 2 of 2; always used in pairs with T
INVALID	0 0 1 0 0	H	Undefined	Transmit Error; used to force signaling errors
	0 0 0 0 0	V	Undefined	Invalid code
	0 0 0 0 1	V	Undefined	Invalid code
	0 0 0 1 0	V	Undefined	Invalid code
	0 0 0 1 1	V	Undefined	Invalid code
	0 0 1 0 1	V	Undefined	Invalid code
	0 0 1 1 0	V	Undefined	Invalid code
	0 1 0 0 0	V	Undefined	Invalid code
	0 1 1 0 0	V	Undefined	Invalid code
	1 0 0 0 0	V	Undefined	Invalid code
	1 1 0 0 1	V	Undefined	Invalid code

Appendix 1.b. – Excerpts for 8B10B encoding

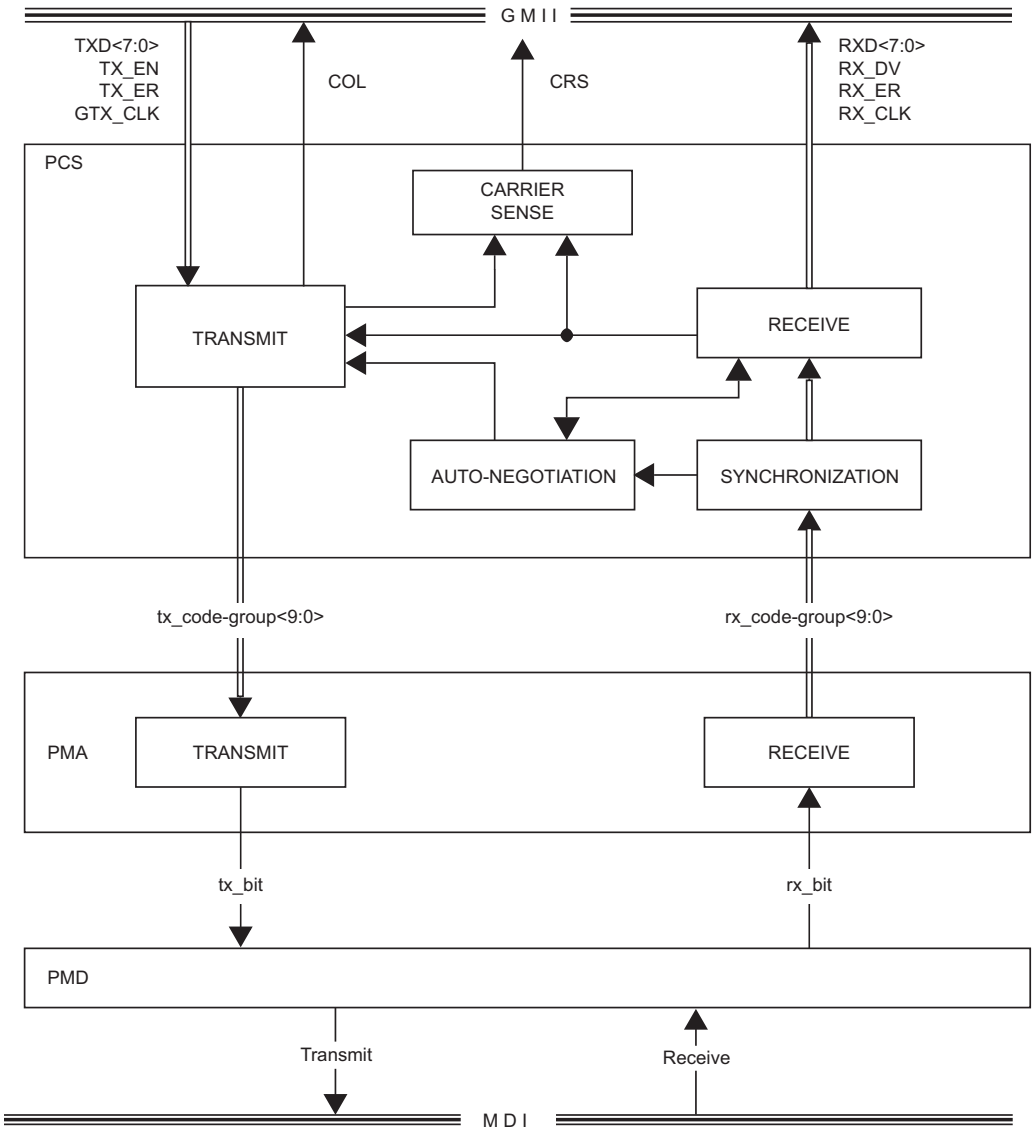

Figure 36–2—Functional block diagram

36.2 Physical Coding Sublayer (PCS)

36.2.1 PCS Interface (GMII)

The PCS Service Interface allows the 1000BASE-X PCS to transfer information to and from a PCS client. PCS clients include the MAC (via the Reconciliation sublayer) and repeater. The PCS Interface is precisely defined as the Gigabit Media Independent Interface (GMII) in Clause 35.

In this clause, the setting of GMII variables to TRUE or FALSE is equivalent, respectively, to “asserting” or “de-asserting” them as specified in Clause 35.

36.2.2 Functions within the PCS

The PCS comprises the PCS Transmit, Carrier Sense, Synchronization, PCS Receive, and Auto-Negotiation processes for 1000BASE-X. The PCS shields the Reconciliation sublayer (and MAC) from the specific nature of the underlying channel. When communicating with the GMII, the PCS uses an octet-wide, synchronous data path, with packet delimiting being provided by separate transmit control signals (TX_EN and TX_ER) and receive control signals (RX_DV and RX_ER). When communicating with the PMA, the PCS uses a ten-bit wide, synchronous data path, which conveys ten-bit code-groups. At the PMA Service Interface, code-group alignment and MAC packet delimiting are made possible by embedding special non-data code-groups in the transmitted code-group stream. The PCS provides the functions necessary to map packets between the GMII format and the PMA Service Interface format.

The PCS Transmit process continuously generates code-groups based upon the TXD <7:0>, TX_EN, and TX_ER signals on the GMII, sending them immediately to the PMA Service Interface via the PMA_UNITDATA.request primitive. The PCS Transmit process generates the GMII signal COL based on whether a reception is occurring simultaneously with transmission. Additionally, it generates the internal flag, transmitting, for use by the Carrier Sense process. The PCS Transmit process monitors the Auto-Negotiation process xmit flag to determine whether to transmit data or reconfigure the link.

The Carrier Sense process controls the GMII signal CRS (see Figure 36–8).

The PCS Synchronization process continuously accepts code-groups via the PMA_UNITDATA.indication primitive and conveys received code-groups to the PCS Receive process via the SYNC_UNITDATA.indicate primitive. The PCS Synchronization process sets the sync_status flag to indicate whether the PMA is functioning dependably (as well as can be determined without exhaustive error-rate analysis).

The PCS Receive process continuously accepts code-groups via the SYNC_UNITDATA.indicate primitive. The PCS Receive process monitors these code-groups and generates RXD <7:0>, RX_DV, and RX_ER on the GMII, and the internal flag, receiving, used by the Carrier Sense and Transmit processes.

The PCS Auto-Negotiation process sets the xmit flag to inform the PCS Transmit process to either transmit normal idles interspersed with packets as requested by the GMII or to reconfigure the link. The PCS Auto-Negotiation process is specified in Clause 37.

36.2.3 Use of code-groups

The PCS maps GMII signals into ten-bit code groups, and vice versa, using an 8B/10B block coding scheme. Implicit in the definition of a code-group is an establishment of code-group boundaries by a PMA code-group alignment function as specified in 36.3.2.4. Code-groups are unobservable and have no meaning outside the PCS. The PCS functions ENCODE and DECODE generate, manipulate, and interpret code-groups as provided by the rules in 36.2.4.

36.2.4 8B/10B transmission code

The PCS uses a transmission code to improve the transmission characteristics of information to be transferred across the link. The encodings defined by the transmission code ensure that sufficient transitions are present in the PHY bit stream to make clock recovery possible at the receiver. Such encoding also greatly increases the likelihood of detecting any single or multiple bit errors that may occur during transmission and reception of information. In addition, some of the special code-groups of the transmission code contain a distinct and easily recognizable bit pattern that assists a receiver in achieving code-group alignment on the incoming PHY bit stream. The 8B/10B transmission code specified for use in this standard has a high transition density, is a run-length-limited code, and is dc-balanced. The transition density of the 8B/10B symbols ranges from 3 to 8 transitions per symbol.

The definition of the 8B/10B transmission code in this standard is identical to that specified in ANSI X3.230-1994 (FC-PH), Clause 11. The relationship of code-group bit positions to PMA and other PCS constructs is illustrated in Figure 36–3.

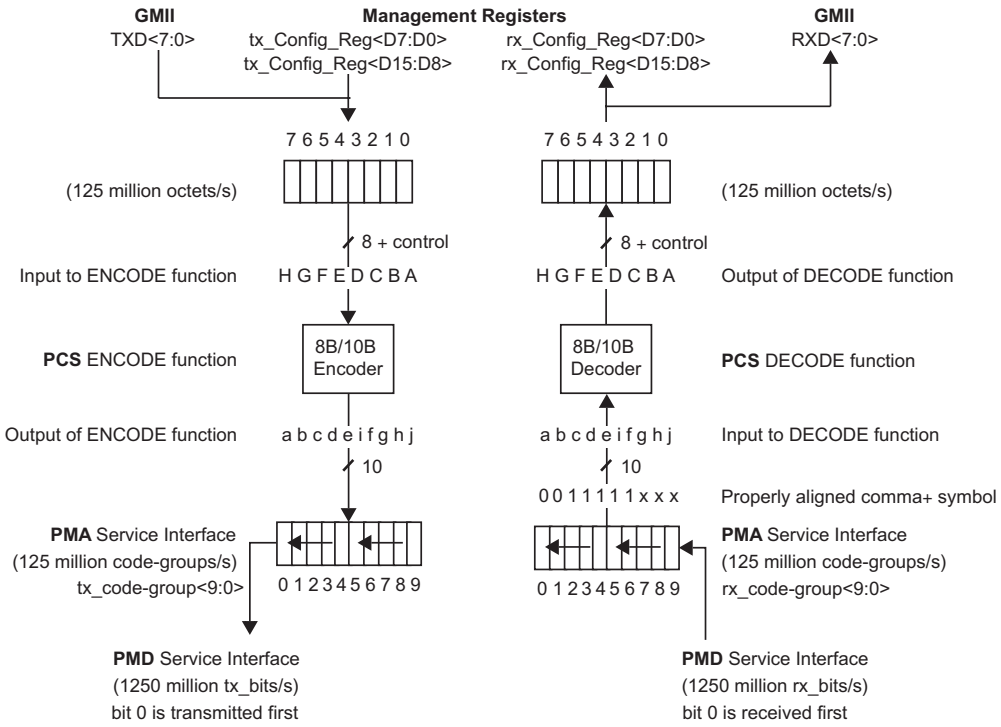

Figure 36–3—PCS reference diagram

36.2.4.1 Notation conventions

8B/10B transmission code uses letter notation for describing the bits of an unencoded information octet and a single control variable. Each bit of the unencoded information octet contains either a binary zero or a binary one. A control variable, Z, has either the value D or the value K. When the control variable associated with an unencoded information octet contains the value D, the associated encoded code-group is referred to as a data code-group. When the control variable associated with an unencoded information octet contains the value K, the associated encoded code-group is referred to as a special code-group.

The bit notation of A,B,C,D,E,F,G,H for an unencoded information octet is used in the description of the 8B/10B transmission code. The bits A,B,C,D,E,F,G,H are translated to bits a,b,c,d,e,i,f,g,h,j of 10-bit transmission code-groups. 8B/10B code-group bit assignments are illustrated in Figure 36–3. Each valid code-group has been given a name using the following convention: /Dx.y/ for the 256 valid data code-groups, and /Kx.y/ for special control code-groups, where x is the decimal value of bits EDCBA, and y is the decimal value of bits HGF.

36.2.4.2 Transmission order

Code-group bit transmission order is illustrated in Figure 36–3.

Code-groups within multi-code-group ordered_sets (as specified in Table 36–3) are transmitted sequentially beginning with the special code-group used to distinguish the ordered_set (e.g., /K28.5/) and proceeding code-group by code-group from left to right within the definition of the ordered_set until all code-groups of the ordered_set are transmitted.

The first code-group of every multi-code-group ordered_set is transmitted in an even-numbered code-group position counting from the first code-group after a reset or power-on. Subsequent code-groups continuously alternate as odd and even-numbered code-groups.

The contents of a packet are transmitted sequentially beginning with the ordered_set used to denote the Start_of_Packet (the SPD delimiter) and proceeding code-group by code-group from left to right within the definition of the packet until the ordered_set used to denote the End_of_Packet (the EPD delimiter) is transmitted.

36.2.4.3 Valid and invalid code-groups

Table 36–1a defines the valid data code-groups (D code-groups) of the 8B/10B transmission code. Table 36–2 defines the valid special code-groups (K code-groups) of the code. The tables are used for both generating valid code-groups (encoding) and checking the validity of received code-groups (decoding). In the tables, each octet entry has two columns that represent two (not necessarily different) code-groups. The two columns correspond to the valid code-group based on the current value of the running disparity (Current RD – or Current RD +). Running disparity is a binary parameter with either the value negative (–) or the value positive (+). Annex 36B provides several 8B/10B transmission code running disparity calculation examples.

36.2.4.4 Running disparity rules

After powering on or exiting a test mode, the transmitter shall assume the negative value for its initial running disparity. Upon transmission of any code-group, the transmitter shall calculate a new value for its running disparity based on the contents of the transmitted code-group.

After powering on or exiting a test mode, the receiver should assume either the positive or negative value for its initial running disparity. Upon the reception of any code-group, the receiver determines whether the code-group is valid or invalid and calculates a new value for its running disparity based on the contents of the received code-group.

The following rules for running disparity shall be used to calculate the new running disparity value for code-groups that have been transmitted (transmitter's running disparity) and that have been received (receiver’s running disparity).

Running disparity for a code-group is calculated on the basis of sub-blocks, where the first six bits (abcdei) form one sub-block (six-bit sub-block) and the second four bits (fghj) form the other sub-block (four-bit sub-block). Running disparity at the beginning of the six-bit sub-block is the running disparity at the end of the last code-group. Running disparity at the beginning of the four-bit sub-block is the running disparity at the end of the six-bit sub-block. Running disparity at the end of the code-group is the running disparity at the end of the four-bit sub-block.

Running disparity for the sub-blocks is calculated as follows:

a) Running disparity at the end of any sub-block is positive if the sub-block contains more ones than zeros. It is also positive at the end of the six-bit sub-block if the six-bit sub-block is 000111, and it is positive at the end of the four-bit sub-block if the four-bit sub-block is 0011;
b) Running disparity at the end of any sub-block is negative if the sub-block contains more zeros than ones. It is also negative at the end of the six-bit sub-block if the six-bit sub-block is 111000, and it is negative at the end of the four-bit sub-block if the four-bit sub-block is 1100;
c) Otherwise, running disparity at the end of the sub-block is the same as at the beginning of the sub-block.

NOTE—All sub-blocks with equal numbers of zeros and ones are disparity neutral. In order to limit the run length of 0's or 1's between sub-blocks, the 8B/10B transmission code rules specify that sub-blocks encoded as 000111 or 0011 are generated only when the running disparity at the beginning of the sub-block is positive; thus, running disparity at the end of these sub-blocks is also positive. Likewise, sub-blocks containing 111000 or 1100 are generated only when the running disparity at the beginning of the sub-block is negative; thus, running disparity at the end of these sub-blocks is also negative.

36.2.4.5 Generating code-groups

The appropriate entry in either Table 36–1a or Table 36–2 is found for each octet for which a code-group is to be generated (encoded). The current value of the transmitter's running disparity shall be used to select the code-group from its corresponding column. For each code-group transmitted, a new value of the running disparity is calculated. This new value is used as the transmitter's current running disparity for the next octet to be encoded and transmitted.

36.2.4.6 Checking the validity of received code-groups

The following rules shall be used to determine the validity of received code groups:

a) The column in Tables 36–1a and 36–2 corresponding to the current value of the receiver's running disparity is searched for the received code-group;
b) If the received code-group is found in the proper column, according to the current running disparity, then the code-group is considered valid and, for data code-groups, the associated data octet determined (decoded);
c) If the received code-group is not found in that column, then the code-group is considered invalid;
d) Independent of the code-group's validity, the received code-group is used to calculate a new value of running disparity. The new value is used as the receiver's current running disparity for the next received code-group.

Detection of an invalid code-group does not necessarily indicate that the code-group in which the invalid code-group was detected is in error. Invalid code-groups may result from a prior error which altered the running disparity of the PHY bit stream but which did not result in a detectable error at the code-group in which the error occurred.

The number of invalid code-groups detected is proportional to the bit error ratio (BER) of the link. Link error monitoring may be performed by counting invalid code-groups.

36.2.4.7 Ordered_sets

Eight ordered_sets, consisting of a single special code-group or combinations of special and data code-groups are specifically defined. Ordered_sets which include /K28.5/ provide the ability to obtain bit and code-group synchronization and establish ordered_set alignment (see 36.2.4.9 and 36.3.2.4). Ordered_sets provide for the delineation of a packet and synchronization between the transmitter and receiver circuits at opposite ends of a link. Table 36–3 lists the defined ordered_sets.

36.2.4.7.1 Ordered_set rules

Ordered_sets are specified according to the following rules:

a) Ordered_sets consist of either one, two, or four code-groups;
b) The first code-group of all ordered_sets is always a special code-group;
c) The second code-group of all multi-code-group ordered_sets is always a data code-group. The second code-group is used to distinguish the ordered set from all other ordered sets. The second code-group provides a high bit transition density.

Table 36–1a—Valid data code-groups

Code Group Name	Octet Value	Octet Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D0.0	00	000 00000	100111 0100	011000 1011
D1.0	01	000 00001	011101 0100	100010 1011
D2.0	02	000 00010	101101 0100	010010 1011
D3.0	03	000 00011	110001 1011	110001 0100
D4.0	04	000 00100	110101 0100	001010 1011
D5.0	05	000 00101	101001 1011	101001 0100
D6.0	06	000 00110	011001 1011	011001 0100
D7.0	07	000 00111	111000 1011	000111 0100
D8.0	08	000 01000	111001 0100	000110 1011
D9.0	09	000 01001	100101 1011	100101 0100
D10.0	0A	000 01010	010101 1011	010101 0100
D11.0	0B	000 01011	110100 1011	110100 0100
D12.0	0C	000 01100	001101 1011	001101 0100
D13.0	0D	000 01101	101100 1011	101100 0100
D14.0	0E	000 01110	011100 1011	011100 0100
D15.0	0F	000 01111	010111 0100	101000 1011
D16.0	10	000 10000	011011 0100	100100 1011
D17.0	11	000 10001	100011 1011	100011 0100
D18.0	12	000 10010	010011 1011	010011 0100
D19.0	13	000 10011	110010 1011	110010 0100
D20.0	14	000 10100	001011 1011	001011 0100
D21.0	15	000 10101	101010 1011	101010 0100
D22.0	16	000 10110	011010 1011	011010 0100
D23.0	17	000 10111	111010 0100	000101 1011
D24.0	18	000 11000	110011 0100	001100 1011
D25.0	19	000 11001	100110 1011	100110 0100
D26.0	1A	000 11010	010110 1011	010110 0100
D27.0	1B	000 11011	110110 0100	001001 1011
D28.0	1C	000 11100	001110 1011	001110 0100
D29.0	1D	000 11101	101110 0100	010001 1011
D30.0	1E	000 11110	011110 0100	100001 1011
D31.0	1F	000 11111	101011 0100	010100 1011
D0.1	20	001 00000	100111 1001	011000 1001
D1.1	21	001 00001	011101 1001	100010 1001
D2.1	22	001 00010	101101 1001	010010 1001
D3.1	23	001 00011	110001 1001	110001 1001
D4.1	24	001 00100	110101 1001	001010 1001
D5.1	25	001 00101	101001 1001	101001 1001
D6.1	26	001 00110	011001 1001	011001 1001
D7.1	27	001 00111	111000 1001	000111 1001
D8.1	28	001 01000	111001 1001	000110 1001
D9.1	29	001 01001	100101 1001	100101 1001
D10.1	2A	001 01010	010101 1001	010101 1001
D11.1	2B	001 01011	110100 1001	110100 1001
D12.1	2C	001 01100	001101 1001	001101 1001
D13.1	2D	001 01101	101100 1001	101100 1001
D14.1	2E	001 01110	011100 1001	011100 1001
D15.1	2F	001 01111	010111 1001	101000 1001
D16.1	30	001 10000	011011 1001	100100 1001
D17.1	31	001 10001	100011 1001	100011 1001
D18.1	32	001 10010	010011 1001	010011 1001
D19.1	33	001 10011	110010 1001	110010 1001
D20.1	34	001 10100	001011 1001	001011 1001
D21.1	35	001 10101	101010 1001	101010 1001
D22.1	36	001 10110	011010 1001	011010 1001
D23.1	37	001 10111	111010 1001	000101 1001
D24.1	38	001 11000	110011 1001	001100 1001
D25.1	39	001 11001	100110 1001	100110 1001
D26.1	3A	001 11010	010110 1001	010110 1001
D27.1	3B	001 11011	110110 1001	001001 1001

(continued)

Table 36–1b—Valid data code-groups

Code Group Name	Octet Value	Octet Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D28.1	3C	001 11100	001110 1001	001110 1001
D29.1	3D	001 11101	101110 1001	010001 1001
D30.1	3E	001 11110	011110 1001	100001 1001
D31.1	3F	001 11111	101011 1001	010100 1001
D0.2	40	010 00000	100111 0101	011000 0101
D1.2	41	010 00001	011101 0101	100010 0101
D2.2	42	010 00010	101101 0101	010010 0101
D3.2	43	010 00011	110001 0101	110001 0101
D4.2	44	010 00100	110101 0101	001010 0101
D5.2	45	010 00101	101001 0101	101001 0101
D6.2	46	010 00110	011001 0101	011001 0101
D7.2	47	010 00111	111000 0101	000111 0101
D8.2	48	010 01000	111001 0101	000110 0101
D9.2	49	010 01001	100101 0101	100101 0101
D10.2	4A	010 01010	010101 0101	010101 0101
D11.2	4B	010 01011	110100 0101	110100 0101
D12.2	4C	010 01100	001101 0101	001101 0101
D13.2	4D	010 01101	101100 0101	101100 0101
D14.2	4E	010 01110	011100 0101	011100 0101
D15.2	4F	010 01111	010111 0101	101000 0101
D16.2	50	010 10000	011011 0101	100100 0101
D17.2	51	010 10001	100011 0101	100011 0101
D18.2	52	010 10010	010011 0101	010011 0101
D19.2	53	010 10011	110010 0101	110010 0101
D20.2	54	010 10100	001011 0101	001011 0101
D21.2	55	010 10101	101010 0101	101010 0101
D22.2	56	010 10110	011010 0101	011010 0101
D23.2	57	010 10111	111010 0101	000101 0101
D24.2	58	010 11000	110011 0101	001100 0101
D25.2	59	010 11001	100110 0101	100110 0101
D26.2	5A	010 11010	010110 0101	010110 0101
D27.2	5B	010 11011	110110 0101	001001 0101
D28.2	5C	010 11100	001110 0101	001110 0101
D29.2	5D	010 11101	101110 0101	010001 0101
D30.2	5E	010 11110	011110 0101	100001 0101
D31.2	5F	010 11111	101011 0101	010100 0101
D0.3	60	011 00000	100111 0011	011000 1100
D1.3	61	011 00001	011101 0011	100010 1100
D2.3	62	011 00010	101101 0011	010010 1100
D3.3	63	011 00011	110001 1100	110001 0011
D4.3	64	011 00100	110101 0011	001010 1100
D5.3	65	011 00101	101001 1100	101001 0011
D6.3	66	011 00110	011001 1100	011001 0011
D7.3	67	011 00111	111000 1100	000111 0011
D8.3	68	011 01000	111001 0011	000110 1100
D9.3	69	011 01001	100101 1100	100101 0011
D10.3	6A	011 01010	010101 1100	010101 0011
D11.3	6B	011 01011	110100 1100	110100 0011
D12.3	6C	011 01100	001101 1100	001101 0011
D13.3	6D	011 01101	101100 1100	101100 0011
D14.3	6E	011 01110	011100 1100	011100 0011
D15.3	6F	011 01111	010111 0011	101000 1100
D16.3	70	011 10000	011011 0011	100100 1100
D17.3	71	011 10001	100011 1100	100011 0011
D18.3	72	011 10010	010011 1100	010011 0011
D19.3	73	011 10011	110010 1100	110010 0011
D20.3	74	011 10100	001011 1100	001011 0011
D21.3	75	011 10101	101010 1100	101010 0011
D22.3	76	011 10110	011010 1100	011010 0011
D23.3	77	011 10111	111010 0011	000101 1100

(continued)

Table 36–1c—Valid data code-groups

Code Group Name	Octet Value	Octet Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D24.3	78	011 11000	110011 0011	001100 1100
D25.3	79	011 11001	100110 1100	100110 0011
D26.3	7A	011 11010	010110 1100	010110 0011
D27.3	7B	011 11011	110110 0011	001001 1100
D28.3	7C	011 11100	001110 1100	001110 0011
D29.3	7D	011 11101	101110 0011	010001 1100
D30.3	7E	011 11110	011110 0011	100001 1100
D31.3	7F	011 11111	101011 0011	010100 1100
D0.4	80	100 00000	100111 0010	011000 1101
D1.4	81	100 00001	011101 0010	100010 1101
D2.4	82	100 00010	101101 0010	010010 1101
D3.4	83	100 00011	110001 1101	110001 0010
D4.4	84	100 00100	110101 0010	001010 1101
D5.4	85	100 00101	101001 1101	101001 0010
D6.4	86	100 00110	011001 1101	011001 0010
D7.4	87	100 00111	111000 1101	000111 0010
D8.4	88	100 01000	111001 0010	000110 1101
D9.4	89	100 01001	100101 1101	100101 0010
D10.4	8A	100 01010	010101 1101	010101 0010
D11.4	8B	100 01011	110100 1101	110100 0010
D12.4	8C	100 01100	001101 1101	001101 0010
D13.4	8D	100 01101	101100 1101	101100 0010
D14.4	8E	100 01110	011100 1101	011100 0010
D15.4	8F	100 01111	010111 0010	101000 1101
D16.4	90	100 10000	011011 0010	100100 1101
D17.4	91	100 10001	100011 1101	100011 0010
D18.4	92	100 10010	010011 1101	010011 0010
D19.4	93	100 10011	110010 1101	110010 0010
D20.4	94	100 10100	001011 1101	001011 0010
D21.4	95	100 10101	101010 1101	101010 0010
D22.4	96	100 10110	011010 1101	011010 0010
D23.4	97	100 10111	111010 0010	000101 1101
D24.4	98	100 11000	110011 0010	001100 1101
D25.4	99	100 11001	100110 1101	100110 0010
D26.4	9A	100 11010	010110 1101	010110 0010
D27.4	9B	100 11011	110110 0010	001001 1101
D28.4	9C	100 11100	001110 1101	001110 0010
D29.4	9D	100 11101	101110 0010	010001 1101
D30.4	9E	100 11110	011110 0010	100001 1101
D31.4	9F	100 11111	101011 0010	010100 1101
D0.5	A0	101 00000	100111 1010	011000 1010
D1.5	A1	101 00001	011101 1010	100010 1010
D2.5	A2	101 00010	101101 1010	010010 1010
D3.5	A3	101 00011	110001 1010	110001 1010
D4.5	A4	101 00100	110101 1010	001010 1010
D5.5	A5	101 00101	101001 1010	101001 1010
D6.5	A6	101 00110	011001 1010	011001 1010
D7.5	A7	101 00111	111000 1010	000111 1010
D8.5	A8	101 01000	111001 1010	000110 1010
D9.5	A9	101 01001	100101 1010	100101 1010
D10.5	AA	101 01010	010101 1010	010101 1010
D11.5	AB	101 01011	110100 1010	110100 1010
D12.5	AC	101 01100	001101 1010	001101 1010
D13.5	AD	101 01101	101100 1010	101100 1010
D14.5	AE	101 01110	011100 1010	011100 1010
D15.5	AF	101 01111	010111 1010	101000 1010
D16.5	B0	101 10000	011011 1010	100100 1010
D17.5	B1	101 10001	100011 1010	100011 1010
D18.5	B2	101 10010	010011 1010	010011 1010
D19.5	B3	101 10011	110010 1010	110010 1010

(continued)

Table 36–1d—Valid data code-groups

Code Group Name	Octet Value	Octet Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D20.5	B4	101 10100	001011 1010	001011 1010
D21.5	B5	101 10101	101010 1010	101010 1010
D22.5	B6	101 10110	011010 1010	011010 1010
D23.5	B7	101 10111	111010 1010	000101 1010
D24.5	B8	101 11000	110011 1010	001100 1010
D25.5	B9	101 11001	100110 1010	100110 1010
D26.5	BA	101 11010	010110 1010	010110 1010
D27.5	BB	101 11011	110110 1010	001001 1010
D28.5	BC	101 11100	001110 1010	001110 1010
D29.5	BD	101 11101	101110 1010	010001 1010
D30.5	BE	101 11110	011110 1010	100001 1010
D31.5	BF	101 11111	101011 1010	010100 1010
D0.6	C0	110 00000	100111 0110	011000 0110
D1.6	C1	110 00001	011101 0110	100010 0110
D2.6	C2	110 00010	101101 0110	010010 0110
D3.6	C3	110 00011	110001 0110	110001 0110
D4.6	C4	110 00100	110101 0110	001010 0110
D5.6	C5	110 00101	101001 0110	101001 0110
D6.6	C6	110 00110	011001 0110	011001 0110
D7.6	C7	110 00111	111000 0110	000111 0110
D8.6	C8	110 01000	111001 0110	000110 0110
D9.6	C9	110 01001	100101 0110	100101 0110
D10.6	CA	110 01010	010101 0110	010101 0110
D11.6	CB	110 01011	110100 0110	110100 0110
D12.6	CC	110 01100	001101 0110	001101 0110
D13.6	CD	110 01101	101100 0110	101100 0110
D14.6	CE	110 01110	011100 0110	011100 0110
D15.6	CF	110 01111	010111 0110	101000 0110
D16.6	D0	110 10000	011011 0110	100100 0110
D17.6	D1	110 10001	100011 0110	100011 0110
D18.6	D2	110 10010	010011 0110	010011 0110
D19.6	D3	110 10011	110010 0110	110010 0110
D20.6	D4	110 10100	001011 0110	001011 0110
D21.6	D5	110 10101	101010 0110	101010 0110
D22.6	D6	110 10110	011010 0110	011010 0110
D23.6	D7	110 10111	111010 0110	000101 0110
D24.6	D8	110 11000	110011 0110	001100 0110
D25.6	D9	110 11001	100110 0110	100110 0110
D26.6	DA	110 11010	010110 0110	010110 0110
D27.6	DB	110 11011	110110 0110	001001 0110
D28.6	DC	110 11100	001110 0110	001110 0110
D29.6	DD	110 11101	101110 0110	010001 0110
D30.6	DE	110 11110	011110 0110	100001 0110
D31.6	DF	110 11111	101011 0110	010100 0110
D0.7	E0	111 00000	100111 0001	011000 1110
D1.7	E1	111 00001	011101 0001	100010 1110
D2.7	E2	111 00010	101101 0001	010010 1110
D3.7	E3	111 00011	110001 1110	110001 0001
D4.7	E4	111 00100	110101 0001	001010 1110
D5.7	E5	111 00101	101001 1110	101001 0001
D6.7	E6	111 00110	011001 1110	011001 0001
D7.7	E7	111 00111	111000 1110	000111 0001
D8.7	E8	111 01000	111001 0001	000110 1110
D9.7	E9	111 01001	100101 1110	100101 0001
D10.7	EA	111 01010	010101 1110	010101 0001
D11.7	EB	111 01011	110100 1110	110100 1000
D12.7	EC	111 01100	001101 1110	001101 0001
D13.7	ED	111 01101	101100 1110	101100 1000
D14.7	EE	111 01110	011100 1110	011100 1000
D15.7	EF	111 01111	010111 0001	101000 1110

(continued)

Table 36–1e—Valid data code-groups

Code Group Name	Octet Value	Octet Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D16.7	F0	111 10000	011011 0001	100100 1110
D17.7	F1	111 10001	100011 0111	100011 0001
D18.7	F2	111 10010	010011 0111	010011 0001
D19.7	F3	111 10011	110010 1110	110010 0001
D20.7	F4	111 10100	001011 0111	001011 0001
D21.7	F5	111 10101	101010 1110	101010 0001
D22.7	F6	111 10110	011010 1110	011010 0001
D23.7	F7	111 10111	111010 0001	000101 1110
D24.7	F8	111 11000	110011 0001	001100 1110
D25.7	F9	111 11001	100110 1110	100110 0001
D26.7	FA	111 11010	010110 1110	010110 0001
D27.7	FB	111 11011	110110 0001	001001 1110
D28.7	FC	111 11100	001110 1110	001110 0001
D29.7	FD	111 11101	101110 0001	010001 1110
D30.7	FE	111 11110	011110 0001	100001 1110
D31.7	FF	111 11111	101011 0001	010100 1110

(concluded)

Table 36–2—Valid special code-groups

Code Group Name	Octet Value	Octet Bits HGF EDCBA	Current RD –	Current RD +	Notes
			abcdei fghj	abcdei fghj	
K28.0	1C	000 11100	001111 0100	110000 1011	1
K28.1	3C	001 11100	001111 1001	110000 0110	1,2
K28.2	5C	010 11100	001111 0101	110000 1010	1
K28.3	7C	011 11100	001111 0011	110000 1100	1
K28.4	9C	100 11100	001111 0010	110000 1101	1
K28.5	BC	101 11100	001111 1010	110000 0101	2
K28.6	DC	110 11100	001111 0110	110000 1001	1
K28.7	FC	111 11100	001111 1000	110000 0111	1,2
K23.7	F7	111 10111	111010 1000	000101 0111	
K27.7	FB	111 11011	110110 1000	001001 0111	
K29.7	FD	111 11101	101110 1000	010001 0111	
K30.7	FE	111 11110	011110 1000	100001 0111	

NOTE 1—Reserved.

NOTE 2—Contains a comma.

Table 36–3 lists the defined ordered_sets.

36.2.4.8 /K28.5/ code-group considerations

The /K28.5/ special code-group is chosen as the first code-group of all ordered_sets that are signaled repeatedly and for the purpose of allowing a receiver to synchronize to the incoming bit stream (i.e., /C/ and /I/), for the following reasons:

a) Bits abcdeif make up a comma. The comma can be used to easily find and verify code-group and ordered_set boundaries of the rx_bit stream.
b) Bits ghj of the encoded code-group present the maximum number of transitions, simplifying receiver acquisition of bit synchronization.

Table 36–3—Defined ordered_sets

Code	Ordered_Set	Number of Code-Groups	Encoding
/C/	**Configuration**		Alternating /C1/ and /C2/
/C1/	Configuration 1	4	/K28.5/D21.5/Config_Reg[a]
/C2/	Configuration 2	4	/K28.5/D2.2/Config_Reg[a]
/I/	**IDLE**		Correcting /I1/, Preserving /I2/
/I1/	IDLE 1	2	/K28.5/D5.6/
/I2/	IDLE 2	2	/K28.5/D16.2/
	Encapsulation		
/R/	Carrier_Extend	1	/K23.7/
/S/	Start_of_Packet	1	/K27.7/
/T/	End_of_Packet	1	/K29.7/
/V/	Error_Propagation	1	/K30.7/

[a]Two data code-groups representing the Config_Reg value.

36.2.4.9 Comma considerations

The seven bit comma string is defined as either b'0011111' (comma+) or b'1100000' (comma-). The /I/ and /C/ ordered_sets and their associated protocols are specified to ensure that comma+ is transmitted with either equivalent or greater frequency than comma- for the duration of their transmission. This is done to ensure compatibility with common components.

The comma contained within the /K28.1/, /K28.5/, and /K28.7/ special code-groups is a singular bit pattern, which, in the absence of transmission errors, cannot appear in any other location of a code-group and cannot be generated across the boundaries of any two adjacent code-groups with the following exception:

The /K28.7/ special code-group is used by 1000BASE-X for diagnostic purposes only (see Annex 36A). This code-group, if followed by any of the following special or data code-groups: /K28.x/, /D3.x/, /D11.x/, /D12.x/, /D19.x/, /D20.x/, or /D28.x/, where x is a value in the range 0 to 7, inclusive, causes a comma to be generated across the boundaries of the two adjacent code-groups. A comma across the boundaries of any two adjacent code-groups may cause code-group realignment (see 36.3.2.4).

36.2.4.10 Configuration (/C/)

Configuration, defined as the continuous repetition of the ordered sets /C1/ and /C2/, is used to convey the 16-bit Configuration Register (Config_Reg) to the link partner. See Clause 37 for a description of the Config_Reg contents.

The ordered_sets, /C1/ and /C2/, are defined in Table 36–3. The /C1/ ordered_set is defined such that the running disparity at the end of the first two code-groups is opposite that of the beginning running disparity. The /C2/ ordered_set is defined such that the running disparity at the end of the first two code-groups is the same as the beginning running disparity. For a constant Config_Reg value, the running disparity after transmitting the sequence /C1/C2/ will be the opposite of what it was at the start of the sequence. This ensures that K28.5s containing comma+ will be sent during configuration.

36.2.4.11 Data (/D/)

A data code-group, when not used to distinguish or convey information for a defined ordered_set, conveys one octet of arbitrary data between the GMII and the PCS. The sequence of data code-groups is arbitrary, where any data code-group can be followed by any other data code-group. Data code-groups are coded and decoded but not interpreted by the PCS. Successful decoding of the data code-groups depends on proper receipt of the Start_of_Packet delimiter, as defined in 36.2.4.13 and the checking of validity, as defined in 36.2.4.6.

36.2.4.12 IDLE (/I/)

IDLE ordered_sets (/I/) are transmitted continuously and repetitively whenever the GMII is idle (TX_EN and TX_ER are both inactive). /I/ provides a continuous fill pattern to establish and maintain clock synchronization. /I/ is emitted from, and interpreted by, the PCS. /I/ consists of one or more consecutively transmitted /I1/ or /I2/ ordered_sets, as defined in Table 36–3.

The /I1/ ordered_set is defined such that the running disparity at the end of the transmitted /I1/ is opposite that of the beginning running disparity. The /I2/ ordered_set is defined such that the running disparity at the end of the transmitted /I2/ is the same as the beginning running disparity. The first /I/ following a packet or Configuration ordered_set restores the current positive or negative running disparity to a negative value. All subsequent /I/s are /I2/ to ensure negative ending running disparity.

Distinct carrier events are separated by /I/s.

Implementations of this standard may benefit from the ability to add or remove /I2/ from the code-group stream one /I2/ at a time without altering the beginning running disparity associated with the code-group subsequent to the removed /I2/.

A received ordered set which consists of two code-groups, the first of which is /K28.5/ and the second of which is a data code-group other than /D21.5/ or /D2.2/ is treated as an /I/ ordered_set.

36.2.4.13 Start_of_Packet (SPD) delimiter

A Start_of_Packet delimiter (SPD) is used to delineate the starting boundary of a data transmission sequence and to authenticate carrier events. Upon each fresh assertion of TX_EN by the GMII, and subsequent to the completion of PCS transmission of the current ordered_set, the PCS replaces the current octet of the MAC preamble with SPD. Upon initiation of packet reception, the PCS replaces the received SPD delimiter with the data octet value associated with the first preamble octet. A SPD delimiter consists of the code-group /S/, as defined in Table 36–3.

SPD follows /I/ for a single packet or the first packet in a burst.

SPD follows /R/ for the second and subsequent packets of a burst.

36.2.4.14 End_of_Packet delimiter (EPD)

An End_of_Packet delimiter (EPD) is used to delineate the ending boundary of a packet. The EPD is transmitted by the PCS following each de-assertion of TX_EN on the GMII, which follows the last data octet comprising the FCS of the MAC packet. On reception, EPD is interpreted by the PCS as terminating a packet. A EPD delimiter consists of the code-groups /T/R/R/ or /T/R/K28.5/. The code-group /T/ is defined in Table 36–3. See 36.2.4.15 for the definition of code-groups used for /R/. /K28.5/ normally occurs as the first code-group of the /I/ ordered_set. See 36.2.4.12 for the definition of code-groups used for /I/.

The receiver considers the MAC interpacket gap (IPG) to have begun two octets prior to the transmission of /I/. For example, when a packet is terminated by EPD, the /T/R/ portion of the EPD occupies part of the region considered by the MAC to be the IPG.

36.2.4.14.1 EPD rules

a) The PCS transmits a /T/R/ following the last data octet from the MAC;
b) If the MAC indicates carrier extension to the PCS, Carrier_Extend rules are in effect. See 36.2.4.15.1;
c) If the MAC does not indicate carrier extension to the PCS, perform the following:
    1) If /R/ is transmitted in an even-numbered code-group position, the PCS appends a single additional /R/ to the code-group stream to ensure that the subsequent /I/ is aligned on an even-numbered code-group boundary and EPD transmission is complete;
    2) The PCS transmits /I/.

36.2.4.15 Carrier_Extend (/R/)

Carrier_Extend (/R/) is used for the following purposes:

a) Carrier extension: Used by the MAC to extend the duration of the carrier event. When used for this purpose, carrier extension is emitted from and interpreted by the MAC and coded to and decoded from the corresponding code-group by the PCS. In order to extend carrier, the GMII must deassert TX_EN. The deassertion of TX_EN and simultaneous assertion of TX_ER causes the PCS to emit an /R/ with a two-octet delay, which gives the PCS time to complete its EPD before commencing transmissions. The number of /R/ code-groups emitted from the PCS equals the number of GMII GTX_CLK periods during which it extends carrier;
b) Packet separation: Carrier extension is used by the MAC to separate packets within a burst of packets. When used for this purpose, carrier extension is emitted from and interpreted by the MAC and coded to and decoded from the corresponding code-group by the PCS;
c) EPD2: The first /R/ following the /T/ in the End_of_Packet delimiters /T/R/I/ or /T/R/R/I/;
d) EPD3: The second /R/ following the /T/ in the End_of_Packet delimiter /T/R/R/I/. This /R/ is used, if necessary, to pad the only or last packet of a burst of packets so that the subsequent /I/ is aligned on an even-numbered code-group boundary. When used for this purpose, Carrier_Extend is emitted from, and interpreted by, the PCS. An EPD of /T/R/R/ results in one /R/ being delivered to the PCS client (see 36.2.4.14.1).

Carrier_Extend consists of one or more consecutively transmitted /R/ ordered_sets, as defined in Table 36–3.

36.2.4.15.1 Carrier_Extend rules

a) If the MAC indicates carrier extension to the PCS, the initial /T/R/ is followed by one /R/ for each octet of carrier extension received from the MAC;
b) If the last /R/ is transmitted in an even-numbered code-group position, the PCS appends a single additional /R/ to the code-group stream to ensure that the subsequent /I/ is aligned on an even-numbered code-group boundary.

36.2.4.16 Error_Propagation (/V/)

Error_Propagation (/V/) indicates that the PCS client wishes to indicate a transmission error to its peer entity. The normal use of Error_Propagation is for repeaters to propagate received errors. /V/ is emitted from the PCS, at the request of the PCS client through the use of the TX_ER signal, as described in Clause 35.

Error_Propagation is emitted from, and interpreted by, the PCS. Error_Propagation consists of the ordered_set /V/, as defined in Table 36–3.

The presence of Error_Propagation or any invalid code-group on the medium denotes a collision artifact or an error condition. Invalid code-groups are not intentionally transmitted onto the medium by DTEs. The PCS processes and conditionally indicates the reception of /V/ or an invalid code-group on the GMII as false carrier, data errors, or carrier extend errors, depending on its current context.

36.2.4.17 Encapsulation

The 1000BASE-X PCS accepts packets from the MAC through the Reconciliation sublayer and GMII. Due to the continuously signaled nature of the underlying PMA, and the encoding performed by the PCS, the 1000BASE-X PCS encapsulates MAC frames into a code-group stream. The PCS decodes the code-group stream received from the PMA, extracts packets from it, and passes the packets to the MAC via the Reconciliation sublayer and GMII.

Figure 36–4 depicts the PCS encapsulation of a MAC packet based on GMII signals.

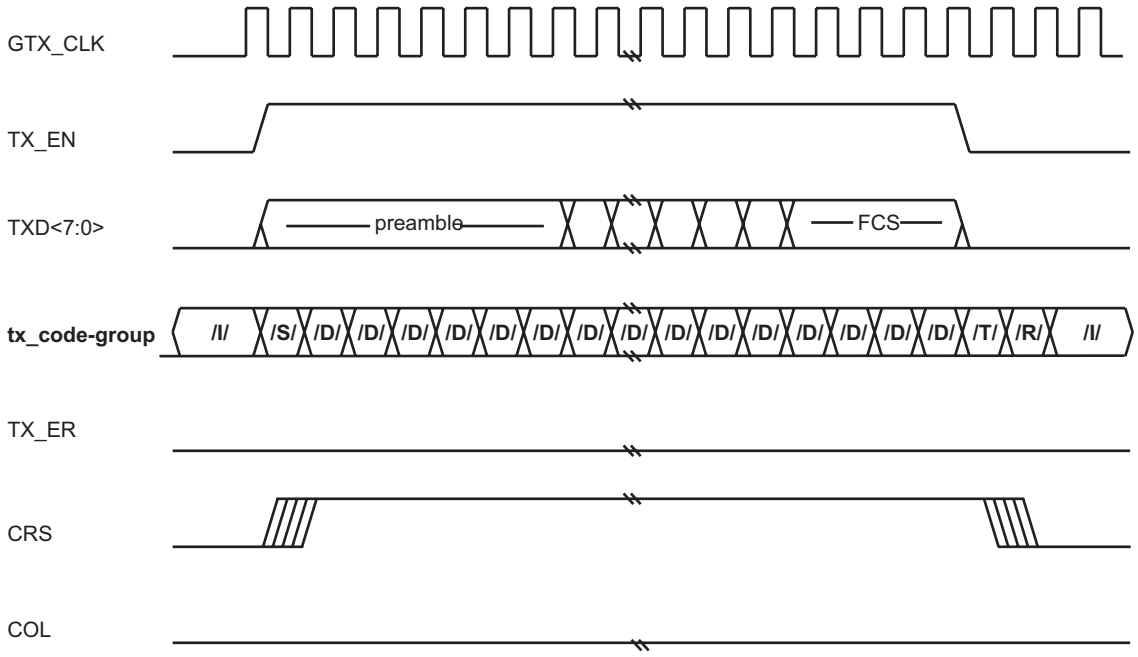

Figure 36–4—PCS encapsulation

36.2.4.18 Mapping between GMII, PCS and PMA

Figure 36–3 depicts the mapping of the octet-wide data path of the GMII to the ten-bit-wide code-groups of the PCS, and the one-bit paths of the PMA/PMD interface.

The PCS encodes an octet received from the GMII into a ten-bit code-group, according to Figure 36–3. Code-groups are serialized into a tx_bit stream by the PMA and passed to the PMD for transmission on the underlying medium, according to Figure 36–3. The first transmitted tx_bit is tx_code-group<0>, and the last tx_bit transmitted is tx_code-group<9>. There is no numerical significance ascribed to the bits within a code-group; that is, the code-group is simply a ten-bit pattern that has some predefined interpretation.

Appendix 1.c. – Excerpts for 4D-PAM5 encoding

40.2.9 PMA_REMRXSTATUS.request

This primitive is generated by PCS Receive to indicate the status of the receive link at the remote PHY as communicated by the remote PHY via its encoding of its loc_rcvr_status parameter. The parameter rem_rcvr_status conveys to the PMA PHY Control function the information on whether reliable operation of the remote PHY is detected or not. The criterion for setting the parameter rem_rcvr_status is left to the implementor. It can be based, for example, on asserting rem_rcvr_status is NOT_OK until loc_rcvr_status is OK and then asserting the detected value of rem_rcvr_status after proper PCS receive decoding is achieved.

40.2.9.1 Semantics of the primitive

PMA_REMRXSTATUS.request (rem_rcvr_status)

The rem_rcvr_status parameter can take on one of two values of the form:

OK	The receive link for the remote PHY is operating reliably.
NOT_OK	Reliable operation of the receive link for the remote PHY is not detected.

40.2.9.2 When generated

The PCS generates PMA_REMRXSTATUS.request messages continuously on the basis on signals received at the MDI.

40.2.9.3 Effect of receipt

The effect of receipt of this primitive is specified in Figure 40–15.

40.2.10 PMA_RESET.indication

This primitive is used to pass the PMA Reset function to the PCS (pcs_reset=ON) when reset is enabled.

The PMA_RESET.indication primitive can take on one of two values:

TRUE	Reset is enabled.
FALSE	Reset is not enabled.

40.2.10.1 When generated

The PMA Reset function is executed as described in 40.4.2.1.

40.2.10.2 Effect of receipt

The effect of receipt of this primitive is specified in 40.4.2.1.

40.3 Physical Coding Sublayer (PCS)

The PCS comprises one PCS Reset function and four simultaneous and asynchronous operating functions. The PCS operating functions are: PCS Transmit Enable, PCS Transmit, PCS Receive, and PCS Carrier Sense. All operating functions start immediately after the successful completion of the PCS Reset function.

The PCS reference diagram, Figure 40–5, shows how the four operating functions relate to the messages of the PCS-PMA interface. Connections from the management interface (signals MDC and MDIO) to other

layers are pervasive, and are not shown in Figure 40–5. Management is specified in Clause 30. See also Figure 40–7, which defines the structure of frames passed from PCS to PMA.

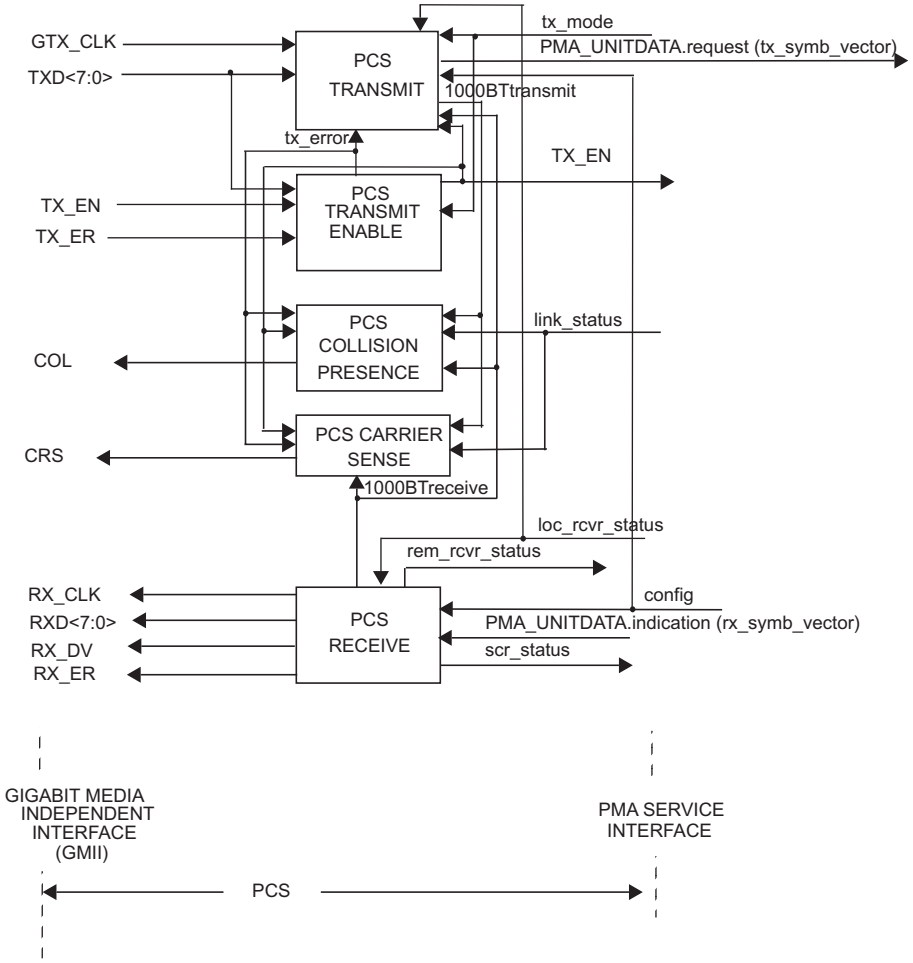

Figure 40–5—PCS reference diagram

40.3.1 PCS functions

40.3.1.1 PCS Reset function

PCS Reset initializes all PCS functions. The PCS Reset function shall be executed whenever one of the following conditions occur:

a) Power on (see 36.2.5.1.3).
b) The receipt of a request for reset from the management entity.

PCS Reset sets pcs_reset=ON while any of the above reset conditions hold true. All state diagrams take the open-ended pcs_reset branch upon execution of PCS Reset. The reference diagrams do not explicitly show the PCS Reset function.

40.3.1.2 PCS Data Transmission Enable

The PCS Data Transmission Enabling process generates the signals tx_enable and tx_error, which PCS Transmit uses for data and carrier extension encoding. The process uses logical operations on tx_mode, TX_ER, TX_EN, and TXD<7:0>. The PCS shall implement the Data Transmission Enabling process as depicted in Figure 40–8 including compliance with the associated state variables as specified in 40.3.3.

40.3.1.3 PCS Transmit function

The PCS Transmit function shall conform to the PCS Transmit state diagram in Figure 40–9.

The PCS Transmit function generates the GMII signal COL based on whether a reception is occurring simultaneously with transmission. The PCS Transmit function is not required to generate the GMII signal COL in a 1000BASE-T PHY that does not support half duplex operation.

In each symbol period, PCS Transmit generates a code-group (A_n, B_n, C_n, D_n) that is transferred to the PMA via the PMA_UNITDATA.request primitive. The PMA transmits symbols A_n, B_n, C_n, D_n over wire-pairs BI_DA, BI_DB, BI_DC, and BI_DD respectively. The integer, n, is a time index that is introduced to establish a temporal relationship between different symbol periods. A symbol period, T, is nominally equal to 8 ns. In normal mode of operation, between streams of data indicated by the parameter tx_enable, PCS Transmit generates sequences of vectors using the encoding rules defined for the idle mode. Upon assertion of tx_enable, PCS Transmit passes a SSD of two consecutive vectors of four quinary symbols to the PMA, replacing the first two preamble octets. Following the SSD, each TXD<7:0> octet is encoded using an 4D-PAM5 technique into a vector of four quinary symbols until tx_enable is de-asserted. If TX_ER is asserted while tx_enable is also asserted, then PCS Transmit passes to the PMA vectors indicating a transmit error. Note that if the signal TX_ER is asserted while SSD is being sent, the transmission of the error condition is delayed until transmission of SSD has been completed. Following the de-assertion of tx_enable, a Convolutional State Reset (CSReset) of two consecutive code-groups, followed by an ESD of two consecutive code-groups, is generated, after which the transmission of idle or control mode is resumed.

If a PMA_TXMODE.indication message has the value SEND_Z, PCS Transmit passes a vector of zeros at each symbol period to the PMA via the PMA_UNITDATA.request primitive.

If a PMA_TXMODE.indication message has the value SEND_I, PCS Transmit generates sequences of code-groups according to the encoding rule in training mode. Special code-groups that use only the values {+2, 0, –2} are transmitted in this case. Training mode encoding also takes into account the value of the parameter loc_rcvr_status. By this mechanism, a PHY indicates the status of its own receiver to the link partner during idle transmission.

In the normal mode of operation, the PMA_TXMODE.indication message has the value SEND_N, and the PCS Transmit function uses an 8B1Q4 coding technique to generate at each symbol period code-groups that represent data, control or idle based on the code-groups defined in Table 40–1 and Table 40–2. During transmission of data, the TXD<7:0> bits are scrambled by the PCS using a side-stream scrambler, then encoded into a code-group of quinary symbols and transferred to the PMA. During data encoding, PCS Transmit utilizes a three-state convolutional encoder.

The transition from idle or carrier extension to data is signalled by inserting a SSD, and the end of transmission of data is signalled by an ESD. Further code-groups are reserved for signaling the assertion of TX_ER within a stream of data, carrier extension, CSReset, and other control functions. During idle and carrier extension encoding, special code-groups with symbol values restricted to the set {2, 0, –2} are used. These code-groups are also generated using the transmit side-stream scrambler. However, the encoding rules for the idle, SSD, and carrier extend code-groups are different from the encoding rules for data, CSReset, CSExtend, and ESD code-groups. During idle, SSD, and carrier extension, the PCS Transmit function reverses the sign of the transmitted symbols. This allows, at the receiver, sequences of code-groups that represent data,

CSReset, CSExtend, and ESD to be easily distinguished from sequences of code-groups that represent SSD, carrier extension, and idle.

PCS encoding involves the generation of the four-bit words $Sx_n[3:0]$, $Sy_n[3:0]$, and $Sg_n[3:0]$ from which the quinary symbols (A_n, B_n, C_n, D_n) are obtained. The four-bit words $Sx_n[3:0]$, $Sy_n[3:0]$, and $Sg_n[3:0]$ are determined (as explained in 40.3.1.3.2) from sequences of pseudorandom binary symbols derived from the transmit side-stream scrambler.

40.3.1.3.1 Side-stream scrambler polynomials

The PCS Transmit function employs side-stream scrambling. If the parameter config provided to the PCS by the PMA PHY Control function via the PMA_CONFIG.indication message assumes the value MASTER, PCS Transmit shall employ

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

as transmitter side-stream scrambler generator polynomial. If the PMA_CONFIG.indication message assumes the value of SLAVE, PCS Transmit shall employ

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

as transmitter side-stream scrambler generator polynomial. An implementation of master and slave PHY side-stream scramblers by linear-feedback shift registers is shown in Figure 40–6. The bits stored in the shift register delay line at time n are denoted by $Scr_n[32:0]$. At each symbol period, the shift register is advanced by one bit, and one new bit represented by $Scr_n[0]$ is generated. The transmitter side-stream scrambler is reset upon execution of the PCS Reset function. If PCS Reset is executed, all bits of the 33-bit vector representing the side-stream scrambler state are arbitrarily set. The initialization of the scrambler state is left to the implementor. In no case shall the scrambler state be initialized to all zeros.

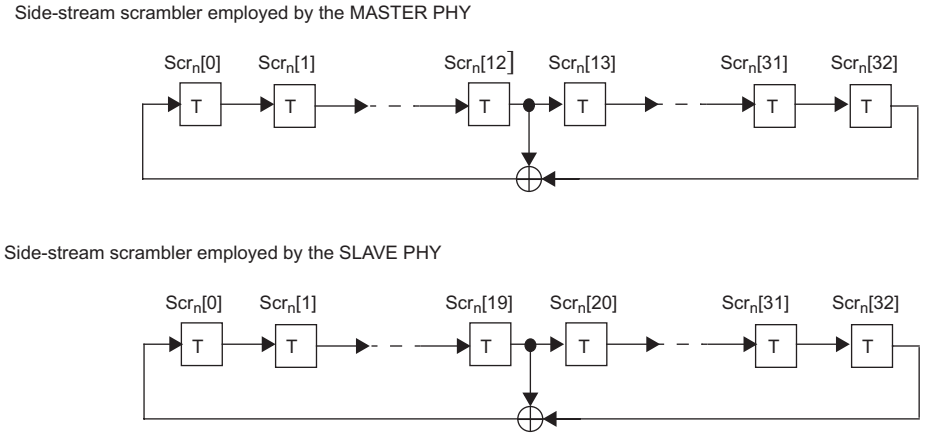

Figure 40–6—A realization of side-stream scramblers by linear feedback shift registers

40.3.1.3.2 Generation of bits $Sx_n[3:0]$, $Sy_n[3:0]$, and $Sg_n[3:0]$

PCS Transmit encoding rules are based on the generation, at time n, of the twelve bits $Sx_n[3:0]$, $Sy_n[3:0]$, and $Sg_n[3:0]$. The eight bits, $Sx_n[3:0]$ and $Sy_n[3:0]$, are used to generate the scrambler octet $Sc_n[7:0]$ for decorrelating the GMII data word TXD<7:0> during data transmission and for generating the idle and training symbols. The four bits, $Sg_n[3:0]$, are used to randomize the signs of the quinary symbols (A_n, B_n, C_n,

D_n) so that each symbol stream has no dc bias. These twelve bits are generated in a systematic fashion using three bits, X_n, Y_n, and $Scr_n[0]$, and an auxiliary generating polynomial, g(x). The two bits, X_n and Y_n, are mutually uncorrelated and also uncorrelated with the bit $Scr_n[0]$. For both master and slave PHYs, they are obtained by the same linear combinations of bits stored in the transmit scrambler shift register delay line. These two bits are derived from elements of the same maximum-length shift register sequence of length $2^{33} - 1$ as $Scr_n[0]$, but shifted in time. The associated delays are all large and different so that there is no short-term correlation among the bits $Scr_n[0]$, X_n, and Y_n. The bits X_n and Y_n are generated as follows:

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

$$Y_n = Scr_n[1] \wedge Scr_n[5]$$

where ^ denotes the XOR logic operator. From the three bits X_n, Y_n, and $Scr_n[0]$, further mutually uncorrelated bit streams are obtained systematically using the generating polynomial

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

The four bits $Sy_n[3:0]$ are generated using the bit $Scr_n[0]$ and g(x) as in the following equations:

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

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

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

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

The four bits $Sx_n[3:0]$ are generated using the bit X_n and g(x) as in the following equations:

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

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

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

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

The four bits $Sg_n[3:0]$ are generated using the bit Y_n and g(x) as in the following equations:

$$Sg_n[0] = Y_n = Scr_n[1] \wedge Scr_n[5]$$

$$Sg_n[1] = g(Y_n) = Scr_n[4] \wedge Scr_n[8] \wedge Scr_n[9] \wedge Scr_n[13]$$

$$Sg_n[2] = g^2(Y_n) = Scr_n[7] \wedge Scr_n[11] \wedge Scr_n[17] \wedge Scr_n[21]$$

$$Sg_n[3] = g^3(Y_n) = Scr_n[10] \wedge Scr_n[14] \wedge Scr_n[15] \wedge Scr_n[19] \wedge Scr_n[20] \wedge Scr_n[24] \wedge Scr_n[25] \wedge Scr_n[29]$$

By construction, the twelve bits $Sx_n[3:0]$, $Sy_n[3:0]$, and $Sg_n[3:0]$ are derived from elements of the same maximum-length shift register sequence of length $2^{33}-1$ as $Scr_n[0]$, but shifted in time by varying delays. The associated delays are all large and different so that there is no apparent correlation among the bits.

40.3.1.3.3 Generation of bits $Sc_n[7:0]$

The bits $Sc_n[7:0]$ are used to scramble the GMII data octet TXD[7:0] and for control, idle, and training mode quartet generation. The definition of these bits is dependent upon the bits $Sx_n[3:0]$ and $Sy_n[3:0]$ that are specified in 40.3.1.3.2, the variable tx_mode that is obtained through the PMA Service Interface, the variable tx_enable_n that is defined in Figure 40–8, and the time index n.

The four bits $Sc_n[7:4]$ are defined as

$$Sc_n[7:4] = \begin{cases} Sx_n[3:0] & \text{if } (tx\_enable_{n-2} = 1) \\ [0\ 0\ 0\ 0] & \text{else} \end{cases}$$

The bits $Sc_n[3:1]$ are defined as

$$Sc_n[3:1] = \begin{cases} [0\ 0\ 0] & \text{if } (tx\_mode = SEND\_Z) \\ Sy_n[3:1] & \text{else if } (n\text{-}n_0) = 0 \pmod 2 \\ (Sy_{n-1}[3:1] \wedge [1\ 1\ 1]) & \text{else} \end{cases}$$

where n_0 denotes the time index of the last transmitter side-stream scrambler reset.

The bit $Sc_n[0]$ is defined as

$$Sc_n[0] = \begin{cases} 0 & \text{if } (tx\_mode = SEND\_Z) \\ Sy_n[0] & \text{else} \end{cases}$$

40.3.1.3.4 Generation of bits $Sd_n[8:0]$

The PCS Transmit function generates a nine-bit word $Sd_n[8:0]$ from Sc_n that represents either a convolutionally encoded stream of data, control, or idle mode code-groups. The convolutional encoder uses a three-bit word $cs_n[2:0]$, which is defined as

$$cs_n[1] = \begin{cases} Sd_n[6] \wedge cs_{n-1}[0] & \text{if } (tx\_enable_{n-2} = 1) \\ 0 & \text{else} \end{cases}$$

$$cs_n[2] = \begin{cases} Sd_n[7] \wedge cs_{n-1}[1] & \text{if } (tx\_enable_{n-2} = 1) \\ 0 & \text{else} \end{cases}$$

$$cs_n[0] = cs_{n-1}[2]$$

from which $Sd_n[8]$ is obtained as

$$Sd_n[8] = cs_n[0]$$

The convolutional encoder bits are non-zero only during the transmission of data. Upon the completion of a data frame, the convolutional encoder bits are reset using the bit $csreset_n$. The bit $csreset_n$ is defined as

$$csreset_n = (tx\_enable_{n-2}) \text{ and } (\text{not } tx\_enable_n)$$

The bits $Sd_n[7:6]$ are derived from the bits $Sc_n[7:6]$, the GMII data bits $TXD_n[7:6]$, and from the convolutional encoder bits as

$$Sd_n[7] = \begin{cases} Sc_n[7] \ \hat{} \ TXD_n[7] & \text{if } (csreset_n = 0 \text{ and } tx\_enable_{n-2} = 1) \\ cs_{n-1}[1] & \text{else if } (csreset_n = 1) \\ Sc_n[7] & \text{else} \end{cases}$$

$$Sd_n[6] = \begin{cases} Sc_n[6] \ \hat{} \ TXD_n[6] & \text{if } (csreset_n = 0 \text{ and } tx\_enable_{n-2} = 1) \\ cs_{n-1}[0] & \text{else if } (csreset_n = 1) \\ Sc_n[6] & \text{else} \end{cases}$$

The bits $Sd_n[5:3]$ are derived from the bits $Sc_n[5:3]$ and the GMII data bits $TXD_n[5:3]$ as

$$Sd_n[5:3] = \begin{cases} Sc_n[5:3] \ \hat{} \ TXD_n[5:3] & \text{if } (tx\_enable_{n-2} = 1) \\ Sc_n[5:3] & \text{else} \end{cases}$$

The bit $Sd_n[2]$ is used to scramble the GMII data bit $TXD_n[2]$ during data mode and to encode loc_rcvr_status otherwise. It is defined as

$$Sd_n[2] = \begin{cases} Sc_n[2] \ \hat{} \ TXD_n[2] & \text{if } (tx\_enable_{n-2} = 1) \\ Sc_n[2] \ \hat{} \ 1 & \text{else if } (loc\_rcvr\_status = \text{OK}) \\ Sc_n[2] & \text{else} \end{cases}$$

The bits $Sd_n[1:0]$ are used to transmit carrier extension information during tx_mode=SEND_N and are thus dependent upon the bits $cext_n$ and $cext\_err_n$. These bits are dependent on the variable tx_error_n, which is defined in Figure 40–8. These bits are defined as

$$cext_n = \begin{cases} tx\_error_n & \text{if } ((tx\_enable_n = 0) \text{ and } (TXD_n[7:0] = 0\text{x}0\text{F})) \\ 0 & \text{else} \end{cases}$$

$$cext\_err_n = \begin{cases} tx\_error_n & \text{if } ((tx\_enable_n = 0) \text{ and } (TXD_n[7:0] \neq 0\text{x}0\text{F})) \\ 0 & \text{else} \end{cases}$$

$$Sd_n[1] = \begin{cases} Sc_n[1] \ \hat{} \ TXD_n[1] & \text{if } (tx\_enable_{n-2} = 1) \\ Sc_n[1] \ \hat{} \ cext\_err_n & \text{else} \end{cases}$$

$$Sd_n[0] = \begin{cases} Sc_n[0] \wedge TXD_n[0] & \text{if } (tx\_enable_{n-2} = 1) \\ Sc_n[0] \wedge cext_n & \text{else} \end{cases}$$

40.3.1.3.5 Generation of quinary symbols TA_n, TB_n, TC_n, TD_n

The nine-bit word $Sd_n[8:0]$ is mapped to a quartet of quinary symbols (TA_n, TB_n, TC_n, TD_n) according to Table 40–1 and Table 40–2 shown as $Sd_n[6:8] + Sd_n[5:0]$.

Encoding of error indication:

If tx_error_n=1 when the condition (tx_enable_n * tx_enable_{n-2}) = 1, error indication is signaled by means of symbol substitution. In this condition, the values of $Sd_n[5:0]$ are ignored during mapping and the symbols corresponding to the row denoted as “xmt_err” in Table 40–1 and Table 40–2 shall be used.

Encoding of Convolutional Encoder Reset:

If tx_error_n=0 when the variable $csreset_n$ = 1, the convolutional encoder reset condition is normal. This condition is indicated by means of symbol substitution, where the values of $Sd_n[5:0]$ are ignored during mapping and the symbols corresponding to the row denoted as “CSReset” in Table 40–1 and Table 40–2 shall be used.

Encoding of Carrier Extension during Convolutional Encoder Reset:

If tx_error_n=1 when the variable $csreset_n$ = 1, the convolutional encoder reset condition indicates carrier extension. In this condition, the values of $Sd_n[5:0]$ are ignored during mapping and the symbols corresponding to the row denoted as “CSExtend” in Table 40–1 and Table 40–2 shall be used when TXD_n = 0x’0F, and the row denoted as “CSExtend_Err” in Table 40–1 and Table 40–2 shall be used when $TXD_n \neq$ 0x’0F. The latter condition denotes carrier extension with error. In case carrier extension with error is indicated during the first octet of CSReset, the error condition shall be encoded during the second octet of CSReset, and during the subsequent two octets of the End-of-Stream delimiter as well. Thus, the error condition is assumed to persist during the symbol substitutions at the End-of-Stream.

Encoding of Start-of-Stream delimiter:

The Start-of-Stream delimiter (SSD) is related to the condition SSD_n, which is defined as (tx_enable_n) * ($!tx\_enable_{n-2}$) = 1, where “*” and “!” denote the logic AND and NOT operators, respectively. For the generation of SSD, the first two octets of the preamble in a data stream are mapped to the symbols corresponding to the rows denoted as SSD1 and SSD2 respectively in Table 40–1. The symbols corresponding to the SSD1 row shall be used when the condition (tx_enable_n) * ($!tx\_enable_{n-1}$) = 1. The symbols corresponding to the SSD2 row shall be used when the condition (tx_enable_{n-1}) * ($!tx\_enable_{n-2}$) = 1.

Encoding of End-of-Stream delimiter:

The definition of an End-of-Stream delimiter (ESD) is related to the condition ESD_n, which is defined as ($!tx\_enable_{n-2}$) * (tn_enable_{n-4}) = 1. This occurs during the third and fourth symbol periods after transmission of the last octet of a data stream.

If carrier extend error is indicated during ESD, the symbols corresponding to the ESD_Ext_Err row shall be used. The two conditions upon which this may occur are

(tx_error_n) * (tx_error_{n-1}) * (tx_error_{n-2}) * ($TXD_n \neq$ 0x0F) = 1, and

$(tx\_error_n) * (tx\_error_{n-1}) * (tx\_error_{n-2}) * (tx\_error_{n-3}) * (TXD_n \neq 0x0F) = 1$.

The symbols corresponding to the ESD1 row in Table 40–1 shall be used when the condition $(!tx\_enable_{n-2}) * (tx\_enable_{n-3}) = 1$, in the absence of carrier extend error indication at time n.

The symbols corresponding to the ESD2_Ext_0 row in Table 40–1 shall be used when the condition $(!tx\_enable_{n-3}) * (tx\_enable_{n-4}) * (!tx\_error_n) * (!tx\_error_{n-1}) = 1$.

The symbols corresponding to the ESD2_Ext_1 row in Table 40–1 shall be used when the condition $(!tx\_enable_{n-3}) * (tx\_enable_{n-4}) * (!tx\_error_n) * (tx\_error_{n-1}) * (tx\_error_{n-2}) * (tx\_error_{n-3}) = 1$.

The symbols corresponding to the ESD2_Ext_2 row in Table 40–1 shall be used when the condition $(!tx\_enable_{n-3}) * (tx\_enable_{n-4}) * (tx\_error_n) * (tx\_error_{n-1}) * (tx\_error_{n-2}) * (tx\_error_{n-3}) * (TXD_n = 0x0F) = 1$, in the absence of carrier extend error indication.

NOTE—The ASCII for Table 40–1 and Table 40–2 is available at http://www.ieee802.org/3/publication/index.html.[1]

Table 40–1—Bit-to-symbol mapping (even subsets)

		$Sd_n[6:8] = [000]$	$Sd_n[6:8] = [010]$	$Sd_n[6:8] = [100]$	$Sd_n[6:8] = [110]$
Condition	**$Sd_n[5:0]$**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**
Normal	000000	0, 0, 0, 0	0, 0,+1,+1	0,+1,+1, 0	0,+1, 0,+1
Normal	000001	–2, 0, 0, 0	–2, 0,+1,+1	–2,+1,+1, 0	–2,+1, 0,+1
Normal	000010	0,–2, 0, 0	0,–2,+1,+1	0,–1,+1, 0	0,–1, 0,+1
Normal	000011	–2,–2, 0, 0	–2,–2,+1,+1	–2,–1,+1, 0	–2,–1, 0,+1
Normal	000100	0, 0,–2, 0	0, 0,–1,+1	0,+1,–1, 0	0,+1,–2,+1
Normal	000101	–2, 0,–2, 0	–2, 0,–1,+1	–2,+1,–1, 0	–2,+1,–2,+1
Normal	000110	0,–2,–2, 0	0,–2,–1,+1	0,–1,–1, 0	0,–1,–2,+1
Normal	000111	–2,–2,–2, 0	–2,–2,–1,+1	–2,–1,–1, 0	–2,–1,–2,+1
Normal	001000	0, 0, 0,–2	0, 0,+1,–1	0,+1,+1,–2	0,+1, 0,–1
Normal	001001	–2, 0, 0,–2	–2, 0,+1,–1	–2,+1,+1,–2	–2,+1, 0,–1
Normal	001010	0,–2, 0,–2	0,–2,+1,–1	0,–1,+1,–2	0,–1, 0,–1
Normal	001011	–2,–2, 0,–2	–2,–2,+1,–1	–2,–1,+1,–2	–2,–1, 0,–1
Normal	001100	0, 0,–2,–2	0, 0,–1,–1	0,+1,–1,–2	0,+1,–2,–1
Normal	001101	–2, 0,–2,–2	–2, 0,–1,–1	–2,+1,–1,–2	–2,+1,–2,–1
Normal	001110	0,–2,–2,–2	0,–2,–1,–1	0,–1,–1,–2	0,–1,–2,–1
Normal	001111	–2,–2,–2,–2	–2,–2,–1,–1	–2,–1,–1,–2	–2,–1,–2,–1
Normal	010000	+1,+1,+1,+1	+1,+1, 0, 0	+1, 0, 0,+1	+1, 0,+1, 0
Normal	010001	–1,+1,+1,+1	–1,+1, 0, 0	–1, 0, 0,+1	–1, 0,+1, 0
Normal	010010	+1,–1,+1,+1	+1,–1, 0, 0	+1,–2, 0,+1	+1,–2,+1, 0

[1]*Copyright release for symbol codes:* Users of this standard may freely reproduce the symbol codes in this subclause so it can be used for its intended purpose. Copies of the symbol codes can be obtained at http://standards.ieee.org/reading/ieee/std/downloads/index.html.

Table 40–1—Bit-to-symbol mapping (even subsets) *(continued)*

		$Sd_n[6:8] = [000]$	$Sd_n[6:8] = [010]$	$Sd_n[6:8] = [100]$	$Sd_n[6:8] = [110]$
Condition	**$Sd_n[5:0]$**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**
Normal	010011	–1,–1,+1,+1	–1,–1, 0, 0	–1,–2, 0,+1	–1,–2,+1, 0
Normal	010100	+1,+1,–1,+1	+1,+1,–2, 0	+1, 0,–2,+1	+1, 0,–1, 0
Normal	010101	–1,+1,–1,+1	–1,+1,–2, 0	–1, 0,–2,+1	–1, 0,–1, 0
Normal	010110	+1,–1,–1,+1	+1,–1,–2, 0	+1,–2,–2,+1	+1,–2,–1, 0
Normal	010111	–1,–1,–1,+1	–1,–1,–2, 0	–1,–2,–2,+1	–1,–2,–1, 0
Normal	011000	+1,+1,+1,–1	+1,+1, 0,–2	+1, 0, 0,–1	+1, 0,+1,–2
Normal	011001	–1,+1,+1,–1	–1,+1, 0,–2	–1, 0, 0,–1	–1, 0,+1,–2
Normal	011010	+1,–1,+1,–1	+1,–1, 0,–2	+1,–2, 0,–1	+1,–2,+1,–2
Normal	011011	–1,–1,+1,–1	–1,–1, 0,–2	–1,–2, 0,–1	–1,–2,+1,–2
Normal	011100	+1,+1,–1,–1	+1,+1,–2,–2	+1, 0,–2,–1	+1, 0,–1,–2
Normal	011101	–1,+1,–1,–1	–1,+1,–2,–2	–1, 0,–2,–1	–1, 0,–1,–2
Normal	011110	+1,–1,–1,–1	+1,–1,–2,–2	+1,–2,–2,–1	+1,–2,–1,–2
Normal	011111	–1,–1,–1,–1	–1,–1,–2,–2	–1,–2,–2,–1	–1,–2,–1,–2
Normal	100000	+2, 0, 0, 0	+2, 0,+1,+1	+2,+1,+1, 0	+2,+1, 0,+1
Normal	100001	+2,–2, 0, 0	+2,–2,+1,+1	+2,–1,+1, 0	+2,–1, 0,+1
Normal	100010	+2, 0,–2, 0	+2, 0,–1,+1	+2,+1,–1, 0	+2,+1,–2,+1
Normal	100011	+2,–2,–2, 0	+2,–2,–1,+1	+2,–1,–1, 0	+2,–1,–2,+1
Normal	100100	+2, 0, 0,–2	+2, 0,+1,–1	+2,+1,+1,–2	+2,+1, 0,–1
Normal	100101	+2,–2, 0,–2	+2,–2,+1,–1	+2,–1,+1,–2	+2,–1, 0,–1
Normal	100110	+2, 0,–2,–2	+2, 0,–1,–1	+2,+1,–1,–2	+2,+1,–2,–1
Normal	100111	+2,–2,–2,–2	+2,–2,–1,–1	+2,–1,–1,–2	+2,–1,–2,–1
Normal	101000	0, 0,+2, 0	+1,+1,+2, 0	+1, 0,+2,+1	0,+1,+2,+1
Normal	101001	–2, 0,+2, 0	–1,+1,+2, 0	–1, 0,+2,+1	–2,+1,+2,+1
Normal	101010	0,–2,+2, 0	+1,–1,+2, 0	+1,–2,+2,+1	0,–1,+2,+1
Normal	101011	–2,–2,+2, 0	–1,–1,+2, 0	–1,–2,+2,+1	–2,–1,+2,+1
Normal	101100	0, 0,+2,–2	+1,+1,+2,–2	+1, 0,+2,–1	0,+1,+2,–1
Normal	101101	–2, 0,+2,–2	–1,+1,+2,–2	–1, 0,+2,–1	–2,+1,+2,–1
Normal	101110	0,–2,+2,–2	+1,–1,+2,–2	+1,–2,+2,–1	0,–1,+2,–1
Normal	101111	–2,–2,+2,–2	–1,–1,+2,–2	–1,–2,+2,–1	–2,–1,+2,–1
Normal	110000	0,+2, 0, 0	0,+2,+1,+1	+1,+2, 0,+1	+1,+2,+1, 0
Normal	110001	–2,+2, 0, 0	–2,+2,+1,+1	–1,+2, 0,+1	–1,+2,+1, 0
Normal	110010	0,+2,–2, 0	0,+2,–1,+1	+1,+2,–2,+1	+1,+2,–1, 0

Table 40–1—Bit-to-symbol mapping (even subsets) *(continued)*

		$Sd_n[6:8] = [000]$	$Sd_n[6:8] = [010]$	$Sd_n[6:8] = [100]$	$Sd_n[6:8] = [110]$
Condition	**$Sd_n[5:0]$**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**
Normal	110011	–2,+2,–2, 0	–2,+2,–1,+1	–1,+2,–2,+1	–1,+2,–1, 0
Normal	110100	0,+2, 0,–2	0,+2,+1,–1	+1,+2, 0,–1	+1,+2,+1,–2
Normal	110101	–2,+2, 0,–2	–2,+2,+1,–1	–1,+2, 0,–1	–1,+2,+1,–2
Normal	110110	0,+2,–2,–2	0,+2,–1,–1	+1,+2,–2,–1	+1,+2,–1,–2
Normal	110111	–2,+2,–2,–2	–2,+2,–1,–1	–1,+2,–2,–1	–1,+2,–1,–2
Normal	111000	0, 0, 0,+2	+1,+1, 0,+2	0,+1,+1,+2	+1, 0,+1,+2
Normal	111001	–2, 0, 0,+2	–1,+1, 0,+2	–2,+1,+1,+2	–1, 0,+1,+2
Normal	111010	0,–2, 0,+2	+1,–1, 0,+2	0,–1,+1,+2	+1,–2,+1,+2
Normal	111011	–2,–2, 0,+2	–1,–1, 0,+2	–2,–1,+1,+2	–1,–2,+1,+2
Normal	111100	0, 0,–2,+2	+1,+1,–2,+2	0,+1,–1,+2	+1, 0,–1,+2
Normal	111101	–2, 0,–2,+2	–1,+1,–2,+2	–2,+1,–1,+2	–1, 0,–1,+2
Normal	111110	0,–2,–2,+2	+1,–1,–2,+2	0,–1,–1,+2	+1,–2,–1,+2
Normal	111111	–2,–2,–2,+2	–1,–1,–2,+2	–2,–1,–1,+2	–1,–2,–1,+2
xmt_err	XXXXXX	0,+2,+2,0	+1,+1,+2,+2	+2,+1,+1,+2	+2,+1,+2,+1
CSExtend_Err	XXXXXX	–2,+2,+2,–2	–1,–1,+2,+2	+2,–1,–1,+2	+2,–1,+2,–1
CSExtend	XXXXXX	+2, 0, 0,+2	+2,+2,+1,+1	+1,+2,+2,+1	+1,+2,+1,+2
CSReset	XXXXXX	+2,–2,–2,+2	+2,+2,–1,–1	–1,+2,+2,–1	–1,+2,–1,+2
SSD1	XXXXXX	+2,+2,+2,+2	—	—	—
SSD2	XXXXXX	+2,+2,+2,–2	—	—	—
ESD1	XXXXXX	+2,+2,+2,+2	—	—	—
ESD2_Ext_0	XXXXXX	+2,+2,+2,–2	—	—	—
ESD2_Ext_1	XXXXXX	+2,+2, –2,+2	—	—	—
ESD2_Ext_2	XXXXXX	+2,–2,+2,+2	—	—	—
ESD_Ext_Err	XXXXXX	–2,+2,+2,+2	—	—	—
Idle/Carrier Extension	000000	0, 0, 0, 0	—	—	—
Idle/Carrier Extension	000001	–2, 0, 0, 0	—	—	—
Idle/Carrier Extension	000010	0,–2, 0, 0	—	—	—
Idle/Carrier Extension	000011	–2,–2, 0, 0	—	—	—
Idle/Carrier Extension	000100	0, 0,–2, 0	—	—	—

Table 40–1—Bit-to-symbol mapping (even subsets) *(continued)*

		$Sd_n[6:8]$ = [000]	$Sd_n[6:8]$ = [010]	$Sd_n[6:8]$ = [100]	$Sd_n[6:8]$ = [110]
Condition	**$Sd_n[5:0]$**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**
Idle/Carrier Extension	000101	–2, 0,–2, 0	—	—	—
Idle/Carrier Extension	000110	0,–2,–2, 0	—	—	—
Idle/Carrier Extension	000111	–2,–2,–2, 0	—	—	—
Idle/Carrier Extension	001000	0, 0, 0,–2	—	—	—
Idle/Carrier Extension	001001	–2, 0, 0,–2	—	—	—
Idle/Carrier Extension	001010	0,–2, 0,–2	—	—	—
Idle/Carrier Extension	001011	–2,–2, 0,–2	—	—	—
Idle/Carrier Extension	001100	0, 0,–2,–2	—	—	—
Idle/Carrier Extension	001101	–2, 0,–2,–2	—	—	—
Idle/Carrier Extension	001110	0,–2,–2,–2	—	—	—
Idle/Carrier Extension	001111	–2,–2,–2,–2	—	—	—

Table 40–2—Bit-to-symbol mapping (odd subsets)

		$Sd_n[6:8]$ = [001]	$Sd_n[6:8]$ = [011]	$Sd_n[6:8]$ = [101]	$Sd_n[6:8]$ = [111]
Condition	**$Sd_n[5:0]$**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**
Normal	000000	0, 0, 0,+1	0, 0,+1, 0	0,+1,+1,+1	0,+1, 0, 0
Normal	000001	–2, 0, 0,+1	–2, 0,+1, 0	–2,+1,+1,+1	–2,+1, 0, 0
Normal	000010	0,–2, 0,+1	0,–2,+1, 0	0,–1,+1,+1	0,–1, 0, 0
Normal	000011	–2,–2, 0,+1	–2,–2,+1, 0	–2,–1,+1,+1	–2,–1, 0, 0
Normal	000100	0, 0,–2,+1	0, 0,–1, 0	0,+1,–1,+1	0,+1,–2, 0
Normal	000101	–2, 0,–2,+1	–2, 0,–1, 0	–2,+1,–1,+1	–2,+1,–2, 0
Normal	000110	0,–2,–2,+1	0,–2,–1, 0	0,–1,–1,+1	0,–1,–2, 0
Normal	000111	–2,–2,–2,+1	–2,–2,–1, 0	–2,–1,–1,+1	–2,–1,–2, 0
Normal	001000	0, 0, 0,–1	0, 0,+1,–2	0,+1,+1,–1	0,+1, 0,–2
Normal	001001	–2, 0, 0,–1	–2, 0,+1,–2	–2,+1,+1,–1	–2,+1, 0,–2

Table 40–2—Bit-to-symbol mapping (odd subsets) *(continued)*

		$Sd_n[6:8] = [001]$	$Sd_n[6:8] = [011]$	$Sd_n[6:8] = [101]$	$Sd_n[6:8] = [111]$
Condition	**$Sd_n[5:0]$**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**
Normal	001010	0,–2, 0,–1	0,–2,+1,–2	0,–1,+1,–1	0,–1, 0,–2
Normal	001011	–2,–2, 0,–1	–2,–2,+1,–2	–2,–1,+1,–1	–2,–1, 0,–2
Normal	001100	0, 0,–2,–1	0, 0,–1,–2	0,+1,–1,–1	0,+1,–2,–2
Normal	001101	–2, 0,–2,–1	–2, 0,–1,–2	–2,+1,–1,–1	–2,+1,–2,–2
Normal	001110	0,–2,–2,–1	0,–2,–1,–2	0,–1,–1,–1	0,–1,–2,–2
Normal	001111	–2,–2,–2,–1	–2,–2,–1,–2	–2,–1,–1,–1	–2,–1,–2,–2
Normal	010000	+1,+1,+1, 0	+1,+1, 0,+1	+1, 0, 0, 0	+1, 0,+1,+1
Normal	010001	–1,+1,+1, 0	–1,+1, 0,+1	–1, 0, 0, 0	–1, 0,+1,+1
Normal	010010	+1,–1,+1, 0	+1,–1, 0,+1	+1,–2, 0, 0	+1,–2,+1,+1
Normal	010011	–1,–1,+1, 0	–1,–1, 0,+1	–1,–2, 0, 0	–1,–2,+1,+1
Normal	010100	+1,+1,–1, 0	+1,+1,–2,+1	+1, 0,–2, 0	+1, 0,–1,+1
Normal	010101	–1,+1,–1, 0	–1,+1,–2,+1	–1, 0,–2, 0	–1, 0,–1,+1
Normal	010110	+1,–1,–1, 0	+1,–1,–2,+1	+1,–2,–2, 0	+1,–2,–1,+1
Normal	010111	–1,–1,–1, 0	–1,–1,–2,+1	–1,–2,–2, 0	–1,–2,–1,+1
Normal	011000	+1,+1,+1,–2	+1,+1, 0,–1	+1, 0, 0,–2	+1, 0,+1,–1
Normal	011001	–1,+1,+1,–2	–1,+1, 0,–1	–1, 0, 0,–2	–1, 0,+1,–1
Normal	011010	+1,–1,+1,–2	+1,–1, 0,–1	+1,–2, 0,–2	+1,–2,+1,–1
Normal	011011	–1,–1,+1,–2	–1,–1, 0,–1	–1,–2, 0,–2	–1,–2,+1,–1
Normal	011100	+1,+1,–1,–2	+1,+1,–2,–1	+1, 0,–2,–2	+1, 0,–1,–1
Normal	011101	–1,+1,–1,–2	–1,+1,–2,–1	–1, 0,–2,–2	–1, 0,–1,–1
Normal	011110	+1,–1,–1,–2	+1,–1,–2,–1	+1,–2,–2,–2	+1,–2,–1,–1
Normal	011111	–1,–1,–1,–2	–1,–1,–2,–1	–1,–2,–2,–2	–1,–2,–1,–1
Normal	100000	+2, 0, 0,+1	+2, 0,+1, 0	+2,+1,+1,+1	+2,+1, 0, 0
Normal	100001	+2,–2, 0,+1	+2,–2,+1, 0	+2,–1,+1,+1	+2,–1, 0, 0
Normal	100010	+2, 0,–2,+1	+2, 0,–1, 0	+2,+1,–1,+1	+2,+1,–2, 0
Normal	100011	+2,–2,–2,+1	+2,–2,–1, 0	+2,–1,–1,+1	+2,–1,–2, 0
Normal	100100	+2, 0, 0,–1	+2, 0,+1,–2	+2,+1,+1,–1	+2,+1, 0,–2
Normal	100101	+2,–2, 0,–1	+2,–2,+1,–2	+2,–1,+1,–1	+2,–1, 0,–2
Normal	100110	+2, 0,–2,–1	+2, 0,–1,–2	+2,+1,–1,–1	+2,+1,–2,–2
Normal	100111	+2,–2,–2,–1	+2,–2,–1,–2	+2,–1,–1,–1	+2,–1,–2,–2
Normal	101000	0, 0,+2,+1	+1,+1,+2,+1	+1, 0,+2, 0	0,+1,+2, 0

Table 40–2—Bit-to-symbol mapping (odd subsets) *(continued)*

		$Sd_n[6:8]$ = [001]	$Sd_n[6:8]$ = [011]	$Sd_n[6:8]$ = [101]	$Sd_n[6:8]$ = [111]
Condition	**$Sd_n[5:0]$**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**	**TA_n,TB_n,TC_n,TD_n**
Normal	101001	–2, 0,+2,+1	–1,+1,+2,+1	–1, 0,+2, 0	–2,+1,+2, 0
Normal	101010	0,–2,+2,+1	+1,–1,+2,+1	+1,–2,+2, 0	0,–1,+2, 0
Normal	101011	–2,–2,+2,+1	–1,–1,+2,+1	–1,–2,+2, 0	–2,–1,+2, 0
Normal	101100	0, 0,+2,–1	+1,+1,+2,–1	+1, 0,+2,–2	0,+1,+2,–2
Normal	101101	–2, 0,+2,–1	–1,+1,+2,–1	–1, 0,+2,–2	–2,+1,+2,–2
Normal	101110	0,–2,+2,–1	+1,–1,+2,–1	+1,–2,+2,–2	0,–1,+2,–2
Normal	101111	–2,–2,+2,–1	–1,–1,+2,–1	–1,–2,+2,–2	–2,–1,+2,–2
Normal	110000	0,+2, 0,+1	0,+2,+1, 0	+1,+2, 0, 0	+1,+2,+1,+1
Normal	110001	–2,+2, 0,+1	–2,+2,+1, 0	–1,+2, 0, 0	–1,+2,+1,+1
Normal	110010	0,+2,–2,+1	0,+2,–1, 0	+1,+2,–2, 0	+1,+2,–1,+1
Normal	110011	–2,+2,–2,+1	–2,+2,–1, 0	–1,+2,–2, 0	–1,+2,–1,+1
Normal	110100	0,+2, 0,–1	0,+2,+1,–2	+1,+2, 0,–2	+1,+2,+1,–1
Normal	110101	–2,+2, 0,–1	–2,+2,+1,–2	–1,+2, 0,–2	–1,+2,+1,–1
Normal	110110	0,+2,–2,–1	0,+2,–1,–2	+1,+2,–2,–2	+1,+2,–1,–1
Normal	110111	–2,+2,–2,–1	–2,+2,–1,–2	–1,+2,–2,–2	–1,+2,–1,–1
Normal	111000	+1,+1,+1,+2	0, 0,+1,+2	+1, 0, 0,+2	0,+1, 0,+2
Normal	111001	–1,+1,+1,+2	–2, 0,+1,+2	–1, 0, 0,+2	–2,+1, 0,+2
Normal	111010	+1,–1,+1,+2	0,–2,+1,+2	+1,–2, 0,+2	0,–1, 0,+2
Normal	111011	–1,–1,+1,+2	–2,–2,+1,+2	–1,–2, 0,+2	–2,–1, 0,+2
Normal	111100	+1,+1,–1,+2	0, 0,–1,+2	+1, 0,–2,+2	0,+1,–2,+2
Normal	111101	–1,+1,–1,+2	–2, 0,–1,+2	–1, 0,–2,+2	–2,+1,–2,+2
Normal	111110	+1,–1,–1,+2	0,–2,–1,+2	+1,–2,–2,+2	0,–1,–2,+2
Normal	111111	–1,–1,–1,+2	–2,–2,–1,+2	–1,–2,–2,+2	–2,–1,–2,+2
xmt_err	XXXXXX	+2,+2, 0,+1	0,+2,+1,+2	+1,+2,+2, 0	+2,+1,+2, 0
CSExtend_Err	XXXXXX	+2,+2, –2,–1	–2,+2,–1,+2	–1,+2,+2,–2	+2,–1,+2,–2
CSExtend	XXXXXX	+2, 0,+2,+1	+2, 0,+1,+2	+1, 0,+2,+2	+2,+1, 0,+2
CSReset	XXXXXX	+2,–2,+2,–1	+2,–2,–1,+2	–1,–2,+2,+2	+2,–1,–2,+2

40.3.1.3.6 Generation of A_n, B_n, C_n, D_n

The four bits $Sg_n[3:0]$ are used to randomize the signs of the quinary symbols (A_n, B_n, C_n, D_n) so that each symbol stream has no dc bias. The bits are used to generate binary symbols (SnA_n, SnB_n, SnC_n, SnD_n) that, when multiplied by the quinary symbols (TA_n, TB_n, TC_n, TD_n), result in (A_n, B_n, C_n, D_n).

PCS Transmit ensures a distinction between code-groups transmitted during idle mode plus SSD and those transmitted during other symbol periods. This distinction is accomplished by reversing the mapping of the sign bits when the condition (tx_enable_{n-2} + tx_enable_{n-4}) = 1. This sign reversal is controlled by the variable $Srev_n$ defined as

$$Srev_n = tx\_enable_{n-2} + tx\_enable_{n-4}$$

The binary symbols SnA_n, SnB_n, SnC_n, and SnD_n are defined using $Sg_n[3:0]$ as

$$SnA_n = \begin{cases} +1 & \text{if } [(Sg_n[0] \wedge Srev_n) = 0] \\ -1 & \text{else} \end{cases}$$

$$SnB_n = \begin{cases} +1 & \text{if } [(Sg_n[1] \wedge Srev_n) = 0] \\ -1 & \text{else} \end{cases}$$

$$SnC_n = \begin{cases} +1 & \text{if } [(Sg_n[2] \wedge Srev_n) = 0] \\ -1 & \text{else} \end{cases}$$

$$SnD_n = \begin{cases} +1 & \text{if } [(Sg_n[3] \wedge Srev_n) = 0] \\ -1 & \text{else} \end{cases}$$

The quinary symbols (A_n, B_n, C_n, D_n) are generated as the product of (SnA_n, SnB_n, SnC_n, SnD_n) and (TA_n, TB_n, TC_n, TD_n) respectively.

$$A_n = TA_n \times SnA_n$$

$$B_n = TB_n \times SnB_n$$

$$C_n = TC_n \times SnC_n$$

$$D_n = TD_n \times SnD_n$$

40.3.1.4 PCS Receive function

The PCS Receive function shall conform to the PCS Receive state diagram in Figure 40–10a including compliance with the associated state variables as specified in 40.3.3.

The PCS Receive function accepts received code-groups provided by the PMA Receive function via the parameter rx_symb_vector. To achieve correct operation, PCS Receive uses the knowledge of the encoding rules that are employed in the idle mode. PCS Receive generates the sequence of vectors of four quinary symbols (RA_n, RB_n, RC_n, RD_n) and indicates the reliable acquisition of the descrambler state by setting the parameter scr_status to OK. The sequence (RA_n, RB_n, RC_n, RD_n) is processed to generate the signals RXD<7:0>, RX_DV, and RX_ER, which are presented to the GMII. PCS Receive detects the transmission

of a stream of data from the remote station and conveys this information to the PCS Carrier Sense and PCS Transmit functions via the parameter 1000BTreceive.

40.3.1.4.1 Decoding of code-groups

When the PMA indicates that correct receiver operation has been achieved by setting the loc_rcvr_status parameter to the value OK, the PCS Receive continuously checks that the received sequence satisfies the encoding rule used in idle mode. When a violation is detected, PCS Receive assigns the value TRUE to the parameter 1000BTreceive and, by examining the last two received vectors (RA_{n-1}, RB_{n-1}, RC_{n-1}, RD_{n-1}) and (RA_n, RB_n, RC_n, RD_n), determines whether the violation is due to reception of SSD or to a receiver error.

Upon detection of SSD, PCS Receive also assigns the value TRUE to the parameter 1000BTreceive that is provided to the PCS Carrier Sense and Collision Presence functions. During the two symbol periods corresponding to SSD, PCS Receive replaces SSD by preamble bits. Upon the detection of SSD, the signal RX_DV is asserted and each received vector is decoded into a data octet RXD<7:0> until ESD is detected.

Upon detection of a receiver error, the signal RX_ER is asserted and the parameter rxerror_status assumes the value ERROR. De-assertion of RX_ER and transition to the IDLE state (rxerror_status=NO_ERROR) takes place upon detection of four consecutive vectors satisfying the encoding rule used in idle mode.

During reception of a stream of data, PCS Receive checks that the symbols RA_n, RB_n, RC_n, RD_n follow the encoding rule defined in 40.3.1.3.5 for ESD whenever they assume values ± 2. PCS Receive processes two consecutive vectors at each time n to detect ESD. Upon detection of ESD, PCS Receive de-asserts the signal RX_DV on the GMII. If the last symbol period of ESD indicates that a carrier extension is present, PCS Receive will assert the RX_ER signal on the GMII. If no extension is indicated in the ESD2 quartet, PCS Receive assigns the value FALSE to the parameter receiving. If an extension is present, the transition to the IDLE state occurs after detection of a valid idle symbol period and the parameter receiving remains TRUE until check_idle is TRUE. If a violation of the encoding rules is detected, PCS Receive asserts the signal RX_ER for at least one symbol period.

A premature stream termination is caused by the detection of invalid symbols during the reception of a data stream. Then, PCS Receive waits for the reception of four consecutive vectors satisfying the encoding rule used in idle mode prior to de-asserting the error indication. Note that RX_DV remains asserted during the symbol periods corresponding to the first three idle vectors, while RX_ER=TRUE is signaled on the GMII. The signal RX_ER is also asserted in the LINK FAILED state, which ensures that RX_ER remains asserted for at least one symbol period.

40.3.1.4.2 Receiver descrambler polynomials

The PHY shall descramble the data stream and return the proper sequence of code-groups to the decoding process for generation of RXD<7:0> to the GMII. For side-stream descrambling, the MASTER PHY shall employ the receiver descrambler generator polynomial $g'_M(x) = 1 + x^{20} + x^{33}$ and the SLAVE PHY shall employ the receiver descrambler generator polynomial $g'_S(x) = 1 + x^{13} + x^{33}$.

40.3.1.5 PCS Carrier Sense function

The PCS Carrier Sense function generates the GMII signal CRS, which the MAC uses for deferral in half duplex mode. The PCS shall conform to the Carrier Sense state diagram as depicted in Figure 40–11 including compliance with the associated state variables as specified in 40.3.3. The PCS Carrier Sense function is not required in a 1000BASE-T PHY that does not support half duplex operation.

Appendix 2 – Source code for Matlab simulations

The following is our full simulation code for the Matlab simulations. We include them for completing the description of our simulations.

Appendix 2.a. – Matlab simulation for 4B5B/MLT-3

```
%%%%% MAIN SIMULATION

%SYMBOLS
n = 40000;
symbols = floor(rand(1,n)*16);
%CONVERT to 4b5b
bits = [1,1,1,1,0];
for i = 1:n
    if symbols(i) == 0
        bits = [bits, 1 1 1 1 0];
    elseif symbols(i) == 1
        bits = [bits, 0 1 0 0 1];
    elseif symbols(i) == 2
        bits = [bits, 1 0 1 0 0];
    elseif symbols(i) == 3
        bits = [bits, 1 0 1 0 1];
    elseif symbols(i) == 4
        bits = [bits, 0 1 0 1 0];
    elseif symbols(i) == 5
        bits = [bits, 0 1 0 1 1];
    elseif symbols(i) == 6
        bits = [bits, 0 1 1 1 0];
    elseif symbols(i) == 7
        bits = [bits, 0 1 1 1 1];
    elseif symbols(i) == 8
        bits = [bits, 1 0 0 1 0];
    elseif symbols(i) == 9
        bits = [bits, 1 0 0 1 1];
    elseif symbols(i) == 10
        bits = [bits, 1 0 1 1 0];
    elseif symbols(i) == 11
        bits = [bits, 1 0 1 1 1];
    elseif symbols(i) == 12
        bits = [bits, 1 1 0 1 0];
    elseif symbols(i) == 13
        bits = [bits, 1 1 0 1 1];
    elseif symbols(i) == 14
        bits = [bits, 1 1 1 0 0];
    elseif symbols(i) == 15
        bits = [bits, 1 1 1 0 1];
    else i/0;
    end
end
%CONVERT to MLT
output = zeros(1,length(bits));
state = 0;
prevState = -1;
for i = 1:length(bits)
    if bits(i)==1
        if state==0 && prevState==-1
            output(i) = 1;
```

```
        elseif state==0 && prevState==1
            output(i) = -1;
        end
        if state==1 || state==-1
            output(i) = 0;
        end
        prevState = state;
    elseif bits(i)==0
        output(i) = state;
    else
        i/0;
    end
    state = output(i);
end
%ANALYSIS
bitsCount00 = 0;
bitsCount01 = 0;
bitsCount11 = 0;
bitsCount10 = 0;
for i = 2:length(bits)
    if bits(i-1)==0 && bits(i)==0
        bitsCount00 = bitsCount00 + 1;
    elseif bits(i-1)==0 && bits(i)==1
        bitsCount01 = bitsCount01 + 1;
    elseif bits(i-1)==1 && bits(i)==1
        bitsCount11 = bitsCount11 + 1;
    elseif bits(i-1)==1 && bits(i)==0
        bitsCount10 = bitsCount10 + 1;
    else
        i/0;
    end
end
%ENERGY
energy = output.^2;
%OUTPUT
n
%lengthCheck = length(bits)/(length(symbols) + 1)
avgSymbols = mean([0, symbols])
avgBits = mean(bits)
%avgDCOutput = mean(output)
avgEnergy = mean(sqrt(energy))
bitsFraction00 = bitsCount00 / (length(bits)-1)
bitsFraction01 = bitsCount01 / (length(bits)-1)
bitsFraction11 = bitsCount11 / (length(bits)-1)
bitsFraction10 = bitsCount10 / (length(bits)-1)
```

Appendix 2.b. – Matlab simulation for 8B10B

```
%%%% MAIN SIMULATION


%SYMBOLS
n = 20000;
symbols = [1 0 1 0 0 0 1 0];

code5 = [ 1 0 0 1 1 1   0 1 1 0 0 0; ...
  0 1 1 1 0 1   1 0 0 0 1 0; ...
  1 0 1 1 0 1   0 1 0 0 1 0; ...
  1 1 0 0 0 1   1 1 0 0 0 1; ...
  1 1 0 1 0 1   0 0 1 0 1 0; ...
                          ...
  1 0 1 0 0 1   1 0 1 0 0 1;  ...
```

```
  0 1 1 0 0 1   0 1 1 0 0 1;
  1 1 1 0 0 0   0 0 0 1 1 1;
  1 1 1 0 0 1   0 0 0 1 1 0;
  1 0 0 1 0 1   1 0 0 1 0 1;

  0 1 0 1 0 1   0 1 0 1 0 1;
  1 1 0 1 0 0   1 1 0 1 0 0;
  0 0 1 1 0 1   0 0 1 1 0 1;
  1 0 1 1 0 0   1 0 1 1 0 0;
  0 1 1 1 0 0   0 1 1 1 0 0;

  0 1 0 1 1 1   1 0 1 0 0 0;
  0 1 1 0 1 1   1 0 0 1 0 0;
  1 0 0 0 1 1   1 0 0 0 1 1;
  0 1 0 0 1 1   0 1 0 0 1 1;
  1 1 0 0 1 0   1 1 0 0 1 0;

  0 0 1 0 1 1   0 0 1 0 1 1;
  1 0 1 0 1 0   1 0 1 0 1 0;
  0 1 1 0 1 0   0 1 1 0 1 0;
  1 1 1 0 1 0   0 0 0 1 0 1;
  1 1 0 0 1 1   0 0 1 1 0 0;

  1 0 0 1 1 0   1 0 0 1 1 0;
  0 1 0 1 1 0   0 1 0 1 1 0;
  1 1 0 1 1 0   0 0 1 0 0 1;
  0 0 1 1 1 0   0 0 1 1 1 0;
  1 0 1 1 1 0   0 1 0 0 0 1;

  0 1 1 1 1 0   1 0 0 0 0 1;
  1 0 1 0 1 1   0 1 0 1 0 0
  ];

code3 = [ 1 0 1 1   0 1 0 0;
          1 0 0 1     1 0 0 1;
          0 1 0 1     0 1 0 1;
          1 1 0 0     0 0 1 1;
          1 1 0 1     0 0 1 0;
          1 0 1 0     1 0 1 0;
          0 1 1 0     0 1 1 0;
          1 1 1 0     0 0 0 1;
        ];

%CONVERT to 8b10b
bits = [];
i = 1;
currentRD = 0;
while i<length(symbols)
    if (i+7 > length(symbols))
        foo = length(symbols) - i;
        current8 = symbols(i:length(symbols));
        current8 = [current10, zeros(foo)];
    else
        current8 = symbols(i:i+7);
    end

    x = code5(getBitValues5(rot90(rot90(current8(4:8)))+1);
    if currentRD == 1
        code = x(7:12);
    else
        code = x(1:6);
    disparity = ones(code) - zeros(code);
    currentRD = compute_RD(currentRD, disparity);
    bits = [bits, code];

    y = code3(getBitValues3(rot90(rot90(current8(1:3)))+1);
    if (currentRD == 1
        code = y(5:8);
    else
        code = y(1:4);
    disparity = ones(code) - zeros(code);
```

```
    currentRD = compute_RD(currentRD, disparity);
    bits = [bits, code];

    i+=8;
end



%%%% SUPPORT FUNCTIONS

function [ number ] = getBitValues5( bits )

if bits == [0 0 0 0 0]
        number = 0;
elseif bits == [0 0 0 0 1]
        number = 1;
elseif bits == [ 0 0 0 1 0]
        number = 2;
elseif bits == [ 0 0 0 1 1]
        number = 3;
elseif bits == [ 0 0 1 0 0]
        number = 4;
elseif bits == [ 0 0 1 0 1]
        number = 5;
elseif bits == [ 0 0 1 1 0]
        number = 6;
elseif bits == [ 0 0 1 1 1]
        number = 7;
elseif bits == [ 0 1 0 0 0]
        number = 8;
elseif bits == [ 0 1 0 0 1]
        number = 9;
elseif bits == [ 0 1 0 1 0]
        number = 10;
elseif bits == [ 0 1 0 1 1]
        number = 11;
elseif bits == [ 0 1 1 0 0]
        number = 12;
elseif bits == [ 0 1 1 0 1]
        number = 13;
elseif bits == [ 0 1 1 1 0]
        number = 14;
elseif bits == [ 0 1 1 1 1]
        number = 15;
elseif bits == [ 1 0 0 0 0]
        number = 16;
elseif bits == [ 1 0 0 0 1]
        number = 17;
elseif bits == [ 1 0 0 1 0]
        number = 18;
elseif bits == [ 1 0 0 1 1]
        number = 19;
elseif bits == [ 1 0 1 0 0]
        number = 20;
elseif bits == [ 1 0 1 0 1]
        number = 21;
elseif bits == [ 1 0 1 1 0]
        number = 22;
elseif bits == [ 1 0 1 1 1]
        number = 23;
elseif bits == [ 1 1 0 0 0]
        number = 24;
elseif bits == [ 1 1 0 0 1]
        number = 25;
elseif bits == [ 1 1 0 1 0]
        number = 26;
elseif bits == [ 1 1 0 1 1]
        number = 27;
elseif bits == [ 1 1 1 0 0]
```

```
        number = 28;
elseif bits == [ 1 1 1 0 1]
        number = 29;
elseif bits == [ 1 1 1 1 0]
        number = 30;
elseif bits == [ 1 1 1 1 1]
        number = 31;
else
        number = 1/0;
end
```

```
function [ RD ] = compute_RD( currentRD, disparity )
%UNTITLED1 Summary of this function goes here
%  Detailed explanation goes here

if currentRD==-1 && disparity == -2
    RD = 0;
elseif currentRD==-1 && disparity == 0
    RD = -1;
elseif currentRD == -1 && disparity == 2
    RD = 1;
elseif currentRD ==1 && disparity == -2
    RD = -1;
elseif currentRD == 1 && disparity == 0
    RD = 1;
elseif currentRD == 1 && disparity == 2
    RD = 0;
else
    RD = 0;
end
```

```
function [ number ] = getBitValues3( bits )

if bits == [0 0 0 ]
        number = 0;
elseif bits == [ 0 0 1]
        number = 1;
elseif bits == [ 0 1 0]
        number = 2;
elseif bits == [ 0 1 1]
        number = 3;
elseif bits == [ 1 0 0]
        number = 4;
elseif bits == [ 1 0 1]
        number = 5;
elseif bits == [ 1 1 0]
        number = 6;
elseif bits == [ 1 1 1]
        number = 7;
else
        number = 1/0;
end
```

Appendix 2.c. – Matlab simulation for 4D-PAM5

```
%%%% MAIN SIMULATION


TXD=[1,1,1,1,1,1,1,1, 0,0,0,0,0,0, 0, 0];
Scr = [];
Syn=[];
Sxn=[];
Sgn=[];
Csn=[];

Sy=zeros(1,4);
Sx=zeros(1,4);
Sg=zeros(1,4);
Cs=zeros(1,3);

even = 1;
i = 1;

[Table1, Table2] = initializeLookupTables;
bits = [];

while(i<= length(TXD))

    %save state
    Syn = Sy;
    Sxn = Sx;
    Sgn = Sg;
    Csn = Cs;

    Scr = getMasterScrambler(Scr);
    Sy = [Scr(1), bitxor(Scr(4), Scr(9)), ...
            bitxor(Scr(7), Scr(17)), ...
            bitxor(bitxor(bitxor(Scr(10), Scr(15)), Scr(20)), Scr(25))];
    Sx = [bitxor(Scr(5), Scr(7)), bitxor(bitxor(bitxor(Scr(8), Scr(10)), Scr(13)), Scr(15)),...
            bitxor(bitxor(bitxor(Scr(11), Scr(13)), Scr(21)), Scr(23)),...
            bitxor(bitxor(bitxor(bitxor(bitxor(bitxor(Scr(14), Scr(16)), Scr(19)),
Scr(21)), Scr(24)), Scr(26)), Scr(29)), Scr(31))];
    Sg = [bitxor(Scr(2), Scr(6)), bitxor(bitxor(bitxor(Scr(5), Scr(9)), Scr(10)), Scr(14)),...
            bitxor(bitxor(bitxor(Scr(8), Scr(12)), Scr(18)), Scr(22)),...
            bitxor(bitxor(bitxor(bitxor(bitxor(bitxor(bitxor(Scr(11), Scr(15)), Scr(16)),
Scr(20)), Scr(21)), Scr(25)), Scr(26)), Scr(30))];

    Sc = zeros(1, 8);
    Sc(5:8) = Sx(1:4);
    if(even)
        Sc(2:4) = Sy(2:4);
    else
        Sc(2:4) = xor(Syn(2:4), [1,1,1]);
    end

    Sc(1) = Sy(1);

    Sd = zeros(1,9);
    Cs(1) = Csn(3);
    Sd(9) = Cs(1);
    Sd(1:8) = xor(Sc(1:8), TXD(i:i+7));
    display(TXD(i:i+7));

    Cs(1) = bitxor(Sd(7), Csn(1));
    Cs(2) = bitxor(Sd(8), Csn(2));

    Tx = lookupSymbol(Table1, Table2, Sd);
```

```
    Ta = Tx(1);
    Tb = Tx(2);
    Tc = Tx(3);
    Td = Tx(4);

    if (bitxor(Sg(1), 0) == 0) SnA = 1; else SnA = -1; end
    if (bitxor(Sg(2), 0) == 0) SnB = 1; else SnB = -1; end
    if (bitxor(Sg(3), 0) == 0) SnC = 1; else SnC = -1; end
    if (bitxor(Sg(4), 0) == 0) SnD = 1; else SnD = -1; end


    bits = [bits, Ta*SnA, Tb*SnB, Tc*SnC, Td*SnD];

    display(Scr);
    display(Sy);
    display(Sx);
    display(Sg);
    display(Cs); display(Csn); display(Sc);
    display(Sd);
    even = ~even;
    i= i+8;
end

bits


%%%% SUPPORT FUNCTIONS

function [Table1, Table2] = initializeLookupTables();

Table1 = [ ...
          0,0,0,0    0,0,1,1    0,1,1, 0    0,1, 0,1;
         -2, 0, 0, 0   -2, 0,1,1   -2,1,1, 0   -2,1, 0,1;
          0,-2, 0, 0    0,-2,1,1    0,-1,1, 0    0,-1, 0,1;
         -2,-2, 0, 0   -2,-2,1,1   -2,-1,1, 0   -2,-1, 0,1;
          0, 0,-2, 0    0, 0,-1,1    0,1,-1, 0    0,1,-2,1;
         -2, 0,-2, 0   -2, 0,-1,1   -2,1,-1, 0   -2,1,-2,1;
          0,-2,-2, 0    0,-2,-1,1    0,-1,-1, 0    0,-1,-2,1;
         -2,-2,-2, 0   -2,-2,-1,1   -2,-1,-1, 0   -2,-1,-2,1;
          0, 0, 0,-2    0, 0,1,-1    0,1,1,-2    0,1, 0,-1;
         -2, 0, 0,-2   -2, 0,1,-1   -2,1,1,-2   -2,1, 0,-1;
          0,-2, 0,-2    0,-2,1,-1    0,-1,1,-2    0,-1, 0,-1;
         -2,-2, 0,-2   -2,-2,1,-1   -2,-1,1,-2   -2,-1, 0,-1;
          0, 0,-2,-2    0, 0,-1,-1    0,1,-1,-2    0,1,-2,-1;
         -2, 0,-2,-2   -2, 0,-1,-1   -2,1,-1,-2   -2,1,-2,-1;
            0,-2,-2,-2    0,-2,-1,-1    0,-1,-1,-2    0,-1,-2,-1;
           -2,-2,-2,-2   -2,-2,-1,-1   -2,-1,-1,-2   -2,-1,-2,-1;
           1,1,1,1   1,1, 0, 0   1, 0, 0,1   1, 0,1, 0;
           -1,1,1,1   -1,1, 0, 0   -1, 0, 0,1   -1, 0,1, 0;
           1,-1,1,1   1,-1, 0, 0   1,-2, 0,1   1,-2,1, 0;
           -1,-1,1,1   -1,-1, 0, 0   -1,-2, 0,1   -1,-2,1, 0;
           1,1,-1,1   1,1,-2, 0   1, 0,-2,1   1, 0,-1, 0;
           -1,1,-1,1   -1,1,-2, 0   -1, 0,-2,1   -1, 0,-1, 0;
           1,-1,-1,1   1,-1,-2, 0   1,-2,-2,1   1,-2,-1, 0;
           -1,-1,-1,1   -1,-1,-2, 0   -1,-2,-2,1   -1,-2,-1, 0;
           1,1,1,-1   1,1, 0,-2   1, 0, 0,-1   1, 0,1,-2;
           -1,1,1,-1   -1,1, 0,-2   -1, 0, 0,-1   -1, 0,1,-2;
           1,-1,1,-1   1,-1, 0,-2   1,-2, 0,-1   1,-2,1,-2;
           -1,-1,1,-1   -1,-1, 0,-2   -1,-2, 0,-1   -1,-2,1,-2;
           1,1,-1,-1   1,1,-2,-2   1, 0,-2,-1   1, 0,-1,-2;
           -1,1,-1,-1   -1,1,-2,-2   -1, 0,-2,-1   -1, 0,-1,-2;
           1,-1,-1,-1   1,-1,-2,-2   1,-2,-2,-1   1,-2,-1,-2;
           -1,-1,-1,-1   -1,-1,-2,-2   -1,-2,-2,-1   -1,-2,-1,-2;
           2, 0, 0, 0   2, 0,1,1   2,1,1, 0   2,1, 0,1;
           2,-2, 0, 0   2,-2,1,1   2,-1,1, 0   2,-1, 0,1;
           2, 0,-2, 0   2, 0,-1,1   2,1,-1, 0   2,1,-2,1;
           2,-2,-2, 0   2,-2,-1,1   2,-1,-1, 0   2,-1,-2,1;
           2, 0, 0,-2   2, 0,1,-1   2,1,1,-2   2,1, 0,-1;
           2,-2, 0,-2   2,-2,1,-1   2,-1,1,-2   2,-1, 0,-1;
           2, 0,-2,-2   2, 0,-1,-1   2,1,-1,-2   2,1,-2,-1;
```

```
           2,-2,-2,-2   2,-2,-1,-1   2,-1,-1,-2   2,-1,-2,-1;
            0, 0,2, 0   1,1,2, 0   1, 0,2,1    0,1,2,1;
           -2, 0,2, 0   -1,1,2, 0   -1, 0,2,1   -2,1,2,1;
            0,-2,2, 0   1,-1,2, 0   1,-2,2,1    0,-1,2,1;
           -2,-2,2, 0   -1,-1,2, 0   -1,-2,2,1   -2,-1,2,1;
            0, 0,2,-2   1,1,2,-2   1, 0,2,-1    0,1,2,-1;
           -2, 0,2,-2   -1,1,2,-2   -1, 0,2,-1   -2,1,2,-1;
            0,-2,2,-2   1,-1,2,-2   1,-2,2,-1    0,-1,2,-1;
           -2,-2,2,-2   -1,-1,2,-2   -1,-2,2,-1   -2,-1,2,-1;
            0,2, 0, 0    0,2,1,1   1,2, 0,1   1,2,1, 0;
           -2,2, 0, 0   -2,2,1,1   -1,2, 0,1   -1,2,1, 0;
            0,2,-2, 0    0,2,-1,1   1,2,-2,1   1,2,-1, 0;
           -2,2,-2, 0   -2,2,-1,1   -1,2,-2,1   -1,2,-1, 0;
            0,2, 0,-2    0,2,1,-1   1,2, 0,-1   1,2,1,-2;
           -2,2, 0,-2   -2,2,1,-1   -1,2, 0,-1   -1,2,1,-2;
            0,2,-2,-2    0,2,-1,-1   1,2,-2,-1   1,2,-1,-2;
           -2,2,-2,-2   -2,2,-1,-1   -1,2,-2,-1   -1,2,-1,-2;
            0, 0, 0,2   1,1, 0,2    0,1,1,2   1, 0,1,2;
           -2, 0, 0,2   -1,1, 0,2   -2,1,1,2   -1, 0,1,2;
            0,-2, 0,2   1,-1, 0,2    0,-1,1,2   1,-2,1,2;
           -2,-2, 0,2   -1,-1, 0,2   -2,-1,1,2   -1,-2,1,2;
            0, 0,-2,2   1,1,-2,2    0,1,-1,2   1, 0,-1,2;
           -2, 0,-2,2   -1,1,-2,2   -2,1,-1,2   -1, 0,-1,2;
            0,-2,-2,2   1,-1,-2,2    0,-1,-1,2   1,-2,-1,2;
           -2,-2,-2,2   -1,-1,-2,2   -2,-1,-1,2   -1,-2,-1,2;
           ];

Table2=[...
               0, 0, 0,1    0, 0,1, 0    0,1,1,1    0,1, 0, 0;
           -2, 0, 0,1   -2, 0,1, 0   -2,1,1,1   -2,1, 0, 0;
            0,-2, 0,1    0,-2,1, 0    0,-1,1,1    0,-1, 0, 0;
           -2,-2, 0,1   -2,-2,1, 0   -2,-1,1,1   -2,-1, 0, 0;
            0, 0,-2,1    0, 0,-1, 0    0,1,-1,1    0,1,-2, 0;
           -2, 0,-2,1   -2, 0,-1, 0   -2,1,-1,1   -2,1,-2, 0;
            0,-2,-2,1    0,-2,-1, 0    0,-1,-1,1    0,-1,-2, 0;
           -2,-2,-2,1   -2,-2,-1, 0   -2,-1,-1,1   -2,-1,-2, 0;
            0, 0, 0,-1    0, 0,1,-2    0,1,1,-1    0,1, 0,-2;
           -2, 0, 0,-1   -2, 0,1,-2   -2,1,1,-1   -2,1, 0,-2;
            0,-2, 0,-1    0,-2,1,-2    0,-1,1,-1    0,-1, 0,-2;
           -2,-2, 0,-1   -2,-2,1,-2   -2,-1,1,-1   -2,-1, 0,-2;
            0, 0,-2,-1    0, 0,-1,-2    0,1,-1,-1    0,1,-2,-2;
           -2, 0,-2,-1   -2, 0,-1,-2   -2,1,-1,-1   -2,1,-2,-2;
            0,-2,-2,-1    0,-2,-1,-2    0,-1,-1,-1    0,-1,-2,-2;
           -2,-2,-2,-1   -2,-2,-1,-2   -2,-1,-1,-1   -2,-1,-2,-2;
           1,1,1, 0   1,1, 0,1   1, 0, 0, 0   1, 0,1,1;
           -1,1,1, 0   -1,1, 0,1   -1, 0, 0, 0   -1, 0,1,1;
           1,-1,1, 0   1,-1, 0,1   1,-2, 0, 0   1,-2,1,1;
           -1,-1,1, 0   -1,-1, 0,1   -1,-2, 0, 0   -1,-2,1,1;
           1,1,-1, 0   1,1,-2,1   1, 0,-2, 0   1, 0,-1,1;
           -1,1,-1, 0   -1,1,-2,1   -1, 0,-2, 0   -1, 0,-1,1;
           1,-1,-1, 0   1,-1,-2,1   1,-2,-2, 0   1,-2,-1,1;
           -1,-1,-1, 0   -1,-1,-2,1   -1,-2,-2, 0   -1,-2,-1,1;
           1,1,1,-2   1,1, 0,-1   1, 0, 0,-2   1, 0,1,-1;
           -1,1,1,-2   -1,1, 0,-1   -1, 0, 0,-2   -1, 0,1,-1;
           1,-1,1,-2   1,-1, 0,-1   1,-2, 0,-2   1,-2,1,-1;
           -1,-1,1,-2   -1,-1, 0,-1   -1,-2, 0,-2   -1,-2,1,-1;
           1,1,-1,-2   1,1,-2,-1   1, 0,-2,-2   1, 0,-1,-1;
           -1,1,-1,-2   -1,1,-2,-1   -1, 0,-2,-2   -1, 0,-1,-1;
           1,-1,-1,-2   1,-1,-2,-1   1,-2,-2,-2   1,-2,-1,-1;
           -1,-1,-1,-2   -1,-1,-2,-1   -1,-2,-2,-2   -1,-2,-1,-1;
           2, 0, 0,1   2, 0,1, 0   2,1,1,1   2,1, 0, 0;
           2,-2, 0,1   2,-2,1, 0   2,-1,1,1   2,-1, 0, 0;
           2, 0,-2,1   2, 0,-1, 0   2,1,-1,1   2,1,-2, 0;
           2,-2,-2,1   2,-2,-1, 0   2,-1,-1,1   2,-1,-2, 0;
           2, 0, 0,-1   2, 0,1,-2   2,1,1,-1   2,1, 0,-2;
           2,-2, 0,-1   2,-2,1,-2   2,-1,1,-1   2,-1, 0,-2;
           2, 0,-2,-1   2, 0,-1,-2   2,1,-1,-1   2,1,-2,-2;
           2,-2,-2,-1   2,-2,-1,-2   2,-1,-1,-1   2,-1,-2,-2;
            0, 0,2,1   1,1,2,1   1, 0,2, 0    0,1,2, 0;
           -2, 0,2,1   -1,1,2,1   -1, 0,2, 0   -2,1,2, 0;
            0,-2,2,1   1,-1,2,1   1,-2,2, 0    0,-1,2, 0;
```

```
          -2,-2,2,1   -1,-1,2,1   -1,-2,2, 0   -2,-1,2, 0;
           0, 0,2,-1   1,1,2,-1   1, 0,2,-2    0,1,2,-2;
          -2, 0,2,-1   -1,1,2,-1   -1, 0,2,-2   -2,1,2,-2;
           0,-2,2,-1   1,-1,2,-1   1,-2,2,-2    0,-1,2,-2;
          -2,-2,2,-1   -1,-1,2,-1   -1,-2,2,-2   -2,-1,2,-2;
           0,2, 0,1    0,2,1, 0   1,2, 0, 0   1,2,1,1;
          -2,2, 0,1   -2,2,1, 0   -1,2, 0, 0   -1,2,1,1;
           0,2,-2,1    0,2,-1, 0   1,2,-2, 0   1,2,-1,1;
          -2,2,-2,1   -2,2,-1, 0   -1,2,-2, 0   -1,2,-1,1;
           0,2, 0,-1    0,2,1,-2   1,2, 0,-2   1,2,1,-1;
          -2,2, 0,-1   -2,2,1,-2   -1,2, 0,-2   -1,2,1,-1;
           0,2,-2,-1    0,2,-1,-2   1,2,-2,-2   1,2,-1,-1;
          -2,2,-2,-1   -2,2,-1,-2   -1,2,-2,-2   -1,2,-1,-1;
          1,1,1,2    0, 0,1,2   1, 0, 0,2    0,1, 0,2;
          -1,1,1,2   -2, 0,1,2   -1, 0, 0,2   -2,1, 0,2;
          1,-1,1,2    0,-2,1,2   1,-2, 0,2    0,-1, 0,2;
          -1,-1,1,2   -2,-2,1,2   -1,-2, 0,2   -2,-1, 0,2;
          1,1,-1,2    0, 0,-1,2   1, 0,-2,2    0,1,-2,2;
          -1,1,-1,2   -2, 0,-1,2   -1, 0,-2,2   -2,1,-2,2;
          1,-1,-1,2    0,-2,-1,2   1,-2,-2,2    0,-1,-2,2;
          -1,-1,-1,2   -2,-2,-1,2   -1,-2,-2,2   -2,-1,-2,2;
    ];



function scrambler = getMasterScrambler( currentScrambler )


if(length(currentScrambler) ==0)
    scrambler = round(random('uniform', zeros(1,33),ones(1,33)));
else
    scrambler = [xor(currentScrambler(13), currentScrambler(33)), currentScrambler(1:32)];
end



function scrambler = getSlaveScrambler( currentScrambler )


if(length(currentScrambler) ==0)
    scrambler = round(random('uniform', zeros(1,33),ones(1,33)));
else
    scrambler = [xor(currentScrambler(13), currentScrambler(33)), currentScrambler(1:32)];
end


function T = lookupSymbol( Table1, Table2, Sd )

row = getBitValues6(rot90(rot90(Sd(1:6))))+1;

if(Sd(7:9) ==  [0,0,0])
    T = Table1(row, 1:4);
elseif(Sd(7:9) == [0,1,0])
    T = Table1(row, 5:8);
elseif(Sd(7:9) == [1,0,0])
    T = Table1(row, 9:12);
elseif(Sd(7:9) == [1,1,0])
    T = Table1(row, 13:16);
elseif(Sd(7:9) ==  [0,0,1])
    T = Table2(row, 1:4);
elseif(Sd(7:9) == [0,1,1])
    T = Table2(row, 5:8);
elseif(Sd(7:9) == [1,0,1])
    T = Table2(row, 9:12);
elseif(Sd(7:9) == [1,1,1])
    T = Table2(row, 13:16);
else
    T = 1/0;
end
```

```
function [ number ] = getBitValues6( bits )

if bits == [0 0 0 0 0 0]
        number = 0;
elseif bits == [0 0 0 0 0 1]
        number = 1;
elseif bits == [0 0 0 0 1 0]
        number = 2;
elseif bits == [0 0 0 0 1 1]
        number = 3;
elseif bits == [0 0 0 1 0 0]
        number = 4;
elseif bits == [0 0 0 1 0 1]
        number = 5;
elseif bits == [0 0 0 1 1 0]
        number = 6;
elseif bits == [0 0 0 1 1 1]
        number = 7;
elseif bits == [0 0 1 0 0 0]
        number = 8;
elseif bits == [0 0 1 0 0 1]
        number = 9;
elseif bits == [0 0 1 0 1 0]
        number = 10;
elseif bits == [0 0 1 0 1 1]
        number = 11;
elseif bits == [0 0 1 1 0 0]
        number = 12;
elseif bits == [0 0 1 1 0 1]
        number = 13;
elseif bits == [0 0 1 1 1 0]
        number = 14;
elseif bits == [0 0 1 1 1 1]
        number = 15;
elseif bits == [0 1 0 0 0 0]
        number = 16;
elseif bits == [0 1 0 0 0 1]
        number = 17;
elseif bits == [0 1 0 0 1 0]
        number = 18;
elseif bits == [0 1 0 0 1 1]
        number = 19;
elseif bits == [0 1 0 1 0 0]
        number = 20;
elseif bits == [0 1 0 1 0 1]
        number = 21;
elseif bits == [0 1 0 1 1 0]
        number = 22;
elseif bits == [0 1 0 1 1 1]
        number = 23;
elseif bits == [0 1 1 0 0 0]
        number = 24;
elseif bits == [0 1 1 0 0 1]
        number = 25;
elseif bits == [0 1 1 0 1 0]
        number = 26;
elseif bits == [0 1 1 0 1 1]
        number = 27;
elseif bits == [0 1 1 1 0 0]
        number = 28;
elseif bits == [0 1 1 1 0 1]
        number = 29;
elseif bits == [0 1 1 1 1 0]
        number = 30;
elseif bits == [0 1 1 1 1 1]
        number = 31;
elseif bits == [1 0 0 0 0 0]
        number = 32;
elseif bits == [1 0 0 0 0 1]
        number = 33;
elseif bits == [1 0 0 0 1 0]
```

```
        number = 34;
elseif bits == [1 0 0 0 1 1]
        number = 35;
elseif bits == [1 0 0 1 0 0]
        number = 36;
elseif bits == [1 0 0 1 0 1]
        number = 37;
elseif bits == [1 0 0 1 1 0]
        number = 38;
elseif bits == [1 0 0 1 1 1]
        number = 39;
elseif bits == [1 0 1 0 0 0]
        number = 40;
elseif bits == [1 0 1 0 0 1]
        number = 41;
elseif bits == [1 0 1 0 1 0]
        number = 42;
elseif bits == [1 0 1 0 1 1]
        number = 43;
elseif bits == [1 0 1 1 0 0]
        number = 44;
elseif bits == [1 0 1 1 0 1]
        number = 45;
elseif bits == [1 0 1 1 1 0]
        number = 46;
elseif bits == [1 0 1 1 1 1]
        number = 47;
elseif bits == [1 1 0 0 0 0]
        number = 48;
elseif bits == [1 1 0 0 0 1]
        number = 49;
elseif bits == [1 1 0 0 1 0]
        number = 50;
elseif bits == [1 1 0 0 1 1]
        number = 51;
elseif bits == [1 1 0 1 0 0]
        number = 52;
elseif bits == [1 1 0 1 0 1]
        number = 53;
elseif bits == [1 1 0 1 1 0]
        number = 54;
elseif bits == [1 1 0 1 1 1]
        number = 55;
elseif bits == [1 1 1 0 0 0]
        number = 56;
elseif bits == [1 1 1 0 0 1]
        number = 57;
elseif bits == [1 1 1 0 1 0]
        number = 58;
elseif bits == [1 1 1 0 1 1]
        number = 59;
elseif bits == [1 1 1 1 0 0]
        number = 60;
elseif bits == [1 1 1 1 0 1]
        number = 61;
elseif bits == [1 1 1 1 1 0]
        number = 62;
elseif bits == [1 1 1 1 1 1]
        number = 63;
else
        number = 1/0;
end
```