Title: Proposal for an Informative Annex for SPI-3, “Protection for the Asynchronous Information Phases (COMMAND, MESSAGE, and STATUS)”

To: T10 Technical committee
From: Mark Evans and Bruce Leshay
Quantum Corporation
500 McCarthy Boulevard
Milpitas, CA USA 95035
Phone: 408-894-4019
Fax: 408-952-3620
Email: mark.evans@quantum.com
bruce.leshay@quantum.com
Date: 28 May 1999

With the advent of CRC on the DT data phases, customers have requested similar types of protection on non-data information phases (COMMAND, MESSAGE, and STATUS) in those cases where the information could become corrupted (e.g., by hot plugging events). The solution in this proposal resolves this issue.

The following recommended changes from the SPI-3 working group meeting May 4th in Manchester are included in this revision: Figure x1 (Protection code generation) has been updated from previous versions of this proposal to correctly reflect the algorithm. Additional explanatory text has been added to x.1 (Code description). “Prototypes” have been added to the C code example. The C code example was moved to x5.1. The Verilog example from the reflector was added in x5.2. Examples of protection code calculations were added in x5.3. Modifications to x2 were made to allow the details of enabling and disabling the feature to be vendor specific, including examples.

Annex x

(Informative)

Improved Error Detection for the Asynchronous Information Phases (COMMAND, MESSAGE, and STATUS)

This annex describes an enhanced error detection method for the COMMAND, MESSAGE, and STATUS asynchronous information transfer phases. In systems not implementing this scheme, these phases only transfer information on the lower eight data bits of an SCSI bus with only normal parity protection on those transfers. Therefore, additional check information can be transferred on the upper eight data bits in order to improve error detection capabilities. Since the upper eight data bits of the bus are used for this scheme, this error detection method is only available on wide SCSI devices that are on wide SCSI busses.

x1 Protection Code

The following are the covered signals to be encoded and details of the protection code to be used on the asynchronous information phases.

x1.1 Covered signals

Table x1 defines the signals to be covered by the protection code and their bit locations in the 21-bit code word. When a device receives an information byte, it also latches the state of the other SCSI signals and values noted in the table.

Table x1

Code Word Bit Location	SCSI Signal	Meaning
0	DB(0)	Data bit 0 of the information byte
1	DB(1)	Data bit 1 of the information byte
2	DB(2)	Data bit 2 of the information byte
3	DB(3)	Data bit 3 of the information byte
4	DB(4)	Data bit 4 of the information byte
5	DB(5)	Data bit 5 of the information byte
6	DB(6)	Data bit 6 of the information byte
7	DB(7)	Data bit 7 of the information byte
8	DB(8)	Reserved (see note 1)
9	DB(9)	Reserved (see note 1)
10	RSVD	Reserved (see note 2)
11	RSVD	Reserved (see note 2)
12	RSVD	Reserved (see note 2)
13	Seq ID 0	Sequence ID bit 0
14	Seq ID 1	Sequence ID bit 1
15	DB(10)	Redundant bit 0 of the code word
16	DB(11)	Redundant bit 1 of the code word
17	DB(12)	Redundant bit 2 of the code word
18	DB(13)	Redundant bit 3 of the code word
19	DB(14)	Redundant bit 4 of the code word
20	DB(15)	Redundant bit 5 of the code word

Note 1: DB(8) and DB(9) are reserved for future use. These signals are negated by the transmitting device and are ignored by the receiving device. Both the transmitter and receiver encode these signals in the protection code.

Note 2: For calculation purposes these signals are zero. However, these virtual signals could be used for other functions in a future standard.

The Sequence IDs are virtual signals that are encoded in the protection code but not transferred on the SCSI bus. A sequence of consecutive information transfers during a MESSAGE, COMMAND, or STATUS phase is a run. The Sequence ID increments during a run. A new run begins on every phase change or on each MESSAGE OUT retry.

For each new run, the Sequence ID is set to zero for the first word transferred, set to one for the second word transferred, set to two for the third word transferred, and set to three for the fourth word transferred. The Sequence ID then cycles back to being set to zero for the fifth word transferred, and so forth until the run is complete. At the beginning of the next run, the Sequence ID is set to zero again.

The Sequence ID provides detection of errors that occur when an information transfer is missed or double clocked. A Sequence ID error causes a protection code error. If a protection code error is detected, then the information transfer is invalid. The method for recovery from these errors is the same as the method for parity error recovery (see x4).

x1.2 Code Description

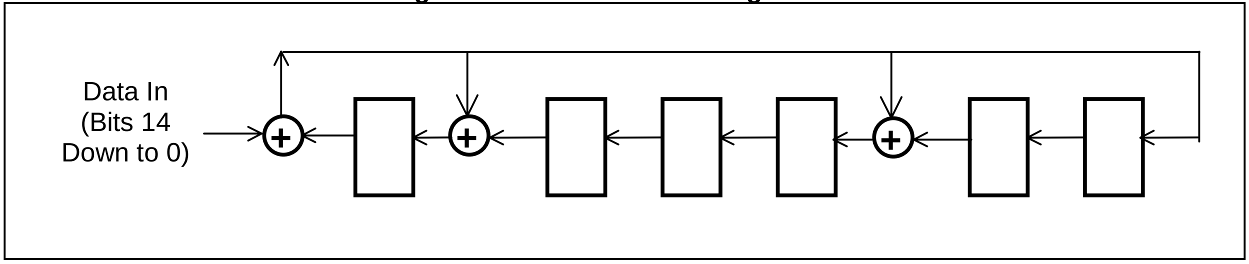
The protection code is a cyclic binary BCH code.

Code	Maximum data bits allowed	Number of redundant bits	Minimum distance of the code
(21,15,4)	15	6	4

The BCH protection code is a cyclic code with a generator polynomial of $x^6 + x^5 + x^2 + 1$.

The canonical form of the code generator is shown in Figure x1. This is a serial implementation: the register is initialized to zero, then the data is fed in one bit at a time, codeword bit 14 (as defined above) first, followed by codeword bits 13, 12, 11, and so on until bit 0. As each data bit is input, the shift register is clocked. When all 15 bits have been clocked into the generator, the check bits are available in the registers, check bit 0 (codeword bit 15) on the right in the diagram and check bit 5 (codeword bit 20) on the left. The **+** signs represent an XOR operation.

Figure x1 – Protection code generator



Using this representation as a baseline, it is possible to construct logic to generate the six check bits from an input data stream of n-bit width, including all 15 bits simultaneously, which is the expected implementation, wherein:

Redundant bit 0 is the XOR of code word bits 0, 1, 2, 3, 5, 6, 7, 10, 11, and 13.
Redundant bit 1 is the XOR of code word bits 1, 2, 3, 4, 6, 7, 8, 11, 12, and 14.
Redundant bit 2 is the XOR of code word bits 0, 1, 4, 6, 8, 9, 10, 11, and 12.
Redundant bit 3 is the XOR of code word bits 1, 2, 5, 7, 9, 10, 11, 12, and 13.
Redundant bit 4 is the XOR of code word bits 2, 3, 6, 8, 10, 11, 12, 13, and 14.
Redundant bit 5 is the XOR of code word bits 0, 1, 2, 4, 5, 6, 9, 10, 12, and 14.

See x.5 for coding examples and examples of protection coding output.

x1.3 Error Detection Properties

This protection code was selected to have adequate detection properties for asynchronous information transfer phases, given that these transfers are inherently less prone to errors and that these transfers have short code words (approximately 20 bits as compared to thousands of bits during a DT data phase). The BCH protection code Hamming distance is a minimum of four, the same as achieved by the data CRC for transfers of less than eight kilobytes. The protection code will detect all errors of three bits or fewer, all errors of an odd number of bits, and 98.4% of all possible errors.

x2 Protection Code Usage

Protection code checking is enabled or disabled on an I_T nexus basis. All COMMAND, MESSAGE, and STATUS phase information is checked for an I_T nexus while checking is enabled. Protection code checking is disabled after a power cycle, after a hard reset, after a TARGET RESET message, and after a change in the transceiver mode (e.g., LVD mode to MSE mode). Protection code checking is always disabled if information unit transfers are enabled.

x2.1 Protection Code Transmission

SCSI devices supporting this protection code transmit the protection code check data during all COMMAND, MESSAGE, and STATUS phases. The check data is transmitted even if detection is not enabled.

x2.2 Enabling Protection Code Checking

An SCSI device enables protection code checking for an I_T nexus when it detects that valid protection code data is being transmitted on the upper byte of the SCSI bus. The frequency that an SCSI device will try to enable protection code checking and the number of valid protection code bytes required is vendor specific. The following are some possible times when an SCSI device could try to enable protection code checking:

1. During the first COMMAND, MESSAGE, or STATUS phase after a power cycle, after a hard reset, after a TARGET RESET message, or after a change in the transceiver mode.

2. Any time that an SCSI device removal and insertion is possible, i.e. after a UNIT ATTENTION condition.

3. During the MESSAGE phases of a negotiation.

x2.3 Disabling Protection Code Checking

The removal and insertion of an SCSI device could require that protection code checking be disabled for a previously enabled I_T nexus. An SCSI device disables protection code checking when it detects that no protection code data is being transmitted on the upper byte. The determination that no protection code data is being transmitted is vendor specific. The following are some possible ways that an SCSI device could determine that no protection code data is being transmitted:

1. The DB(15-8) and DB(P1) signals are continuously deasserted while there is good parity on DB(7-0) and DB(P_CRCA).

2. The protection code has a consistent error while there is good parity on DB(7-0) and DB(P_CRCA).

x3 Parity

Normal wide parity is used during a protected transfer of COMMAND, MESSAGE, or STATUS information. DB(P_CRCA) contains the parity for DB(7-0), and DB(P1) contains the parity for DB(15-8).

x4 Error handling

Protection code errors are handled exactly like parity errors during COMMAND, MESSAGE, or STATUS phases as defined in the relevant subclauses on exception condition handling in clause 11.1.

x5 Examples

x5.1 C code example

The following is an example of a program written in the C programming language that would generate the check bits for the protection scheme described in this annex.

```
/* C-code implementation of (21,15,4) cyclic code calculated in parallel.   */
/* Input a 15-bit word, output six check bits.                              */
/* The implementation splits the 15-bit word into an 8-bit and 7-bit word.  */
/* Each word is input to its own lookup table, producing two 6-bit results  */
/* The two 6-bit results are then XOR'd together to create the final six    */
/* check bits.                                                              */

/* The correspondence between the input data and SCSI bus is:               */
/* data[0]  :   SCSI DB[0]                                                  */
/* data[1]  :   SCSI DB[1]                                                  */
/* data[2]  :   SCSI DB[2]                                                  */
/* data[3]  :   SCSI DB[3]                                                  */
/* data[4]  :   SCSI DB[4]                                                  */
/* data[5]  :   SCSI DB[5]                                                  */
/* data[6]  :   SCSI DB[6]                                                  */
/* data[7]  :   SCSI DB[7]                                                  */
/* data[8]  :   SCSI DB[8]                                                  */
/* data[9]  :   SCSI DB[9]                                                  */
/* data[10] :   0 (Reserved)                                                */
/* data[11] :   0 (Reserved)                                                */
/* data[12] :   0 (Reserved)                                                */
/* data[13] :   Sequence ID 0                                               */
/* data[14] :   Sequence ID 1                                               */

#include <stdio.h>
unsigned short gen_poly = 0145U;
unsigned short degree_term = 0100U;
unsigned short table_lower[256];
unsigned short table_upper[128];

unsigned short encode( unsigned short data );
void making_tables( void );

void main( void ) {
   unsigned short data = 1U;

   making_tables();

   /* to exit the loop, enter 0.  Data of 0 produces result of 0 */

   while( data ) {
     /* read in an input data value */
     printf( "Enter an fifteen-bit data in <hex> format:\n" );
     scanf( "%hx",&data );
     data &= 0x7fffU; /** making sure that there are 15 bits in data **/
     printf( "data = <hex>%04x, parallel-encoded edc  = <hex>%02x\n",
             data, encode( data ));
   }
}

/* creates the lookup tables used to generate the check bits */
/* Look up table for low 8 bits of input data, 256 entries x 6 bits */
/* Look up table for high 7 bits of data, 128 entries x 6 bits */
void making_tables( void ) {
   unsigned int k;
   unsigned int power_of_two, one;

   /** making table for the lower order 8 bits of data ***/
   for( k = 0U; k < 256U; k++ )
      table_lower[k] = 0U;

   table_lower[1] = gen_poly ^ degree_term;
   power_of_two = 1U;

   for( k = 1U; k < 8U; k++ ) {
      table_lower[power_of_two << 1] = table_lower[power_of_two] << 1;
      power_of_two <<= 1;

      if(table_lower[power_of_two] & degree_term)
         table_lower[power_of_two] ^= gen_poly;
   }

   for( k = 0U; k < 256U; k++ ) {
      if(table_lower[k])
         continue;

      for( one = 128U; one; one >>=1 ) {
         if( one & k )
            table_lower[k] ^= table_lower[one];
      }
   }

   /* Can uncomment this loop to print out the table values */
   /* for( k = 0U; k < 256U; k++ ) {
         printf( "k = %03d, data = %04x\n", k, table_lower[k] );
      } */

   /** making table for the upper order 7 bits of data ****/
   for( k = 0U; k < 128U; k++ )
      table_upper[k] = 0U;

   table_upper[1] = table_lower[128] << 1;

   if( table_upper[1] & degree_term )
      table_upper[1] ^= gen_poly;

   power_of_two = 1U;

   for( k = 1U; k < 7U; k++ ) {
      table_upper[power_of_two << 1] = table_upper[power_of_two] << 1;
      power_of_two <<= 1;

      if( table_upper[power_of_two] & degree_term )
         table_upper[power_of_two] ^= gen_poly;
   }
   for( k = 0U; k < 128U; k++ ) {
      if( table_upper[k] )
         continue;

      for( one = 64U; one; one >>= 1 ) {
         if( one & k )
            table_upper[k] ^= table_upper[one];
      }
   }

   /* Can uncomment this loop to print out the table values */
   /* for( k = 0U; k < 128U; k++ ) {
         printf( "k = %03d, data = %04x\n", k, table_upper[k] );
      } */
}

/* actual encoding is just two lookups and then XOR the results */
/** data contains 15 bits **/
unsigned short encode( unsigned short data ) {
   unsigned short edc, lower_index, upper_index;

   lower_index = data & 0xffU;          /** lower_index contains 8 bits **/
   upper_index = (data >> 8) & 0x7fU;   /** only 7 bits in upper_index **/
   edc = table_lower[lower_index] ^ table_upper[upper_index];

   return(edc);
}
```

x5.2 Verilog example

The following is an example of a Verilog implementation that would generate the check bits for the protection scheme described in this annex.

```
// Verilog Implementation of Protection Code generator
//
//  Input:         cw       - 15-bit Code word
//  Returns:       check    - 6 Check bits
//
function[5:0]      check;
input[14:0]        cw;
begin
   check[0] = cw[00] ^ cw[01] ^ cw[02] ^ cw[03] ^ cw[05] ^ cw[06] ^ cw[07] ^ cw[10] ^ cw[11] ^ cw[13];
   check[1] = cw[01] ^ cw[02] ^ cw[03] ^ cw[04] ^ cw[06] ^ cw[07] ^ cw[08] ^ cw[11] ^ cw[12] ^ cw[14];
   check[2] = cw[00] ^ cw[01] ^ cw[04] ^ cw[06] ^ cw[08] ^ cw[09] ^ cw[10] ^ cw[11] ^ cw[12];
   check[3] = cw[01] ^ cw[02] ^ cw[05] ^ cw[07] ^ cw[09] ^ cw[10] ^ cw[11] ^ cw[12] ^ cw[13];
   check[4] = cw[02] ^ cw[03] ^ cw[06] ^ cw[08] ^ cw[10] ^ cw[11] ^ cw[12] ^ cw[13] ^ cw[14];
   check[5] = cw[00] ^ cw[01] ^ cw[02] ^ cw[04] ^ cw[05] ^ cw[06] ^ cw[09] ^ cw[10] ^ cw[12] ^ cw[14];
end
endfunction
```

x5.3 Protection code examples

x5.3.1 Example of a sequence of an IDENTIFY message with a SIMPLE task attribute message having a tag field of zero.

Information Byte Contents	Codeword bits (7:0) [DB(7:0)]	Codeword bits (9:8) [DB(9:8)]	Codeword bits (14:13) [Seq ID(1:0)]	Calculated redundant bits (5:0)	Output on DB(15:8)
IDENTIFY message	10000000	00	00	001011	00101100
SIMPLE message	00100000	00	01	110000	11000000
TAG = 0	00000000	00	10	110010	11001000

x5.3.2 Example of a sequence of a CDB for a READ(6) command with a Logical Block Address of 1A BC DEh, and a Transfer Length of 55h.

Information Byte Contents	Codeword bits (7:0) [DB(7:0)]	Codeword bits (9:8) [DB(9:8)]	Codeword bits (14:13) [Seq ID(1:0)]	Calculated redundant bits (5:0)	Output on DB(15:8)
OPERATION CODE 08h (READ(6))	00001000	00	00	010011	01001100
LBA(20:16)	00011010	00	01	000011	00001100
LBA(15:8)	10111100	00	10	011110	01111000
LBA(7:0)	11011110	00	11	110110	11011000
TRANSFER LENGTH	01010101	00	00	001111	00111100
CONTROL	00000000	00	01	011001	01100100

x5.3.3 Example of a “shifting ones” sequence.

The following is not an SCSI sequence, but demonstrates the calculation results and theoretical output for twelve different codewords each containing all zeroes except for a single one in a different bit in each word.

Information Byte Contents	Codeword bits (7:0) [DB(7:0)]	Codeword bits (9:8) [DB(9:8)]	Codeword bits (14:13) [Seq ID(1:0)]	Calculated redundant bits (5:0)	Theoretical Output on DB(15:8)
Not applicable for this example	00000001	00	00	100101	10010100
	00000010	00	00	101111	10111100
	00000100	00	00	111011	11101100
	00001000	00	00	010011	01001100
	00010000	00	00	100110	10011000
	00100000	00	00	101001	10100100
	01000000	00	00	110111	11011100
	10000000	00	00	001011	00101100
	00000000	01	00	010110	01011001
	00000000	10	00	101100	10110010
	00000000	00	01	011001	01100100
	00000000	00	10	110010	11001000

x5.3.4 Example of a “shifting zeroes” sequence.

The following is not an SCSI sequence, but demonstrates the calculation results and theoretical output for twelve different codewords each containing all ones except for a single zero in a different bit in each word.

Information Byte Contents	Codeword bits (7:0) [DB(7:0)]	Codeword bits (9:8) [DB(9:8)]	Codeword bits (14:13) [Seq ID(1:0)]	Calculated redundant bits (5:0)	Theoretical Output on DB(15:8)
Not applicable for this example	11111110	11	11	100101	10010111
	11111101	11	11	101111	10111111
	11111011	11	11	111011	11101111
	11110111	11	11	010011	01001111
	11101111	11	11	100110	10011011
	11011111	11	11	101001	10100111
	10111111	11	11	110111	11011111
	01111111	11	11	001011	00101111
	11111111	10	11	010110	01011010
	11111111	01	11	101100	10110001
	11111111	11	10	011001	01100111
	11111111	11	01	110110	11011011