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: **23 April 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 may become corrupted (e.g., by hot plugging events). The solution in this proposal resolves this issue.

The following takes the original proposal and puts it in the form of an informative annex based on the recommendation of the SPI-3 working group meeting the week of April 5th in Monterey.

Annex x

(Informative)

Protection for the Asynchronous Information Phases (COMMAND, MESSAGE, and STATUS)

This annex describes a protection method that provides improved error detection for asynchronous information phases (COMMAND, MESSAGE, and STATUS) by transferring redundant information on the upper eight data bits on the wide bus. During these phases asynchronous information is sent along the 8-bit (narrow) bus with parity as normal. This protection method is only available for wide SCSI devices.

x1 Protection Code

The following are the items 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 the information byte, it also latches the state of the other SCSI signals and values noted in the table.

Table x1

Codeword 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) should be included in the calculation as read from the bus, so they can be used in the future. These signals are reserved and should be negated. Note 2: For calculation purposes these signals are zero. However, these signals may be used for other functions in a future standard.		

The Sequence ID is a "virtual" signal that is carried in the encoding, but not actually sent on the bus. The Sequence ID increments during a "run". A "run" is a sequence of transfers during a MESSAGE IN, MESSAGE OUT, COMMAND, or STATUS phase. A new run begins with every phase change and every time that ATN is negated.

For each new run, the Sequence ID is set to zero for the first word transferred, one for the second word transferred, two for the third word transferred, and three for the fourth word transferred. The Sequence ID then cycles back 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 starts at zero again.

The Sequence ID provides protection for errors that occur when a data transfer is missed or double clocked. If a BCH code error is detected or if a sequence ID value is missing during a run, then the transfer is invalid. Recovery from these protection errors is the same as parity error recovery (see x4).

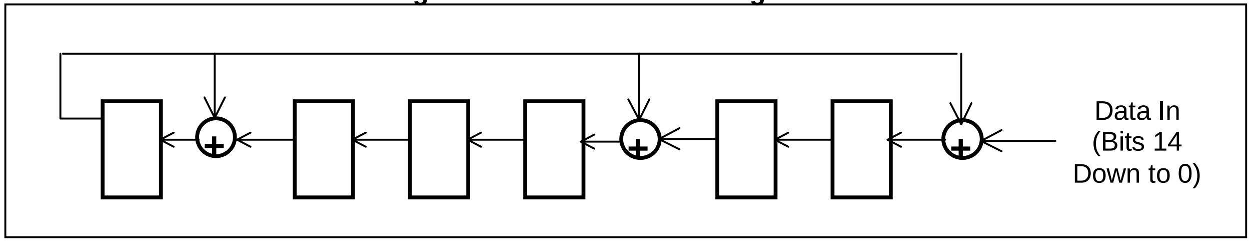
x1.2 Code Description

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 bit 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 in, 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 would be the expected implementation.

x1.3 Error Detection Properties

The 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 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 3 bits or fewer, all errors of an odd number of bits, and 98.4% of all possible errors.

x1.4 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 6 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 six bit results are then XOR'd together to create the final 6    */
/* 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 = 0145;
unsigned short degree_term = 0100;
unsigned short table_lower[256];
unsigned short table_upper[128];

int main()
{

  unsigned short data,edc_single,edc_parallel;
  unsigned short serial_encode();
  unsigned short parellel_encode();

   short making_tables();

   making_tables();

  while(1)
   {
     /* read in an input data value */
     printf("Enter an fifteen-bit data in <hex> format:\n");
     scanf("%hx",&data);
     data &= 0x7fff; /** making sure that there are 15 bits in data **/
     edc_parallel = parellel_encode(data);
     printf(" data = <hex>%04x, parallel-encoded edc  = <hex>%02x\n",data,
     edc_parallel);
     /* to exit the loop, enter 0.  Data of 0 produces result of 0 */
     if(data==0) break;
   }

}

/* 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 */
short making_tables()
{short k;
 short power_of_two,one;

 /** making table for the lower order 8 bits of data ***/
  for(k=0; k < 256; k++) table_lower[k] = 0;
  table_lower[1] = gen_poly ^ degree_term;

  power_of_two = 1;
   for(k=1; k < 8; 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 = 0; k < 256; k++)
      {if(table_lower[k]) continue;
       one = 128;
       while(one)
```

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

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

 /** making table for the upper order 7 bits of data ****/
   for(k=0; k < 128; k++) table_upper[k] = 0;
   table_upper[1] = table_lower[128] << 1;
   if(table_upper[1] & degree_term) table_upper[1] ^= gen_poly;
  power_of_two = 1;
   for(k=1; k < 7; 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 = 0; k < 128; k++)
      {if(table_upper[k]) continue;
       one = 64;
       while(one)
         {if(one & k) table_upper[k] ^= table_upper[one];
         one >>= 1;
         }
      }

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

 return(1);
 }

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

 lower_index = data & 0xff; /** lower_index contains 8 bits **/
 upper_index = (data >> 8) & 0x7f;   /** only 7 bits in upper_index **/
 edc = table_lower[lower_index] ^ table_upper[upper_index];
 return(edc);

}
```

x2 Enabling

1) All SCSI devices supporting the protection code generate and transmit the code during all COMMAND, MESSAGE, and STATUS phases.
2) If detection of the protection code is enabled, then the SCSI device checks the code when receiving all COMMAND, MESSAGE, and STATUS information.
3) If the protection code is detected during the first message or command received by a target after a power on or reset condition, then that target uses protection code detection on that I_T nexus for all subsequent I/O processes until a subsequent power on or reset condition occurs.
4) If the protection code is detected on the status and command completion message received by the initiator after a power on or reset condition occurs, the initiator uses protection code detection on that I_T nexus for subsequent I/O processes until a subsequent power on or reset condition occurs. (Note: An initiator could check for the protection code on each UNIT ATTENTION condition generated by a target in order to detect when a target not capable of generating protection code is "hot swap" replaced by a target that is capable of generating protection code).
5) If an SCSI device receives two consecutive bytes of COMMAND, MESSAGE, or STATUS information with no parity error across (DB(0) – DB(7), P(0)) but having a protection code error or a parity error on (DB(8) – DB(15), P(1)), the device should disable protection code checking for that I_T nexus. Detection of protection code should not be re-enabled for the device until a subsequent power on or reset condition occurs (or, if implemented by the initiator, a subsequent UNIT ATTENTION condition occurs).

x3 Parity

During a protected transfer of COMMAND, MESSAGE, or STATUS information P(0) contains parity for the low order byte (DB(0) through DB(7)) as normal. P(1) contains parity for the high order byte of the transfer (DB(8) through DB(15)).

x4 Error handling

Protection code errors shall be 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.