Date: June 20, 02
To: T10 Technical Committee
From: Jim Coomes (jim.comes@seagate.com)
Subject: Bit order for scrambling and CRC

Below is a C program to generate CRC and scrambled output for a sample SAS COMMAND frame. The content of the COMMAND IU and the resulting output are provided. A revision of this document will break the program into separate codes for the CRC and scrambler as they will be proposed for inclusion into the SAS draft.

After the program, equations for generating each bit of the CRC and scrambled output are listed.

```
#include <stdio.h>

unsigned long calculate_crc( unsigned long *frame, int length); /* returns crc
value */
unsigned long scrambler(boolean reset, unsigned long dword); /* returns
scrambled dword */

void main(void)
{  static unsigned long READ_IU[] = {0x06D0B992L,0x00B5DF59L,0x00000000L,  /*
header */
                                     0x00000000L,0x1234FFFFL,0x00000000L,
                                     0x00000000L,0x00000000L,0x00000000L,  /* IU
*/
                                     0x08000012L,0x01000000L,0x00000000L,
                                     0x00000000L};
/*  INFORMATION UNIT TYPE                                       06h
    HASHED DESTINATION DEVICE NAME for 500107534F0CFC88h        D0B992h
    HASHED SOURCE DEVICE NAME for 50010B2B3CBF639h              B5DF59h
    TIMEOUT                                                     0
    NUMBER OF FILL BYTES                                        0
    COMMAND ID                                                  0
    TAG                                                         1234h
    TARGET PORT TRANSFER TAG                                    FFFFh
    INFORMATION UNIT (CMD)
       LUN                                                      0
       TASK ATTRIBRUTE                                          0
       ADDITIONAL CDB LENGTH                                    0
       CDB (6 byte read block 12h)                              080000120100h
    CRC                                                         3F4F1C26h
*/
   unsigned long crc;
   FILE *out;
   union {unsigned long lword; unsigned char byte[4];} big_endian;

    /* EXAMPLE CRC and SCRAMBLE generation */
   crc = calculate_crc(READ_IU,13);

   out =fopen("Crc_rslt.txt", "w");
   fprintf(out, "SOF (4 bytes) K28.5 0x18 0xE4 0x67\r\n");
   big_endian.lword = scrambler(true, READ_IU[0]);
   fprintf(out, "0x%08X --> 0x%02X 0x%02X 0x%02X 0x%02X\r\n", READ_IU[0],
big_endian.byte[3], big_endian.byte[2], big_endian.byte[1], big_endian.byte[0]);
   for (int i=1; i<13; i++)
```

```
   {   big_endian.lword = scrambler(false, READ_IU[i]);
       fprintf(out, "0x%08X --> 0x%02X 0x%02X 0x%02X 0x%02X\r\n", READ_IU[i],
big_endian.byte[3], big_endian.byte[2], big_endian.byte[1], big_endian.byte[0]);
   }
   big_endian.lword = scrambler(false, crc);
   fprintf(out, "0x%08X --> 0x%02X 0x%02X 0x%02X 0x%02X\r\n", crc,
big_endian.byte[3], big_endian.byte[2], big_endian.byte[1], big_endian.byte[0]);
   fprintf(out, "EOF (4 bytes) K28.5 0x18 0xF0 0x9B\r\n");
   fclose(out);
}
//-----------------------------------------------------------------------------


unsigned long calculate_crc( unsigned long *frame, int length) /* returns crc
value */
{   long poly = 0x04C11DB7L;
    unsigned long crc_gen, x;
    union {unsigned long lword; unsigned char byte[4];} b_access;
    static unsigned char xpose[] = {0x0,0x8,0x4,0xC,0x2,0xA,0x6,0xE,
                                    0x1,0x9,0x5,0xD,0x3,0xB,0x7,0xF};
    int i, j, fb;

    crc_gen = ~0;   /* seed generator with all ones */
    for (i=0; i<length; i++)
    {   x = *frame++; /* get word */
        b_access.lword = x; /* transpose bits in byte */
        for (j=0; j<4; j++)
        {   b_access.byte[j] = xpose[b_access.byte[j]>>4] |
xpose[b_access.byte[j]&0xF]<<4;
        }
        x = b_access.lword;

        for (j=0; j<32; j++) /* serial shift register implementation */
        {   fb = ((x & 0x80000000L)>0) ^ ((crc_gen & 0x80000000L)>0);
            x <<= 1;
            crc_gen <<= 1;
            if (fb)
                crc_gen ^= poly;
        }
    }

    b_access.lword = crc_gen; /* transpose bits in CRC */
    for (j=0; j<4; j++)
    {   b_access.byte[j] = xpose[b_access.byte[j]>>4] |
xpose[b_access.byte[j]&0xF]<<4;
    }
    crc_gen = b_access.lword;

    return ~crc_gen; /* invert output */
}

/* This sample code generates the entire sequence of Dwords produced by
   the scrambler defined in the SAS specification. The specification calls for
   an LFSR to generate a string of bits that will be packaged into 32 bit
   Dwords to be XORed with the data Dwords. The generator polynomial specified
   is:  G(x) = x^16 + x^15 + x^13 + x^4 + 1
```

```
   Parallelized versions of the scrambler are initialized to a value derived
   from the initialization value of 0xFFFF defined in the specification. This
   implementation is initialized to 0xF0F6. Other parallel implementations
   will have different initial values. The important point is that the first
   Dword output of any implementation must equal 0xC2D2768D followed by
   0x1F26B368 */

void scrambler_mult(unsigned char *now, unsigned char *next)
/* Notice that there are lots of shared terms in these assignments. */
/* The following 16 assignments implement the matrix multiplication
   G(x) = x^16 + x^15 + x^13 + x^4 + 1 */
{   next[15] = now[15] ^ now[14] ^ now[12] ^ now[10] ^ now[6] ^ now[3] ^ now[0];
    next[14] = now[15] ^ now[13] ^ now[12] ^ now[11] ^ now[9] ^ now[5] ^ now[3]
^ now[2];
    next[13] = now[14] ^ now[12] ^ now[11] ^ now[10] ^ now[8] ^ now[4] ^ now[2]
^ now[1];
    next[12] = now[13] ^ now[11] ^ now[10] ^ now[9] ^ now[7] ^ now[3] ^ now[1] ^
now[0];
    next[11] = now[15] ^ now[14] ^ now[10] ^ now[9] ^ now[8] ^ now[6] ^ now[3] ^
now[2] ^ now[0];
    next[10] = now[15] ^ now[13] ^ now[12] ^ now[9] ^ now[8] ^ now[7] ^ now[5] ^
now[3] ^ now[2] ^ now[1];
    next[9] = now[14] ^ now[12] ^ now[11] ^ now[8] ^ now[7] ^ now[6] ^ now[4] ^
now[2] ^ now[1] ^ now[0];
    next[8] = now[15] ^ now[14] ^ now[13] ^ now[12] ^ now[11] ^ now[10] ^ now[7]
^ now[6] ^ now[5] ^ now[1] ^ now[0];
    next[7] = now[15] ^ now[13] ^ now[11] ^ now[10] ^ now[9] ^ now[6] ^ now[5] ^
now[4] ^ now[3] ^ now[0];
    next[6] = now[15] ^ now[10] ^ now[9] ^ now[8] ^ now[5] ^ now[4] ^ now[2];
    next[5] = now[14] ^ now[9] ^ now[8] ^ now[7] ^ now[4] ^ now[3] ^ now[1];
    next[4] = now[13] ^ now[8] ^ now[7] ^ now[6] ^ now[3] ^ now[2] ^ now[0];
    next[3] = now[15] ^ now[14] ^ now[7] ^ now[6] ^ now[5] ^ now[3] ^ now[2] ^
now[1];
    next[2] = now[14] ^ now[13] ^ now[6] ^ now[5] ^ now[4] ^ now[2] ^ now[1] ^
now[0];
    next[1] = now[15] ^ now[14] ^ now[13] ^ now[5] ^ now[4] ^ now[1] ^ now[0];
    next[0] = now[15] ^ now[13] ^ now[4] ^ now[0];
}
unsigned long scrambler(boolean reset, unsigned long dword)
{   int j;
    static unsigned short context; /* The 16 bit register that holds the context
or state */
    unsigned long scrambler; /* The 32 bit output of the circuit */
    unsigned char reg[16]; /* The individual bits of context */
    unsigned char next[32]; /* The computed bits of scrambler */

/* Shall we startup the scrambler with the first word? */
    if (reset) context = 0xF0F6;
/* Split the register contents (the variable context) up into its individual
   bits for easy handling. */
    for (j = 0; j < 16; ++j)
    {   reg[j] = (context >> j) & 0x01;
    }
    scrambler_mult(reg,next);       /* the first 16 bits */
    scrambler_mult(next,&next[16]); /* the second 16 bits */
/* The 32 bits of the output have been generated in the "next" array. */
/* Reassemble the bits into a 32 bit Dword. */
```

```
    for (scrambler = 0,j = 31; j >= 0; --j)
    {   scrambler = scrambler << 1;
        scrambler |= next[j];
    }
/* The upper half of the scrambler output is stored backed into the register
   as the saved context for the next cycle. */
    context = scrambler >> 16;
    return dword ^ scrambler;
}
```

```
File output:

SOF (4 bytes) K28.5 0x18 0xE4 0x67
0x06D0B992 --> 0xC4 0x02 0xCF 0x1F
0x00B5DF59 --> 0x1F 0x93 0x6C 0x31
0x00000000 --> 0xA5 0x08 0x43 0x6C
0x00000000 --> 0x34 0x52 0xD3 0x54
0x1234FFFF --> 0x98 0x61 0x6A 0xFD
0x00000000 --> 0xBB 0x1A 0xBE 0x1B
0x00000000 --> 0xFA 0x56 0xB7 0x3D
0x00000000 --> 0x53 0xF6 0x0B 0x1B
0x00000000 --> 0xF0 0x80 0x9C 0x41
0x08000012 --> 0x7C 0x7F 0xC3 0x58
0x01000000 --> 0xBF 0x86 0x52 0x91
0x00000000 --> 0x7A 0x6F 0xA7 0xB6
0x00000000 --> 0x31 0x63 0xE6 0xD6
0x3F4F1C26 --> 0xCF 0x79 0xE2 0x2A
EOF (4 bytes) K28.5 0x18 0xF0 0x9B
```

```
 d: IN STD_LOGIC_VECTOR(31 DOWNTO 0);
 crc  : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
```

These equations generate the 32 bit Cyclic Redundancy Check for dword transmission. The ^ symbol represents an XOR operation.

```
crc00 <= d00^d06^d09^d10^d12^d16^d24^d25^d26^d28^d29^d30^d31;
crc01 <= d00^d01^d06^d07^d09^d11^d12^d13^d16^d17^d24^d27^d28;
crc02 <= d00^d01^d02^d06^d07^d08^d09^d13^d14^d16^d17^d18^d24^d26^d30^d31;
crc03 <= d01^d02^d03^d07^d08^d09^d10^d14^d15^d17^d18^d19^d25^d27^d31;
crc04 <= d00^d02^d03^d04^d06^d08^d11^d12^d15^d18^d19^d20^d24^d25^d29^d30^d31;
crc05 <= d00^d01^d03^d04^d05^d06^d07^d10^d13^d19^d20^d21^d24^d28^d29;
crc06 <= d01^d02^d04^d05^d06^d07^d08^d11^d14^d20^d21^d22^d25^d29^d30;
crc07 <= d00^d02^d03^d05^d07^d08^d10^d15^d16^d21^d22^d23^d24^d25^d28^d29;
crc08 <= d00^d01^d03^d04^d08^d10^d11^d12^d17^d22^d23^d28^d31;
crc09 <= d01^d02^d04^d05^d09^d11^d12^d13^d18^d23^d24^d29;
crc10 <= d00^d02^d03^d05^d09^d13^d14^d16^d19^d26^d28^d29^d31;
crc11 <= d00^d01^d03^d04^d09^d12^d14^d15^d16^d17^d20^d24^d25^d26^d27^d28^d31;
crc12 <= d00^d01^d02^d04^d05^d06^d09^d12^d13^d15^d17^d18^d21^d24^d27^d30^d31;
crc13 <= d01^d02^d03^d05^d06^d07^d10^d13^d14^d16^d18^d19^d22^d25^d28^d31;
crc14 <= d02^d03^d04^d06^d07^d08^d11^d14^d15^d17^d19^d20^d23^d26^d29;
crc15 <= d03^d04^d05^d07^d08^d09^d12^d15^d16^d18^d20^d21^d24^d27^d30;
crc16 <= d00^d04^d05^d08^d12^d13^d17^d19^d21^d22^d24^d26^d29^d30;
```

```
crc17 <= d01^d05^d06^d09^d13^d14^d18^d20^d22^d23^d25^d27^d30^d31;
crc18 <= d02^d06^d07^d10^d14^d15^d19^d21^d23^d24^d26^d28^d31;
crc19 <= d03^d07^d08^d11^d15^d16^d20^d22^d24^d25^d27^d29;
crc20 <= d04^d08^d09^d12^d16^d17^d21^d23^d25^d26^d28^d30;
crc21 <= d05^d09^d10^d13^d17^d18^d22^d24^d26^d27^d29^d31;
crc22 <= d00^d09^d11^d12^d14^d16^d18^d19^d23^d24^d26^d27^d29^d31;
crc23 <= d00^d01^d06^d09^d13^d15^d16^d17^d19^d20^d26^d27^d29^d31;
crc24 <= d01^d02^d07^d10^d14^d16^d17^d18^d20^d21^d27^d28^d30;
crc25 <= d02^d03^d08^d11^d15^d17^d18^d19^d21^d22^d28^d29^d31;
crc26 <= d00^d03^d04^d06^d10^d18^d19^d20^d22^d23^d24^d25^d26^d28^d31;
crc27 <= d01^d04^d05^d07^d11^d19^d20^d21^d23^d24^d25^d26^d27^d29;
crc28 <= d02^d05^d06^d08^d12^d20^d21^d22^d24^d25^d26^d27^d28^d30;
crc29 <= d03^d06^d07^d09^d13^d21^d22^d23^d25^d26^d27^d28^d29^d31;
crc30 <= d04^d07^d08^d10^d14^d22^d23^d24^d26^d27^d28^d29^d30;
crc31 <= d05^d08^d09^d11^d15^d23^d24^d25^d27^d28^d29^d30^d31;
```

These equations generate the 16 bit scrambling bytes for transmission. The ^ symbol represents an XOR operation.

```
scr15 <= d15^d14^d12^d10^d06^d03^d00;
scr14 <= d15^d13^d12^d11^d09^d05^d03^d02;
scr13 <= d14^d12^d11^d10^d08^d04^d02^d01;
scr12 <= d13^d11^d10^d09^d07^d03^d01^d00;
scr11 <= d15^d14^d10^d09^d08^d06^d03^d02^d00;
scr10 <= d15^d13^d12^d09^d08^d07^d05^d03^d02^d01;
scr09 <= d14^d12^d11^d08^d07^d06^d04^d02^d01^d00;
scr08 <= d15^d14^d13^d12^d11^d10^d07^d06^d05^d01^d00;
scr07 <= d15^d13^d11^d10^d09^d06^d05^d04^d03^d00;
scr06 <= d15^d10^d09^d08^d05^d04^d02;
scr05 <= d14^d09^d08^d07^d04^d03^d01;
scr04 <= d13^d08^d07^d06^d03^d02^d00;
scr03 <= d15^d14^d07^d06^d05^d03^d02^d01;
scr02 <= d14^d13^d06^d05^d04^d02^d01^d00;
scr01 <= d15^d14^d13^d05^d04^d01^d00;
scr00 <= d15^d13^d04^d00;
```