

Practical Application of the Hamming Code

Jon Fick
Westford MicroSystems
Westford, Vermont, USA 05494

April 03, 2003

© Westford MicroSystems 2003

The Hamming code, one type of “forward error correction,” provides error detection and correction for single-bit errors such as what might come from EPROM. (It fails to provide benefit for burst errors, in which there may be two or more bits in a row that are lost, such as those found over a wireless link.)

In the Hamming code:

1. If a *single-bit error* occurs and the generated syndrome value is non-zero and not an uncorrectable syndrome, and the syndrome is then used as a pointer into the coset-leader table to get the correction vector.
2. If a *double-bit error* occurs, all that can be guaranteed is that the syndrome will not be zero; there is no way to correct the such errors.
3. If *more than two bits are in error*, all bets are off for even detecting them! Only a checksum byte transmitted with the multi-character message could hope to validate a fixed-length or known-length message and flag an error.

An analogy that helps to explain the theory of the Hamming code is that it is like the phonetic alphabet used by airline pilots (“alpha”, “bravo”, “charlie”, “echo”, “foxtrot”, etc) in which none of the words sound like each other. Even if part of a word is garbled during transmission or reception, there’s only a small chance of mistaking it for a different word.

Code example of the Detection and Correction process

The example code below is an experiment with a (12:8) Hamming code. It generates an eight-bit data byte and concatenates four parity bits to form a 12-bit “codeword”, after which a single-bit error is induced anywhere in the 12 bits. It then decodes the codeword and corrects the data byte. The whole process is displayed on a 9600-8-N-1 terminal screen.

```
/******  
HAMMING1.C  
  
External XTAL oscillator = 4MHz  
Cycle time = 1uS  
  
In the CCS PCB compiler:   CHAR and INT are 8-bit values  
                           LONG is a 16-bit value  
*****/  
  
#include <16c57.h>  
#list  
/*  
Set configuration bits in the PIC processor.  Or use #FUSES statement.  
*/  
#ROM 0x800 = { 0x0F09 }          /* XTAL oscillator, no internal watchdog, no code protect */  
  
/* DEFINES */  
#define   UCHAR          char  
#define   UINT           long  
#define   ERRORS        0x0001  
  
/* PRE-PROCESSOR STUFF */  
#use delay ( clock=4000000 )     /* sets appropriate compiler constants */  
#use fast_io ( A )              /* Fast I/O doesn't handle TRIS */  
#use rs232 ( BAUD=9600, XMIT=PIN_A1, PARITY=N, BITS=8 )  
  
/* VARIABLES */  
  
UCHAR const g [ 4 ] = { 0xE2, 0x99, 0x57, 0x2F };          /* parity generation matrix */  
UINT const h [ 4 ] = { 0x0E28, 0x0994, 0x0572, 0x02F1 }; /* parity check matrix */  
/*  
Each entry in the coset leader table below points to one bit that must be inverted in the incorrect  
data.  If no bits need correcting, then zero is returned.  
*/  
UINT const cl [ 13 ] = { 0x000, 0x001, 0x002, 0x040, 0x004, 0x080,  
                        0x100, 0x010, 0x008, 0x200, 0x400, 0x020, 0x800 };  
static UCHAR value;      /* value to encode */  
static UCHAR parity;     /* Hamming bits */  
static UINT tempval;  
static UCHAR i;          /* counts parity calculations */  
static UCHAR j;          /* counts bits */  
static UCHAR syndrome;  
static UINT databyte;   /* combined byte */  
static UINT errorbits;  /* creates bit errors */  
  
void main( void )  
{  
    set_tris_a ( 0b11111101 ); /* RA1 is serial output */  
    printf ( "%c", 0x1A );     /* clear screen */  
    errorbits = ERRORS;       /* preload error bits */  
  
    for ( value = 0; value < 255; value++ )  
    {  
  
        /* Encoding for a (12,8) code */  
        /* Accept an 8-bit byte, return the byte with a 4-bit parity code */  
        parity = 0;  
        for ( i = 0; i < 4; i++ ) /* do for four parity bits */  
        {  
            parity <<= 1; /* Make room for new parity bit */  
            tempval = value & g [ i ]; /* Multiply mod 2 */  
            for ( j = 0; j < 8; j++ ) /* Compute parity on 8-bit data byte */  
            {  
                if ( tempval & 0x01 ) /* is bit = 1 ? */  
                {  
                    parity ^= 0x01; /* hash into parity bit */  
                }  
            }  
        }  
    }  
}
```

```

        }
        tempval /= 2;          /* shift right one */
    }
}

/* Combine into a 12-bit word, introduce single-bit errors */
databyte = value;
databyte *= 16;             /* shift over four bits */
databyte |= parity;        /* combine data and parity */

databyte ^= errorbits;     /* introduce moving single-bit error */
errorbits <<= 1;          /* move error to another bit */
if ( ( errorbits & 0b0001000000000000 ) != 0 )
{
    errorbits = ERRORS;    /* restart error bits */
}

printf ( "\n\rOrig data,parity = %2X,%1X  but sent = %4LX", value, parity, databyte );

/* Decoding for a (12,8) code */
/*
Enter with 8-bit value and 4-bit parity in 12-bit "databyte".  Essentially compute "parity" on
the whole word, if parity is non-zero there is an error that must be corrected unless not
possible.
*/
syndrome = 0;
for ( i = 0; i < 4; i++ )    /* do for four parity bits */
{
    syndrome <<= 1;          /* Make room for new syndrome bit */
    tempval = databyte & h [ i ]; /* Multiply mod 2 */
    for ( j = 0; j < 12; j++ ) /* Compute parity on whole 12-bit word */
    {
        if ( tempval & 0x01 ) /* is bit = 1 ? */
        {
            syndrome ^= 0x01; /* hash into parity bit */
        }
        tempval /= 2;        /* shift right one */
    }
}
printf ( "  Syn = %1X", syndrome );

/* Correct the error */
if ( syndrome <= 12 ) /* syndrome values are 0 through 12 */
{
    databyte ^= cl [ syndrome ]; /* invert the erring bit from the coset leader table */
    i = databyte / 16; /* shift right four bits to drop parity bits */
    printf ( "  Corrected data = %2X", i );
}
else
{
    printf ( "  Non-correctable error");
}
if ( value == 0xFE )
{
    value = 0;
}
}
while ( true );
}

```

In theory the multi-bit parity value is concatenated to the data and transmitted as one word, such as a 12-bit word. However, standard UART's aren't equipped such word lengths. Thus, a practical application of a (12:8) Hamming code that uses a standard 8-bit UART scheme is one in which a fixed-length or known-length message is transmitted, each 8-bit character followed by a 8-bit parity byte (only four bits of which are used.) At least one problem with this scheme lies in ensuring that the receiver, after somehow getting out of sync, is able to determine whether the 8-bit byte just received is *data* or *parity* and get back into sync. *The solution is left to the reader* (I've always wanted to say that!)

A transmission code module

The following code is a transmission module that transmits one character at 9600-8-N-1, waits 5mS for decoding and correction time in the receiving module, then transmits the parity byte and waits another 5mS. This continues for the entire message. When the fixed-length message is done, a checksum byte, followed by it's parity byte, is transmitted to validate the message at the receiver.

```

/*****
TEST2_TX.C

External XTAL oscillator = 4MHz
Cycle time = 1uS

In the CCS PCB compiler:  CHAR and INT are 8-bit values
                          LONG is a 16-bit value

Jon Fick  07/15/98

*****/

#include <16c57.h>
#include <list>
/*
Set configuration bits in the PIC processor.  Or use #FUSES statement.
*/
#ROM 0x800 = { 0x0F09 }          /* XTAL oscillator, no internal watchdog, no code protect */

/* DEFINES */
#define UCHAR          char
#define UINT           long

/* PRE-PROCESSOR STUFF */
#define delay ( clock=4000000 )  /* sets appropriate compiler constants */
#define fast_io ( A )          /* Fast I/O doesn't handle TRIS */
#define rs232 ( BAUD=9600, XMIT=PIN_A0, PARITY=N, BITS=8 )

/* VARIABLES */
UCHAR const g [ 4 ] = { 0xE2, 0x99, 0x57, 0x2F }; /* parity generation matrix */
static UCHAR cCount;
static UCHAR cChecksum;

/* PROTOTYPES */
void WriteCodeWord ( UCHAR cData );

void main( void )
{
    set_tris_a ( 0b11111110 ); /* RA0 is serial output */
    cCount = 0;
    cChecksum = 0;
    while ( true )
    {
        if ( cCount == 0 )          /* ensure runs from 100-255 */
        {
            cCount = 100;
        }
        printf ( WriteCodeWord, "\n\rCount = %u", cCount++ );
        printf ( WriteCodeWord, "%c", cChecksum );
        cChecksum = 0; /* re-initialize */
        delay_ms ( 993 ); /* total pass time ~ 1 second */
    }
}

void WriteCodeWord ( UCHAR cData )
{
    UCHAR cI, cJ, cTmp, cParity;

    /* Hash data byte into checksum */
    cChecksum += cData;

    /* Send character byte */
    putchar ( cData );
    delay_ms ( 5 ); /* allow processing time at the other end */
}

```

```

/* Generate a 4-bit (12:4) Hamming parity code for cX */
cParity = 0;
for ( cI = 0; cI < 4; cI++ )      /* do for four parity bits */
{
    cParity <<= 1;                /* Make room for new parity bit */
    cTmp = cData & g [ cI ];      /* Multiply mod 2 */
    for ( cJ = 0; cJ < 8; cJ++ ) /* Compute parity on 8-bit data byte */
    {
        if ( cTmp & 0x01 )       /* is bit = 1 ? */
        {
            cParity ^= 0x01;     /* hash into parity bit */
        }
        cTmp /= 2;              /* shift right one */
    }
}

/* Send parity byte */
putchar ( cParity );
delay_ms ( 5 );                /* allow processing time at the other end */
}

```

A receiving code module

The receiving module attempts to sync up to characters that are possibly garbled. The first character of the message from the transmission module is a LF. which resets the message count and the checksum. The remainder of the characters and parity are received, processed, and displayed, and validated by the checksum.

```

/*****
TEST2_RX.C

The problems encountered with this scenario are that it uses the UNCORRECTED
0x0A to key off and reset the byte count and the checksum. If the 0x0A is
corrupted, it doesn't find it and has resyncing problems.

External XTAL oscillator = 4 MHz
Cycle time = 1uS

In the CCS PCB compiler:   CHAR and INT are 8-bit values
                           LONG is a 16-bit value

Jon Fick  07/15/98

*****/

#include <16c57.h>
#include <list>
/*
Set configuration bits in the PIC processor.  Or use #FUSES statement.
*/
#ROM 0x800 = { 0x0F09 }          /* XTAL oscillator, no internal watchdog, no code protect */

/* DEFINES */
#define UCHAR      char
#define UINT       long
#define OFF        0
#define ON         1

/* MAP HARDWARE TO VARIABLES */
#define PORT_A     5
#define PORT_B     6
#define PORT_C     7
#define LED_RED    PORT_B.0
#define LED_GRN    PORT_B.1
#define SERIAL_IN  PORT_A.0
#define SERIAL_OUT PORT_A.1

/* PRE-PROCESSOR STUFF */
#define delay(clock) /* sets appropriate compiler constants */
#define fast_io(A)   /* Fast I/O doesn't handle TRIS */
#define fast_io(B)
#define rs232(BAUD, XMIT, RCV, PARITY, BITS) /* BAUD = 9600, XMIT = PIN_A1, RCV = PIN_A0, PARITY = N, BITS = 8 */

/* VARIABLES */
#define h [ 4 ] = { 0x0E28, 0x0994, 0x0572, 0x02F1 }; /* parity check matrix */
#define c1 [ 13 ] = { 0x000, 0x001, 0x002, 0x040, 0x004, 0x080,
                    0x100, 0x010, 0x008, 0x200, 0x400, 0x020, 0x800 }; /* coset leader table */
static UCHAR cChecksum;
static UCHAR cCount;

void main( void )
{
    UCHAR cData, cParity, cI, cJ, cSyndrome;
    UINT iCodeWord, iTmp, iSyndrome;

    set_tris_a ( 0b11111101 ); /* RA1 is output */
    printf ( "%c\n\rReceive test program #1\n\r", 0x1a );

    while ( TRUE )          /* simply repeat input to output */
    {
        /* Receive first character */
        cData = getchar();

        /* Receive parity byte */

```

```

do
{
    cParity = getchar();
    } while ( ( cParity & 0xF0 ) != 0 );           /* to get back in sync */

/* combine into a 12-bit codeword */
iCodeWord = cData;
iCodeWord *= 16;           /* shift left four bits */
iCodeWord |= cParity;     /* combine data and parity */

/* Check character for error */
/* Generate a 4-bit (12:4) Hamming parity code for received cData */

cSyndrome = 0;
for ( cI = 0; cI < 4; cI++ )           /* do for four parity bits */
{
    cSyndrome <<= 1;           /* Make room for new syndrome bit */
    iTmp = iCodeWord & h [ cI ]; /* Multiply mod 2 */
    for ( cJ = 0; cJ < 12; cJ++ )       /* Compute parity on whole 8-bit byte */
    {
        if ( iTmp & 0x01 )           /* is bit = 1 ? */
        {
            cSyndrome ^= 0x01;       /* hash into parity bit */
        }
        iTmp /= 2;           /* shift right one */
    }
}

/* Correct any error if possible */
if ( cSyndrome <= 12 )           /* syndrome values are 0 through 12 */
{
    iCodeWord ^= c1 [ cSyndrome ]; /* invert the erring bit from the coset leader table */
    cData = iCodeWord / 16;       /* shift right four bits to drop parity bits */
    if ( cSyndrome != 0 )
    {
        putchar ( '-' );           /* prepend indicator that correction occurred */
    }
}
else
{
    if ( ( cData & 0xF0 ) != 0x00 )
    {
        cData = 'X';
    }
}

/* checksum */
if ( cData == 0x0A )
{
    cChecksum = 0;
    cCount = 0;
}
cChecksum += cData;           /* hash into checksum */
cCount++;

/* Finish up after checksum */
if ( cCount == 14 )
{
    if ( cChecksum != 0 )
    {
        printf ( " ?" );           /* not much time here to print a long message */
    }
}
else
{
    /* Send correct character to terminal */
    putchar ( cData );
}
}
}

```