AN1221

Hamming Error Control Coding Techniques with the HC08 MCU

by Mark McQuilken & Mark Glenewinkel
CSIC Applications

INTRODUCTION

This application note is intended to demonstrate the use of error control coding (ECC) in a digital transmission system. The HC08 MCU will be used to illustrate the code development of this process. A message frame consisting of a 4-bit data field with three parity bits will be encoded to allow the original four bits to be recovered, even if any single bit is corrupted during the transmission and reception processes. This process is based upon a class of linear error-correcting codes called Hamming codes. The process of using time diversity is also discussed as a way to control burst errors in a transmission system.

BACKGROUND

MODEL FOR A DATA TRANSMISSION SYSTEM

Generally, the processing of digital data (in a whole system and/or within a system) can be modelled as a generalized data transmission or storage system. Such a system consists of an information source, the "channel" (medium or storage), and the destination device. Some possible specific implementations that fit this generalized model are shown in Table 1 below:

Table 1. Digital Data Transmission System Implementations

Example System	Information Source	Channel	Destination
Personal computer	CPU	RAM or hard disk	CPU or serial interface
Digital telephone	Your digitized voice	Telephone company's wires and switching system	Whoever you're calling (another telephone)
Stereo system	Compact disc	Laser optics and electronics	Audio output (ultimately to transducers like speakers)
Avionic weapon development system	Cockpit command to release weapon stores	Wiring/electronics between cockpit and weapons	Electro-mechanical assembly which activates and deploys weapons

There are many other systems that also fit this simplistic model of a data transmission system.

Problems with Data Transmission Systems

In a perfect world, data would be transmitted and received completely intact. In the real world, we must often combat the effects of errors induced in our transmitted/received information. For example, in the above-mentioned avionics system, if a command from the cockpit to release the weapons was transmitted and corrupted so that the deployment electronics interpreted the message as "hold" rather

©MOTOROLA, INC., 1993

than “release,” the intended target may never be hit. A worse case would be the pilot wanting to abort deployment and the “abort” command was corrupted by noise to become “release” at the deployment assembly. A system must be built in a way to avoid disastrous effects of data corruption. ECC is the field of study that deals with methods of coding information to reduce the effects of errors.

One of the general challenges facing the system designer is to implement ECCs without degrading the data transfer rates too significantly. For example, one of the simplest ways to hedge against channel-induced errors, is to specify a transmission protocol that requires multiple transmissions of the same data. Such a protocol will reduce the effective rate of transmission by a factor equal to the number of retransmissions per information block. Even if this retransmission strategy guaranteed that at least one of the data blocks was correct at the receiver, there are two problems:

1) How will we know which of the data blocks is the correct one?
2) How do we make up for the channel bandwidth lost by transmitting the same data multiple times?

ECCs can actually be more effective in using the channel's existing bandwidth than the above scenario. Instead of merely retransmitting data, ECCs take up some of the channel bandwidth (but less than retransmissions) to send some data redundancies that may be used for the detection and, sometimes, correction of corrupted received data.

Types of Noise

There are many types of ECCs. Each ECC has its own set of strengths and weaknesses. One important aspect of all the codes is their ability or inability to deal with each of the error types: random channel errors/noise, burst-type errors/noise, or a combination of each. The codes discussed in this presentation (Hamming codes) are a class of algebraic codes that deal effectively with random channel noise. Burst errors are typically longer in duration than random errors and thus require more robust code types to prevent unrecoverable data. This application note also describes time-diversity coding used to prevent burst-type noise. The technique of coupling time-diversity ECCs with random error ECCs can help improve the performance of the ECC in the presence of burst errors.

One-way Data Transmission

In a system where only one-way communication is possible and precautions must be taken against data corruption, error control codes must be employed by using forward error correction (FEC). FEC is accomplished by employing codes that allow for “automatic” correction of specific types of errors induced by noise in the channel and received by the receiver. Hamming codes are one of the simplest classes of FEC codes and are characterized by the following traits:

— The number of parity-check symbols (bits) must be greater than or equal to 3. Let these symbols be represented by the variable m.

— The number, k, of information symbols (bits) is:

$$k = 2^m - m - 1$$

— The total length, n, of the code is:

$$n = 2^m - 1$$

— Error correcting capability is exactly one symbol.

— The error detecting capability is all error patterns of two errors or less.

— The parity check matrix is:

$$\mathbf{H} = [\ \mathbf{I}_m\ \ \mathbf{Q}_{m,k}\ ]$$

where $\mathbf{I}_m$ is an m x m identity matrix and $\mathbf{Q}_{m,k}$ is the submatrix that consists of k columns which are of weight two or more (i.e., have two or more ones in them).

—The generator matrix is:

$$\mathbf{G} = [\ \mathbf{Q}^T_{m,k}\ \mathbf{I}_k\ ]$$

where $\mathbf{I}_k$ is a k x k identity matrix and $\mathbf{Q}^T_{m,k}$ is the transpose of the submatrix **Q** that consists of m columns and k rows.

Two-way Data Transmission

Error control for a two-way system can be accomplished with both error control coding as well as some rational scheme of data retransmission. A system that utilizes retransmission of messages is said to use an automatic repeat request (ARQ) protocol.

HAMMING ENCODING

HAMENC1 — HAMMING ENCODER #1, TABLE LOOK-UP

The 4-bit information word to be encoded is used as an index into a look-up table. A (7,4) Hamming code represents a 7-bit word with four data bits and three code bits. A (7,4) Hamming code will have 2^4 (16) different codeword possibilities. The 16-element look-up table consists of the pre-encoded (i.e., pre-calculated) codewords. This is the speediest of the encoding techniques, but consumes a lot of memory for anything except the smallest code length. For example, the next Hamming codeword size up from the (7,4) Hamming shown in this example would be a (15,11) Hamming code according to the above formulas. This means that the look-up table would have to be 2^{11} (2,048) elements deep (a total of 4,096 bytes @ 2 bytes per element). The flowchart and HC08 assembly code for HAMENC1 is listed in Appendix A.

HAMENC2 — HAMMING ENCODER #2, MATRIX CALCULATION

HAMENC2 actually performs the matrix arithmetic used to encode information into Hamming codewords. This is done with the generator matrix, **G**, described earlier. The generator matrix consists of two submatrices: a k x k identity matrix, $\mathbf{I}_k$, and the transpose of matrix Q, $\mathbf{Q}^T_{m,k}$, where matrix Q consists of k columns which are of weight two or more. The generator matrix used to generate the look-up table in HAMENC1 is:

$$\mathbf{G} = [\,\mathbf{Q}^{T}_{m,k}\ \mathbf{I}_{k}\,]$$

$$\mathbf{G} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

where m = 3 and k = 4.

The 4-bit information word is first bit-wise multiplied (logically anded) by each column in the generator matrix. Each bit in each of the column products is then added together, modulo-2, to create a parity bit for each product. An example of a 4-bit infoword and its generated codeword is given below.

Information Word		**G** Matrix		Code Word
[1 0 1 0]		$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$	=	[0 0 1 1 0 1 0]

The flowchart and HC08 assembly code for HAMENC2 is listed in Appendix B. Once an information word is encoded, it may be transmitted over the channel for recovery by the HAMDEC routine explained next.

HAMMING DECODING

ERROR DETECTION AND CORRECTION OF RECEIVED CODEWORD

A 7-bit Hamming codeword will be decoded into the original 4-bit information word even with up to one bit in the codeword being corrupted by the channel. This routine will use matrix math to recover the information word. The parity-check matrix used in the HAMENC routine(s) is:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

The parity-check matrix has the property such that when a 7-bit codeword generated by the generator matrix and uncorrupted by the channel is multiplied by the transpose of the parity-check matrix, $\mathbf{H}^{T}$, a zero vector is obtained. The three-element result is called the syndrome. For uncorrupted data, the syndrome should be a zero vector. An example is given below.

Code Word		**H** Transpose Matrix		Syndrome
[0 0 1 1 0 1 0]		$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$	=	[0 0 0]

Suppose that we transmit a code vector, **v**, and that an error occurs in the fourth bit position. This is the same as adding a vector, **e**, to the codeword **v** where **e** looks like:

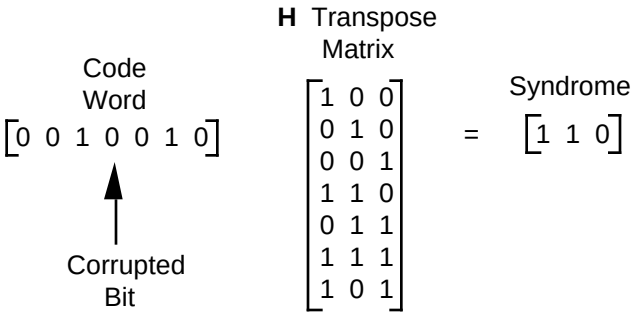
$$\mathbf{e} = [\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ ]$$

By multiplying the received vector by the transpose of the parity-check matrix, we obtain:

$$(\ \mathbf{v} + \mathbf{e}\ )\ \mathbf{H}^T = \mathbf{v}\mathbf{H}^T + \mathbf{e}\mathbf{H}^T = \mathbf{e}\mathbf{H}^T$$

since $\mathbf{v}\mathbf{H}^T = 0$.

The resultant non-zero vector, $\mathbf{e}\mathbf{H}^T$, indicates a problem in the received codeword. As mentioned above, this product of the received vector and the transpose of the parity-check matrix is called the syndrome. An example of a corrupted codeword and its syndrome is given below.



The syndrome bit pattern of a single bit error will be the pattern of one row within $\mathbf{H}^T$.

The row number is numerically equivalent to the corrupted bit position in the received codeword. Thus, by calculating the syndrome and obtaining a non-zero vector we have:

1) Identified that one or two errors have occurred.
2) In the case of a single bit error, we have identified the location of the error within the received word.

The last item to be accomplished is to correct the identified error, which is accomplished by complementing the bit position in the received codeword. The flowchart and HC08 assembly code for HAMDEC is listed in Appendix C.

TIME DIVERSITY PACK AND UNPACK

SOME LIMITATIONS OF HAMMING PERFORMANCE

Although the Hamming ECC allows recovery of "mildly" corrupted codewords, it is limited in its effectiveness against some "real world" corruptions. It was stated earlier that Hamming ECCs are useful in combating the effects of random noise. This applies only for random bit corruptions that do not exceed one bit time per codeword since Hamming codes can only correct up to one bit error per codeword. Where burst errors occur—noise corruptions typically considered longer in duration than random noise—other FEC types are the preferred antidote. However, the FECs useful for combating burst noise require considerable processing power and, as such, should only be used under duress.

HOW TO IMPROVE HAMMING PERFORMANCE

Time-diversity coding is another way to combat the effects of burst noise. By combining bits of many codewords into a transmission packet, it is possible to limit the effect of long noise bursts on a given codeword. Specifically, we view a collection of eight Hamming-encoded codewords arranged in RAM like this:

$$\begin{bmatrix} a07 & a06 & a05 & a04 & \dots & a00 \\ a17 & a16 & a15 & a14 & \dots & a10 \\ a27 & a26 & a25 & a24 & \dots & a20 \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots \\ a77 & a76 & a75 & a74 & \dots & a70 \end{bmatrix}$$

where each element in the matrix, a_{ij} , is a single bit within each byte-wide codeword.

In the case of a (7,4) Hamming code, the eighth bit (bit 7) of each codeword is zeroed. A matrix is used to represent the data bits as they would appear in RAM, that is, the first row is the first data byte with the LSB to the right and MSB to the left, the second row is the next data byte, etc.

By taking one bit from each of the eight Hamming-encoded words and assembling eight bytes where each byte is made of one bit from each of the original codewords, we distribute the information from one codeword over eight different transmissions:

$$\begin{bmatrix} a07 & a17 & a27 & a37 & \dots & a77 \\ a06 & a16 & a26 & a36 & \dots & a76 \\ a05 & a15 & a25 & a35 & \dots & a75 \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots \\ a00 & a10 & a20 & a30 & \dots & a70 \end{bmatrix}$$

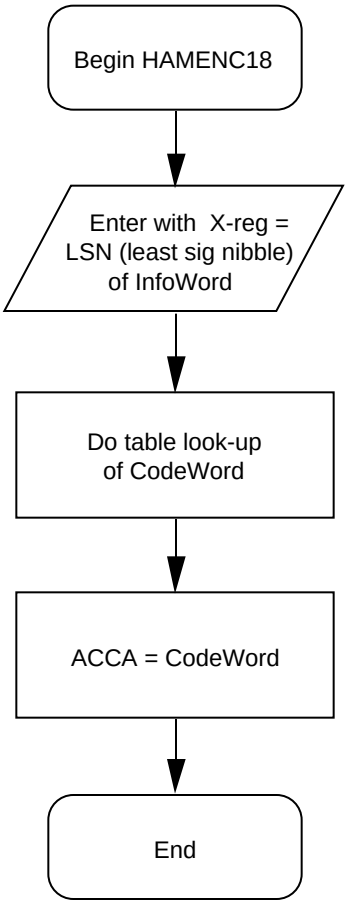
If a "deep fade" in the channel should take out an entire transferred byte, only one bit within each codeword would be affected because each codeword is effectively spread over many data transfers. As shown in the above matrix, the first row (i.e., first data byte to be transferred) consists of the eighth bit position (MSB) of the pre-diversity coded data matrix. If this first row was transmitted and completely corrupted on the receive end, only one bit from each of the pre-diversity encoded codewords would be corrupted. This allows the relatively modest Hamming decoder to detect and completely correct all of the corrupted codewords despite losing one entire byte of data out of eight.

TDPACK — TIME DIVERSITY PACK AND UNPACK

TDPACK transposes an 8 x 8 matrix consisting of 8 bits wide (eight columns) and 8 bytes deep (eight rows). Normally such a matrix would be transmitted, following convention, the first row, most significant bit first. Each subsequent row would be sequentially transmitted, again with MSB first. By executing TDPACK on such an array of bits, the first byte to be transferred would actually contain the seventh bit (MSB) of each byte. Each subsequent byte of data transmitted out of this transposed version of the original data matrix actually only contains one bit from each of the original data bytes. In this way, each byte transmission contains only one bit from each of the original data bytes. Should anything happen to corrupt up to an entire eight bits of transmission, no more than one bit per original data byte would be affected.

After the codeword is received, it must be unpacked. The same routine used to pack (transpose) the matrix is used to unpack it to its original order. It is the output of the transpose that would be used as the input to an ECC decode routine (such as HamDec). The flowchart and HC08 assembly code for TDPACK is listed in Appendix D.

APPENDIX A
HAMENC1 FLOWCHART AND CODE LISTING

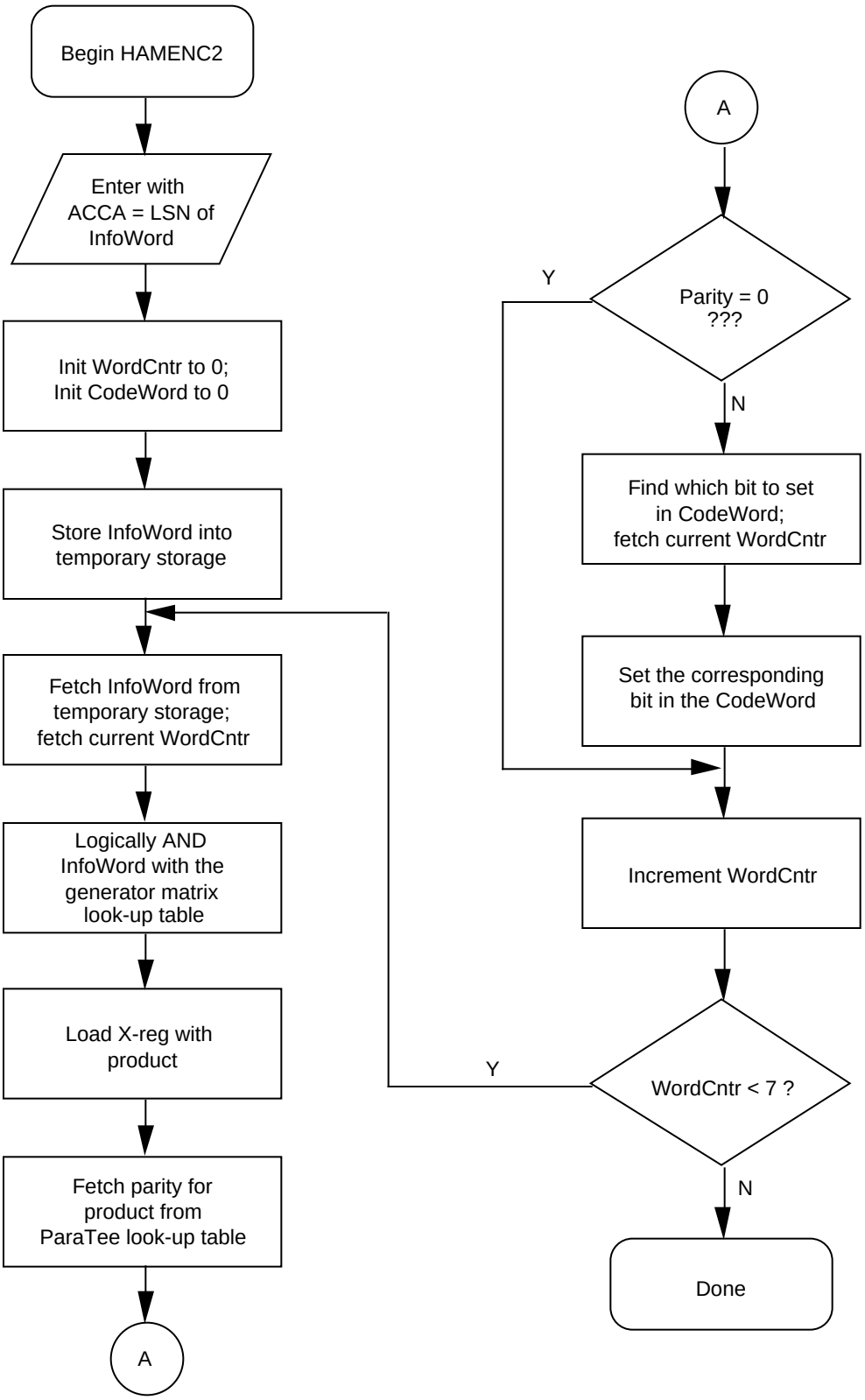


```
*****************************************************************
*
* Program Name: HAMENC1.ASM (Hamming Encoder #1 - Table Look-up)
* Revision: 1.00
* Date: January 21,1993
*
* Written By: Mark Glenewinkel
*             Motorola CSIC Applications
*
* Assembled Under: P&E Microcomputer Systems IASM08
*
*       ********************************
*       *       Revision History       *
*       ********************************
*
*       Rev     0.50    12/15/92        M.A. McQuilken
*                       HC05 version to be translated to HC08 code
*
*       Rev     0.60    01/21/93        M.R. Glenewinkel
*                       Added more comments
*
*       Rev     1.00    01/22/93        M.R. Glenewinkel
*                       HC08 version
*
*****************************************************************
*
* Program Description:
*       This routine will encode a four-bit info word into a (7,4)
*       Hamming encoding codeword.
*
*       This source code is an example of using a table look-up
*       method to encode a four bit info word to a seven bit code
*       word. For a more detailed description of the process of error
*       control codes and Hamming codes in particular, please
*       refer to Motorola Application Note 1221.
*       This routine consists of only one instruction, a
*       look-up table fetch, where the encoding has been already
*       done and inserted into this look-up table.
*
*       "info word" is the word you want to encode
*       "codeword" is an encoded info word
*
* TASK DATA:
*  Input Variables      Output Variables        Description
*  ----------------     -----------------       -------------
*  X                                            Enter routine with
*                                                X-reg=LSN of info
*                                                word.
*                       ACCA                    Leave routine with
*                                                7-bit codeword in here
```

```
*
*
* LOCAL DATA:
*  Input Variables      Output Variables        Description
*  ----------------     -----------------       -------------
*  ACCA                                         Misc. computational
*                                                use.
*
************************************************************************
*
* Register and Variable Equates
*
*       None
*
************************************************************************
*
* Memory
*
*       None
*
************************************************************************

                ORG     $1000           ;beginning of program area
START           EQU     *

************************************************************************

* Main Routine

HAMENC1         lda     CodeWords,X     ;ACCA <- (CodeWords+X)
                                        ;X contains the offset
                                        ; from CodeWords

DONE            nop                     ;done !!!
                bra     DONE

************************************************************************
```

```
* Tables

                ORG     $2000
CodeWords       FCB     %00000000
                FCB     %01010001
                FCB     %01110010
                FCB     %00100011
                FCB     %00110100
                FCB     %01100101
                FCB     %01000110
                FCB     %00010111
                FCB     %01101000
                FCB     %00111001
                FCB     %00011010
                FCB     %01001011
                FCB     %01011100
                FCB     %00001101
                FCB     %00101110
                FCB     %01111111

*************************************************************************

* Vector Setup

                ORG     $FFFE
                DW      START           ;set up reset vector

*************************************************************************
```

APPENDIX B
HAMENC2 FLOWCHART AND CODE LISTING



```
*****************************************************************
*
* Program Name: HAMENC28.ASM (Hamming Encoder - Matrix Calculation)
* Revision: 1.00
* Date: January 21,1993
*
* Written By: Mark Glenewinkel
*             Motorola CSIC Applications
*
* Assembled Under:  P&E Microcomputer Systems IASM08
*
*       *********************************
*       *       Revision History        *
*       *********************************
*
*       Rev     0.50    12/15/92        M.A. McQuilken
*                               HC05 version to be translated to HC08 code
*
*       Rev     0.60    01/20/93        M.R. Glenewinkel
*                               Fixed logic bugs
*
*       Rev     1.00    01/22/93        M.R. Glenewinkel
*                               HC08 version
*
*****************************************************************
*
* Program Description:
*
*       This routine will encode a four-bit info word into a (7,4)
*       Hamming encoding codeword.
*
*
*       This routine differs from HAMENC1 not in results, but in
*       method. Whereas HAMENC1 was basically an easy look-up of
*       pre-encoded (7,4) Hamming codewords, HAMENC2 actually
*       performs the matrix arithmetic to encode the 4-bit info word
*       input. Fortunately when working with modulo arithmetic
*       (particularly mod-2), things like multiplication and such
*       are reduced to fairly easy functions.
*
*       This routine follows this basic overall flow: Workspace is
*       cleared, the info word is first multiplied, each bit in the
*       product is then added together (this is effectively
*       calculating the parity of the product), the actual final
*       codeword is constructed one-bit at a time, at that point
*       the next row from the generator matrix is fetched and this
*       process repeated until all of the generator matrix rows have
*       been combined with the info word.
*
```

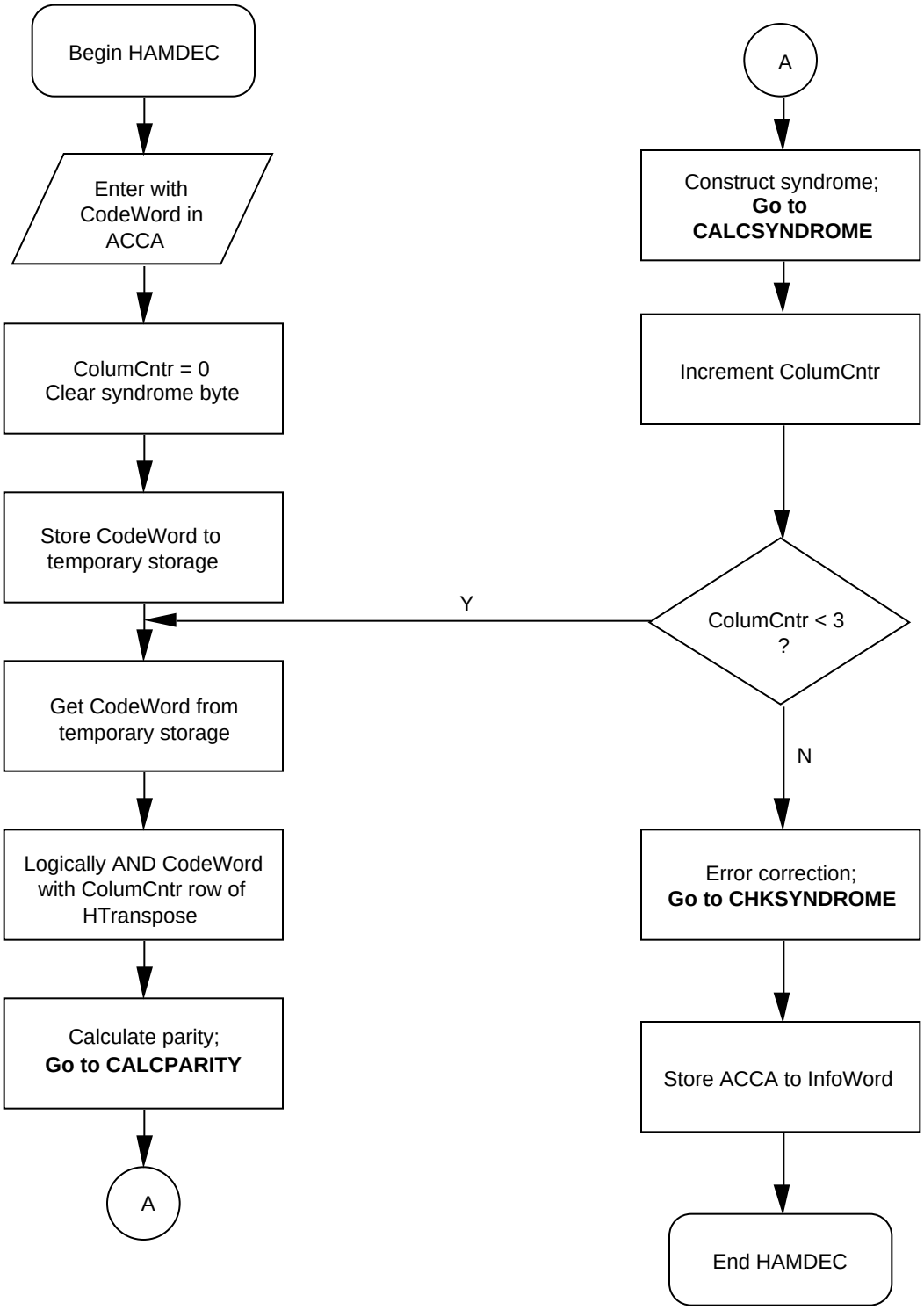
```
*       For a more detailed description of the process of error
*       control codes and Hamming codes in particular, please
*       refer to Motorola Application Note 1221.
*
* TASK DATA:
*  Input Variables      Output Variables        Description
*  ----------------     -----------------       -------------
*  ACCA                                         Enter routine with
*                                                ACCA=LSN of info
*                                                word (4-bit).
*                       ACCA                    Leave routine with
*                                                7-bit codeword in
*                                                here.
*
*
* LOCAL DATA:
*  Input Variables      Output Variables        Description
*  ----------------     -----------------       -------------
*  CodeWord             CodeWord                Yup, you guessed it
*                                                ...this is where we
*                                                keep a temp version
*                                                of the codeword.
*  WordCntr             WordCntr                Keeps track of the
*                                                GenMatrix row that
*                                                is combined with
*                                                the info word.
*  ACCA                 ACCA                    Misc. computational
*                                                use.
*  X                    X                       Misc. computational
*                                                use.
*
*
*************************************************************************
*
* Register and Variable Equates
*
*       None
*
*************************************************************************
*
* Memory
*
                org     $50
CodeWord        RMB     1
InfoWord        RMB     1
WordCntr        RMB     1
*
*************************************************************************
```

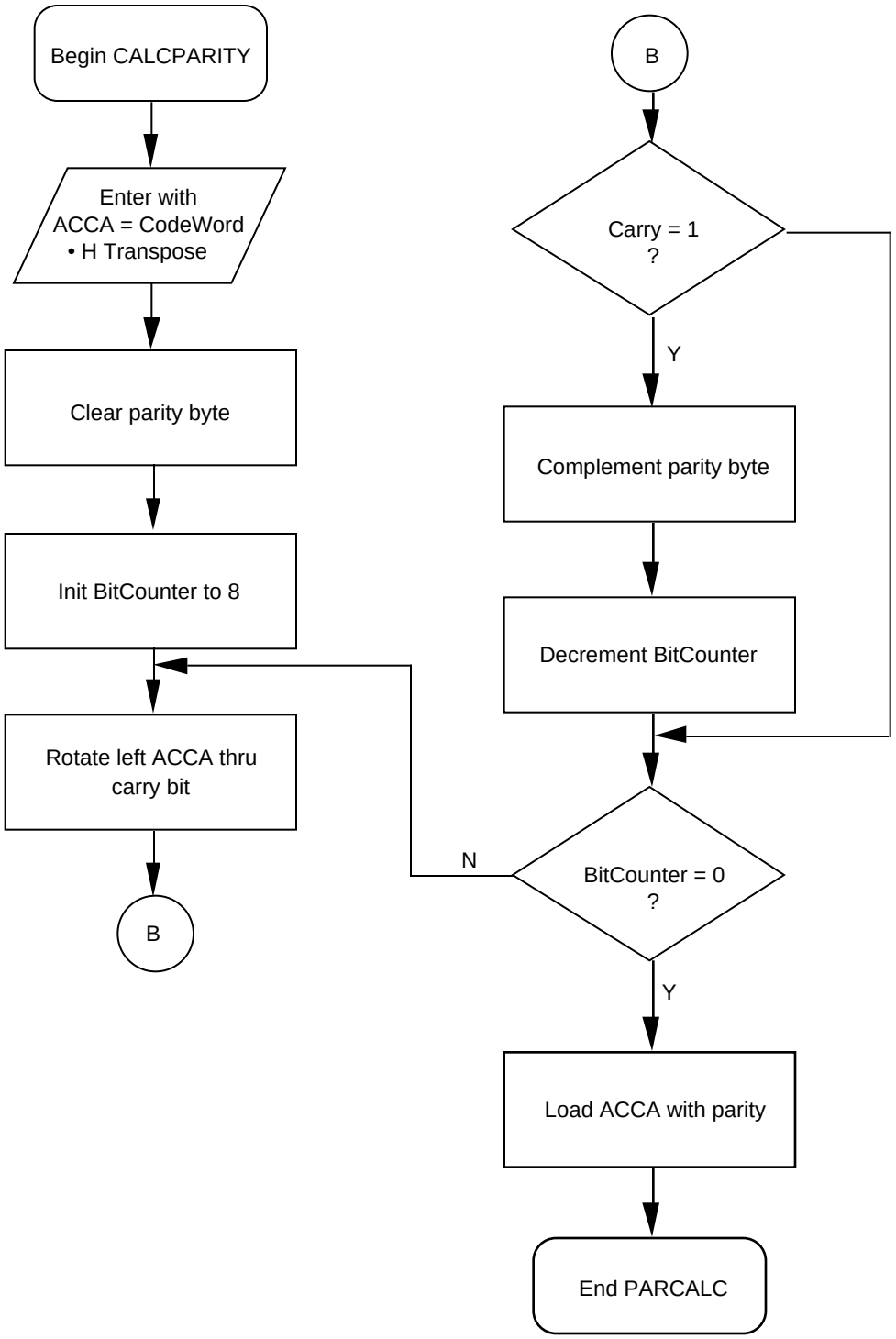
```
                ORG     $1000
START           EQU     *               ;beginning of program area

*************************************************************************

* Main Routine

* As stated in the header, the first place to start is
* the clearing of data space:

HamEnc2         clr     WordCntr
                clr     CodeWord

* Next, before we begin the process of multiplying and adding the
* codeword with the info word, we need to save a copy of the info word
* (remember, we are entering the routine with ACCA having the
* info word):

SaveInfo        sta     InfoWord

* Now we begin the fun stuff...doing the actual multiplication and
* addition that is required to this super-fun Hamming stuff. Each
* multiplication with binary data is actually a logical AND on a
* bitwise basis:

GetInfoWord     lda     InfoWord        ;get the info word for the
                                        ; multiplication.
                ldx     WordCntr        ;get the current row count
                                        ; into the generator matrix.

                and     GenMatrix,X     ;Go forth and multiply...

* Well...that wasn't so bad, was it? Now that the multiplication
* is complete we begin the task of adding the product, bit-by-bit
* (so to speak). Rather than go through the actual tedium of adding
* each bit within the product one bit at a time, I've made a look-up
* table that has it done for you. It even has the clever name of
* PARATEE to remind you that the process of adding bits within a
* byte is determining the byte's parity:

                tax                     ;the byte to have parity
                                        ; encoded resides in ACCA.

CalcParity      lda     ParaTee,X       ;get the parity value
                                        ; from LUT.
```

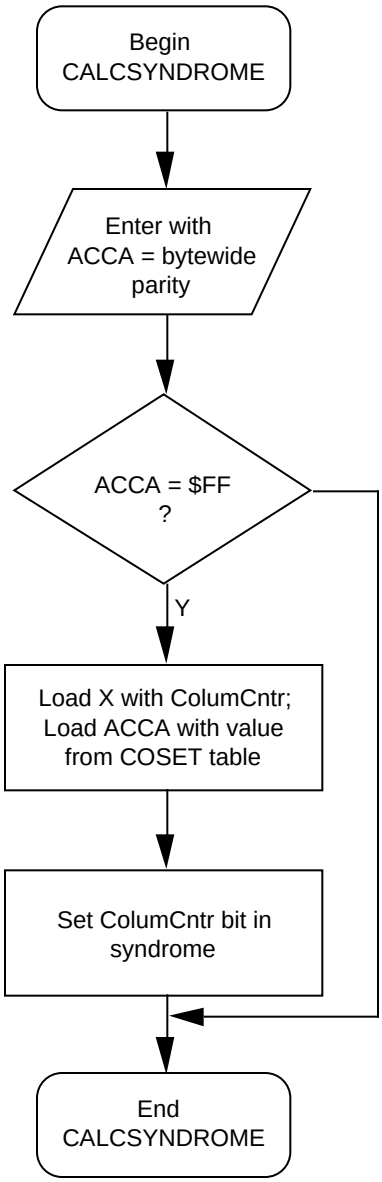
```
* As mentioned in the header, the actual codeword is constructed
* one-bit at a time. For each multiplication and addition we do,
* a single bit within the final codeword results. At this point,
* then, we must construct another bit of the codeword from the
* last multiplcation and addition:

MakeCW          cbeqa   #0,BumpCntr1    ;if parity is odd, then
                                        ; we do nothing to the
                                        ; final codeword.
                                        ;here's the branch to do
                                        ; nothing, otherwise
                                        ; we add a positive bit
                                        ; to the correct position
                                        ; in the codeword.

* If we've made it here, then we know to set a bit in the codeword.
* So even though the value in ACCA gets "stepped on" in this part
* of the routine, the contents of ACCA have done its job and got
* us to this point. A look-up table (called CoSet) was used as the
* mechanism to construct the codeword one bit at time. CoSet
* contains only one bit=1...in each case only the bit that we wish
* to set within the byte:

                ldx     WordCntr        ;WordCntr has the current bit
                                        ; position in it.
                lda     CoSet,X         ;get bit (in correct position)
                                        ; to be set.

                ora     CodeWord        ;set the bit.
                sta     CodeWord        ;modify CodeWord for next use.

* This completes a single cycle in the process of encoding an info
* word into a bit within the final codeword. All that is left to do
* is to check to see if we have completed the entire codeword. If
* we haven't, then pointers get modified so we can do the next one:

BumpCntr1       inc     WordCntr        ;inc current row/bit position
                lda     WordCntr        ;load updated WordCntr
                cmp     #7              ;check to see if we're done.
                blo     GetInfoWord     ;Go back, Jack, and
                                        ; do it again...

DONE            nop                     ;done !!!
                bra     DONE

*********************************************************************
```

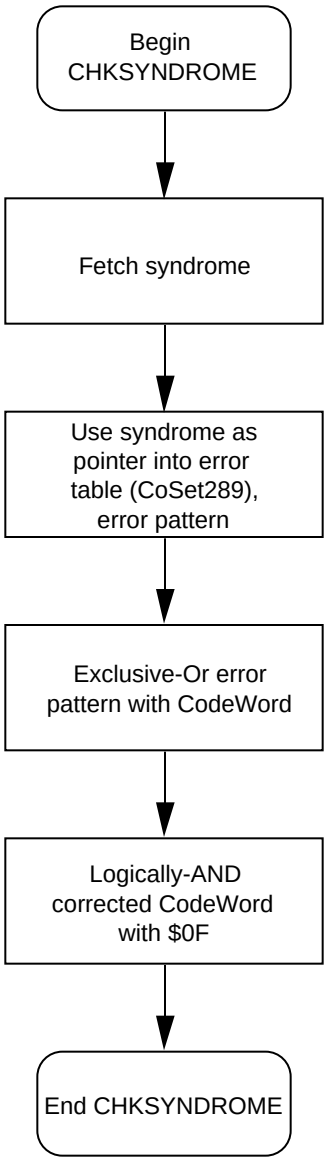
```
* Tables

GenMatrix       FCB     %00001011
                FCB     %00001110
                FCB     %00000111
                FCB     %00001000
                FCB     %00000100
                FCB     %00000010
                FCB     %00000001

ParaTee         FCB     $00
                FCB     $FF
                FCB     $FF
                FCB     $00
                FCB     $FF
                FCB     $00
                FCB     $00
                FCB     $FF
                FCB     $FF
                FCB     $00
                FCB     $00
                FCB     $FF
                FCB     $00
                FCB     $FF
                FCB     $FF
                FCB     $00

CoSet           FCB     %01000000
                FCB     %00100000
                FCB     %00010000
                FCB     %00001000
                FCB     %00000100
                FCB     %00000010
                FCB     %00000001

*****************************************************************

* Vector Setup

                ORG     $FFFE
                DW      START           ;set up reset vector

*****************************************************************
```

APPENDIX C
HAMDEC FLOWCHART AND CODE LISTING




Begin CALCPARITY
Enter with
ACCA = CodeWord
• H Transpose
Clear parity byte
Init BitCounter to 8
Rotate left ACCA thru
carry bit
B
B
Carry = 1
?
Y
Complement parity byte
Decrement BitCounter
N
BitCounter = 0
?
Y
Load ACCA with parity
End PARCALC


Begin
CALCSYNDROME
Enter with
ACCA = bytewide
parity
ACCA = $FF
?
Y
Load X with ColumCntr;
Load ACCA with value
from COSET table
Set ColumCntr bit in
syndrome
End
CALCSYNDROME


Begin
CHKSYNDROME
Fetch syndrome
Use syndrome as
pointer into error
table (CoSet289),
error pattern
Exclusive-Or error
pattern with CodeWord
Logically-AND
corrected CodeWord
with $0F
End CHKSYNDROME

```
*****************************************************************
*
* Program Name: HAMDEC8.ASM ( Hamming Decoder - Matrix Calculation )
* Revision: 1.00
* Date: January 21,1993
*
* Written By: Mark Glenewinkel
*             Motorola CSIC Applications
*
* Assembled Under: P&E Microcomputer Systems IASM08
*
*       *********************************
*       *       Revision History        *
*       *********************************
*
*       Rev     0.50    12/15/92        M.A. McQuilken
*                       HC05 version to be translated to HC08 code
*
*       Rev     0.60    01/21/93        M.R. Glenewinkel
*                       Fixed logic bugs
*
*       Rev     1.00    01/22/93        M.R. Glenewinkel
*                       HC08 version
*
*****************************************************************
*
* Program Description:
*
*       This routine will evaluate a received Hamming-encoded
*       codeword and decode it into its original form, thereby
*       receiving the originally encoded data. HAMDEC will
*       successfully do this even in the presence of up to 1
*       single-bit error induced into the received codeword
*       by the channel.
*
*       This routine works similarly to the HAMENC2 routine, in
*       that the calculations that are performed are actual
*       matrix-type arithmetic operations. The routine works like
*       this: The workspace is prepared (cleared), the received
*       codeword is multiplied by each column of a matrix referred
*       to as "the transpose of the parity check" matrix, byte wide
*       parity is generated, and a 3-bit word is created (it is
*       called the syndrome). The non-zero syndrome is then used
*       to identify the bit location of the error. The syndrome is
*       used to correct the corrupted bit position and recover the
*       original four-bit info word. If the syndrome is zero, then
*       no detectable errors have occurred and the info word
*       is recovered.
*
```

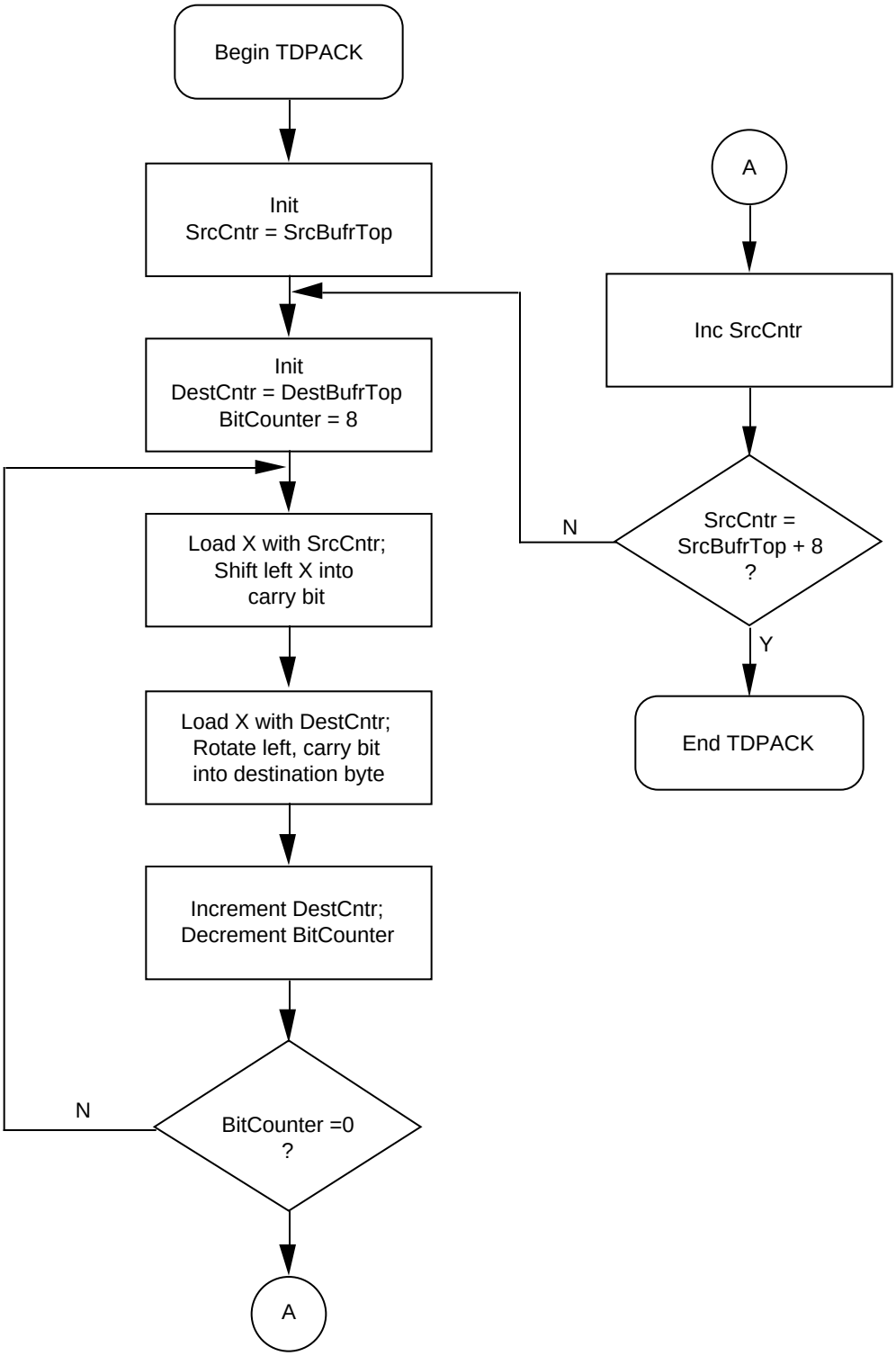
```
* TASK DATA:
*  Input Variables      Output Variables        Description
*  ----------------     -----------------       -------------
*  ACCA                                         Enter routine with
*                                                ACCA=Codeword.
*                       ACCA                    Same contents
*                                                as InfoWord.
*                       InfoWord                The recovered
*                                                info word.
*
* LOCAL DATA:
*  Input Variables      Output Variables        Description
*  ----------------     -----------------       -------------
*  BitCounter           BitCounter              Keeps track of the
*                                                column # within
*                                                HTranspose.
*  CodeWord             CodeWord                Yup, you guessed it
*                                                ...this is where we
*                                                keep a temp version
*                                                of the codeword.
*  ColumCntr            ColumCntr               Keeps track of the
*                                                H-transpose column
*                                                that is combined
*                                                with the codeword.
*                       InfoWord                The recovered
*                                                info word.
*  Syndrome             Syndrome                Data which gives us
*                                                the location of any
*                                                errors (if any).
*  ACCA                 ACCA                    Misc. computational
*                                                use.
*  X                    X                       Misc. computational
*                                                use.
*
*
*************************************************************************
*
* Register and Variable Equates
*
*       None
*
*************************************************************************
```

```
*
* Memory
*
                ORG     $50
BitCounter      RMB     1
CodeWord        RMB     1
ColumCntr       RMB     1
InfoWord        RMB     1
Parity          RMB     1
Syndrome        RMB     1
*
*****************************************************************

                ORG     $1000           ;beginning of program
START           EQU     *

*****************************************************************

* Main Routine

* We must prepare the workspace...any nonzero stuff in some variables
* could really mess up our process. So...

HamDec          clr     ColumCntr
                clr     Syndrome

* Since we enter this routine with the codeword contained in the
* accumulator, and use the codeword multiple times, a copy is first
* made into the location called "CodeWord":

                sta     CodeWord


GetCodeWord     lda     CodeWord        ;get the first argument
                                        ; used in our multiplication.
                ldx     ColumCntr       ;get the current column to
                                        ; be worked on
MultEm          and     HTranspose,X    ;Multiply!

* The next step in the process is to calculate the parity of the
* received codeword. This is accomplished by rotating each bit
* through the carry bit and then complementing a byte called "parity":

CalcParity      clr     Parity          ;clear the workspace
                    mov     #8,BitCounter   ;prep loop counter x to do
                                        ; all eight bits in received
                                        ; codeword.
                                        ;it's now prepped.
```

```
RotateIt        lsla                    ;start the process of deter-
                                        ; mining the state of each
                                        ; bit within the rec'd
                                        ; codeword.

                bcc     BumpCntr2       ;if carry is not positive,
                                        ; then do nothing but
                                        ; bump counter.

* Otherwise, fall through to here:

CompParity      com     Parity

* Bump pointer for the next bit to do. If the counter is zero, then
* the process stops:

BumpCntr2       dbnz    BitCounter,RotateIt
                lda     Parity

* The next step in our overall decoding of the codeword into an
* information word, is to calculate the syndrome. Remember, the
* syndrome will tell us whether a detectable error has occurred and
* allow us to find out where it occurred. Again, if the syndrome is
* zero, then no detectable error has occurred and we may recover the
* original info word by merely "looking it up" in a look-up table.

CalcSyndrome    cmp     #$FF
                bne     BildSynDun      ;if syndrome (bit#=X)
                                        ; is a 0, then
                                        ; branch else fall through...

                ldx     ColumCntr       ;find out which part of the
                                        ; syndrome we're working on.
                lda     CoSet,X         ;get ith bit of syndrome
                                        ; and set to a one.
                ora     Syndrome
                sta     Syndrome        ;save it for later use.

* We're finally to the point where we want to see if all of the bits
* have been processed. This is done by updating the column counter
* (ColumCntr) and branching back to the top of the process if we
* must finish constructing the syndrome. Else, we move down into
* correcting the error and/or just recovering the codeword.

BildSynDun      equ     *
ChkColumCnt     inc     ColumCntr       ;inc current column counter
                lda     ColumCntr       ;load column counter
                cmpa    #3
                blo     GetCodeWord     ;branch if not done with
                                        ; all 3 columns
```

```
ChkSyndrome     ldx     Syndrome
                lda     CodeWord        ;get codeword for correction
                eor     CoSet2,X        ;correct the codeword. ACCA
                                        ; now contains the corrected
                                        ; codeword.

* One of the traits of Hamming codes is that part of the codeword
* is the info word (nibble, in this case). If you look at the
* listing for HamEnc1, you'll notice that the least significant
* nibble contains the info word. Hence, the recovery process for
* the Hamming decode merely consists of ANDing the LSN of the
* corrected codeword. So...

                and     #$0F            ;ACCA now has the original
                                        ; info word.
                sta     InfoWord

DONE            nop
                bra     DONE            ;done !!!

*********************************************************************

* Tables

HTranspose      FCB     %01001011
                FCB     %00101110
                FCB     %00010111

CoSet           FCB     %00000100
                FCB     %00000010
                FCB     %00000001

CoSet2          FCB     %00000000
                FCB     %00010000
                FCB     %00100000
                FCB     %00000100
                FCB     %01000000
                FCB     %00000001
                FCB     %00001000
                FCB     %00000010

*********************************************************************

* Vector Setup

                ORG     $FFFE
                DW      START           ;set up reset vector

*********************************************************************
```

APPENDIX D
TDPACK FLOWCHART AND CODE LISTING



```
*****************************************************************
*
* Program Name: TDPACK.ASM (Time Diversity Pack and UnPack)
*
* Revision: 1.00
* Date: January 20,1993
*
* Written By: Mark Glenewinkel
*             Motorola CSIC Applications
*
* Assembled Under: P&E Microcomputer Systems IASM08
*
*       *********************************
*       *       Revision History        *
*       *********************************
*
*       Rev     0.50    12/15/92        M.A. McQuilken
*                       HC05 version to be translated to HC08 code
*
*       Rev     0.60    01/20/93        M.R. Glenewinkel
*                       Fixed logic bugs
*
*       Rev     1.00    01/22/93        M.R. Glenewinkel
*                       HC08 version
*
*****************************************************************
*
* Program Description:
*
*       This routine will take a matrix that is 8 bits wide and 8
*       bytes deep and transpose the matrix so that the first row
*       becomes the 1st column, the second row becomes the 2nd
*       column...the last row becomes the 8th column. This is to
*       distribute data bits over several byte transfers so that no
*       channel errors "wipe out" a complete single byte...only
*       individual bits within each source byte will be hit (if the
*       channel fades are "deep" and not frequent). It is expected
*       that the complementary process of "unpacking" (un-transposing)
*       on the receive end must be done to recover the data as it was
*       intended. By executing another transpose on the data, the data
*       will be "unpacked" to its original form.
*
*       As stated above, this routine transposes a matrix consisting
*       of 64 bits. It does this by left shifting each bit of one
*       source row into its corresponding destination column. After
*       all eight bits of the source are shifted to the destination
*       column, the next source byte is left shifted to the next
*       destination column. This entire process is repeated until
*       all eight source bytes and destination bytes have had all of
*       their bits moved.
```

```
*
* Task Data:
*
*  Input Variables      Output Variables        Description
*  ----------------     -----------------       -------------
*  SrcBuffer                                    Eight byte buffer for
*                                                info that is to be
*                                                transposed before
*                                                transmission on
*                                                channel.
*                       DestBuffer              Eight byte buffer that
*                                                contains transposed
*                                                version of data
*                                                previously contained
*                                                in SrcBuffer.
*
*
* LOCAL DATA:
*  Input Variables      Output Variables        Description
*  ----------------     -----------------       -------------
*  BitCounter           BitCounter              Keeps track of the
*                                                bits being shifted
*                                                from the source
*                                                buffer
*  DestCntr             DestCntr                Current byte location
*                                                in destination
*                                                buffer.
*  SrcCntr              SrcCntr                 Current byte location
*                                                in source buffer.
*  ACCA                 ACCA                    Misc. computational
*                                                use.
*  X                    X                       Misc. computational
*                                                use.
*
************************************************************************
*
* Register and Variable Equates
*
*       None
*
************************************************************************
```

```
*
* Memory
*
                ORG     $50
BitCounter      RMB     1
DestCntr        RMB     1
SrcCntr         RMB     1
SrcBuffer       EQU     *
SrcBufrTop      RMB     8
DestBuffer      EQU     *
DestBufrTop     RMB     8
*
*****************************************************************

                ORG     $1000           ;beginning of program area
START           EQU     *

*****************************************************************

* Main Routine

* Initialization of the variables must occur before the data can
* be manipulated.

TDPack          mov     #SrcBufrTop,SrcCntr
                                        ;the starting place for
                                        ; source data manipulation

* Although the next couple of lines could also be considered
* basic workspace initialization, the initialization occurs
* every 8-bits of source data manipulation:

SetUpDestPntr   mov     #DestBufrTop,DestCntr
                                        ;these two lines allow us to
                                        ; point to DestBufrTop


* A separate bit counter is maintained to ease the hassle of
* evaluating the loop point for the 8 bits:

                mov     #8,BitCounter   ;(BitCounter) <- #8


* At this point, the inner loop (which counts the bits shifted out
* of the source buffer) begins.
```

```
InnerLoop       ldx     SrcCntr         ;SrcCntr contains the
                                        ; location of the current
                                        ; byte to be shifted
                                        ; out of the source buffer.
                lsl     ,X              ;this moves from source
                                        ; data buffer
                                        ; into the carry bit.

* Time to move the data into the destination buffer:

                ldx     DestCntr        ;like the SrcCntr, this
                                        ; pointer contains the
                                        ; address of the destination
                                        ; byte.
                rol     ,X              ;move data from carry into
                                        ; destination byte.

* Here's where we determine if we've shifted enough data bits:

                inc     DestCntr        ;proceed to next
                                        ; destination byte
                dbnz    BitCounter,InnerLoop
                                        ;branch if we've not moved
                                        ; eight bits in eight of
                                        ; the destination bytes.


* If we've made it here, then we've moved an eight bit chunk of the
* source buffer. These next few lines of code determine the actual
* value of the pointer into the source buffer. It also contains
* the test for completion of movement into the destination buffer:

                inc     SrcCntr         ;update counter
                lda     SrcCntr         ;get counter in ACCA

                cmp     #SrcBufrTop+8   ;test
                bne     SetUpDestPntr   ;branch if we haven't
                                        ; moved everything.

DONE            nop                     ;done !!!
                bra     DONE

*****************************************************************

* Vector Setup

                ORG     $FFFE
                DW      START           ;set up reset vector

*****************************************************************
```