DOC-CR-BT-83-003

LKC
QA
268
.M673
1983
c.2

ENCODING AND DECODING

DATA BLOCK AND BUNDLE CODES

IC

B.C. Mortimer and M.J. Moore

Department of
Mathematics and Statistics

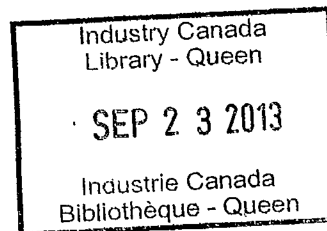
Carleton University
Ottawa, Canada

QA
268
M888
1983

9

ENCODING AND DECODING
DATA BLOCK AND BUNDLE CODES

B.C. Mortimer and M.J. Moore


Industry Canada
Library - Queen
SEP 2 3 2013
Industrie Canada
Bibliothèque - Queen

JUNE, 1983

part of

DSS Contract No. OSU82-00164

Scientific Authority:
Dr. Mike Sablatash
Communications Research Centre
Department of Communications
Ottawa, Canada

Principal Investigator:
Dr. Brian Mortimer
NSERC University Research Fellow
Department of Mathematics and Statistics
Carleton University
Ottawa, Canada

TABLE OF CONTENTS

		Page
CHAPTER 1	ENCODING THE DATA BLOCK CODES	1
CHAPTER 2	FINITE FIELD ARITHMETIC AND ENCODING CODE C	5
CHAPTER 3	DECODING DATA BLOCK CODES	11
CHAPTER 4	BUNDLE CODES - ENCODING AND DECODING	18
REFERENCES		24
APPENDIX A	LOOK-UP TABLE FOR LOG, EXP AND PARITY	A-1
APPENDIX B	IMPLEMENTATIONS IN BASIC	B-1
	PROGRAM B.2: An encoder for the Bundle Code	B-2
	PROGRAM B.3: A Decoder for the Bundle Code	B-3
	PROGRAM B.4: A demonstration program for the complete Bundle encoding and decoding	B-5

CHAPTER 1

ENCODING THE DATA BLOCK CODES

The basic unit in the Telidon system is the byte. The bytes always use their high order bit to give them an overall odd parity. A data block with no further protection is encoded in what we will call here mode $M = 0$. We call this mode PARITY-ONLY. Encoding consists of deciding the parity of the 7 data bits and setting or clearing the eighth bit (b8) appropriately. For a byte B we will write this as

give B odd parity;

an instruction which could be implemented in many ways. We will indicate a method using a look-up table below as part of a more complex decoder.

The second mode, $M = 1$, is the PRODUCT CODE. The 27 odd-parity data bytes are EOR-ed together to produce a byte σ. We want, in fact, the complement of σ so it is best to start with $\sigma = (1\ 1\ 1\ 1\ 1\ 1\ 1\ 1)$. Then σ is added as the 28th byte of the data block. Let B_j be a byte of the data block for $j = 0,1,\ldots, N$ and $N = 27$. Then the encoder can be written as follows:

$$
\left\{
\begin{array}{l}
\sigma \leftarrow (1\ 1\ 1\ 1\ 1\ 1\ 1\ 1) = (255) \\
\text{for } j = 0 \text{ to } N-1 \\
\qquad \text{give } B_j \text{ odd parity} \\
\qquad \sigma \leftarrow \sigma \oplus B_j \\
B_N \leftarrow \sigma
\end{array}
\right.
$$

Figure 1. PRODUCT encoder; M = 1

We note that B_N will automatically have odd parity and that since σ starts as (255) the "longitudinal parity checks" are odd as well. This agrees with BS-14 1 . The symbol $\oplus$ stands for the exclusive-or (EOR) of bytes.

The third option for the data block code is a Reed-Solomon 1-byte-error correcting code we call C . This is mode $M = 2$. The symbols of code C are bytes. Since the bytes all have odd parity, there are only 128 legitimate symbols and we take these to represent elements of the finite field $GF(2^7)$ of 128 elements. This means that we can conveniently add and multiply bytewise. Addition is exclusive-or. Multiplication is more complex and this is the source of power for code C (and all other algebraic codes).

To encode a set of $N-1$ data bytes $B_0, \ldots, B_{N-2}$ with code C we must calculate two more bytes B_{N-1}, B_N . By definition we have

$$B_0 \oplus B_1 \quad \ldots \quad B_{N-2} \oplus B_{N-1} \oplus B_N = 0$$

$$B_0 \oplus B_1 \beta \quad \ldots \quad B_{N-1} \beta^{N-1} \oplus B_N \cdot \beta^N = 0 .$$

Here $\cdot$ represents multiplication in $GF(2^7)$ and the symbol β is a particular element of $GF(2^7)$. This element β is a root of $X^7 + X^3 + 1$: (This is an arbitrary choice of a primitive element of $GF(2^7)$; a different choice would give an essentially equivalent code.) It is more convenient to work with $\gamma = \beta^{-1}$ which is a root of $X^7 + X^4 + 1$. The defining equations can now

be written

$$B_0 \oplus B_1 \oplus \cdots \oplus B_{N-2} \oplus B_{N-1} \oplus B_n = 0$$

$$B_0 \cdot \gamma^N \oplus B_1 \cdot \gamma^{N-1} \oplus \cdots + B_{N-2} \cdot \gamma^2 \oplus B_{N-1} \cdot \gamma + B_N = 0$$

We should clarify the field representation. We have found that it is best if the bit b8 is used to fix the parity of the byte and not be used in the field representations at all. (We then represent (b8, b7, ..., b1) as $b_7\gamma^6 + b_6\gamma^5 + \ldots + b_1$. This is not important however, except in calculating the logarithm and exponential tables in Chapter 2.)

To encode we calculate σ and τ :

$$\sigma = B_0 \oplus \cdots \oplus B_{N-2}$$

$$\tau = B_0\gamma^{N-2} \oplus \cdots \oplus B_{N-3}\gamma \oplus B_{N-2} .$$

Then we solve the system,

$$B_{N-1} \oplus B_N = \sigma$$

$$B_{N-1}\gamma \oplus B_N = \tau\gamma^2$$

for the check bytes B_{N-1} and B_N. The solution is,

$$B_{N-1} = (\sigma + \tau\gamma^2)/(1 + \gamma)$$

$$B_N = \sigma \oplus B_{N-1}$$

as with PRODUCT code we start with $\sigma = (255)$ so that B_N gives the same longitudinal odd parity checks used in PRODUCT code.

We see that we must have a way of multiplying by γ and γ^2 and dividing by $1 + \gamma$ in the finite field $GF(2^7)$. This can be accomplished in a variety of ways. We will use a pair of look-up tables each 128 bytes long. These tables store logarithms and exponentials base γ in $GF(2^7)$ and are discussed in Chapter 2.

$\sigma \leftarrow (1\ 1\ 1\ 1\ 1\ 1\ 1\ 1)$

$\tau \leftarrow (0\ 0\ 0\ 0\ 0\ 0\ 0\ 0)$

for $j = 0$ to $N-2$

$$\left[\begin{array}{l} \text{give } B_j \text{ odd parity} \\ \sigma \leftarrow \sigma \oplus B_j \\ \tau \leftarrow B_j \oplus \tau \cdot \gamma \end{array}\right.$$

$B_{N-1} \leftarrow (\sigma + \tau\gamma^2)/(1 + \gamma)$

give B_{N-1} odd parity

$B_N \leftarrow \sigma \oplus B_{N-1}$

Figure 2 Code C Encoder; M = 2

CHAPTER 2

FINITE FIELD ARITHMETIC AND ENCODING CODE C

The key to speed in the implementation of a Reed-Solomon code is an efficient method of carrying out the finite field arithmetic. Addition is not a problem with our representation of the field since it is the exclusive-or of bytes. We have selected look-up tables for multiplication in the finite field as a very quick easily handled method which can be incorporated into a microprocessor based teletext decoder in a straightforward way.

Every non-zero element of $GF(2^7)$ can be written as a power of γ. This gives us two functions LOG and EXP which behave much like the classical log and exp functions. The base is γ and we have

$$\mathrm{LOG}(\gamma^j) = j$$

$$\mathrm{EXP}(j) = \gamma^j .$$

Of course LOG(0) is undefined. Also, $\gamma^{127} = \gamma^0 = 1$ so j in these formulae can always be taken so that $0 \leq j < 127$. (In fact we allow $j = 127$ as it speeds up the decoder.) The function LOG is calculated by using the binary representation of γ^j (i.e. the byte which represents γ^j) as an offset from the top of the LOG table. The integer j is stored at the calculated address. Thus

LOG is a table of integers between 0 and 127 which are indexed by the field elements. The zero element (offset) does not actually represent a logarithm but we will need this location later so we leave it in. The EXP table is also 128 bytes long and at offset j it contains γ^j for $j = 0, 1, \ldots, 127$. As mentioned above, the first and last entries are the same.

To actually construct the tables we have to be precise about the way that bytes represent field elements. The field $GF(2^7)$ has a basis $\{\gamma^6, \gamma^5, \ldots, \gamma, 1\}$ over $GF(2) = \{0,1\}$. The relation $\gamma^7 = \gamma^4 + 1$ allows us to calculate successive powers of γ in this basis. For example

$$\begin{aligned}
\gamma^7 &= \gamma^4 + 1 \\
\gamma^8 &= \gamma^5 + \gamma \\
\gamma^9 &= \gamma^6 + \gamma^2 \\
\gamma^{10} &= \gamma^7 + \gamma^3 = \gamma^4 + \gamma^3 + 1 \\
\gamma^{11} &= \gamma^5 + \gamma^4 + \gamma \\
\gamma^{12} &= \gamma^6 + \gamma^5 + \gamma^2 \\
\gamma^{13} &= \gamma^7 + \gamma^6 + \gamma^3 = \gamma^6 + \gamma^4 + \gamma^3 + 1 \quad \text{etc.}
\end{aligned}$$

So each field element is represented by seven bits. We will attach a zero at the left end of the seven bits to make a full byte (of odd or even parity). This byte is used as an offset from the start of the LOG table to find the position of the log of the field element base γ. For example, $\gamma^{13} = \gamma^6 + \gamma^4 + \gamma^3 + 1 \equiv (0\ 1\ 0\ 1\ 1\ 0\ 0\ 1)$.

Now this byte is binary for 89 and in the 89th byte of the LOG table we store the integer 13 ≡ (0 0 0 0 1 1 0 1). The EXP table is constructed in the opposite way. The jth entry is the byte representing γ^j .

	LOG		EXP
LOG:	0 0 0 0 0 0 0 0	EXP:	0 0 0 0 0 0 0 1
+1:	0 0 0 0 0 0 0 0	+1:	0 0 0 0 0 0 1 0
+2:	0 0 0 0 0 0 0 1	+2:	0 0 0 0 0 1 0 0
+3:	0 1 1 0 0 0 0 1	+3:	0 0 0 0 1 0 0 0
+4:	0 0 0 0 0 0 1 0	+4:	0 0 0 1 0 0 0 0
+5:	0 1 0 0 0 0 1 1	+5:	0 0 1 0 0 0 0 0
+6:	0 1 1 0 0 0 1 0	+6:	0 1 0 0 0 0 0 0
+7:	0 0 0 1 1 0 1 0	+7:	0 0 0 1 0 0 0 1
+8:	0 0 0 0 0 0 1 1	+8:	0 0 1 0 0 0 1 0
	⋮		⋮

The rule then for using the LOG table is to first ensure that the high order bit is zero in the byte B whose log we wish to calculate. Then add this byte, as a rational integer, to the address LOG (= top of LOG table) and retrieve the $\log_\gamma(B)$ (in the 7 low order bits at this address). For EXP we check that $0 \leq j \leq 127$, adjusting by a multiple of 127 if necessary, then use j as an offset from EXP (= top of EXP table) to find the byte which represents γ^j (in its low 7 bits).

The final table that we need is PARITY. This table allows us to determine the parity of an arbitrary byte. The alternative is to

run through all the bits of the block which of course takes at least 8 times longer. The table Parity needs to have 256 bytes since any byte could appear as the test byte. In fact, only one bit is needed at each address, set to 1 if and only if the offset from PARITY is an odd parity byte. Our combined LOG and EXP tables have 256 bytes and each byte has a free high order bit. Thus, we can construct our PARITY table "on top of" the LOG and EXP tables. We store the LOG and EXP tables contiguously in memory and set PARITY to be the top of the combined table. For any byte B we store the parity of B in the high order bit of PARITY +B. We must remember to always clear this bit when we use the LOG and EXP tables. This is crucial.

The full table is Appendix A.

Address		Memory
PARITY =	LOG	0 0 0 0 0 0 0 0
+1	+1	1 0 0 0 0 0 0 0
+2	+2	1 0 0 0 0 0 0 1
+3	+3	0 1 1 0 0 0 0 1
+4	+4	1 0 0 0 0 0 1 0
⋮	⋮	⋮
+127	+127	1 0 1 0 0 0 1 0
+128	EXP	1 0 0 0 0 0 0 1
+129	+1	0 0 0 0 0 0 1 0
+130	+2	0 0 0 0 0 1 0 0
⋮	⋮	⋮
+254	+126	1 1 0 0 1 0 0 0
+255	+127	0 0 0 0 0 0 0 1

Figure 3. LOG, EXP and PARITY tables .

We will now work through the use of the tables LOG, EXP and PARITY in encoding code C. First, the statement "give B odd parity" can be made precise:

$$\text{if } \{\text{PARITY} + B\} < 128 \quad \text{then} \quad B \leftarrow B \oplus (128) .$$

Here we use $[x]$ to denote the contents of location x. Now we can write out the encoder for code C. Again the data bytes are denoted B_i for $i = 0$ to $N-2$ and we initialize σ with (255) to maintain compatibility with the PRODUCT code.

```
σ ← (255)
τ ← (0)
for  i = 0  to  N-2
          ┌ give B_i odd parity
          │ β ← B_i
          │ σ ← σ ⊕ β
          │ β ← β  and (127)          * clears high order bits
          │ τ ← τ  and (127)          * clear high order bit
          │ if  τ = 0  then skip      * don't take log of 0
          │ τ ← [LOG + τ] + 1         * τ is log(τγ)
          │ τ ← τ and (127)
          │ τ ← [EXP+ τ]              * τ is τγ
  Skip 1  │ τ ← τ ⊕ β                 *
          └
```

Thus far we have calculated σ and τ. We must now successively calculate $\tau\gamma^2$, $\tau + \tau\gamma^2$, $(\sigma + \tau\gamma^2)\gamma^{30} = B_{n-1}$ since $\gamma^{30} = 1/(1 + \gamma)$. Finally, we give B_{N-1} odd parity and set $B_N = \sigma \oplus B_{N-1}$. The encoder continues;

```
         τ ← τ  and  (127)
         if   τ = 0  then  Skip 2
         τ ← 127  and  [ LOG + τ ]
         τ ← τ + 2
         if  τ > 127  then  τ ← τ - 127
         τ ← 127  and  [EX + τ ]
Skip 2   τ ← τ ⊕ σ
         if   τ = 0  then  ⎡B_N-1 ← (128)
                           ⎣go to Skip 3
         τ = (127 and  [LOG + τ]) + 30
         if   τ > 126  then  τ ← τ - 127
         B_N-1 ← 127  and  [EXP + τ]
         give  B_N-1 the right parity
Skip 3   B_N ← B_N-1 ⊕ σ                        * done
```

This encoder illustrates the rules for using the look-up tables. Note that the program could be much shorter if larger tables were used so that much of the checking could be avoided.

CHAPTER 3

DECODING DATA BLOCK CODES

To decode the PARITY ONLY code, mode $M = 0$, we simply look at the parity of each byte and declare a failure if the parity is even.

```
        for  i = 0  to  N
              [if  [PARITY + B_i]  < 128  then  DF ]
        Decoding Failure Flag ← 0
        go to DONE
     DF Decoding Failure Flag ← 1
   DONE end
```

Figure 4 PARITY-ONLY DECODER; M = 0

For the PRODUCT code, we must note the number of parity failures, in a variable PFC (Parity Failure Counter) and the index of the first failed byte. We also calculate the longitudinal parity check σ of the bytes. We start with (255) so that an error-free codeword will give (0) from this calculation. Then, if there are no parity failures and $\sigma = 0$ then we are DONE. If there is one byte parity failure (PFC = 1) and σ has weight 1 then we correct a single error. Otherwise, we declare a decoding failure.

```
PFC ← 0
σ ← (255)
for i = 0 to N-1
    ⎡ if [ PARITY + B_i ] < 128 then
    ⎢                 if PFC = 1 then DF
    ⎢                 else | PFC ← PFC + 1
    ⎢                      | BADBYTE ← i
    ⎣ σ ← σ ⊕ B_i
if PFC = 0 and σ = 0 then ⎡ DFF ← 0
                          ⎣ go to DONE
if PFC = 1 and weight (σ) = 1 then ⎡ DFF ← 0
                                   ⎢ i ← BADBYTE
                                   ⎢ B_i ← B_i ⊕ σ
                                   ⎣ go to DONE

DF      DFF ← 1
DONE    END
```

Figure 5 PRODUCT Decoder; $M = 1$

Now we come to the decoder for code C. We will first outline the decoding method. Suppose that an error corrupts one byte B_i by adding an error pattern E. Thus $B_i \oplus E$ is received while B_i was sent. (Thus E contains a 0 in the position of each correct bit and a 1 in the position of each error.) At the decoder we calculate,

$$\sigma = B_0 \oplus \ldots \oplus (B_i \oplus E_i) \oplus \ldots \oplus B_N \oplus (255)$$

$$\tau = B_0 \gamma^N \oplus B_1 \gamma^{N-1} \oplus \ldots \oplus (B_i \oplus E) \gamma^{N-1} \oplus \ldots \oplus B_n .$$

So we obtain in fact

$$\sigma = E$$

$$\tau = E\gamma^{N-i} .$$

To correct the error we express σ/τ as a power γ^j of γ. Then $i = N - j$ is the erroneous byte and we correct this byte,

$$B_i \leftarrow B_i \oplus \sigma .$$

The determination of j is accomplished with LOG table.

If there are two erroneous bytes say B_i and B_j then we find (if the error patterns are E_i and E_j),

$$\sigma = E_i \oplus E_j$$

$$\tau = E_i \gamma^{N-i} \oplus E_j \gamma^{N-j} .$$

If bytes i and j show parity failures or are otherwise identifiable, then the exponents $N-i$ and $N-j$ are known, then we can calculate E_i and E_j:

$$E_i = (\sigma\gamma^{N-j} \oplus \tau)/(\gamma^{N-i} \oplus \gamma^{N-j})$$

$$E_j = \sigma \oplus E_i .$$

Thus we must count the number of bytes with a parity failure. If there are two such then their indices are passed on and we perform erasure decoding while if there are 0 or 1 bytes with parity violations then we perform single byte correction.

In fact, there are other places in which the decoder should declare a decoding failure. In the case of one byte correction, the calculated byte index may not correspond to the byte with parity failure (if there was one) or fall outside 0 ... N then we declare a decoding failure. Moreover, if one of σ and τ is zero and the other is not, then we declare decoding failure. In the case of two-byte erasure, if E_i or E_j is zero then again declare decoding failure.

In this decoder we still store N-i instead of i if there is a parity failure in the ith byte. We will describe the algorithm in pseudo-code below and then give the look-up table implementation. An implementation of the look-up table decoder in BASIC is in Appendix B.

Figure 6 Code C Decoder; M = 2

```
PFC ← 0                                   * parity failure counter
σ ← (255)
τ ← (0)
For  i = 0  to  N
   If parity (B_i) = 0  then
        if  PFC = 2  then  Decoding Failure
        else  PFC ← PFC + 1
              Bad Byte (PFC) ← i
        end if
   end if
σ ← σ ⊕ B_i
τ ← τ.γ ⊕B_i
if  σ = 0  and  τ = 0  then Done
if  PFC < 2  then
   if σ = 0  or τ = 0  then Decoding Failure
   j ← N-log_2 (τ/σ)
   if  j ≠ Badbyte (1) and PFC = 1  then Decoding Failure
   if  j > N  then Decoding Failure
   B_j ← B_j ⊕  σ
   Done

endif
i ← Bad Byte (1)
j ← Bad Byte (2)
E ← (σγ^(N-i) +τ) / (γ^(N-i) +γ^(N-j))
if  E = 0  OR  E = σ  then Decoding Failure
B_i ← B_i ⊕E
B_j ← B_j ⊕E⊕σ
Done.
```

Figure 7 Code C Decoder, M = 2: look-up table details

```
DFF ← 0
N ← 27
PFC ← 0
σ ← (255)
τ ← (0)
for  i = 0  to  N
   ┌ if [PARITY + B_i] < 128  then
   │                       ┌if  PFC = 2  then DF
   │                       └else ┌PFC ← PFC + 1
   │                             └BAD BYTE (PFC) ← N-i
   │ σ ← σ ⊕ B_i
   │ τ ← τ   and  (127)
   │ if τ = 0  then SKIP 1
SKIP 1 │ τ ←  (127) and [LOG + T]
   │ τ ←  [EXP + τ + 1]
   └ τ ← τ   ⊕  B_i
 E ← σ
 σ ← (127) and σ
 τ ← (127) and τ
 if   σ = 0  and  τ = 0  then Done
 if  PFC = 2  then ERASURE

SINGLE BYTE  if  σ = 0  or   τ = 0  then DF
 τ ← ([LOG + τ] and (127)) - ([LOG + σ] and (127))  *τ is now n-j*
 if   τ < 0   then  τ ← τ + 127
 if   PFC = 1  and  BAD BYTE (1) ≠ τ  then  DF
 if   τ > N  then  DF
 j  ← N- τ
 B_j ← B_j ⊕ E
 go to DONE
```

```
ERASURE   if σ = 0  then ⎡c ← 0
                         ⎣go to  SKIP 2
          c ← BAD BYTE (1) + ((127) and [ LOG + σ ] )
          if  c > 127  then  c ← c-127
          c ← (127)  and  [EXP + c]                      * c  is σγ^{n-j} *

SKIP 2    c ← c ⊕ τ                                      * c  is σγ^{n-i} + τ*
          if  c = 0  then DF
          A ←  (127)  and  [EXP + Bad Byte (1)]          * γ^{n-j} *
          B ←  (127)  and  [EXP + Bad Byte (2)]          * γ^{n-i} *
          A ←  (127)  and  [LOG + (A⊕B)]                 * NOTE:  A ⊕ B ≠ 0 *
          C ←  (127)  and  [LOG + c]
          E ← c-A
          if   E ← 0  then  E ← E + 127
          E ← (127)  and  [EXP + E]
          If   E = 0  or  E = σ   then  DF
          j ← N-Bad Byte (1)
          i ← N-Bad Byte (2)
          B_i ← B_i ⊕ E
          B_j ← B_j ⊕ B_i ⊕ E
          give  B_i  and  B_j  the odd parity
          go to DONE

   DF     DFF ← 1
   DONE   END
```

CHAPTER 4

BUNDLE CODES - ENCODING AND DECODING

A bundle is a collection of $h + 1$ data blocks of which the first h actually contain data and the final one contains check bytes. The data bytes are encoded in v vertical codewords each a codeword of code C. Since each such codeword has two check bytes, the vertical codewords each contain two bytes from each of the $h + 1$ data blocks in the bundle. The two bytes from the ith vertical codeword occupy positions i and $i + v$ so as to be as far apart as possible (an aid to correcting bursts).

The data blocks of the bundle are first encoded with one of the data block codes depending on the encoding mode EMODE:

EMODE	CODE ON DATA BLOCKS
0	Parity only
1	PRODUCT
2	C

The vertical codewords are then encoded with code C (MODE 2). In fact, these two passes could easily be done in one pass through the data. After encoding the data blocks and the vertical codewords, the vertical checks form the $(h + 1)$th data block. This data block is automatically PRODUCT encoded if 14 vertical codewords $(v = 14)$ are used. In this case the vertical codewords encode the horizontal check bytes. On the other hand, if the EMODE is 2 and code C has been

used horizontally, encoding the horizontal check bytes will not give us a codeword of C as the (h + 1)th-data block. In this case only 13 verticals are used (v = 13) and the horizontal checks are not encoded vertically. We must encode the (h + 1)th data block with code C separately at the end of the encoding when EMODE is 2.

Our model for the encoder is that the data bytes are stored in memory with "gaps" left to accomodate the check bytes once they are calculated. We remarked above that it would be straightforward to process the bytes one at a time, say as they are entered, and to write the check bytes in their appropriate places. This could be done by effectively running v + 1 encoders in parallel - one for the current data block and one for each vertical codeword.

The encoder has subroutines for each of the data block codes so that the code used can easily be selected by the EMODE. The vertical codewords must then be formed. This involves stepping through the data blocks approximately v at a time starting in bytes 0, 1, ..., v-1 of block number 0 . The complication is that if EMODE = 0 or 1 then the vertical codewords cover all the bytes so we simply compose our code word by moving through the data in 14 byte increments. When EMODE = 2 we must first increment by 13(= v) and then by 15 so as to skip the check bytes. What we require then is an encoder (and later a decoder) for code C which accepts a start address then increments by 1 for horizontal encoding, by 14 for vertical encoding in EMODE 0 or 1 and by 13 and 15 alternately in EMODE 2. This has been accomplished by passing the code C encoder

a start address v, a shift SH and a toggle TT. After each byte is processed the shift changes to TT-SH. The values used are:

Case	SH	TT	TT-SH
horizontal	1	2	1
vertical, EMODE 0,1	14	28	14
vertical, EMODE 2	13	28	15

The encoder then has the following structure.

```
SH ← 1 //  TT ← 2  //  L ← N+1
for  v = 0  to  h*N  Step L
 on EMODE gosub ENCODE PARITY, ENCODE PRODUCT, ENCODEC
TT ← 28 //  L = 2*h+2 //  SH ← 14 ;
if EMODE = 2  then  SH ← 13
for  v = 0   to  SH-1
       [ENCODEC
if  EMODE = 2  then
             [SH ← 1 //  TT ← 2 //  L ← N+1 //  u = h*L
             [ENCODEC .
```

We have already discussed the encoders for the three codes. The only change is in ENCODEC. The form of this encoder has to change to accomodate the new variables SH, TT and L. The form of this encoder is the following:

```
          A ← v                                  * start address *
          k ← 0                                  * byte counter *
          s ← 255 //  T ← 0
LOOP      process byte  B(A)
          A ← A + SH
          SH ← TT - SH
          if  k = L - 3   then  CONTINUE
          k ← k + 1 //  go to LOOP
CONTINUE  calculate check bytes
          put them in  B(A)  and  B(A + SH)
```

An implementation in BASIC is contained in Appendix B.

The bundle decoder works much like the encoder. The decoder moves through the constituent codewords in the same order except that the last data block is not treated any differently than the rest. There are several additional parameters to consider for decoding. First, the decoder has a decoding mode DMODE which is set by the teletext decoder. This is 0,1 or 2 depending on the level of coding which the teletext decoder can and wants to use. The decoder first determines EMODE (from the packet structure bytes) and then uses the mode min(EMODE, DMODE).

The decoder also uses the number of missing blocks NM, a parameter passed to the decoder by the prefix processor. The missing

data blocks are written as 0 so that they will produce parity failures for the decoder. If NM > 1 then declare decoding failure.

The decoder returns a flag to indicate if it recognized a decoding failure. Note that decoding failures in the horizontal codewords are not recorded, in this decoder (although a decoder might, if NM = 0, set such a block to 0 and attempt erasure correction).

The other major change in the decoder is that we must calculate the real address of a byte before we can correct it. This leads us to the formula

$$A = v + TT* [i/2] + SH* (i \bmod 2)$$

for the address of the ith byte in the vertical codeword which starts with byte v. For a horizontal codeword the formula is simply A = v + i .

The outline of the decoder is as follows.

```
if  NM > 1  then  DECODING FAILURE
SH ← 1
TT ← 2
L ← N
L1 ← L + 1
MODE ← min(EMODE, DMODE)
for  v = 0  to  h * L1  Step L1
        [on MODE gosub DECPARITY, DECPRODUCT, DECCODEC
```

```
TT ← 28
SH ← 14
L ← 2 * h - 1
if  EMODE = 2  THEN  SH = 13
for  v = 0  to  SH - 1
        ┌ DECODEC
        │
        └ if  DECODING FAILURE FLAG (DE) = 1  then
                                    DECODING FAILURE
```

In fact, DECPARITY is nothing since the parity could only signal a decoding failure in a horizontal codeword and we ignore these. DECPRODUCT is a straightforward decoder and DECODEC is variation of our earlier code C decoder with the changes noted above. All these programs, as implemented in BASIC are in Appendix B. A complete demonstration program is listed in Appendix C.

REFERENCES

[1] "Television Broadcast Videotex", Broadcast Specification No. 14, Issue 1, Provisional, Telecommunications Regulatory Services, Department of Communications, Canada, June 19, 1981.

[2] B.C. Mortimer and M.J. Moore, "More Powerful Error-correction Schemes for the Broadcast Telidon System", Final Report, March, 1983, DSS Contract OSU82-00164, Department of Communications, Ottawa, Canada.

Appendix A.

LOOK-UP TABLE FOR LOG,EXP AND PARITY

PARITY =	LOG	00000000	+ 128	EXP	10000001
+ 1	+ 1	10000000	+ 129	+ 1	00000010
+ 2	+ 2	10000001	+ 130	+ 2	00000100
+ 3	+ 3	01100001	+ 131	+ 3	10001000
+ 4	+ 4	10000010	+ 132	+ 4	00010000
+ 5	+ 5	01000011	+ 133	+ 5	10100000
+ 6	+ 6	01100010	+ 134	+ 6	11000000
+ 7	+ 7	10011010	+ 135	+ 7	00010001
+ 8	+ 8	10000011	+ 136	+ 8	00100010
+ 9	+ 9	01111011	+ 137	+ 9	11000100
+ 10	+ 10	01000100	+ 138	+ 10	10011001
+ 11	+ 11	10101110	+ 139	+ 11	00110010
+ 12	+ 12	01100011	+ 140	+ 12	11100100
+ 13	+ 13	11110011	+ 141	+ 13	01011001
+ 14	+ 14	10011011	+ 142	+ 14	00100011
+ 15	+ 15	00100101	+ 143	+ 15	11000110
+ 16	+ 16	10000100	+ 144	+ 16	00011101
+ 17	+ 17	00000111	+ 145	+ 17	10111010
+ 18	+ 18	01111100	+ 146	+ 18	11110100
+ 19	+ 19	11111000	+ 147	+ 19	01111001
+ 20	+ 20	01000101	+ 148	+ 20	11100011
21	+ 21	10110100	+ 149	+ 21	01010111
22	+ 22	10101111	+ 150	+ 22	00111111
+ 23	+ 23	01010101	+ 151	+ 23	11111110
+ 24	+ 24	01100100	+ 152	+ 24	11101101
+ 25	+ 25	10001010	+ 153	+ 25	01001011
+ 26	+ 26	11110100	+ 154	+ 26	00000111
+ 27	+ 27	01011101	+ 155	+ 27	10001110
+ 28	+ 28	10011100	+ 156	+ 28	00011100
+ 29	+ 29	00010000	+ 157	+ 29	10111000
+ 30	+ 30	00100110	+ 158	+ 30	11110000
+ 31	+ 31	11010000	+ 159	+ 31	01110001
+ 32	+ 32	10000101	+ 160	+ 32	01110011
+ 33	+ 33	00110010	+ 161	+ 33	11110111
+ 34	+ 34	00001000	+ 162	+ 34	11111111
+ 35	+ 35	10001110	+ 163	+ 35	01101111
+ 36	+ 36	01111101	+ 164	+ 36	11001111
+ 37	+ 37	11000001	+ 165	+ 37	00001111
+ 38	+ 38	11111001	+ 166	+ 38	00011110
+ 39	+ 39	01110001	+ 167	+ 39	10111100
+ 40	+ 40	01000110	+ 168	+ 40	11111000
+ 41	+ 41	11011000	+ 169	+ 41	01100001
+ 42	+ 42	10110101	+ 170	+ 42	01010011
+ 43	+ 43	01101011	+ 171	+ 43	10110111
+ 44	+ 44	10110000	+ 172	+ 44	01101110
+ 45	+ 45	00111111	+ 173	+ 45	11001101
+ 46	+ 46	01010110	+ 174	+ 46	10001011
47	+ 47	10111101	+ 175	+ 47	00010110
+ 48	+ 48	01100101	+ 176	+ 48	10101100
+ 49	+ 49	11001000	+ 177	+ 49	01011000
+ 50	+ 50	10001011	+ 178	+ 50	00100001

51	+ 51	01101000	+ 179	+ 51	11000010
+ 52	+ 52	11110101	+ 180	+ 52	00010101
+ 53	+ 53	01011010	+ 181	+ 53	10101010
+ 54	+ 54	01011110	+ 182	+ 54	11010100
+ 55	+ 55	10101011	+ 183	+ 55	00111001
+ 56	+ 56	10011101	+ 184	+ 56	01110010
+ 57	+ 57	00110111	+ 185	+ 57	11110101
+ 58	+ 58	00010001	+ 186	+ 58	11111011
+ 59	+ 59	11001011	+ 187	+ 59	01100111
+ 60	+ 60	00100111	+ 188	+ 60	11011111
+ 61	+ 61	11101101	+ 189	+ 61	00101111
+ 62	+ 62	11010001	+ 190	+ 62	01011110
+ 63	+ 63	00010110	+ 191	+ 63	10101101
+ 64	+ 64	10000110	+ 192	+ 64	01011010
+ 65	+ 65	01110111	+ 193	+ 65	10100101
+ 66	+ 66	00110011	+ 194	+ 66	11001010
+ 67	+ 67	11010100	+ 195	+ 67	00000101
+ 68	+ 68	00001001	+ 196	+ 68	10001010
+ 69	+ 69	11011100	+ 197	+ 69	00010100
+ 70	+ 70	10001111	+ 198	+ 70	00101000
+ 71	+ 71	01001111	+ 199	+ 71	11010000
+ 72	+ 72	01111110	+ 200	+ 72	10110001
+ 73	+ 73	11100000	+ 201	+ 73	01100010
+ 74	+ 74	11000010	+ 202	+ 74	01010101
75	+ 75	00011001	+ 203	+ 75	10111011
76	+ 76	11111010	+ 204	+ 76	01110110
+ 77	+ 77	00101101	+ 205	+ 77	11111101
+ 78	+ 78	01110010	+ 206	+ 78	11101011
+ 79	+ 79	10100100	+ 207	+ 79	01000111
+ 80	+ 80	01000111	+ 208	+ 80	10011111
+ 81	+ 81	11100111	+ 209	+ 81	00111110
+ 82	+ 82	11011001	+ 210	+ 82	01111100
+ 83	+ 83	00101010	+ 211	+ 83	11101001
+ 84	+ 84	10110110	+ 212	+ 84	01000011
+ 85	+ 85	01001010	+ 213	+ 85	10010111
+ 86	+ 86	01101100	+ 214	+ 86	10101110
+ 87	+ 87	10010101	+ 215	+ 87	01011100
+ 88	+ 88	10110001	+ 216	+ 88	00101001
+ 89	+ 89	00001101	+ 217	+ 89	11010010
+ 90	+ 90	01000000	+ 218	+ 90	10110101
+ 91	+ 91	11110000	+ 219	+ 91	01101010
+ 92	+ 92	01010111	+ 220	+ 92	11000101
+ 93	+ 93	11101010	+ 221	+ 93	00011011
+ 94	+ 94	10111110	+ 222	+ 94	00110110
+ 95	+ 95	00111100	+ 223	+ 95	11101100
+ 96	+ 96	01100110	+ 224	+ 96	11001001
+ 97	+ 97	10101001	+ 225	+ 97	00000011
+ 98	+ 98	11001001	+ 226	+ 98	00000110
+ 99	+ 99	00010100	+ 227	+ 99	10001100
+ 100	+ 100	10001100	+ 228	+ 100	00011000

+ 101	+ 101	01101111	+ 229	+ 101	10110000
+ 102	+ 102	01101001	+ 230	+ 102	11100000
+ 103	+ 103	10111011	+ 231	+ 103	01010001
+ 104	+ 104	11110110	+ 232	+ 104	00110011
105	+ 105	01010011	+ 233	+ 105	11100110
+ 106	+ 106	01011011	+ 234	+ 106	11011101
+ 107	+ 107	11001110	+ 235	+ 107	00101011
+ 108	+ 108	01011111	+ 236	+ 108	11010110
+ 109	+ 109	10011000	+ 237	+ 109	00111101
+ 110	+ 110	10101100	+ 238	+ 110	01111010
+ 111	+ 111	00100011	+ 239	+ 111	11100101
+ 112	+ 112	10011110	+ 240	+ 112	01011011
+ 113	+ 113	00011111	+ 241	+ 113	10100111
+ 114	+ 114	00111000	+ 242	+ 114	11001110
+ 115	+ 115	10100000	+ 243	+ 115	00001101
+ 116	+ 116	00010010	+ 244	+ 116	10011010
+ 117	+ 117	10111001	+ 245	+ 117	00110100
+ 118	+ 118	11001100	+ 246	+ 118	01101000
+ 119	+ 119	00100001	+ 247	+ 119	11000001
+ 120	+ 120	00101000	+ 248	+ 120	10010011
+ 121	+ 121	10010011	+ 249	+ 121	00100110
+ 122	+ 122	11101110	+ 250	+ 122	01001100
+ 123	+ 123	00111010	+ 251	+ 123	10001001
+ 124	+ 124	11010010	+ 252	+ 124	00010010
+ 125	+ 125	01001101	+ 253	+ 125	10100100
+ 126	+ 126	00010111	+ 254	+ 126	11001000
+ 127	+ 127	10100010	+ 255	+ 127	00000001

```
1 'TABLE MAKER
2 '
3 'This program creates the LOG,EXP, and PARITY
4 'look up tables.LOG starts at 4000, EXP starts
5 'at 4128, and PARITY at 4000.
6 '
7 '
8 S=1
9 A=4000
10 FOR I= 0 TO 127
11 X=I:GOSUB 28
12 'entry in the EXP table.
13 POKE A+128+I,S+128*(1-P)
14 X=S:GOSUB 28
15 'entry in the LOG table.
16 POKE A+S,I+128*P
17 '
18 'Make S the next power of Gamma.
19 '
20 S=S*2:IF S>=128 THEN S=(S OR 145)-(S AND 145)
21 '
22 NEXT I
23 'Enter the LOG of 1.
24 POKE A+1,128
25 END ' end of TABLE MAKER.
26 '
27 '
28 ' ROUTINE to calculaate the binary parity of X<256.
29 '
30 P=0
31 FOR J=7 TO 0 STEP -1
32 IF X>=INT(2^J) THEN P=1-P:X=X-INT(2^J):NEXT ELSE NEXT
33 RETURN
```

This appendix contains implementations in BASIC for some of the more important algorithms of the text. These programs try to stay close to machine language techniques.

```
1 N=27
2 PFC=0:S=255:T=0:FOR I=0 TO N
3 IF PEEK(PARITY+B(I)) < 128 THEN IF PFC=2 THEN 45 ELSE PFC=PFC+
1:BADBYTE(PFC)=N-I
4 S=(S OR B(I))-(S AND B(I))
5 T=T AND 127:IF T=0 THEN 8
6 T=PEEK(LG+T):T=T AND 127
7 T=PEEK(EX+T+1)
8 T=(T OR B(I))-(T AND B(I))
9 NEXT I
10 ERR=S
11 S=S AND 127:T=T AND 127
12 IF S=0 AND T=0 THEN 47
13 REM SINGLE BYTE CORRECTION
14 IF PFC =2 THEN 24
15 IF S=0 OR T=0 THEN 45
16 A=PEEK(LG+T) AND 127
17 B=PEEK(LG+S) AND 127
18 A=A-B
19 IF A<0 THEN A=A+127
20 IF PFC=1 AND BADBYTE(1) <> A THEN 45
21 IF A>N THEN 45
22 I=N-A
23 B(I)=(B(I) OR ERR)-(B(I) AND ERR): GOTO 47
24 REM DOUBLE BYTE ERASURE
25 A=PEEK(EX+BADBYTE(1)):B=PEEK(EX+BADBYTE(2))
26 A=A AND 127:B=B AND 127
27 IF S=0 THEN C=0:GOTO 31
28 C=PEEK(LG+S):C=C AND 127:C=C+BADBYTE(1)
29 IF C>127 THEN C=C-127
30 C=127 AND PEEK(EX+C)
31 C=(C OR T)-(C AND T)
32 IF C=0 THEN 45
33 C=127 AND PEEK(LG+C)
34 A=127 AND PEEK(LG+((A OR B)-(A AND B)))
35 E=C-A:IF E<0 THEN E=E+127
36 E=PEEK(EX+E) AND 127
37 IF E=0 OR E=S THEN 45
38 J=N-BADBYTE(1)
39 I=N-BADBYTE(2)
40 B(I)=(B(I) OR E)-(B(I) AND E)
41 B(J)=(B(J) OR E)-(B(J) AND E):B(J)=(B(J) OR S)-(B(J) AND S)
42 IF PEEK(PARITY+B(I))<128 THEN B(I)=B(I) OR 128
43 IF PEEK(PARITY+B(J))<128 THEN B(J)=B(J) OR 128
44 GOTO 47
45 REM DECODING FAILURE (DF)
46 DECFAIL=1:GOTO49
47 REM DONE
48 DECFAIL=0:GOTO49
49 END
```

PROGRAM B.1 : A decoder for code C.

PROGRAM B.2 : An encoder for the Bundle Code

```
1 'encoder...."
2 '
3 SH=1:TT=2:L=N+1
4 FOR U=0 TO H*L-L STEP L
5 ON EMODE+1 GOSUB 14,19,28
6 NEXT U
7 TT=28:L=2*H+2:SH=14:IF EMODE=2 THEN SH=13
8 FOR U=0 TO SH-1
9 GOSUB 28 :NEXT U
10 IF M=2 THEN SH=1:TT=2:L=N+1:U=H*L:GOSUB 28
11 RETURN 'end of encoder
12 '
13 '
14 'PARITY ONLY encoder
15 '
16 FOR A=U TO U+N:GOSUB 55:NEXT:RETURN
17 '
18 '
19 'PRODUCT encoder
20 '
21 S=255
22 FOR A=U TO U+N-1
23 GOSUB 55:S=(S OR B(A))-(S AND B(A)):NEXT
24 B(U+N)=S:A=U+N
25 RETURN
26 '
27 '
28 'CODE C encoder
29 '
30 K=0:A=U 'COUNTER AND START
31 S=255:T=0
32 GOSUB 55
33 S=(S OR B(A))-(S AND B(A))
34 T=T AND 127:IF T=0 THEN 36
35 T=127 AND (PEEK(LG+T)+1):T=127 AND (PEEK(EX+T))
36 T=(T OR B(A))-(T AND B(A))
37 A=A+SH:SH=TT-SH:IF K=L-3 THEN 39
38 K=K+1:GOTO 32
39 T=127 AND T:S=127 AND S
40 IF T=0 THEN 43
41 T=127 AND PEEK(LG+T):T=T+2:IF T>127 THEN T=T-127
42 T=127 AND PEEK(EX+T)
43 T=(T OR S)-(T AND S)
44 IF T=0 THEN B(N-1)=0:GOTO48
45 T=(127 AND PEEK(LG+T))+30
46 IF T>126 THEN T=T-127
47 B(A)=127 AND PEEK(EX+T)
48 B(A+SH)=(B(A) OR S)-(B(A) AND S)
49 GOSUB 55:A=A+SH:GOSUB 55
50 RETURN ' END OF CODEC ENCODER
51 '
52 '
53 ' give B(A) odd parity
54 '
55 IF PEEK(PARITY+B(A))<128 THEN B(A)=(B(A) OR 128)-(B(A) AND 128)
56 RETURN
```

PROGRAM B.3 : A Decoder for the Bundle Code.

```
1 'decoder..........
2 '
3 '
4 IF NM>1 THEN 80 'DEC.FAIL.
5 SH=1:TT=2:L=N:L1=L+1
6 FOR U=0 TO H*L1 STEP L1
7 M=EMODE:IF DMODE<EMODE THEN M=DMODE
8 ON M+1 GOSUB 17,22,32
9 NEXT U
10 TT=28:SH=14:L=2*H+1 :IF EMODE=2 THEN SH=13
11 FOR U=0 TO SH-1
12 GOSUB 32
13 NEXT U
14 RETURN 'END OF DECODER
15 '
16 '
17 'PARITY DECODER
18 '
19 RETURN
20 '
21 '
22 'PRODUCT DECODER
23 '
24 PFC=0:S=255
25 FOR A=U TO U+N:IF PEEK(PARITY+B(A))<128 THEN IF PFC>0 THEN DE=1:RE
TURN ELSE PFC=1:BADBYTE(1)=A
26 S=(S OR B(A))-(S AND B(A)):NEXT
27 IF S=0 AND PFC=0 THEN DE=0:RETURN
28 W=0:FOR Q=0 TO 7:IF S=INT(2^Q) THEN W=1:NEXT ELSE NEXT
29 IF W<>1 THEN DE=1:RETURN ELSE A=BADBYTE(1):B(A)=(S OR B(A))-(S AND
 B(A)):RETURN
30 '
31 '
32 'CODE C DECODER
33 '
34 PFC=0:S=255:T=0:I=0:A=U
35 IF PEEK(PARITY+B(A)) < 128 THEN IF PFC=2 THEN 80 ELSE PFC=PFC+1:BA
DBYTE(PFC)=L-I
36 S=(S OR B(A))-(S AND B(A))
37 T=T AND 127:IF T=0 THEN 40
38 T=PEEK(LG+T):T=T AND 127
39 T=PEEK(EX+T+1)
40 T=(T OR B(A))-(T AND B(A))
41 A=A+SH:SH=TT-SH:IF I=L THEN 44
42 I=I+1:GOTO 35
43 '
44 T=T AND 127
45 ERR=S
46 S=S AND 127:T=T AND 127
47 IF S=0 AND T=0 THEN 82
48 ' single byte correction
49 IF PFC =2 THEN 59
50 IF S=0 OR T=0 THEN 80
```

```
51 A=PEEK(LG+T) AND 127
52 B=PEEK(LG+S) AND 127
53 A=A-B
54 IF PFC=1 AND BADBYTE(1) <> A THEN 80
55 IF A<0 OR A>L THEN 80
56 A=L-A
57 A=U+TT*INT(A/2)+SH*(A-2*INT(A/2))
58 B(A)=(B(A) OR ERR)-(B(A) AND ERR): GOTO 82
59 ' double byte erasure correction
60 A=PEEK(EX+BADBYTE(1)):B=PEEK(EX+BADBYTE(2))
61 A=A AND 127:B=B AND 127
62 IF S=0 THEN C=0:GOTO 66
63 C=PEEK(LG+S):C=C AND 127:C=C+BADBYTE(1)
64 IF C>127 THEN C=C-127
65 C=127 AND PEEK(EX+C)
66 C=(C OR T)-(C AND T)
67 IF C=0 THEN 80
68 C=127 AND PEEK(LG+C)
69 A=127 AND PEEK(LG+((A OR B)-(A AND B)))
70 E=C-A:IF E<0 THEN E=E+127
71 E=PEEK(EX+E) AND 127
72 IF E=0 OR E=S THEN 80
73 I=L-BADBYTE(1):I=U+TT*INT(I/2)+SH*(I AND 1)
74 J=L-BADBYTE(2):J=U+TT*INT(J/2)+SH*(J AND 1)
75 B(J)=(B(J) OR E)-(B(J) AND E)
76 B(I)=(B(I) OR E)-(B(I) AND E):B(I)=(B(I) OR S)-(B(I) AND S)
77 IF PEEK(PARITY+B(I))<128 THEN B(I)=B(I) OR 128
78 IF PEEK(PARITY+B(J))<128 THEN B(J)=B(J) OR 128
79 GOTO 82
80 ' decoding failure
81 DECFAIL=1:GOTO84
82 ' decoding done
83 DECFAIL=0:GOTO84
84 RETURN
```

PROGRAM B.4 A demonstration program for the complete

Bundle encoding and decoding.


```
1 REM THE BUNDLE CODEC
2 CLEAR 512:DIM B(600)
3 GOSUB 18
4 '*** menu ***
5 PRINT@0,"<S>ET PARAMETERS &DEFINE TEXT"
6 PRINT"<A>LTER BYTES OR REMOVE BLOCKS"
7 PRINT"<E>NCODE BUNDLE -- MODE:";EMODE
8 PRINT"<D>ECODE BUNDLE -- MODE:";DMODE
9 PRINT"<Q>UIT"
10 PRINTSTRING$(32,"-");
11 A$=INKEY$:IF A$="" THEN 11 ELSE ON INSTR(1,"SAEDQ",A$) GOSUB 18,32
,41,83,39
12 GOTO 4
13 ' DISPLAY BUNDLE
14 FOR I=0 TO H:PRINT@192+32*I, I;:IF NM=1 AND MINDEX=I THEN 16
15 FOR J=0 TO N:PRINT@195+I*32+J,CHR$(B(I*N1+J) AND 127);:NEXT:PRINT
16 NEXT
17 RETURN
18 'SET PARAMETERS ANF DEFINE DATA BYTES
19 CLS:PRINT"****** setup ******":PRINT:PRINT
20 N=27:N1=28
21 INPUT"LENGTH OF BUNDLE";H:H=H-1
22 INPUT "ENCODING MODE(0,1,2):";EMODE:INPUT "DECODING MODE(0,1,2):";
DMODE
23 PARITY=4000:LG=4000:EX=4128
24 N1=N+1:NM=0
25 RESTORE:READ B$
26 C$=B$:PRINT:PRINT"one momemt please"
27 FOR I=0 TO H-1:FOR J=0 TO N-EMODE:B(I*N1+J)=ASC(LEFT$(C$,1)):C$=RI
GHT$(C$,LEN(C$)-1):IF LEN(C$)=0 THENC$=B$
28 NEXT J,I
29 DATA "MARY HAD A LITTLE LAMB AND FOUND THAT IT WENT WELL WITH MINT
 SAUCE.MYSELF, I FOUND THAT IT WAS TOO GREASY."
30 GOSUB 13
31 RETURN
32 'alterations *********
33 PRINT@0,"alterations......"
34 INPUT "BLOCK , BYTE , ERROR";I,J,S
35 INPUT "# MISSING,MISS.INDEX:";NM,MINDEX
36 A=I*N1+J:B(A)=(B(A) OR S)-(B(A) AND S):GOSUB 158
37 IF NM=1 THEN FOR A=MINDEX*N1 TO MINDEX*N1+N:B(A)=0:GOSUB 158:NEXT
38 RETURN
39 'END
40 END
41 'encoder...."
42 PRINT@0,"encoding":PRINT:PRINT:PRINT:PRINT
43 SH=1:TT=2:L=N+1
44 FOR U=0 TO H*L-L STEP L
45 ON EMODE+1 GOSUB 52,54,60
46 NEXT U
47 TT=28:L=2*H+2:SH=14:IF EMODE=2 THEN SH=13
48 FOR U=0 TO SH-1
49 GOSUB 60 :NEXT U
50 IF M=2 THEN SH=1:TT=2:L=N+1:U=H*L:GOSUB 60
```

```
51 RETURN 'END OF ENCODER
52 'PARITY ONLY ENCODER
53 FOR A=U TO U+N:GOSUB 82:NEXT:RETURN
54 'PRODUCT ENCODER
55 S=255
56 FOR A=U TO U+N-1
57 GOSUB 82:S=(S OR B(A))-(S AND B(A)):NEXT
58 B(U+N)=S:A=U+N:GOSUB 158
59 RETURN
60 'CODE C ENCODER
61 K=0:A=U 'COUNTER AND START
62 S=255:T=0
63 GOSUB 82
64 S=(S OR B(A))-(S AND B(A))
65 T=T AND 127:IF T=0 THEN 67
66 T=127 AND (PEEK(LG+T)+1):T=127 AND (PEEK(EX+T))
67 T=(T OR B(A))-(T AND B(A))
68 A=A+SH:SH=TT-SH:IF K=L-3 THEN 70
69 K=K+1:GOTO 63
70 T=127 AND T:S=127 AND S
71 IF T=0 THEN 74
72 T=127 AND PEEK(LG+T):T=T+2:IF T>127 THEN T=T-127
73 T=127 AND PEEK(EX+T)
74 T=(T OR S)-(T AND S)
75 IF T=0 THEN B(N-1)=0:GOTO79
76 T=(127 AND PEEK(LG+T))+30
77 IF T>126 THEN T=T-127
78 B(A)=127 AND PEEK(EX+T)
79 B(A+SH)=(B(A) OR S)-(B(A) AND S)
80 GOSUB 82:A=A+SH:GOSUB 82
81 RETURN ' END OF CODEC ENCODER
82 GOSUB 158:IF PEEK(PARITY+B(A))<128 THEN B(A)=(B(A) OR 128)-(B(A) A
ND 128):RETURN ELSE RETURN
83 PRINT@0,"decoding ***********":PRINT:PRINT:PRINT:PRINT
84 IF NM>1 THEN 153 'DEC.FAIL.
85 SH=1:TT=2:L=N:L1=L+1
86 FOR U=0 TO H*L1 STEP L1
87 M=EMODE:IF DMODE<EMODE THEN M=DMODE
88 ON M+1 GOSUB 97,99,106
89 PRINT@32,"decoding failure flag:";DE
90 NEXT U
91 TT=28:SH=14:L=2*H+1 :IF EMODE=2 THEN SH=13
92 FOR U=0 TO SH-1
93 GOSUB 106
94 PRINT@32,"decoding failure flag:";DE
95 NEXT U
96 RETURN 'END OF DECODER
97 'PARITY DECODER
98 RETURN
99 'PRODUCT DECODER
100 PFC=0:S=255
```

```
101 FOR A=U TO U+N:GOSUB 158:IF PEEK(PARITY+B(A))<128 THEN IF PFC>0 T
HEN DE=1:RETURN ELSE PFC=1:BADBYTE(1)=A
102 S=(S OR B(A))-(S AND B(A)):NEXT
103 IF S=0 AND PFC=0 THEN DE=0:RETURN
104 W=0:FOR Q=0 TO 7:IF S=INT(2^Q) THEN W=1:NEXT ELSE NEXT
105 IF W<>1 THEN DE=1:RETURN ELSE A=BADBYTE(1):GOSUB 158:B(A)=(S OR B
(A))-(S AND B(A)):GOSUB 158:RETURN
106 'CODE C DECODER
107 PFC=0:S=255:T=0:I=0:A=U
108 GOSUB 158:IF PEEK(PARITY+B(A)) < 128 THEN IF PFC=2 THEN 153 ELSE
PFC=PFC+1:BADBYTE(PFC)=L-I
109 S=(S OR B(A))-(S AND B(A))
110 T=T AND 127:IF T=0 THEN 113
111 T=PEEK(LG+T):T=T AND 127
112 T=PEEK(EX+T+1)
113 T=(T OR B(A))-(T AND B(A))
114 A=A+SH:SH=TT-SH:IF I=L THEN 116
115 I=I+1:GOTO 108
116 T=T AND 127
117 ERR=S
118 S=S AND 127:T=T AND 127
119 IF S=0 AND T=0 THEN 155
120 REM SINGLE BYTE CORRECTION
121 IF PFC =2 THEN 131
122 IF S=0 OR T=0 THEN 153
123 A=PEEK(LG+T) AND 127
124 B=PEEK(LG+S) AND 127
125 A=A-B
126 IF PFC=1 AND BADBYTE(1) <> A THEN 153
127 IF A<0 OR A>L THEN 153
128 A=L-A
129 A=U+TT*INT(A/2)+SH*(A-2*INT(A/2))
130 B(A)=(B(A) OR ERR)-(B(A) AND ERR):GOSUB 158: GOTO 155
131 REM DOUBLE BYTE ERASURE
132 A=PEEK(EX+BADBYTE(1)):B=PEEK(EX+BADBYTE(2))
133 A=A AND 127:B=B AND 127
134 IF S=0 THEN C=0:GOTO 138
135 C=PEEK(LG+S):C=C AND 127:C=C+BADBYTE(1)
136 IF C>127 THEN C=C-127
137 C=127 AND PEEK(EX+C)
138 C=(C OR T)-(C AND T)
139 IF C=0 THEN 153
140 C=127 AND PEEK(LG+C)
141 A=127 AND PEEK(LG+((A OR B)-(A AND B)))
142 E=C-A:IF E<0 THEN E=E+127
143 E=PEEK(EX+E) AND 127
144 IF E=0 OR E=S THEN 153
145 I=L-BADBYTE(1):I=U+TT*INT(I/2)+SH*(I AND 1)
146 J=L-BADBYTE(2):J=U+TT*INT(J/2)+SH*(J AND 1)
147 B(J)=(B(J) OR E)-(B(J) AND E)
148 B(I)=(B(I) OR E)-(B(I) AND E):B(I)=(B(I) OR S)-(B(I) AND S)
149 IF PEEK(PARITY+B(I))<128 THEN B(I)=B(I) OR 128
150 IF PEEK(PARITY+B(J))<128 THEN B(J)=B(J) OR 128
```

```
151 A=I:GOSUB158:A=J:GOSUB158
152 GOTO 155
153 REM DECODING FAILURE (DF)
154 DECFAIL=1:GOTO157
155 REM DONE
156 DECFAIL=0:GOTO157
157 RETURN
158 Y=INT(A/N1):X=A-N1*Y
159 LL=195+X+32*Y
160 PRINT@LL,"*";:PRINT@LL,CHR$(B(A) AND127);:RETURN
```