Capítulo IV

Códigos Corretores de Erro

Vimos no Capítulo I que o Teorema Fundamental de Shannon estabelece a existência de um código corretor de erro tal que a informação pode ser transmitida através do canal de comunicação com uma taxa de erro arbitrariamente baixa caso a taxa de transmissão R $[bits/s]$ seja menor ou igual que a capacidade do canal C $[bits/s]$.

Neste capítulo estudaremos como construir tais códigos. Especificamente, estudaremos os membros mais importantes de duas grandes classes de códigos para correção de erro: os códigos de bloco e os códigos convolucionais.

Estudaremos também, através da análise de um sistema para transmissão digital específico, a classe de códigos concatenados. Os códigos concatenados são formados pela combinação de um código de bloco seguido por um código convolucional, interligados através de um "embaralhador" de blocos de bits denominado *interleaver*. O sistema para transmissão digital escolhido para análise e estudo dos códigos concatenados é o sistema de televisão digital 8-VSB da ATSC (ATSC – *Advanced Television Systems Committee*) [ATSC1].

É importante lembrar que o processo de correção de erros através de codificação/decodificação é realizado no Codificador/Decodificador de Canal (ver Figura 1.1 do Capítulo I). Algumas vezes este processo é referido como FEC (FEC – *Forward Error Correction*).

Um processo alternativo para correção de erros em um sistema digital é o processo ARQ (ARQ – *Automatic Repeat Request*). O processo FEC procura inferir e imediatamente corrigir erros pelas características do sinal recebido, enquanto que o processo ARQ simplesmente detecta a existência de erros no receptor e solicita ao transmissor que envie a informação original novamente. Isto implica na existência de um canal de *feedback* do receptor para o transmissor, de modo que o processo ARQ não é muito utilizado exceto quando o objetivo são baixíssimas taxas de erro [Taub].

4.1 Códigos de Bloco

Podemos considerar um código de bloco de maneira semelhante àquela que adotamos para os códigos compressores por entropia vistos no Capítulo III. Ou seja, um código de bloco pode ser considerado como um operador $\boldsymbol{\theta}\{\cdot\}$, tal que $\mathbf{C} = \boldsymbol{\theta}\{\mathbf{X}\}$, onde $\mathbf{X} = \{\underline{x}_i\} = \{\underline{x}_0, \underline{x}_1, \cdots, \underline{x}_{M-1}\}$ é o conjunto de M possíveis **mensagens** $\underline{x}_i$ a serem codificadas e $\mathbf{C} = \{\underline{c}_i\} = \{\underline{c}_0, \underline{c}_1, \cdots, \underline{c}_{M-1}\}$ é o conjunto de M possíveis **palavras-código** $\underline{c}_i$ resultantes da codificação, com $i = 0, 1, \cdots, M-1$. O operador $\boldsymbol{\theta}\{\cdot\}$ efetua um mapeamento unívoco entre cada mensagem $\underline{x}_i$ e a respectiva palavra-código $\underline{c}_i$.

O **conjunto de caracteres do código** ou **alfabeto do código** é o conjunto $\mathbf{A} = \{a_0, a_1, \cdots, a_{D-1}\}$ composto por D elementos, de cuja composição são formadas cada mensagem e sua respectiva palavra-código.

No contexto de códigos corretores de erro, cada mensagem $\underline{x}_i \in \mathbf{X}$ é considerada como um vetor $\underline{x}_i = [x_{i(k-1)} \quad x_{i(k-2)} \quad \cdots \quad x_{i1} \quad x_{i0}]$ de k componentes $x_{ij} \in \mathbf{A}$, $j = k-1, k-2, \cdots, 1, 0$. Visto que os k componentes da i-ésima mensagem $\underline{x}_i$ pertencem ao alfabeto $\mathbf{A}$, é válida a relação de pertinência $\underline{x}_i \in \mathbf{A}^k$.

Da mesma forma, cada palavra-código $\underline{c}_i \in \mathbf{C}$ é considerada como um vetor $\underline{c}_i = [c_{i(n-1)} \quad c_{i(n-2)} \quad \cdots \quad c_{i1} \quad c_{i0}]$ de n componentes $c_{ij} \in \mathbf{A}$, $j = n-1, n-2, \cdots, 1, 0$. Visto que os n componentes da i-ésima palavra-código $\underline{c}_i$ pertencem ao alfabeto $\mathbf{A}$, é válida a relação de pertinência $\underline{c}_i \in \mathbf{A}^n$.

Por exemplo, a palavra-código binária 0101, de $n = 4$ bits, é representada pelo vetor $\underline{c} = [0 \quad 1 \quad 0 \quad 1]$, $\underline{c} \in \mathbf{A}^4$, $\mathbf{A} = \{0,1\}$.

Para um código D – ário cujo D é uma potência inteira de 2, (i.e. $D = 2^b$, onde b é um inteiro positivo) cada caractere D – ário terá uma representação binária equivalente formada por uma seqüência de b bits. Portanto, um tal código D – ário cujo tamanho da palavra-código é de N caracteres D – ários pode ser mapeado em um código binário cujo tamanho da palavra-código é $n = bN$. Como, em geral, $D = 2^b$ em sistemas práticos, neste estudo focalizaremos os códigos binários ($\mathbf{A} = \{0,1\}$), visto que a qualquer instante o código D – ário pode ser mapeado no código binário equivalente e vice-versa.

Exemplo 4.1: Determine o código binário equivalente ao código $\boldsymbol{\theta}\{\cdot\}$ abaixo.

Mensagem $\underline{x}_i$	Palavra-código $\underline{c}_i$ associada a $\underline{x}_i$ por $\underline{c}_i = \boldsymbol{\theta}\{\underline{x}_i\}$
00	00
01	03
02	11
03	12
10	21
11	22
12	30
13	33

Solução:

O código original é quaternário ($D = 4$).

O número de caracteres quaternários utilizado na representação das palavras-código do código quaternário é $N = 2$.

Existem $M = 8$ mensagens quaternárias, logo o código binário equivalente necessita $k = \log_2 M = 3$ bits em cada mensagem binária para representá-las. Assim, ao mapear o conjunto de M mensagens quaternárias no conjunto de M mensagens binárias, obtemos:

Mensagem Quaternária	Mensagem Binária Equivalente de $k = 3$ bits
00	000
01	001
02	010
03	011
10	100
11	101
12	110
13	111

O número de bits necessários a cada palavra-código binária equivalente é $n = bN$, onde $b = \log_2 D = 2$. Logo $n = bN = 2 \times 2 = 4$. Assim, ao mapear o conjunto de M palavras-código quaternárias no conjunto de M palavras-código binárias, obtemos:

Palavra-Código Quaternária	Palavra-Código Binária Equivalente de $n = 4$ bits
00	0000
03	0011
11	0101
12	0110
21	1001
22	1010
30	1100
33	1111

Portanto o código binário equivalente ao código quaternário é

Tabela 4.1: Código binário equivalente.

Mensagem $\underline{x}_i$ de $k = 3$ bits	Palavra-código $\underline{c}_i$ de $n = 4$ bits associada a $\underline{x}_i$ por $\underline{c}_i = \boldsymbol{\theta}\{\underline{x}_i\}$.
000	0000
001	0011
010	0101
011	0110
100	1001
101	1010
110	1100
111	1111

4.2 Códigos de Bloco Binários

Um código de bloco binário $\boldsymbol{\theta}\{\}$ mapeia um conjunto $\mathbf{X} = \{\underline{x}_i\} = \{\underline{x}_0, \underline{x}_1, \cdots, \underline{x}_{M-1}\}$ de $M = 2^k$ mensagens binárias, cada uma delas com k bits, em um conjunto $\mathbf{C} = \{\underline{c}_i\} = \{\underline{c}_0, \underline{c}_1, \cdots, \underline{c}_{M-1}\}$ de M palavras-código binárias, cada uma delas com n bits, onde $n > k$. Um código de bloco $\boldsymbol{\theta}\{\}$ binário cujas mensagens a serem codificadas apresentam

k bits e são mapeadas em palavras-código de *n* bits é representado pelo operador $\boldsymbol{\theta}(n,k)\{\cdot\}$ ou simplesmente $\boldsymbol{\theta}(n,k)$.

Um código $\boldsymbol{\theta}(n,k)$ é **sistemático** quando cada palavra-código de *n* bits é formada pelos *k* bits da respectiva mensagem associada, acrescidos (por justaposição) de *r* bits adicionais destinados ao controle e correção de erros, denominados de **bits de paridade**. Portanto, em um código sistemático cada mensagem contendo *k* bits de informação é expandida em uma palavra-código de $n = k + r$ bits onde *r* é o número de bits representativos da informação redundante adicionada visando o controle e correção de erro. Um código $\boldsymbol{\theta}(n,k)$ é **não-sistemático** quando nas palavras-códigos de *n* bits não aparecem explicitamente representados os *k* bits de informação da respectiva mensagem associada. Na Seção 4.2.3 veremos como converter um código não-sistemático em um código sistemático. Em função disto, inicialmente restringiremos nossa atenção aos códigos sistemáticos. Por exemplo, o código $\boldsymbol{\theta}(4,3)$ da Tabela 4.1 é sistemático porque cada palavra-código $\underline{c}_i$ de $n = 4$ bits é formada pela justaposição de 1 bit de paridade aos $k = 3$ bits de informação da mensagem $\underline{x}_i$ associada.

Observe que, como $n > k$, no conjunto de todas as 2^n possíveis palavras-códigos de *n* bits existem $2^n - 2^k$ elementos que não são associados a qualquer elemento do conjunto $\mathbf{X} = \{\underline{x}_i\} = \{\underline{x}_0, \underline{x}_1, \cdots, \underline{x}_{M-1}\}$ de $M = 2^k$ mensagens binárias de *k* bits. Por exemplo, para o código binário $\boldsymbol{\theta}(4,3)$ da Tabela 4.1 existem $2^n - 2^k = 2^4 - 2^3 = 8$ elementos no conjunto de todas as $2^n = 2^4 = 16$ possíveis palavras-códigos de 4 bits sem associação com qualquer mensagem do conjunto $\mathbf{X} = \{000,001,010,011,100,101,110,111\}$.

Em geral é desejável que o tempo $n\tau_s$ de duração de uma palavra-código seja igual (idealmente menor ou igual) ao tempo de duração $k\tau_x$ de uma mensagem, onde τ_s representa a largura (duração) dos bits em uma palavra-código e τ_x representa a largura dos bits em uma mensagem. Assim, se $n\tau_s = k\tau_x$, então a razão de codificação R_c (ver Capítulo I) de um código de bloco é $R_c = k/n = \tau_s/\tau_x$.

O **peso** de uma palavra-código é definido como o número de dígitos "1" nela presentes. Em geral cada palavra-código tem seu próprio peso. O conjunto de todos os pesos de um código constitui a **distribuição de pesos** do código. Quando todas as *M* palavras-código tem pesos iguais o código é denominado de **código de peso constante**. Por exemplo, o peso da palavra-código $\underline{c} = [0 \quad 1 \quad 0 \quad 1]$ é 2.

O processo de codificação/decodificação de um código de bloco baseia-se na propriedade algébrica de que o conjunto de palavras-código $\mathbf{C} = \{\underline{c}_i\} = \{\underline{c}_0, \underline{c}_1, \cdots, \underline{c}_{M-1}\}$ pode ser mapeado em um conjunto de polinômios $\{C_i(p)\} = \{C_0(p), C_1(p), \cdots, C_{M-1}(p)\}$.

Os componentes do vetor $\underline{c}_i = [c_{i(n-1)} \quad c_{i(n-2)} \quad \cdots \quad c_{i1} \quad c_{i0}]$ que representa a *i*-ésima palavra-código correspondem aos coeficientes do polinômio $C_i(p) = c_{i(n-1)}p^{n-1} + c_{i(n-2)}p^{n-2} + \cdots + c_{i1}p + c_{i0}$ associado à palavra-código.

A mesma propriedade algébrica pode ser aplicada sobre o conjunto de mensagens $\mathbf{X} = \{\underline{x}_i\} = \{\underline{x}_0, \underline{x}_1, \cdots, \underline{x}_{M-1}\}$ de modo que este também pode mapeado em um conjunto de polinômios $\{X_i(p)\} = \{X_0(p), X_1(p), \cdots, X_{M-1}(p)\}$.

Os componentes do vetor $\underline{x}_i = [x_{i(k-1)} \quad x_{i(k-2)} \quad \cdots \quad x_{i1} \quad x_{i0}]$ que representa a *i*-ésima mensagem correspondem aos coeficientes do polinômio $X_i(p) = X_{i(k-1)}p^{n-1} + c_{i(k-2)}p^{n-2} + \cdots + c_{i1}p + c_{i0}$ associado à mensagem.

Por este motivo os códigos de bloco são também denominados de **códigos polinomiais**. Por exemplo, a representação polinomial do código da Tabela 4.1 é mostrada na Tabela 4.2.

Tabela 4.2: Representação polinomial do código da Tabela 4.1.

Mensagem $\underline{x}_i$	Polinômio $X_i(p)$	Palavra-código $\underline{c}_i$ associada a $\underline{x}_i$ por $\underline{c}_i = \boldsymbol{\theta}\{\underline{x}_i\}$	Polinômio $C_i(p)$
000	0	0000	0
001	1	0011	$p+1$
010	p	0101	p^2+1
011	$p+1$	0110	p^2+p
100	p^2	1001	p^3+1
101	p^2+1	1010	p^3+p
110	p^2+p	1100	p^3+p^2
111	p^2+p+1	1111	p^3+p^2+p+1

O processo de codificação/decodificação envolve operações aritméticas de adição e multiplicação realizadas sobre o conjunto de polinômios $\{C_i(p)\} = \{C_0(p), C_1(p), \cdots, C_{M-1}(p)\}$ que representam as palavras-código, conforme veremos.

Um código corretor de erro deve ser tal que o conjunto $\{C_i(p)\}$ e as operações aritméticas sobre ele definidas obedeçam a determinadas restrições, caso contrário a unicidade e o custo computacional do processo de codificação/decodificação resultarão prejudicados. Especificamente, os coeficientes dos polinômios em $\{C_i(p)\}$ devem pertencer a um tipo especial de conjunto denominado de **campo algébrico** (*field*) [Chen].

Um campo algébrico é uma entidade matemática estudada em Álgebra Linear. Para efeito de nosso estudo, um campo $\mathbf{F}$ é um conjunto de elementos que permite duas operações sobre seus elementos – adição e multiplicação – e que satisfaz as seguintes propriedades (axiomas):

Adição

1- O conjunto $\mathbf{F}$ é fechado sob adição, i.e., se $a, b \in \mathbf{F}$ então $a + b \in \mathbf{F}$.

2- A adição em $\mathbf{F}$ é associativa, i.e., se $a, b, c \in \mathbf{F}$ então $a + (b + c) = (a + b) + c$.

3- A adição em $\mathbf{F}$ é comutativa, i.e., se $a,b \in \mathbf{F}$ então $a+b=b+a$.

4- O conjunto $\mathbf{F}$ contém um único elemento denominado **zero**, representado por "0", que satisfaz a condição $a+0=a$, $\forall a \in \mathbf{F}$.

5- Cada elemento em $\mathbf{F}$ tem o seu elemento negativo (simétrico). Se $b \in \mathbf{F}$ então seu simétrico é denotado por $-b$ tal que $b+(-b)=0$. Se $a \in \mathbf{F}$ então a subtração $a-b$ entre os elementos a e b é definida como $a+(-b)$.

Multiplicação

1- O conjunto $\mathbf{F}$ é fechado sob multiplicação, i.e., se $a,b \in \mathbf{F}$ então $ab \in \mathbf{F}$.

2- A multiplicação em $\mathbf{F}$ é associativa, i.e., se $a,b,c \in \mathbf{F}$ então $a(bc)=(ab)c$.

3- A multiplicação em $\mathbf{F}$ é comutativa, i.e., se $a,b \in \mathbf{F}$ então $ab=ba$.

4- A multiplicação em $\mathbf{F}$ é distributiva sobre a adição, i.e., se $a,b,c \in \mathbf{F}$ então $a(b+c)=ab+ac$.

5- O conjunto $\mathbf{F}$ contém um único elemento denominado **identidade**, representado por "1", que satisfaz a condição $1a=a$, $\forall a \in \mathbf{F}$.

6- Cada elemento de $\mathbf{F}$, exceto o elemento 0, possui um elemento **inverso**. Assim, se $b \in \mathbf{F}$ e $b \neq 0$ então o inverso de b é definido como b^{-1} tal que $bb^{-1}=1$. Se $a \in \mathbf{F}$ então a divisão $a-b$ entre os elementos a e b é definida como ab^{-1}.

Por exemplo, o conjunto $\Re$ dos números reais é um campo algébrico com infinitos elementos, assim como também o é conjunto dos números complexos $\mathbb{C}$. Estes dois conjuntos obedecem aos axiomas acima.

Um campo algébrico finito com D elementos é denominado de Campo de Galois (*Galois Field*) e é designado por $\mathbf{GF}(D)$. Nem para todos os valores de D é possível formar um campo. Em geral, quando D é primo (ou uma potência inteira de um número primo) é possível construir o campo finito $\mathbf{GF}(D)$ consistindo dos elementos $\{0,1,\cdots,D-1\}$ desde que as operações de adição e multiplicação sobre $\mathbf{GF}(D)$ sejam operações **módulo *D*** [Clark].

Nota: Uma operação **op** é módulo D quando pode ser representada por $(a\ \mathbf{op}\ b)\bmod D$, onde $x \bmod y$ é o operador que resulta no resto da divisão x/y. Por exemplo, a operação de soma módulo 5 entre os números 4 e 3 resulta em 2 visto que $(4+3)\bmod 5=2$.

Por exemplo, o Campo de Galois $\mathbf{GF}(2)$ é formado pelo conjunto $\{0,1\}$ e pelas operações módulo 2 de soma e multiplicação dadas pelas Tabelas 4.3 e 4.4

Tabela 4.3: Soma sobre $\mathbf{GF}(2)$

+	0	1
0	0	1
1	1	0

Tabela 4.4: Multiplicação sobre $\mathbf{GF}(2)$

.	0	1
0	0	0
1	0	1

Note nas Tabelas 4.3 e 4.4 que a soma entre dois elementos a e b pertencentes a $\mathbf{GF}(2)$ é implementado pela operação lógica $a \oplus b$ (ou $a\,\text{XOR}\,b$) e que a multiplicação entre dois elementos a e b pertencentes a $\mathbf{GF}(2)$ é implementado pela operação lógica $a.b$ (ou $a\,\text{AND}\,b$). Por isto é usual os códigos corretores serem construídos em $\mathbf{GF}(2)$ dada a facilidade de implementação com portas lógicas AND e XOR.

Assim, um código corretor de erro binário ($\mathbf{A} = \{0,1\}$) é tal que os coeficientes dos polinômios em $\{C_i(p)\}$ pertencem a $\mathbf{GF}(2) = \mathbf{A} = \{0,1\}$ e as operações aritméticas realizadas sobre o conjunto de polinômios $\{C_i(p)\} = \{C_0(p), C_1(p), \cdots, C_{M-1}(p)\}$ (ou, equivalentemente, sobre o conjunto de palavras-código $\mathbf{C} = \{\underline{c}_i\} = \{\underline{c}_0, \underline{c}_1, \cdots, \underline{c}_{M-1}\}$) durante o processo de codificação/decodificação obedecem às Tabelas 4.3 e 4.4.

4.2.1 Capacidade de Detecção e Correção de Erro

Suponhamos que $\underline{c}_i$ e $\underline{c}_j$ sejam duas palavras-código quaisquer do código $\boldsymbol{\theta}(n,k)$. Uma medida da diferença entre as duas palavras-código é o número de bits em posições correspondentes que diferem entre si. Esta medida é denominada de **<u>Distância de Hamming</u>** e é denotada por d_{ij}. Por exemplo, sejam $\underline{c}_i = [0 \;\; 1 \;\; 0 \;\; 1]$ e $\underline{c}_j = [1 \;\; 0 \;\; 0 \;\; 0]$. Então $d_{ij} = 3$.

Observe que d_{ij} sempre satisfaz a condição $0 < d_{ij} \leq n$, $i \neq j$, para duas palavras-código $\underline{c}_i$ e $\underline{c}_j$, ambas de n bits (por definição, em um código $\boldsymbol{\theta}(n,k)$, $\underline{c}_i \neq \underline{c}_j$ $\forall i$ e $\forall j$ com $i \neq j$).

O menor valor no conjunto $\{d_{ij}\}$, $i, j = 0,1,\cdots,M-1$, $i \neq j$, $M = 2^k$ é denominado **<u>distância mínima</u>** do código e é denotado como $d_{\min}$. Por exemplo, $d_{\min} = 2$ para o código $\boldsymbol{\theta}(4,3)$ da Tabela 4.1

A Distância de Hamming d_{ij} é uma medida do <u>grau de separação</u> entre duas palavras-código $\underline{c}_i$ e $\underline{c}_j$. Assim, o grau de correlação temporal entre dois sinais modulados $v_i(t)$ e $v_j(t)$ gerados no modulador de um transmissor digital em decorrência de $\underline{c}_i$ e $\underline{c}_j$ é implicitamente associado à d_{ij} [Proakis]. Portanto, $d_{\min}$ está associado à capacidade do código $\boldsymbol{\theta}(n,k)$ em identificar palavra-código demoduladas no receptor quando estas são recebidas em erro como conseqüência do ruído e interferência presentes no canal. Em outras palavras, **<u>quanto maior $d_{\min}$ maior a capacidade de um código $\boldsymbol{\theta}(n,k)$ detectar e corrigir erros</u>**.

Demonstra-se que [Ash][Proakis]:

Seja $\boldsymbol{\theta}(n,k)$ um código corretor binário. Seja d o número máximo de erros que $\boldsymbol{\theta}(n,k)$ é capaz de **detetar** e seja t o número máximo de erros que $\boldsymbol{\theta}(n,k)$ é capaz de **corrigir**. Seja $d_{\min}$ a distância mínima de $\boldsymbol{\theta}(n,k)$. Então:

$$d = d_{\min} - 1 \tag{4.1}$$

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor \tag{4.2}$$

$$d_{\min} \leq n - k + 1 \tag{4.3}$$

sendo $\lfloor . \rfloor$ o operador que resulta no inteiro mais próximo e menor que o argumento.

Por exemplo, $d_{\min} = 2$ para o código $\boldsymbol{\theta}(4,3)$ da Tabela 4.1. Daí, de (4.1) e (4.2), temos que $d = d_{\min} - 1 = 2 - 1 = 1$ e $t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor = \left\lfloor \frac{2-1}{2} \right\rfloor = 0$. Portanto o código $\boldsymbol{\theta}(4,3)$ da Tabela 4.1 detecta no máximo 1 erro por palavra-código mas não tem capacidade de correção. De fato, este código é um simples código *parity-check*.

4.2.2 Códigos $\boldsymbol{\theta}(n,k)$ no Contexto de Espaços Vetorais

Faremos nesta seção uma breve interpretação de códigos de bloco à luz da Álgebra Linear. Embora esta interpretação não seja obrigatoriamente necessária para a compreensão da operação de um codificador, ela lança as bases para a interpretação matricial da operação de codificação/decodificação.

Um conjunto de vetores de n componentes forma um espaço vetorial $\boldsymbol{S}$ de dimensão n. Se formarmos um sub-conjunto $\boldsymbol{\Psi}$ selecionando $k < n$ vetores linearmente independentes de $\boldsymbol{S}$ e , com estes vetores de $\boldsymbol{\Psi}$, construirmos o conjunto $\boldsymbol{S_c}$ de todas as combinações lineares dos vetores de $\boldsymbol{\Psi}$, então $\boldsymbol{S_c}$ é um sub-espaço de $\boldsymbol{S}$ de dimensão k.

Qualquer conjunto de k vetores linearmente independentes no sub-espaço $\boldsymbol{S_c}$ constitui uma base de $\boldsymbol{S_c}$. Portanto, o conjunto $\boldsymbol{\Psi}$ de vetores é uma base de $\boldsymbol{S_c}$. Diz-se, então, que os vetores do sub-espaço $\boldsymbol{S_c}$ são gerados (*spanned*) pela base $\boldsymbol{\Psi}$ através de todas as possíveis combinações lineares passíveis de serem feitas com os vetores de $\boldsymbol{\Psi}$.

Consideremos agora o conjunto $\mathbf{N}$ de todos os vetores no espaço $\boldsymbol{S}$ que são ortogonais aos vetores de $\boldsymbol{\Psi}$**.** Então os vetores de $\mathbf{N}$ são também ortogonais a todos os vetores em $\boldsymbol{S_c}$, visto que $\boldsymbol{S_c}$ é gerado pela base $\boldsymbol{\Psi}$. O conjunto $\mathbf{N}$ é também um sub-espaço de $\boldsymbol{S}$ e é chamado **espaço nulo** do sub-espaço $\boldsymbol{S_c}$. Se a dimensão de $\boldsymbol{S_c}$ é k então a dimensão do espaço nulo $\mathbf{N}$ é $n-k$ [Chen].

No contexto de um código de bloco $\boldsymbol{\theta}(n,k)$ binário, o espaço vetorial $\boldsymbol{S}$ consiste de 2^n vetores de n componentes pertencentes a $\mathbf{GF}(2)$. O conjunto $\mathbf{C}$ de 2^k palavras-código de n bits é interpretado como um conjunto de 2^k vetores de n componentes, o qual forma o sub-espaço $\boldsymbol{S_c}$ de dimensão k.

Visto que existem 2^k vetores em $\boldsymbol{S_c}$ (i.e., em $\mathbf{C}$) então uma base $\boldsymbol{\Psi}$ de $\boldsymbol{S_c}$ deve conter k vetores (palavras-código). Isto porque são necessários k palavras-código linearmente independentes para construir 2^k combinações lineares em $\mathbf{GF}(2)$ e, desta maneira, gerar todo conjunto $\mathbf{C}$ a partir de $\boldsymbol{\Psi}$.

O espaço nulo de $\boldsymbol{S_c}$ (i.e., de $\mathbf{C}$) constitui o conjunto $\mathbf{C'}$ de palavras código, o qual define um outro código $\boldsymbol{\theta}'(n,n-k)$. O código $\boldsymbol{\theta}'(n,n-k)$ é formado por um conjunto de 2^{n-k} palavras-código de n bits, cada uma delas associada à respectiva mensagem de n–k bits. O código $\boldsymbol{\theta}'(n,n-k)$ é denominado de **código dual** do código $\boldsymbol{\theta}(n,k)$ [Clark]. As palavras-código de $\boldsymbol{\theta}'(n,n-k)$, por pertencerem ao espaço nulo, são todas ortogonais às palavras-código de $\boldsymbol{\theta}(n,k)$.

4.2.3 A Matriz Geradora de um Código $\boldsymbol{\theta}(n,k)$

Seja a i-ésima mensagem de um código binário $\boldsymbol{\theta}(n,k)$ representada pelo vetor $\underline{x}_i = [x_{i0} \quad x_{i1} \quad \cdots \quad x_{i(k-1)}]$ e seja a i-ésima palavra-código de $\boldsymbol{\theta}(n,k)$ representada pelo vetor $\underline{c}_i = [c_{i0} \quad c_{i1} \quad \cdots \quad c_{i(n-1)}]$, onde $i = 0,1,\cdots,M-1$, $M = 2^k$.

O processo de codificação da mensagem $\underline{x}_i = [x_{i0} \quad x_{i1} \quad \cdots \quad x_{i(k-1)}]$ na respectiva palavra-código $\underline{c}_i = [c_{i0} \quad c_{i1} \quad \cdots \quad c_{i(n-1)}]$ efetuado por um código binário $\boldsymbol{\theta}(n,k)$ pode ser representado em forma matricial por

$$\underline{c}_i = \underline{x}_i\mathbf{G} \tag{4.4}$$

onde a matriz $\mathbf{G}_{k\times n}$ é denominada de **matriz geradora** do código $\boldsymbol{\theta}(n,k)$ e é dada por

$$\mathbf{G} = \begin{bmatrix} g_{00} & g_{01} & \cdots & g_{0(n-1)} \\ g_{10} & g_{11} & \cdots & g_{1(n-1)} \\ \vdots & \vdots & & \vdots \\ g_{(k-1)0} & g_{(k-1)1} & \cdots & g_{(k-1)(n-1)} \end{bmatrix}. \tag{4.5}$$

Podemos interpretar a matriz $\mathbf{G}$ como um conjunto de k vetores-linha $\underline{g}_j$, $j = 0,1,\cdots,k-1$, tal que

$$\mathbf{G} = \begin{bmatrix} g_{00} & g_{01} & \cdots & g_{0(n-1)} \\ g_{10} & g_{11} & \cdots & g_{1(n-1)} \\ \vdots & \vdots & & \vdots \\ g_{(k-1)0} & g_{(k-1)1} & \cdots & g_{(k-1)(n-1)} \end{bmatrix} = \begin{bmatrix} \leftarrow & \underline{g}_0 & \rightarrow \\ \leftarrow & \underline{g}_1 & \rightarrow \\ & \vdots & \\ \leftarrow & \underline{g}_{(k-1)} & \rightarrow \end{bmatrix}. \tag{4.6}$$

Desta maneira, de (4.4) e (4.6), cada palavra-código $\underline{c}_i = [c_{i0} \quad c_{i1} \quad \cdots \quad c_{i(n-1)}]$ é simplesmente uma combinação linear dos vetores $\underline{g}_j$ com coeficientes da combinação determinados pela mensagem associada $\underline{x}_i = [x_{i0} \quad x_{i1} \quad \cdots \quad x_{i(k-1)}]$, isto é:

$$\underline{c}_i = x_{i0}\underline{g}_0 + x_{i1}\underline{g}_1 + \cdots + x_{i(k-1)}\underline{g}_{(k-1)} \tag{4.7}$$

Conforme discutimos na Seção 4.2.2, o conjunto **C** de 2^k palavras-código de um código $\boldsymbol{\theta}(n,k)$ é um sub-espaço de dimensão *k*. Logo, os k vetores-linha $\underline{g}_j$ que formam a matriz **G** devem ser linearmente independentes para que possam, conforme estabelece (4.7), gerar o sub-espaço **C** em *k* dimensões. Em outras palavras, o conjunto de vetores $\underline{g}_j$ é uma base para o sub-espaço **C** .

Exemplo 4.2: Verifique se a matriz $\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$ é a matriz geradora do código $\boldsymbol{\theta}(4,3)$ da Tabela 4.1.

Solução:

Cada palavra-código $\underline{c}_i = [c_{i0} \quad c_{i1} \quad \cdots \quad c_{i(n-1)}]$ de $n = 4$ bits é gerada através de (4.4) a partir da respectiva mensagem $\underline{x}_i = [x_{i0} \quad x_{i1} \quad \cdots \quad x_{i(k-1)}]$ de $k = 3$ bits. No total, existem $2^k = 8$ palavras-código em $\boldsymbol{\theta}(4,3)$. Assim

$\underline{x}_i$	$\underline{x}_i\mathbf{G} = \underline{c}_i$
$[0 \quad 0 \quad 0]$	$[0 \quad 0 \quad 0]\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} = [0 \quad 0 \quad 0 \quad 0]$
$[0 \quad 0 \quad 1]$	$[0 \quad 0 \quad 1]\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} = [0 \quad 0 \quad 1 \quad 1]$
$[0 \quad 1 \quad 0]$	$[0 \quad 1 \quad 0]\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} = [0 \quad 1 \quad 0 \quad 1]$
$[0 \quad 1 \quad 1]$	$[0 \quad 1 \quad 1]\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} = [0 \quad 1 \quad 1 \quad 0]$
$[1 \quad 0 \quad 0]$	$[1 \quad 0 \quad 0]\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} = [1 \quad 0 \quad 0 \quad 1]$

$[1\ \ 0\ \ 1]$	$[1\ \ 0\ \ 1]\begin{bmatrix}1 & 0 & 0 & 1\\ 0 & 1 & 0 & 1\\ 0 & 0 & 1 & 1\end{bmatrix}=[1\ \ 0\ \ 1\ \ 0]$
$[1\ \ 1\ \ 0]$	$[1\ \ 1\ \ 0]\begin{bmatrix}1 & 0 & 0 & 1\\ 0 & 1 & 0 & 1\\ 0 & 0 & 1 & 1\end{bmatrix}=[1\ \ 1\ \ 0\ \ 0]$
$[1\ \ 1\ \ 1]$	$[1\ \ 1\ \ 1]\begin{bmatrix}1 & 0 & 0 & 1\\ 0 & 1 & 0 & 1\\ 0 & 0 & 1 & 1\end{bmatrix}=[1\ \ 1\ \ 1\ \ 1]$

Portanto **G** é geradora de $\boldsymbol{\theta}(4,3)$.

Qualquer matriz geradora **G** de um código $\boldsymbol{\theta}(n,k)$ pode, através de operações elementares em suas linhas e permutações em suas colunas, ser reduzida à **forma sistemática** quando, então, o código gerado é sistemático. Uma matriz geradora **G** encontra-se na forma sistemática quando

$$\mathbf{G}=[\mathbf{I}_k \;\vdots\; \mathbf{P}]=\left[\begin{array}{ccccc|cccc}1 & 0 & 0 & \cdots & 0 & p_{00} & p_{01} & \cdots & p_{0(n-k-1)}\\ 0 & 1 & 0 & \cdots & 0 & p_{10} & p_{11} & \cdots & p_{1(n-k-1)}\\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots\\ 0 & 0 & 0 & \cdots & 1 & p_{(k-1)0} & p_{(k-1)1} & \cdots & p_{(k-1)(n-k-1)}\end{array}\right] \tag{4.8}$$

onde $\mathbf{I}_k$ é a matriz identidade $k\times k$ e $\mathbf{P}$ é uma matriz $k\times(n-k)$ que determina os $n-k$ bits de paridade na palavra-código $\underline{c}_i$ de n bits, a partir dos k bits da mensagem $\underline{x}_i$. Por exemplo, a matriz **G** do Exemplo 4.2 está na forma sistemática e o código $\boldsymbol{\theta}(4,3)$ gerado é um código sistemático, i.e., cada palavra-código de n bits é formada pelos k bits da respectiva mensagem associada, acrescidos (por justaposição) de $n-k$ bits de paridade.

No contexto de comunicação digital, as palavras-código passam por um processo de modulação no trasmissor e são enviadas através de um canal com ruido/interferência. Dois códigos que diferem somente na ordem (arranjo) de suas palavras-código, apresentam a mesma probabilidade de erro de decodificação no receptor, porque as distâncias de Hamming entre as palavras-código são as mesmas [Peterson]. Tais dois códigos são denominados **equivalentes**.

Especificamente, o código $\boldsymbol{\theta}_e(n,k)$ é equivalente ao código $\boldsymbol{\theta}(n,k)$ se a matriz geradora $\mathbf{G}_e$ de $\boldsymbol{\theta}_e(n,k)$ puder ser obtida através da permutação de **colunas** da matriz **G** geradora de $\boldsymbol{\theta}(n,k)$ ou através de operações elementares realizado entre as **linhas** de **G**. Uma operação elementar em $\mathbf{GF}(2)$ entre duas linhas de uma matriz consiste em permutar as linhas ou em substituir uma linha pela soma dela com outra linha.

Assim sempre podemos transformar uma matriz $\mathbf{G}$ qualquer para a forma sistemática $\mathbf{G}^*$ dada por (4.8) , mantendo a equivalência entre os respectivos códigos gerados.

Exemplo 4.3: Dada a matriz geradora $\mathbf{G} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$, colocá-la na forma sistemática $\mathbf{G}^*$. Verifique se $\mathbf{G}^*$ gera um código equivalente ao gerado por $\mathbf{G}$.

Solução:

Visto que a matriz geradora é uma matriz $\mathbf{G}_{3\times4}$, então o código gerado será um código $\boldsymbol{\theta}(4,3)$. $\mathbf{G}^*$ pode ser obtido pelo seguinte conjunto de operações elementares feito sobre as linhas de $\mathbf{G}$:

Operação Elementar	Matriz $\mathbf{G}$ alterada
$L_0 \leftrightarrow L_2$	$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$
$L_0 \leftarrow (L_0 + L_1)$	$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$
$L_0 \leftarrow (L_0 + L_2)$	$\mathbf{G}^* = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$

O código gerado por $\mathbf{G}$ é

$\underline{x}_i$	$\underline{x}_i\mathbf{G} = \underline{c}_i$
$[0 \;\; 0 \;\; 0]$	$[0 \;\; 0 \;\; 0]\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} = [0 \;\; 0 \;\; 0 \;\; 0]$
$[0 \;\; 0 \;\; 1]$	$[0 \;\; 0 \;\; 1]\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} = [1 \;\; 1 \;\; 1 \;\; 1]$

$[0\ \ 1\ \ 0]$	$[0\ \ 1\ \ 0]\begin{bmatrix}0&0&1&1\\0&1&0&1\\1&1&1&1\end{bmatrix}=[0\ \ 1\ \ 0\ \ 1]$
$[0\ \ 1\ \ 1]$	$[0\ \ 1\ \ 1]\begin{bmatrix}0&0&1&1\\0&1&0&1\\1&1&1&1\end{bmatrix}=[1\ \ 0\ \ 1\ \ 0]$
$[1\ \ 0\ \ 0]$	$[1\ \ 0\ \ 0]\begin{bmatrix}0&0&1&1\\0&1&0&1\\1&1&1&1\end{bmatrix}=[0\ \ 0\ \ 1\ \ 1]$
$[1\ \ 0\ \ 1]$	$[1\ \ 0\ \ 1]\begin{bmatrix}0&0&1&1\\0&1&0&1\\1&1&1&1\end{bmatrix}=[1\ \ 1\ \ 0\ \ 0]$
$[1\ \ 1\ \ 0]$	$[1\ \ 1\ \ 0]\begin{bmatrix}0&0&1&1\\0&1&0&1\\1&1&1&1\end{bmatrix}=[0\ \ 1\ \ 1\ \ 0]$
$[1\ \ 1\ \ 1]$	$[1\ \ 1\ \ 1]\begin{bmatrix}0&0&1&1\\0&1&0&1\\1&1&1&1\end{bmatrix}=[1\ \ 0\ \ 0\ \ 1]$

O código gerado por $\mathbf{G}^*$ é o mesmo código gerado no Exemplo 4.2. Portanto os códigos gerados por $\mathbf{G}^*$ e $\mathbf{G}$ são equivalentes porque diferem apenas na ordem (arranjo) de suas palavras-código.

Um código sistemático $\boldsymbol{\theta}(n,k)$ pode ser implementado em *hardware* através de um *shift-register* de n bits totais e $n-k$ somadores módulo 2 . Por exemplo, seja o código $\boldsymbol{\theta}(7,4)$ sistemático cuja matriz geradora é

$$\mathbf{G}=\left[\begin{array}{cccc:ccc}1&0&0&0&1&0&1\\0&1&0&0&1&1&1\\0&0&1&0&1&1&0\\0&0&0&1&0&1&1\end{array}\right]=[\mathbf{I}_4 \ \vdots\ \mathbf{P}] \tag{4.9}$$

Qualquer palavra-código $\underline{c}_i=[c_{i0}\ \ c_{i1}\ \ c_{i2}\ \ c_{i3}\ \ c_{i4}\ \ c_{i5}\ \ c_{i6}]$ deste código pode ser expressa na forma $\underline{c}_i=[x_{i0}\ \ x_{i1}\ \ x_{i2}\ \ x_{i3}\ \ c_{i4}\ \ c_{i5}\ \ c_{i6}]$ onde os 4 primeiros bits representam a respectiva mensagem $\underline{x}_i=[x_{i0}\ \ x_{i1}\ \ x_{i2}\ \ x_{i3}]$ e onde os 3 últimos bits são dados por,

$$c_{i4} = x_{i0} + x_{i1} + x_{i2} \tag{4.10}$$

$$c_{i5} = x_{i1} + x_{i2} + x_{i3} \tag{4.11}$$

$$c_{i6} = x_{i0} + x_{i1} + x_{i3} \tag{4.12}$$

A Figura 4.1. mostra uma possível implementação em *hardware* do código $\boldsymbol{\theta}(7,4)$ cuja matriz geradora é dada por (4.9).

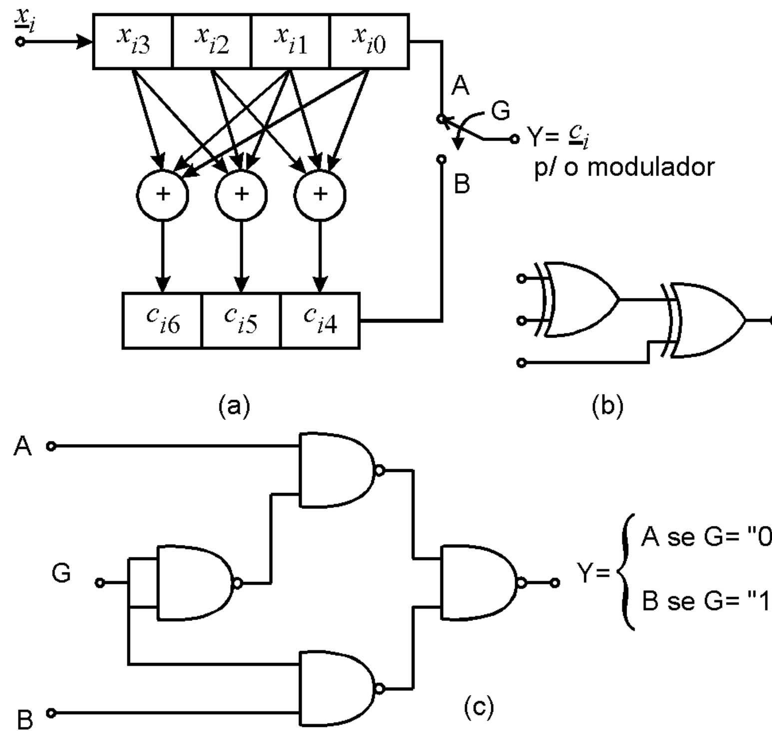


Figura 4.1: Implementação em *hardware* do código $\boldsymbol{\theta}(7,4)$ cuja matriz geradora é dada por (4.9). (a) Conjunto de *shift-registers* e somadores módulo 2. (b) Implementação de um somador módulo 2 de 3 bits com portas XOR. (c) Implementação da chave comutadora com portas NAND.

Inicialmente os 4 bits da mensagem $\underline{x}_i$ são armazenados no *shift-register* de $k = 4$ bits na ordem mostrada na Figura 4.1. Os somadores módulo 2 executam as operações definidas por (4.10),(4.11) e (4.12) e armazenam o resultado no *shift-register* de $n - k = 3$ bits. Então o *shift-register* de $k = 4$ bits é lido com o sinal de controle G da chave mantido no nível lógico "0". Após o último bit da mensagem ser lido, G recebe o nível lógico "1" e então o *shift-register* de $n - k = 3$ bits é lido.

4.2.4 A Matriz de Paridade de um Código $\boldsymbol{\theta}(n,k)$

Seja um código $\boldsymbol{\theta}(n,k)$ com matriz geradora **G** dada na forma sistemática, isto é,

$$\mathbf{G} = [\mathbf{I}_k \;\vdots\; \mathbf{P}] \tag{4.13}$$

Conforme discutimos na Seção 4.2.3, a i-ésima palavra-código $\underline{c}_i = [c_{i0} \quad c_{i1} \quad \cdots \quad c_{i(n-1)}]$ relaciona-se com a respectiva mensagem $\underline{x}_i = [x_{i0} \quad x_{i1} \quad \cdots \quad x_{i(k-1)}]$ através de $\underline{c}_i = \underline{x}_i\mathbf{G}$.

Já que **G** encontra-se na forma sistemática, a palavra-código $\underline{c}_i$ pode ser decomposta em

$$\underline{c}_i = [\underline{x}_i \quad \underline{a}_i] \tag{4.14}$$

onde $\underline{a}_i = \underline{x}_i\mathbf{P}$ é um vetor-linha que contém os $n-k$ bits de paridade de $\underline{c}_i$.

Visto que $\underline{a}_i = \underline{x}_i\mathbf{P}$, e considerando que a soma em $\mathbf{GF}(2)$ é uma operação módulo 2 (ver Tabela 4.3) então

$$\underline{x}_i\mathbf{P} + \underline{a}_i = \underline{0} \tag{4.15}$$

que pode ser escrita matricialmente como

$$[\underline{x}_i \quad \underline{a}_i]\begin{bmatrix} \mathbf{P} \\ \mathbf{I}_{n-k} \end{bmatrix} = \underline{0} \tag{4.16}$$

Definindo $\mathbf{H}^T = \begin{bmatrix} \mathbf{P} \\ \mathbf{I}_{n-k} \end{bmatrix}$ temos de (4.16) que

$$\underline{c}_i\mathbf{H}^T = \underline{0} \tag{4.17}$$

sendo

$$\mathbf{H} = \left(\mathbf{H}^T\right)^T = \begin{bmatrix} \mathbf{P} \\ \mathbf{I}_{n-k} \end{bmatrix}^T = \left[\mathbf{P}^T \;\vdots\; (\mathbf{I}_{n-k})^T\right] = \left[\mathbf{P}^T \;\vdots\; \mathbf{I}_{n-k}\right] \tag{4.18}$$

Portanto, de (4.17), infere-se que cada palavra-código do código $\boldsymbol{\theta}(n,k)$ é ortogonal a cada linha da matriz **H** (se $\underline{u} \cdot \underline{v}^T = 0$ então os vetores $\underline{u}$ e $\underline{v}$ são ortogonais [Chen]). Em conseqüência, como as palavras-código do código $\boldsymbol{\theta}(n,k)$ são geradas por **G**, então

$$\mathbf{G}\mathbf{H}^T = \underline{0} \tag{4.19}$$

Observe que a matriz **H** pode ser usada no receptor digital para detectar e localizar em qual bit da palavra-código recebida ocorreu erro como conseqüência da degradação imposta pelo canal de transmissão. Sempre que a palavra-código $\underline{y}_i$ recebida no receptor digital resultar $\underline{y}_i\mathbf{H}^T \neq \underline{0}$ então $\underline{y}_i$ apresenta erros. Por este motivo, $\mathbf{H}_{(n-k)\times n}$ é denominada de **matriz de paridade**. Discutiremos esta propriedade na Seção 4.2.5.

Conforme visto na Seção 4.2.2, associado com qualquer código $\boldsymbol{\theta}(n,k)$ existe o código dual $\boldsymbol{\theta}'(n,n-k)$, cujas 2^{n-k} palavras-código são ortogonais às 2^k palavras-código de $\boldsymbol{\theta}(n,k)$. Portanto a matriz **H** é uma matriz geradora para o código dual $\boldsymbol{\theta}'(n,n-k)$.

Exemplo 4.4:

Determine a matriz de paridade **H** do código $\boldsymbol{\theta}(4,3)$ do Exemplo 4.3 e verifique se $\mathbf{GH}^T = \underline{0}$ e se $\underline{c}_i\mathbf{H}^T = \underline{0}$.

Solução:

A matriz geradora de $\boldsymbol{\theta}(4,3)$ na forma sistemática é $\mathbf{G} = [\mathbf{I}_3 \,\vdots\, \mathbf{P}] = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$. De (4.18) temos

$$\mathbf{H} = [\mathbf{P}^T \,\vdots\, \mathbf{I}_{n-k}] = [1 \quad 1 \quad 1 \quad 1] \tag{4.20}$$

$$\mathbf{GH}^T = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{4.21}$$

Verificando se $\underline{c}_i\mathbf{H}^T = \underline{0}$:

$[0 \; 0 \; 0 \; 0]\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = [0]$	$[0 \; 0 \; 1 \; 1]\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = [0]$	$[0 \; 1 \; 0 \; 1]\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = [0]$	$[0 \; 1 \; 1 \; 0]\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = [0]$
$[1 \; 0 \; 0 \; 1]\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = [0]$	$[1 \; 0 \; 1 \; 0]\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = [0]$	$[1 \; 1 \; 0 \; 0]\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = [0]$	$[1 \; 1 \; 1 \; 1]\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = [0]$

Exemplo 4.5: Determine a matriz de paridade do código gerado pela equação (4.9).

Solução:

De (4.18)

$$\mathbf{H} = [\mathbf{P}^T \,\vdots\, \mathbf{I}_{n-k}] = \left[\begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right] \tag{4.22}$$

4.2.5 Decodificação pela Mínima Distância **(Decodificacão MLD - *Maximum-Likelihood Decoding*)**

Nesta seção estudaremos como os erros nas palavras-código são detectados e corrigidos no receptor digital.

No receptor digital, os n bits provenientes do demodulador correspondentes à i-ésima palavra-código $\underline{y}_i$ recebida são entregues ao decodificador do código $\boldsymbol{\theta}(n,k)$. O decodificador compara $\underline{y}_i$ com as $M = 2^k$ possíveis palavras-código $\underline{c}_j$ de $\boldsymbol{\theta}(n,k)$, $j = 0,1,\cdots,M-1$, e decide em favor daquela palavra-código (portanto, em favor da mensagem associada) que é mais próxima da palavra-código recebida em termos da Distância de Hamming. Matematicamente esta operação pode ser expressa por

$$\boldsymbol{\theta}^{-1}\{\underline{y}_i\} = \arg\min_{\underline{c}_j} \left|\underline{y}_i - \underline{c}_j\right|_H \tag{4.23}$$

onde $\underline{c}_j \in \mathbf{C}$, $\mathbf{C} = \{\underline{c}_i\} = \{\underline{c}_0, \underline{c}_1, \cdots, \underline{c}_{M-1}\}$ e $\left|\underline{y}_i - \underline{c}_j\right|_H$ denota a Distância de Hamming entre $\underline{y}_i$ e $\underline{c}_j$.

Esta decodificação com base na distância mínima é ótima no sentido de que ela resulta na mínima probabilidade de erro de decodificação se:

a) As palavras-código do conjunto $\mathbf{C} = \{\underline{c}_i\} = \{\underline{c}_0, \underline{c}_1, \cdots, \underline{c}_{M-1}\}$ apresentam distribuição de probabilidade uniforme (i.e., as palavras-código são equiprováveis).

b) O canal de transmissão não altera esta distribuição de probabilidade.

Um decodificador baseado no critério de distância mínima é denominado de **Decodificador de Máxima Verossimilhança** ou **Decodificador ML** (ML - *Maximum-Likelihood*). Em geral, pelo menos uma das duas condições a) ou b) não pode ser obedecida na prática. Nesta situação, o decodificador ótimo passa a ser o **Decodificador MAP** (MAP - *maximum a posteriori*), muito embora este não seja encontrado com muita freqüência na implementação de sistemas de comunicações digitais.

Um decodificador MAP leva em conta a distribuição estatística das palavras-código e toma a decisão sobre qual palavra-código foi recebida com base em Estatística Bayesiana [Proakis]. Na prática, decodificadores MAP não são muito utilizados porque sua operação depende do conhecimento exato da distribuição de probabilidade das palavras-código e da variância de ruído no canal, informação quase sempre não disponível. A maioria dos sistemas digitais utiliza decodificadores ML ao invés de decodificadores MAP principalmente devido à insignificante melhora obtida com o uso de um decodificador MAP às expensas de um considerável aumento da complexidade computacional do decodificador [Messerschmitt]. Focalizaremos nossa atenção, portanto, em decodificadores ML.

Embora a decodificação ML possa ser realizada através de (4.23), existe uma maneira mais eficiente de implementar um decodificador ML, aproveitando as propriedades da matriz de paridade $\mathbf{H}_{(n-k)\times n}$ de um código $\boldsymbol{\theta}(n,k)$, denominada de **Decodificação por**

Arranjo Padrão (*Standard Array Decoding*) . A desvantagem de (4.23) é a necessidade de calcular $M = 2^k$ Distâncias de Hamming para decodificar a palavra-código recebida. Veremos a seguir como reduzir este número de distâncias calculadas para 2^{n-k} utilizando o conceito de **Arranjo Padrão**, já que, na prática, usualmente $n-k<k$.

Seja $\underline{c}_i$ a palavra-código transmitida pelo transmissor digital através do canal de transmissão e seja $\underline{y}$ a palavra-código recebida resultante na saída do demodulador do receptor digital. Devido à degradação do sinal no canal em conseqüência de ruído/interferência, a palavra-código $\underline{y}$ recebida pode conter erros, de modo que $\underline{y}$ pode ser expressa por

$$\underline{y} = \underline{c}_i + \underline{e} \tag{4.24}$$

onde $\underline{e}$ é o vetor-linha de n bits que representa o **padrão de erro** (i.e., os bits errados em $\underline{y}$) resultante da degradação do sinal no canal. Note que o peso do padrão de erro é a Distância de Hamming entre $\underline{y}$ e $\underline{c}_i$.

Pós-multiplicando (4.24) por $\mathbf{H}^T$ obtemos

$$\underline{y}\mathbf{H}^T = (\underline{c}_i + \underline{e})\mathbf{H}^T = \underline{c}_i\mathbf{H}^T + \underline{e}\mathbf{H}^T = \underline{e}\mathbf{H}^T \tag{4.25}$$

Define-se o vetor $n-k$ dimensional $\underline{s}$, denominado **síndrome do padrão de erro**, ou simplesmente **síndrome**, como

$$\underline{s} = \underline{e}\mathbf{H}^T \tag{4.26}$$

É importante enfatizar que o conjunto de síndromes $\{\underline{s}\}$ é determinado pelo conjunto de padrões de erro $\{\underline{e}\}$ mas **não** pelo conjunto $\mathbf{C}$ de palavras-código transmitidas, como podemos inferir de (4.25).

Ainda, visto que $\underline{e}$ é um vetor de n bits (i.e., $\underline{e}$ é um vetor n dimensional em $\mathbf{GF}(2)$) então existem 2^n possíveis padrões de erro no conjunto $\{\underline{e}\}$. Mas, observe que $\underline{s}$ é um vetor de $n-k$ bits e, portanto, existem 2^{n-k} possíveis síndromes no conjunto $\{\underline{s}\}$. Em conseqüência, (4.26) mapeia diferentes padrões de erro $\underline{e}$ na mesma síndrome $\underline{s}$.

O Arranjo Padrão (AP) resulta em uma tabela, denominada **Tabela de Síndromes**, a qual é implementada em ROM (ROM - *Read Only Memory*) no receptor digital. A Tabela de Síndromes é consultada pelo decodificador ML para identificação e correção de erro em cada palavra-código $\underline{y}$ recebida. Veremos como construir a Tabela de Síndromes no Exemplo 4.6. Por enquanto, focalizaremos nossa atenção na construção do AP, visto que a capacidade de detecção/correção de um código pode ser detalhadamente dele obtida.

O AP também é uma tabela que possui 2^{n-k} linhas, cada uma delas associada a uma das 2^{n-k} possíveis síndromes. O número de colunas do AP é 2^k, correspondendo ao número de palavras-código do código $\boldsymbol{\theta}(n,k)$. Quando implementado manualmente, a linha superior do AP recebe a designação de L0 e a coluna mais à esquerda recebe a designação

C0. A Tabela 4.5 mostra a forma geral de um AP, o qual, portanto, é formado de $2^{n-k} \times 2^k = 2^n$ células.

Tabela 4.5: Forma geral de um Arranjo Padrão					
	C0	C1	C2	$\cdots$	$\text{C}(2^k - 1)$
L0	$\underline{e}_0 = \underline{c}_0 = \underline{0}$	$\underline{c}_1$	$\underline{c}_2$	$\cdots$	$\underline{c}_{(2^k-1)}$
L1	$\underline{e}_1$	$\underline{c}_1 + \underline{e}_1$	$\underline{c}_2 + \underline{e}_1$	$\cdots$	$\underline{c}_{(2^k-1)} + \underline{e}_1$
L2	$\underline{e}_2$	$\underline{c}_1 + \underline{e}_2$	$\underline{c}_2 + \underline{e}_2$	$\cdots$	$\underline{c}_{(2^k-1)} + \underline{e}_2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
$\text{L}(2^{n-k} - 1)$	$\underline{e}_{(2^{n-k}-1)}$	$\underline{c}_1 + \underline{e}_{(2^{n-k}-1)}$	$\underline{c}_2 + \underline{e}_{(2^{n-k}-1)}$	$\cdots$	$\underline{c}_{(2^k-1)} + \underline{e}_{(2^{n-k}-1)}$

Na linha L0 do AP são listadas da esquerda para a direita as 2^k palavras-código de $\boldsymbol{\theta}(n,k)$, cada uma delas representada por um vetor n dimensional em $\mathbf{GF}(2)$. A palavra-código $\underline{c}_0$ pertencente à célula identificada pela intersecção da coluna C0 com a linha L0 (célula $\text{L0} \times \text{C0}$) obrigatoriamente deve ser aquela representada pelo vetor $\underline{0}$.

Na coluna C0, abaixo da palavra-código $\underline{0}$, são listados, de alto a baixo, os $2^{n-k} - 1$ padrões de erro relativos à palavra-código $\underline{c}_0 = \underline{0}$. Primeiramente são listados todos os n padrões de erro de peso 1, isto é, primeiramente são listados todos os padrões de erro que resultam de uma Distância de Hamming unitária entre a palavra-código $\underline{y}$ recebida e $\underline{c}_0 = \underline{0}$. Se $2^{n-k} > n$, então lista-se a seguir em C0 todos os possíveis padrões de erro de peso 2. Em seguida lista-se em C0 todos os possíveis padrões de erro de peso 3, e assim sucessivamente até que todas as 2^{n-k} células de C0 estejam preenchidas. Neste contexto, $\underline{e}_0 = \underline{c}_0 = \underline{0}$ representa o padrão de erro de peso 0, isto é, representa a não-ocorrência de erro.

Nota: Visto que cada linha do AP necessita corresponder a uma única síndrome dentre as 2^{n-k} possíveis síndromes, devemos ter o cuidado de, na construção de C0, assegurar que distintos padrões de erro de peso maior que 1 em C0 correspondam a síndromes que são distintas entre si e que são simultaneamente distintas daquelas que correspondem a padrões de erro de peso 1.

Então, dando prosseguimento à construção do AP, adicionamos o padrão de erro contido na *i*-ésima célula de C0 à palavra-código na célula $\text{L0} \times \text{C1}$ e colocamos o resultado na *i*-ésima célula em C1. Em seguida, adicionamos o padrão de erro contido na *i*-ésima célula de C0 à palavra-código na célula $\text{L0} \times \text{C2}$ e colocamos o resultado na *i*-ésima célula em C2, e assim sucessivamente até completar a última coluna $\text{C}(2^k - 1)$, mais à direita do AP, sendo $i = 0,1,2,\cdots,2^{n=k} - 1$.

Observe que a i-ésima linha Li do AP assim construído (incluindo L0) representa o conjunto das 2^k possíveis palavras-código $\underline{y}_j = \underline{c}_j + \underline{e}_i$, $j = 0,1,\cdots,2^k - 1$, que serão recebidas caso a degradação do canal gere o padrão de erro $\underline{e}_i$ contido na célula $\mathrm{L}i \times \mathrm{C0}$.

Cada linha Li do AP é denominada de ***coset*** e a célula $\mathrm{L}i \times \mathrm{C0}$ é denominada **líder do *coset***. Portanto, um *coset* é o conjunto de todas as palavras-código possíveis de serem recebidas quando o canal impõe o padrão de erro definido pelo líder do *coset*.

Exemplo 4.6:

Seja o codificador de canal no transmissor de um sistema de comunicação digital que utiliza o código de bloco gerado por $\mathbf{G} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$.

a) Determine um possível AP para este código e a Tabela de Síndromes associada, visando o projeto do decodificador no receptor.

b) Suponha que o transmissor digital envie a palavra-código $\underline{c} = [1 \quad 0 \quad 1 \quad 0 \quad 1]$ através do canal. O canal degrada o sinal de forma que o demodulador no receptor envia para o decodificador a palavra-código $\underline{y} = [1 \quad 1 \quad 1 \quad 0 \quad 1]$ (erro no bit b_3). Verifique a capacidade do decodificador em detectar e corrigir este erro.

c) Suponha que o ruído/interferência no canal seja alto de forma que o demodulador no receptor envia para o decodificador a palavra-código $\underline{y} = [1 \quad 1 \quad 1 \quad 1 \quad 1]$ (erro no bit b_0 e no b_3). Verifique a capacidade do decodificador em detectar e corrigir este erro duplo.

Solução:

a) Visto que $\mathbf{G}_{k \times n} = \mathbf{G}_{2 \times 5}$, o código em questão é $\boldsymbol{\theta}(5,2)$.

As $2^k = 2^2 = 4$ palavras-código de $\boldsymbol{\theta}(5,2)$ gerado por $\mathbf{G}$ são obtidas de (4.4):

$$\underline{c}_0 = [0 \quad 0]\mathbf{G} = [0 \quad 0 \quad 0 \quad 0 \quad 0]$$

$$\underline{c}_1 = [0 \quad 1]\mathbf{G} = [0 \quad 1 \quad 0 \quad 1 \quad 1]$$

$$\underline{c}_2 = [1 \quad 0]\mathbf{G} = [1 \quad 0 \quad 1 \quad 0 \quad 1]$$

$$\underline{c}_3 = [1 \quad 1]\mathbf{G} = [1 \quad 1 \quad 1 \quad 1 \quad 0]$$

A matriz geradora não necessita ser transformada por permutação de colunas ou por operações elementares em linhas visto que já encontra-se na forma sistemática, isto é, $\mathbf{G} = \left[\begin{array}{cc|ccc} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{array}\right] = [\mathbf{I}_2 \mid \mathbf{P}]$.

Daí, de (4.18) temos que $\mathbf{H}_{(n-k)\times n} = \left[\mathbf{P}^T \;\vdots\; \mathbf{I}_{n-k}\right] = \left[\begin{array}{cc:ccc} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{array}\right]$.

Para determinar os padrões de erro da coluna C0 do AP precisamos verificar quais as síndromes resultantes dos $n = 5$ padrões de erro de peso 1 para que não ocorra igualdade com as síndromes resultantes dos padrões de erro de peso maior que 1. Os padrões de erro de peso 1 são: $[0\ \ 0\ \ 0\ \ 0\ \ 1], [0\ \ 0\ \ 0\ \ 1\ \ 0], [0\ \ 0\ \ 1\ \ 0\ \ 0], [0\ \ 1\ \ 0\ \ 0\ \ 0]$ e $[1\ \ 0\ \ 0\ \ 0\ \ 0]$.

Verificando as síndromes resultantes dos padrões de erro de peso 1:

$\underline{e}_i$	$\underline{e}_i \mathbf{H}^T = \underline{s}_i$
$[0\ \ 0\ \ 0\ \ 0\ \ 1]$	$[0\ \ 0\ \ 0\ \ 0\ \ 1]\mathbf{H}^T = [0\ \ 0\ \ 1]$
$[0\ \ 0\ \ 0\ \ 1\ \ 0]$	$[0\ \ 0\ \ 0\ \ 1\ \ 0]\mathbf{H}^T = [0\ \ 1\ \ 0]$
$[0\ \ 0\ \ 1\ \ 0\ \ 0]$	$[0\ \ 0\ \ 1\ \ 0\ \ 0]\mathbf{H}^T = [1\ \ 0\ \ 0]$
$[0\ \ 1\ \ 0\ \ 0\ \ 0]$	$[0\ \ 1\ \ 0\ \ 0\ \ 0]\mathbf{H}^T = [0\ \ 1\ \ 1]$
$[1\ \ 0\ \ 0\ \ 0\ \ 0]$	$[1\ \ 0\ \ 0\ \ 0\ \ 0]\mathbf{H}^T = [1\ \ 0\ \ 1]$

Obviamente a síndrome resultante do padrão de erro de peso 0 (inexistência de erro) é $[0\ \ 0\ \ 0\ \ 0\ \ 0]\mathbf{H}^T = [0\ \ 0\ \ 0]$.

O AP a ser construído possui $2^{n-k} = 2^{5-2} = 8$ linhas (correspondentes às 2^{n-k} síndromes). Já determinamos $n+1=6$ síndromes. Ainda faltam determinar $2^{n-k} - (n+1) = 8 - (5+1) = 2$ síndromes. Estas 2 síndromes faltantes devem **<u>obrigatoriamente</u>** ser distintas entre si e distintas das $n+1=6$ síndromes já determinadas. Tendo esta condição em mente, verifica-se na tabela acima que elas são as síndromes $[1\ \ 1\ \ 0]$ e $[1\ \ 1\ \ 1]$.

Os padrões de erro que resultam nestas 2 síndromes (que estamos buscando determinar para formar a coluna C0 do AP) devem ser padrões de erro de peso 2, visto que já esgotamos os possíveis padrões de erro de peso 0 e de peso 1.

Se expressarmos o padrão de erro por $\underline{e} = [b_4\ \ b_3\ \ b_2\ \ b_1\ \ b_0]$, onde b_i representa a ordem do bit, e considerando que $\underline{s} = \underline{e}\mathbf{H}^T$ (Equação (4.26)), temos que para a síndrome $[1\ \ 1\ \ 0]$:

$$\begin{bmatrix}1 & 1 & 0\end{bmatrix} = \begin{bmatrix}b_4 & b_3 & b_2 & b_1 & b_0\end{bmatrix}\begin{bmatrix}1 & 0 & 1\\0 & 1 & 1\\1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$$

o que resulta no seguinte sistema de equações em $\mathbf{GF}(2)$:

$$b_4 + b_2 = 1 \rightarrow b_4 = b_2 + 1 \rightarrow b_4 = \overline{b_2}$$

$$b_3 + b_1 = 1 \rightarrow b_3 = b_1 + 1 \rightarrow b_3 = \overline{b_1}$$

$$b_4 + b_3 + b_0 = 0 \rightarrow b_2 + 1 + b_1 + 1 + b_0 = 0 \rightarrow b_2 + b_1 + b_0 = 0$$

onde $\overline{b}$ representa a negação do valor lógico do bit *b*. Um possível padrão de erro de peso 2 que obedece às equações acima é $\underline{e} = \begin{bmatrix}1 & 1 & 0 & 0 & 0\end{bmatrix}$. Portanto este será o padrão de erro que associaremos à síndrome $\begin{bmatrix}1 & 1 & 0\end{bmatrix}$.

Para a síndrome $\begin{bmatrix}1 & 1 & 0\end{bmatrix}$ temos que:

$$\begin{bmatrix}1 & 1 & 1\end{bmatrix} = \begin{bmatrix}b_4 & b_3 & b_2 & b_1 & b_0\end{bmatrix}\begin{bmatrix}1 & 0 & 1\\0 & 1 & 1\\1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$$

o que resulta no seguinte sistema de equações em $\mathbf{GF}(2)$:

$$b_4 + b_2 = 1 \rightarrow b_4 = b_2 + 1 \rightarrow b_4 = \overline{b_2}$$

$$b_3 + b_1 = 1 \rightarrow b_3 = b_1 + 1 \rightarrow b_3 = \overline{b_1}$$

$$b_4 + b_3 + b_0 = 1 \rightarrow b_2 + 1 + b_1 + 1 + b_0 = 1 \rightarrow b_2 + b_1 + b_0 = 1$$

Um possível padrão de erro de peso 2, distinto do anterior, que obedece às equações acima é $\underline{e} = \begin{bmatrix}1 & 0 & 0 & 1 & 0\end{bmatrix}$. Portanto este será o padrão de erro que associaremos à síndrome $\begin{bmatrix}1 & 1 & 1\end{bmatrix}$.

De posse destes resultados, o AP é construído como:

Arranjo Padrão:				
	C0	C1	C2	C3
L0	$[0\ 0\ 0\ 0\ 0]$	$[0\ 1\ 0\ 1\ 1]$	$[1\ 0\ 1\ 0\ 1]$	$[1\ 1\ 1\ 1\ 0]$
L1	$[0\ 0\ 0\ 0\ 1]$	$[0\ 1\ 0\ 1\ 0]$	$[1\ 0\ 1\ 0\ 0]$	$[1\ 1\ 1\ 1\ 1]$
L2	$[0\ 0\ 0\ 1\ 0]$	$[0\ 1\ 0\ 0\ 1]$	$[1\ 0\ 1\ 1\ 1]$	$[1\ 1\ 1\ 0\ 0]$
L3	$[0\ 0\ 1\ 0\ 0]$	$[0\ 1\ 1\ 1\ 1]$	$[1\ 0\ 0\ 0\ 1]$	$[1\ 1\ 0\ 1\ 0]$
L4	$[0\ 1\ 0\ 0\ 0]$	$[0\ 0\ 0\ 1\ 1]$	$[1\ 1\ 1\ 0\ 1]$	$[1\ 0\ 1\ 1\ 0]$
L5	$[1\ 0\ 0\ 0\ 0]$	$[1\ 1\ 0\ 1\ 1]$	$[0\ 0\ 1\ 0\ 1]$	$[0\ 1\ 1\ 1\ 0]$
L6	$[1\ 1\ 0\ 0\ 0]$	$[1\ 0\ 0\ 1\ 1]$	$[0\ 1\ 1\ 0\ 1]$	$[0\ 0\ 1\ 1\ 0]$
L7	$[1\ 0\ 0\ 1\ 0]$	$[1\ 1\ 0\ 0\ 1]$	$[0\ 0\ 1\ 1\ 1]$	$[0\ 1\ 1\ 0\ 0]$

E a Tabela de Síndromes para implentação do decodificador é:

Tabela de Síndromes (implementada em ROM):	
Síndrome $\underline{s}_i$	Padrão de Erro $\underline{e}_i$
$[0\ 0\ 0]$	$[0\ 0\ 0\ 0\ 0]$
$[0\ 0\ 1]$	$[0\ 0\ 0\ 0\ 1]$
$[0\ 1\ 0]$	$[0\ 0\ 0\ 1\ 0]$
$[0\ 1\ 1]$	$[0\ 1\ 0\ 0\ 0]$
$[1\ 0\ 0]$	$[0\ 0\ 1\ 0\ 0]$
$[1\ 0\ 1]$	$[1\ 0\ 0\ 0\ 0]$
$[1\ 1\ 0]$	$[1\ 1\ 0\ 0\ 0]$
$[1\ 1\ 1]$	$[1\ 0\ 0\ 1\ 0]$

b) De (4.25) temos que $\underline{y}\mathbf{H}^T = \underline{e}\mathbf{H}^T = \underline{s}$. Dado $\underline{y} = [1 \ \ 1 \ \ 1 \ \ 0 \ \ 1]$, então $\underline{s} = \underline{y}\mathbf{H}^T = [0 \ \ 1 \ \ 1]$. Consultando a Tabela de Síndromes verifica-se que o padrão de erro correspondente é $\underline{e} = [0 \ \ 1 \ \ 0 \ \ 0 \ \ 0]$. De (4.24), $\underline{c}_i = \underline{y} + \underline{e} = [1 \ \ 0 \ \ 1 \ \ 0 \ \ 1]$. Portanto o decodificador detectou e corrigiu o erro simples.

c) De (4.25) temos que $\underline{y}\mathbf{H}^T = \underline{e}\mathbf{H}^T = \underline{s}$. Dado $\underline{y} = [1 \ \ 1 \ \ 1 \ \ 1 \ \ 1]$, então $\underline{s} = \underline{y}\mathbf{H}^T = [0 \ \ 0 \ \ 1]$. Consultando a Tabela de Síndromes verifica-se que o padrão de erro correspondente é $\underline{e} = [0 \ \ 0 \ \ 0 \ \ 0 \ \ 1]$. De (4.24), $\underline{c}_i = \underline{y} + \underline{e} = [1 \ \ 1 \ \ 1 \ \ 1 \ \ 0]$. Portanto o decodificador detectou o erro mas **não** corrigiu o erro duplo.

A impossibilidade deste código corrigir todos os padrões de erro com peso maior que 1 pode ser também verificada bastando consultar a coluna C0 do AP. Por inspeção da coluna C0 conclue-se que este código corrige todos os 5 padrões de erro de peso 1 possíveis e somente 2 padrões de erro de peso 2, quais sejam, $\underline{e} = [1 \ \ 1 \ \ 0 \ \ 0 \ \ 0]$ e $\underline{e} = [1 \ \ 0 \ \ 0 \ \ 1 \ \ 0]$.

Em geral o projetista do código escolhe os padrões de erro de peso w que corrigem w erros de modo incompleto com base em alguma peculiaridade do sistema digital. Por exemplo, no Exemplo 4.6 o número total de padrões de erro de peso 2 é dado pela combinação de $n = 5$ bits tomados $m = 2$ a m, i.e. $\text{Comb}(n, m) = \text{Comb}(5,2) = 10$, onde $\text{Comb}(n, m) = n!/[m!(n-m)!]$. No entanto, na construção do AP foi possível utilizar apenas 2 deles: $\underline{e} = [1 \ \ 1 \ \ 0 \ \ 0 \ \ 0]$ e $\underline{e} = [1 \ \ 0 \ \ 0 \ \ 1 \ \ 0]$. A razão da escolha destes dois padrões poderia ser, por exemplo, o fato de que o bit b_4 é um bit crucial à supervisão e controle do sistema (supondo que o código seja sistemático) e que, em menor grau, o b_3 também o seja.

4.2.6 Códigos Estendidos

Qualquer código $\boldsymbol{\theta}(n, k)$ pode ser **estendido**, isto é, a partir da matriz de paridade **H** de $\boldsymbol{\theta}(n, k)$ a matriz estendida $\mathbf{H}_E$ é obtida de **H** conforme segue:

$$\mathbf{H}_E = \left[\begin{array}{ccc|c} & & & 0 \\ & \mathbf{H} & & \vdots \\ & & & 0 \\ \hline 1 & 1 & \cdots & 1 \end{array}\right] \qquad (4.27)$$

Demonstra-se que a distância mínima de um código estendido é igual à distância mínima do código não-estendido acrescida de 1, isto é, $d_{\min}E = d_{\min} + 1$ [Clark].

4.2.7 Principais Códigos de Bloco Binários

Há uma extensa coleção de códigos de bloco binários (e não binários). Entre eles citamos:

- Códigos de Hadamard $\boldsymbol{\theta}(n,k)=\boldsymbol{\theta}(2^m, m+1)$, caracterizados por $d_{\min}=m+1$, onde $m \geq 1$ é um número inteiro. Em geral, os Códigos de Hadamard apresentam baixa razão de codificação $R_c = k/n = \tau_s/\tau_x = (m+1)/2^m$, onde onde τ_s representa a largura (duração no tempo) dos bits em uma palavra-código e τ_x representa a largura dos bits na respectiva mensagem. Portanto, como τ_s/τ_x é pequeno, o uso de um Código de Hadamard implica em um considerável **aumento** na banda-passante do sistema, e, por isso, não é muito utilizado.
- Código de Golay $\boldsymbol{\theta}(23,12)$, caracterizado por $d\min = 7$, o que significa (ver Equação (4.2)) uma capacidade de correção de até $t=\left\lfloor \frac{d\min-1}{2} \right\rfloor = \left\lfloor \frac{7-1}{2} \right\rfloor = 3$ erros simultâneos e de uma capacidade de detecção de até (ver Equação (4.1)) $d = d_{\min}-1 = 7-1 = 6$ erros simultâneos. Este código é peculiar porque ele é o único código conhecido de 23 bits capaz de corrigir até 3 erros simultâneos [Taub].
- Códigos de Hamming $\boldsymbol{\theta}(2^m-1, 2^m-1-m)$, bastante populares por serem caracterizados pela extrema facilidade de construção aliada a uma distância mínima $d\min = 3$ [Peterson][Proakis] (detecta até 2 erros simultâneos e corrige até 1 erro), sendo $m = n-k$ um inteiro positivo. Por exemplo, se $m=3$ então obtemos um Código de Hamming $\boldsymbol{\theta}(7,4)$.

Em geral, a construção de um código de bloco $\boldsymbol{\theta}(n,k)$ consiste em definirmos a sua matriz de paridade $\mathbf{H}_{(n-k)\times n}$ (preferencialmente na forma dada por (4.18)), e, a seguir, a partir de **H** e de (4.8), obtermos a sua matriz geradora $\mathbf{G}_{k\times n}$.

A matriz **H** de um Código de Hamming caracteriza-se pelas suas $n = 2^m - 1$ colunas serem formadas por todos os vetores distintos m dimensionais em $\mathbf{GF}(2)$, exceto o vetor $\underline{0}$. Por exemplo, o código $\boldsymbol{\theta}(7,4)$ gerado pela Equação (4.9) e cuja matriz **H** é dada por (4.22) é um Código de Hamming com $m=3$. Observe que **H** para este código é formado pelos $n=7$ vetores colunas $[0 \;\; 0 \;\; 1]^T$, $[0 \;\; 1 \;\; 0]^T$, $[1 \;\; 0 \;\; 0]^T$, $[0 \;\; 1 \;\; 1]^T$, $[1 \;\; 1 \;\; 0]^T$, $[1 \;\; 1 \;\; 1]^T$ e $[1 \;\; 0 \;\; 1]^T$.

4.2.8 Códigos Cíclicos

Em um código $\boldsymbol{\theta}(n,k)$ **cíclico**, se o vetor $\underline{c}_i = [c_{(n-1)} \;\; c_{(n-2)} \;\; \cdots \;\; c_1 \;\; c_0]$ é uma palavra-código de $\boldsymbol{\theta}(n,k)$, então também é o vetor $\underline{c}_j = [c_{(n-2)} \;\; c_{(n-3)} \;\; \cdots \;\; c_0 \;\; c_{(n-1)}]$ resultante de um **deslocamento cíclico** de $\underline{c}_i$. A direção do deslocamento é irrelevante sob o ponto de vista do $d_{\min}$, mas afeta a ordem em que as palavras-código são geradas.

Os Códigos Cíclicos constituem um sub-conjunto dos Códigos de Bloco, e sua popularidade resulta da facilidade de implementação da operação de deslocamento cíclico através de *shift-registers* .

Um Códigos Cíclico $\boldsymbol{\theta}(n,k)$ baseia-se na representação polinomial dos Códigos de Bloco, já discutida na Seção 4.2. Embora a implementação em *hardware* de Códigos Cíclicos seja simples, a teoria que rege esta implementação é algo sofisticada porque envolve a determinação das raízes de polinômios em $\mathbf{GF}(2)$. Portanto, limitaremos nosso estudo à descrição de um caso específico.

Seja, então, a palavra-código $\underline{c} = [c_{(n-1)} \quad c_{(n-2)} \quad \cdots \quad c_1 \quad c_0]$ de $\boldsymbol{\theta}(n,k)$ cíclico representada polinômio $C(p) = c_{(n-1)}p^{n-1} + c_{(n-2)}p^{n-2} + \cdots + c_1 p + c_0$. Seja a mensagem $\underline{x} = [x_{(k-1)} \quad x_{(k-2)} \quad \cdots \quad x_1 \quad x_0]$ representada pelo polinômio $X(p) = X_{(k-1)}p^{n-1} + c_{(k-2)}p^{n-2} + \cdots + c_1 p + c_0$.

O **polinômio gerador** de um código cíclico $\boldsymbol{\theta}(n,k)$ é o polinômio $g(p)$, de ordem $n-k$, caracterizado por ser fator do polinômio $p^n + 1$ (i.e., $g(p)$ divide exatamente $p^n + 1$, com resto nulo) e que possui a forma geral

$$g(p) = p^{(n-k)} + g_{(n-k-1)}p_{(n-k-1)} + \cdots + g_1 p + 1 \tag{4.28}$$

Exemplo 4.7:

Gere as palavras-código do código cíclico $\boldsymbol{\theta}(7,4)$ e determine a matriz geradora $\mathbf{G}$.

Solução:

$n - k = 3$

O polinômio $p^7 + 1$ pode ser fatorado na seguinte maneira em $\mathbf{GF}(2)$ ❖:

$p^7 + 1 = (p+1)(p^3 + p^2 + 1)(p^3 + p + 1)$

❖ **Nota 1**: Obviamente, não adianta acharmos as 7 raízes z_i de $p^7 + 1$ pelos métodos usuais de obtenção de raízes de polinômios com coeficientes em $\Re$ (visando formar os fatores a partir delas, i.e., $p^7 + 1 = \prod_{i=0}^{6}(p - z_i)$) porque os coeficientes dos polinômios pertencem a $\mathbf{GF}(2)$ e não a $\Re$. Foge ao escopo deste estudo a fatoração de polinômios em $\mathbf{GF}(2)$, no entanto em [Peterson] e em [Costello] pode ser encontrado uma descrição abrangente deste aspecto de códigos cíclicos.

❖ **Nota 2**: Um polinômio é irredutível em $\mathbf{GF}(D)$ quando não pode ser fatorado em polinômios de grau menor do que o próprio. Um polinômio irredutível $P(p)$ de grau m é um polinômio primitivo em $\mathbf{GF}(D)$ se o menor inteiro n para o qual $P(p)$ divide exatamente $p^n - 1$ é $n = D^m - 1$.

Tanto $p^3 + p^2 + 1$ como $p^3 + p + 1$ são possíveis candidatos à representar $g(p)$ visto que ambos são fatores de $p^7 + 1$ cuja maior potência em p é $n - k = 3$. Aliás, ambos são polinômios primitivos em $\mathbf{GF}(2)$. Adotaremos $g(p) = p^3 + p^2 + 1$.

As palavras-código de um código cíclico podem ser obtidas de

$$C_i(p) = g(p)X_i(p) \tag{4.29}$$

ou seja, se um polinômio $c_{(n-1)}p^{n-1} + c_{(n-2)}p^{n-2} + \cdots + c_1 p + c_0$ é uma palavra-código **válida** (i.e., sem erro) então ele é exatamente divisível (o resto da divisão é zero) pelo polinômio gerador $g(p)$. É desta maneira (pelo resto da divisão entre a palavra-código e $g(p)$) que um código cíclico detecta uma palavra-código errada. O cuidado em garantir que $g(p)$ seja irredutível e primitivo garante que ele seja um bom "detetor" de erros, isto é, sem ambigüidades.

Para o código $\boldsymbol{\theta}(7,4)$ em questão, as palavras-códigos resultam em

$X_i(p)$				$C_i(p) = g(p)X_i(p) = (p^3 + p^2 + 1)X_i(p)$						
p^3	p^2	p^1	p^0	p^6	p^5	p^4	p^3	p^2	p^1	p^0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1	1	0	1
0	0	1	0	0	0	1	1	0	1	0
0	0	1	1	0	0	1	0	1	1	1
0	1	0	0	0	1	1	0	1	0	0
0	1	0	1	0	1	1	1	0	0	1
0	1	1	0	0	1	0	1	1	1	0
0	1	1	1	0	1	0	0	0	1	1
1	0	0	0	1	1	0	1	0	0	0
1	0	0	1	1	1	0	0	1	0	1
1	0	1	0	1	1	1	0	0	1	0
1	0	1	1	1	1	1	1	1	1	1
1	1	0	0	1	0	1	1	1	0	0
1	1	0	1	1	0	1	0	0	0	1
1	1	1	0	1	0	0	0	1	1	0
1	1	1	1	1	0	0	1	0	1	1

As *i*-ésima linha $\underline{\ell}_i$ da matriz geradora **G** de um código cíclico relaciona-se com o polinômio gerador $g(p)$ através de

$$\underline{\ell}_i \leftrightarrow p^i g(p), \; i = k-1, k-2, \cdots, 0 \tag{4.30}$$

Assim, para o código $\boldsymbol{\theta}(7,4)$ em questão $\underline{\ell}_i \leftrightarrow p^i g(p) = p^{3+i} + p^{2+i} + p^i$, $i = 3,2,1,0$ e **G** resulta em

$$\mathbf{G} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{4.31}$$

4.3 Códigos Reed-Solomon

Os Códigos Reed-Solomon constituem uma sub-classe de uma ampla classe de códigos cíclicos denominada de Códigos BCH (Bose – Chaudhuri – Hocquenghem) [Peterson][Costello].

Os Códigos Reed-Solomon (RS) encontram-se entre os códigos mais poderosos no que diz respeito à capacidade de correção de erro, sendo largamente utilizados em muitos sistemas digitais como:

- Dispositivos de armazenamento (Fita Magnética,CDs, DVD, códigos de barra, etc.).
- Comunicações Móveis e *wireless* (Telefonia celular, *links* de microondas, etc.)
- Comunicações via Satélite.
- Televisão Digital

Vimos anteriormente que um código de bloco binário $\boldsymbol{\theta}(n,k)$ codifica mensagens de k **bits** em palavras-código de n **bits**, podendo corrigir até $t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor$ **bits** errados.

Um Código Reed-Solomon $\boldsymbol{\theta}(n,k)$, representado por $\mathbf{RS}(n,k)$, codifica mensagens de k **símbolos** em palavras-código de n **símbolos**, sendo capaz de corrigir até $t = \left\lfloor \frac{n-k}{2} \right\rfloor$ **símbolos** errados. Cada **símbolo** em uma palavra-código (ou em uma mensagem) de um código $\mathbf{RS}(n,k)$ é um bloco de m bits. Daí, portanto, o poder de correção de erro de um código $\mathbf{RS}(n,k)$: Mesmo que **todos** os m bits de cada um dos t símbolos recebidos estejam errados, o código $\mathbf{RS}(n,k)$ efetua a correção não importando a localização dos símbolos na palavra-código. Ainda, não importando o número e a posição dos bits errados em cada símbolo, o código $\mathbf{RS}(n,k)$ corrigirá até t símbolos e, caso o número de símbolos errados ultrapassar t, o código $\mathbf{RS}(n,k)$ detectará esta situação. No contexto do codificador de canal de um sistema de comunicações digitais esta característica é extremamente vantajosa porque permite a correção de um surto de $m \times t$ bits sequenciais recebidos em erro (*error burst correction*). Se o número de erros ultrapassar t, então o código $\mathbf{RS}(n,k)$ avisa o sistema de que não foi capaz de corrigir todos os erros.

É de especial interesse o caso em que $m = 8$, quando cada símbolo representa 1 *byte*. Um *byte* representa um bloco de 8 bits, que é o menor bloco de informação usualmente encontrado em sistemas microprocessados. Por exemplo, consideremos um código $\mathbf{RS}(20,16)$ com $m = 8$. Suponhamos que queiramos codificar a mensagem de $k = 16$ bytes abaixo:

255	100	012	098	120	003	233	111	077	163	000	001	088	200	101	007

O código $\mathbf{RS}(20,16)$ adiciona $n - k = 4$ bytes de paridade e codifica a mensagem acima na palavra-código em forma sistemática abaixo:

255	100	012	098	120	003	233	111	077	163	000	001	088	200	101	007	208	107	221	076

Observe que nenhum símbolo é maior do que 255, valor máximo decimal para 1 byte. Observe também que as operações entre polinômios são todas executadas em $\mathbf{GF}(2^m) = \mathbf{GF}(2^8) = \mathbf{GF}(256)$.

Foge ao escopo deste texto o estudo da álgebra de polinômios em $\mathbf{GF}(2^m)$, e, portanto, não nos aprofundaremos na teoria dos Códigos Reed-Solomon. No entanto, a Seção 4.3.1 apresenta o código fonte rscodec.c, em linguagem C ANSI, de um codec (codificador/decodificador) Reed-Solomon para $\mathbf{GF}(2^m)$, $m = 8$, o qual segue o embasamento teórico em [Costello]. O código fonte permite alteração imediata do valor de *m*, desde que $1 \leq m \leq 8$.

Segue abaixo o conjunto de linhas de comando executadas em uma janela MS-DOS do Windows 98 com o intuito de demonstrar a capacidade de correção de um código $\mathbf{RS}(20,16)$ sob ocorrência de $2 = t$ e de $3 > t$ erros. Os erros simulam a degradação do sinal no canal de comunicações. A mensagem a ser transmitida encontra-se no arquivo message.txt. O programa executado é o rscodec.exe, resultante da compilação do fonte rscodec.c em um compilador DJGPP/GCC da GNU. O compilador DJGPP é um compilador de 32 bits e pode ser *dowloaded* de http://www.delorie.com .

Tabela 4.6: Capacidade de correção de um código $\mathbf{RS}(20,16)$

```
H:\Out>rscodec -h

RS_Codec - Reed-Solomon Codec Simulator

Usage:
Option&Default:   Meaning:
-h                This page.
-n 32             Block (codeword) size.
-k 28             Number of information symbols (message size).
-o 0              Error offset in codeword (init error location).
-e 1              Number of errored symbols per codeword.
-d msg.txt        Message ascii input file.

V1.2 February 1999 by Cristina & Fernando De Castro
H:\Out>type message.txt
255 100 012 098 120 003 233 111 077 163 000 001 088 200 101 007

H:\Out>rscodec -n 20 -k 16 -o 4 -e 2 -d message.txt > result.txt

H:\Out>type result.txt

RS(20,16) Codec:
Generator polynomial coefficients in GF(256):
Dec   Bin
116   01110100
231   11100111
216   11011000
030   00011110
001   00000001

1 message in file message.txt
```

```
****************************************
Encoding message number 0

The channel imposes an error at symbol locations: 4,5.
Received codeword 0 after encoding with errors:
Loc   Dec   Bin
0     255   11111111
1     100   01100100
2     012   00001100
3     098   01100010
4     135   10000111
5     252   11111100
6     233   11101001
7     111   01101111
8     077   01001101
9     163   10100011
10    000   00000000
11    001   00000001
12    088   01011000
13    200   11001000
14    101   01100101
15    007   00000111
16    208   11010000
17    107   01101011
18    221   11011101
19    076   01001100

Decoding codeword #0

Syndrome Polynomial:
Dec   Bin
001   00000001
249   11111001
071   01000111
055   00110111
030   00011110



Detected errors: 2
Decoded Codeword at Receiver:
Dec   Bin
255   11111111
100   01100100
012   00001100
098   01100010
120   01111000
003   00000011
233   11101001
111   01101111
077   01001101
163   10100011
000   00000000
001   00000001
088   01011000
200   11001000
101   01100101
```

```
007   00000111
208   11010000
107   01101011
221   11011101
076   01001100

H:\Out>rscodec -n 20 -k 16 -o 4 -e 3 -d message.txt > result.txt

H:\Out>type result.txt

RS(20,16) Codec:
Generator polynomial coefficients in GF(256):
Dec   Bin
116   01110100
231   11100111
216   11011000
030   00011110
001   00000001

1 messages in file message.txt

****************************************
Encoding message number 0

The channel imposes an error at symbol locations: 4,5,6.
Received codeword 0 after encoding with errors:
Loc   Dec   Bin
0     255   11111111
1     100   01100100
2     012   00001100
3     098   01100010
4     135   10000111
5     252   11111100
6     022   00010110
7     111   01101111
8     077   01001101
9     163   10100011
10    000   00000000
11    001   00000001
12    088   01011000
13    200   11001000
14    101   01100101
15    007   00000111
16    208   11010000
17    107   01101011
18    221   11011101
19    076   01001100

Decoding codeword #0

Syndrome Polynomial:
Dec   Bin
001   00000001
092   01011100
127   01111111
206   11001110
142   10001110
```

```
Uncorrectable error detected !
Decoded Codeword at Receiver:
Dec   Bin
255   11111111
100   01100100
012   00001100
098   01100010
135   10000111
252   11111100
022   00010110
111   01101111
077   01001101
163   10100011
000   00000000
001   00000001
088   01011000
200   11001000
101   01100101
007   00000111
208   11010000
107   01101011
221   11011101
076   01001100
```

4.3.1 Fonte ANSI C para um Codec Reed-Solomon

A Tabela 4.7 apresenta o código fonte rscodec.c, em linguagem C ANSI, de um codec (codificador/decodificador) Reed-Solomon para $\mathbf{GF}(2^m)$, $m=8$. O código fonte permite alteração imediata do valor de *m*, desde que $1 \leq m \leq 8$. Para $m > 8$ a constante RS_NN_MAX deve ser alterada de acordo.

Para converter a Tabela 4.7 no arquivo rscodec.c, sugere-se que a tabela inteira seja selecionada, copiada e colada no bloco de notas do Windows (notepad.exe), e, então, salva como rscodec.c. Este procedimento filtrará todos os caracteres de formatação do Word.

```
/* Tabela 4.7: Codigo Fonte de um Codec Reed-Solomon */
/***************************************************************************
*    rscodec.c: Reed-Solomon encoder-decoder.
*
*    Adapted from Christian Schuler & Phil Karn
*    & Robert Morelos-Zaragoza C source code "new_rs_erasures.c".
*
*    By Cristina & Fernando de Castro - February 1999
*
*    Note: This program should be compiled with DJGPP or GCC compiler.
*          DJGPP is avaliable at http://www.delorie.com
*
***************************************************************************/
/****************************************************************************
*
* HEADERS:
*
***************************************************************************/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

/****************************************************************************
*
* PROGRAM DEFINITIONS:
*
****************************************************************************/
#define RS_NN_MAX 256 /* size for decoding arrays -> mm max. 8 ! */

#define BUFSIZE 0x7fff /* in&out file buffer size */

#define MAXLINESIZE 8192 /* max line size for  GetSheetNumLines() and
NumberOfFloatsInLine() */

/****************************************************************************
*
* COMPILING DIRECTIVES:
*
****************************************************************************/

/****************************************************************************
*
* MACROS:
*
****************************************************************************/
#define     min(a,b)    ((a) < (b) ? (a) : (b))

#define     CLEAR(a,n) {\
      int ci;\
      for(ci=(n)-1;ci >=0;ci--)\
            (a)[ci] = 0;\
      }
#define     COPY(a,b,n) {\
      int ci;\
      for(ci=(n)-1;ci >=0;ci--)\
            (a)[ci] = (b)[ci];\
      }

/****************************************************************************
*
* CONSTANTS:
*
****************************************************************************/

#define B0 1 /* Alpha exponent for the first root of the generator polynomial */

/****************************************************************************
*
* DATA TYPE STRUCTURES:
*
****************************************************************************/
```

```
typedef enum {

      NO_ERROR,        /*" ", */
      MEMORY_ERR,      /* "Not enough memory", */
      READ_OPEN_ERR,   /* "I can't find the file", */
      READ_ERR,        /* "Error reading file", */
      WRITE_OPEN_ERR,  /* "I can't open the file", */
      WRITE_ERR,       /* "I can't write the file", */
      BUFFER_ERR,      /* "I can't attach buffer to file" */
      SIZE_ERR,        /* "Specified size doesn't match data size in file", */
      MAXIT_ERR,       /* "Exceeded max number of iterations in function" */
      OFFS_ERR,        /*  "Cmd. line error: o + e must be < n" */
      VALUE_ERR        /*  "Max numerical value must be 255 in file", */

}ERR;

/*****************************************************************************
*
* FUNCTION PROTOTYPES:
*
*****************************************************************************/
void InitRS(int nn, int kk, int **Gg_ind, int **Gg_poly);
int RSEncodeTab(int *data, int *bb, int nn, int kk, int *Gg_poly);
int RSSyndrome(int *data, int *s, int nn, int kk);
void GetPp(int mm);
void GenerateGf(int mm);
int Modnn(int x);
int GfMulTab(int m1, int m2);
void PrintPolyEl(int poly_el, int mode, FILE *fp);
void Help(void);
void Quit(ERR err, char *name);
int RowColCounter(FILE *Fp, unsigned *RowCount, unsigned *ColCount);
int RSDecode(int *data, int *s, int nn, int kk, int verbose, FILE *LogF);
/*****************************************************************************
*
* GLOBAL VARIABLES:
*
*****************************************************************************/
int A0; /* No legal value in index form represents zero, so we need a special
          value for this purpose */
int gf_mm,gf_nn_max;
int *Alpha_to,*Index_of;
int *Pp;
/*****************************************************************************
*
*                       MAIN():
*
*****************************************************************************/
int main( int argc, char *argv[] )

{
int i,offset,err_pattern;
int Byte,test;

int optind;
extern char *optarg;
```

```
int e;
int mm,nn,kk,no_p;
unsigned char *Msg; /* message - size kk */
int *cw; /* original codeword - size nn */
int *recvd; /* received (with errors) codeword - size nn */
int *synd; /* syndrome vector */
int *Gg_ind;  /* generator polynomial - index form */
int *Gg_poly; /* generator polynomial – poly form */

FILE *MsgF; /* message file filepointer */

char MsgFile[128]; /* message file name.ext */


char *IO_buffer;

unsigned long CodewordCounter=0;
unsigned long SymbolCounter;
unsigned NumMessages,NSymbolsPerMessage;
unsigned LineMaxLength;
int err_cnt = 0;

/* Default parameters: */

e = 1; /* number of errors */
offset = 0; /* symbol location */
mm = 8;     /* symbol size: 1 byte = 8 bits */
nn = 32; /* number of symbols per codeword */
kk = 28; /* number of symbols per message */
strcpy(MsgFile,"msg.txt"); /* message input file */


      while ( (optind = getopt(argc, argv, "he:n:o:k:d:")) != EOF) {

            switch (optind) {

                  case 'h':
                        Help();
                        break;
                  case 'e':
                        sscanf(optarg,"%d",&e);
                        break;
                  case 'o':
                        sscanf(optarg,"%d",&offset);
                        break;
                  case 'n':
                        sscanf(optarg,"%d",&nn);
                        break;
                  case 'k':
                        sscanf(optarg,"%d",&kk);
                        break;
                         case 'd':
                        sscanf(optarg,"%s",MsgFile);
                        break;

            }/*switch*/
```

```
      }/*while*/

      if((offset + e) > nn)Quit(OFFS_ERR,"");

/* Open I/O files: */

IO_buffer=(char *)calloc(BUFSIZE,sizeof(char));
if(!IO_buffer)Quit(MEMORY_ERR,"");

if((MsgF=fopen(MsgFile,"rt"))==NULL)Quit(WRITE_OPEN_ERR,MsgFile);
if (setvbuf(MsgF, IO_buffer, _IOFBF, BUFSIZE) != 0)Quit(BUFFER_ERR,MsgFile);

/* Init procedures: */

fprintf(stdout,"RS(%d,%d) Codec:\n",nn,kk);

no_p = nn-kk; /* Number of parity symbols */

Msg = (unsigned char *) malloc( kk * sizeof(unsigned char));
if(!Msg)Quit(MEMORY_ERR,"");

cw = (int *) malloc( nn * sizeof(int));
if(!cw)Quit(MEMORY_ERR,"");

recvd = (int *) malloc( nn * sizeof(int));
if(!recvd)Quit(MEMORY_ERR,"");

synd = (int *) malloc( (no_p+1) * sizeof(int));
if(!synd)Quit(MEMORY_ERR,"");

GenerateGf(mm); /* generate galois field */

InitRS(nn,kk,&Gg_ind,&Gg_poly); /* init RS vectors */

fprintf(stdout,"Generator polynomial coefficients in GF(%d):\n",1<<mm);
fprintf(stdout,"Dec\tBin\n");

for (i = 0; i <= nn-kk; i++)PrintPolyEl(Gg_poly[i],4,stdout);

/* Get the total number of messages in MsgF: */

if(RowColCounter(MsgF,&NumMessages,&NSymbolsPerMessage)){
printf("Read/size error in file %s!",MsgFile);exit(1);
}
printf("\n%d messages in file %s\n",NumMessages,MsgFile);

if(NSymbolsPerMessage!=kk)Quit(SIZE_ERR,MsgFile);

/* Encoding/Decoding Loop: */

while(CodewordCounter<NumMessages){

/* Read message: */

for(i=0;i<kk;i++){
    test=fscanf(MsgF,"%d",&Byte);
    if(Byte>255)Quit(VALUE_ERR,MsgFile);
```

```
    Msg[i]=(unsigned char)Byte;
}

/* Reed-Solomon encoder: */

for(i=0;i<nn;i++) cw[i] = 0; /* clear codeword (symbol vector) */

fprintf(stdout,"\n");
for (i=0;i<40;i++)fprintf(stdout,"*");

fprintf(stdout,"\nEncoding message number %lu\n\n",CodewordCounter);

for(i=0;i<kk;i++) cw[i] = Msg[i]; /* transfer message in buffer to codeword */

RSEncodeTab(cw,&cw[kk],nn,kk,Gg_poly); /* encoder */

for(i=0;i<nn;i++) recvd[i] = cw[i];

/* Simulate errors imposed by the transmission channel: */

err_pattern = pow(2.0,mm)-1; /* If mm=8 then 2^mm-1=255=0xFF=11111111b */

for(i=offset;i<e+offset;i++)recvd[i] ^= err_pattern;

fprintf(stdout,"The channel imposes an error at symbol locations: ");

for(i=0;;i++){
if(i>=e-1){fprintf(stdout,"%d.",i+offset);break;}
fprintf(stdout,"%d,",i+offset);
}
fprintf(stdout,"\n");

fprintf(stdout,"Received codeword %lu after encoding with \
errors:\n",CodewordCounter);

fprintf(stdout,"Loc\tDec\tBin\n");

for (i = 0; i < nn; i++) {
fprintf(stdout,"%d\t",i);
PrintPolyEl(recvd[i],4,stdout);
}
fprintf(stdout,"\n");

/* Reed-Solomon decoder: */

for(i=0;i<no_p+1;i++) synd[i] = 0; /* clear syndrome vector */

fprintf(stdout,"Decoding codeword #%lu\n\n",CodewordCounter);

if (RSSyndrome(recvd,synd,nn,kk) != -1) {
fprintf(stdout,"Syndrome zero, no errors !\n");
}

else{

fprintf(stdout,"Syndrome Polynomial:\n");
fprintf(stdout,"Dec\tBin\n");
```

```
for (i = 0; i <= nn-kk; i++)PrintPolyEl(Alpha_to[synd[i]],4,stdout);
fprintf(stdout,"\n");


err_cnt = RSDecode(recvd,synd,nn,kk,0,stdout); /* decoder */

if(err_cnt == -1) fprintf(stdout,"Uncorrectable error detected !\n");
else fprintf(stdout,"Detected errors: %d\n",err_cnt);

fprintf(stdout,"Decoded Codeword at Receiver:\n");
fprintf(stdout,"Dec\tBin\n");

for (i = 0; i < nn; i++)PrintPolyEl(recvd[i],4,stdout);
fprintf(stdout,"\n");

} /* else */

CodewordCounter++;

} /* while */

fclose(MsgF);

return 0;

}

/*****************************************************************************
* FUNC: void Quit(int err, char *name)
*
* DESC: Prints Error message and quit.
*****************************************************************************/
void Quit(ERR err, char *name)
{
#define BELL 7
static char *ErrMess[] = {

      " ",
      "Not enough memory",
      "I can't find the file",
      "Error reading file",
      "I can't open the file",
      "I can't write the file",
      "I can't attach buffer to file",
      "Specified size doesn't match data size in file",
      "Exceeded max number of iterations in function",
      "Cmd. Line error: o + e must be < n",
      "Max numerical value must be 255 in file"
      };

system("cls");

if (err != NO_ERROR) {

if(name[0]){
fprintf (stderr,"%s %s!\n", ErrMess[err], name);
putch(BELL);
```

```
exit (err);
}
else{
fprintf (stderr,"%s!\n", ErrMess[err]);
putch(BELL);
exit (err);
}

}
exit (0);
}


/*************************************************************************
* FUNC: void Help(void)
*
* DESC: Defines usage syntax
*************************************************************************/
void Help(void)
{

const char usgtxt[] = "\nRS_Codec - Reed-Solomon Codec Simulator\n\n\
Usage:\n\
Option&Default:   Meaning:\n\
-h                This page.\n\
-n 32             Block (codeword) size.\n\
-k 28             Number of information symbols (message size).\n\
-o 0              Error offset in codeword (init error location).\n\
-e 1              Number of errored symbols per codeword.\n\
-d msg.txt        Message ascii input file.\n\n\
V1.2 February 1999 by Cristina & Fernando De Castro\n\
" ;
      printf("%s",usgtxt);
    exit(0);
}

/*************************************************************************
* FUNC: int RSEncodeTab(int *data, int *bb, int nn, int kk, int *Gg_poly)
*
* DESC: Reed-Solomon encoder.
* NOTE1: data[0] is transmitted first.
* NOTE2: data[0] is interpreted as x^(rs_nn-1) at the syndrome calculation.
*************************************************************************/
int RSEncodeTab(int *data, int *bb, int nn, int kk, int *Gg_poly)
{
int i,j,no_p;
int feedback;

no_p = nn-kk; /* Number of parity symbols */

CLEAR(bb,no_p);

for (i = 0; i < kk; i++) {
feedback = data[i] ^ bb[0];

    for (j = 0; j < no_p-1; j++){
    bb[j] = bb[j+1] ^ GfMulTab(Gg_poly[no_p-1-j],feedback);
```

```
    }

    bb[no_p-1] = GfMulTab(Gg_poly[0],feedback);
}

return 0;
}

/**************************************************************************
* FUNC: int RSSyndrome(int *data, int *s, int nn, int kk)
*
* DESC: Multiplication/division by alpha^i.Syndrome is stored in index form.
**************************************************************************/
int RSSyndrome(int *data, int *s, int nn, int kk)
{
int i,j,no_p;
int sum,product;

no_p = nn-kk; /* Number of parity symbols */

/* reset A0: */
for (j = 1; j <= no_p; j++) s[j] = A0;

for (i = 0; i < nn; i++) {
      for (j = 1; j <= no_p; j++) {
            if (s[j] != A0) product = Modnn(j+s[j]);
            else product = A0;
            sum = Alpha_to[product] ^ data[i];
            s[j] = Index_of[sum];
      }
}

/* check for errors: */
for (j = 1; j <= no_p; j++) {
      if(s[j] != A0) return(-1);
}
return(0);
}
/**************************************************************************
* FUNC: void InitRS(int nn, int kk, int **Gg_ind, int **Gg_poly)
*
* DESC: Generate poly and index form of the RS generator polynomial. The GF
*       must have been generated before.
**************************************************************************/
void InitRS(int nn, int kk, int **Gg_ind, int **Gg_poly)
{
int i, j;

if(nn > gf_nn_max) {
fprintf(stderr,"nn value > pow(2,gf_mm)-1 !\n");
exit(-1);
}

if(kk > nn) {
fprintf(stderr,"kk value > nn !\n");
exit(-1);
}
```

```
*Gg_poly = (int *) malloc( (nn - kk + 1) * sizeof(int));
 if(!(*Gg_poly))Quit(MEMORY_ERR,"for Gg_poly");

*Gg_ind = (int*) malloc( (nn - kk + 1) * sizeof(int));
if(!(*Gg_ind))Quit(MEMORY_ERR,"for Gg_ind");

(*Gg_poly)[0] = Alpha_to[B0];

(*Gg_poly)[1] = 1; /* g(x) = (X+@^B0) initially */

for (i = 2; i <= nn - kk; i++) {

(*Gg_poly)[i] = 1;

/* Multiply (Gg[0]+Gg[1]*x + ... +Gg[i]x^i) by (@^(B0+i-1) + x): */

for (j = i - 1; j > 0; j--)
(*Gg_poly)[j] = (*Gg_poly)[j - 1] ^ GfMulTab((*Gg_poly)[j], Alpha_to[B0+i-1]);

(*Gg_poly)[0] = GfMulTab((*Gg_poly)[0], Alpha_to[B0+i-1]);
}

/* generate index form for quicker encoding: */
for (i = 0; i <= nn - kk; i++) (*Gg_ind)[i] = Index_of[(*Gg_poly)[i]];
}
/*************************************************************************
* FUNC: int GfMulTab(int m1, int m2)
*
* DESC: Galois Field elements multiplication.
* NOTE: m1, m2 and result in poly-form.
*************************************************************************/
int GfMulTab(int m1, int m2)
{
int m1_i,m2_i,prod_i,result;

if((m1 == 0) || (m2 == 0)) return(0);

m1_i = Index_of[m1];
m2_i = Index_of[m2];
prod_i = Modnn(m1_i + m2_i);
result = Alpha_to[prod_i];
return(result);
}
/*************************************************************************
* FUNC: int Modnn(int x)
*
* DESC: Compute x % NN, where NN is 2^gf_mm - 1, without a slow divide
*************************************************************************/
int Modnn(int x)
{
while (x >= gf_nn_max) {
x -= gf_nn_max;
x = (x >> gf_mm) + (x & gf_nn_max);
}
return x;
}
```

```
/*************************************************************************
* FUNC: void GenerateGf(int mm)
*
* DESC: Generate Galois Field GF(2^mm).
*************************************************************************/
void GenerateGf(int mm)
{
int i, mask;

GetPp(mm);
gf_mm = mm;
gf_nn_max = pow(2,mm)-1;

A0 = gf_nn_max;
Alpha_to = (int*) malloc( (gf_nn_max + 1) * sizeof(int));
if(!Alpha_to)Quit(MEMORY_ERR,"");

Index_of = (int*) malloc( (gf_nn_max + 1) * sizeof(int));
if(!Index_of)Quit(MEMORY_ERR,"");

for (i = 0; i <= gf_nn_max; i++ ) Alpha_to[i] = 0;
for (i = 0; i <= gf_nn_max; i++ ) Index_of[i] = 0;

mask = 1;
Alpha_to[mm] = 0;

for (i = 0; i < mm; i++) {
Alpha_to[i] = mask;
Index_of[Alpha_to[i]] = i;
/* If Pp[i] == 1 then, term @^i occurs in poly-repr of @^gf_mm: */
if (Pp[i] != 0) Alpha_to[mm] ^= mask;     /* Bit-wise XOR operation */
mask <<= 1; /* single left-shift */
}

Index_of[Alpha_to[mm]] = mm;

/* Note: Have obtained poly-repr of @^gf_mm. Poly-repr of @^(i+1) is given by
         poly-repr of @^i shifted left one-bit and accounting for any @^gf_mm
         term that may occur when poly-repr of @^i is shifted. */

mask >>= 1;
for (i = mm + 1; i < gf_nn_max; i++) {

      if (Alpha_to[i - 1] >= mask)

            Alpha_to[i] = Alpha_to[mm] ^ ((Alpha_to[i - 1] ^ mask) << 1);
      else
            Alpha_to[i] = Alpha_to[i - 1] << 1;

Index_of[Alpha_to[i]] = i;

}

Index_of[0] = A0;
Alpha_to[gf_nn_max] = 0;
}
```

```
/*************************************************************************
* FUNC: void GetPp(int mm)
*
* DESC: Get Primitive Polynomial
*************************************************************************/
void GetPp(int mm)
{
int i;

Pp = (int *) malloc( (mm + 1) * sizeof(int));
if(!Pp)Quit(MEMORY_ERR,"");

for (i = 0; i <= mm; i++) Pp[i] = 0;

Pp[0] = 1;
Pp[mm] = 1;

      switch(mm) {
            case 2:
                  Pp[1] = 1;
                  break;
            case 3:
                  Pp[1] = 1;
                  break;
            case 4:
                  Pp[1] = 1;
                  break;
            case 5:
                  Pp[2] = 1;
                  break;
            case 6:
                  Pp[1] = 1;
                  break;
            case 7:
                  Pp[3] = 1;
                  break;
            case 8:
                  Pp[2] = 1;
                  Pp[3] = 1;
                  Pp[4] = 1;
                  break;
            case 9:
                  Pp[4] = 1;
                  break;
            case 10:
                  Pp[3] = 1;
                  break;
            case 11:
                  Pp[2] = 1;
                  break;
            case 12:
                  Pp[1] = 1;
                  Pp[4] = 1;
                  Pp[6] = 1;
                  break;
```

```
            case 13:
                  Pp[1] = 1;
                  Pp[3] = 1;
                  Pp[4] = 1;
                  break;
            case 14:
                  Pp[1] = 1;
                  Pp[6] = 1;
                  Pp[10] = 1;
                  break;
            case 15:
                  Pp[1] = 1;
                  break;
            case 16:
                  Pp[1] = 1;
                  Pp[3] = 1;
                  Pp[12] = 1;
                  break;
            default:
                  fprintf(stderr,"symbol size must be in range 2-16 bits !\n");
                  exit(-1);
                  break;
      }/*switch*/
}

/**************************************************************************
* FUNC: void PrintPolyEl(int poly_el, int mode, FILE *fp)
*
* DESC:  Print polynomial elements.
**************************************************************************/
void PrintPolyEl(int poly_el, int mode, FILE *fp)
{
int k;

switch(mode) {

            case 0:
                  for (k = gf_mm-1; k >= 0; k--)
fprintf(fp,"%d",(poly_el>>k)&1);
                  fprintf(fp,"\n");
                  break;
            case 1:
                  fprintf(fp,"%02x\t",poly_el);
                  for (k = gf_mm-1; k >= 0; k--)
fprintf(fp,"%d",(poly_el>>k)&1);
                  fprintf(fp,"\n");
                  break;
            case 2:
                  fprintf(fp,"%3d, %02x ",Index_of[poly_el],poly_el);
                  for (k = gf_mm-1; k >= 0; k--)
fprintf(fp,"%d",(poly_el>>k)&1);
                  fprintf(fp,"\n");
                  break;
```

```
            case 3: /* VHDL */
                  fprintf(fp,"\"");
                  for (k = gf_mm-1; k >= 0; k--)
fprintf(fp,"%d",(poly_el>>k)&1);
                  fprintf(fp,"\"");
                  break;

            case 4:
                  fprintf(fp,"%03d\t",poly_el);
                  for (k = gf_mm-1; k >= 0; k--)
fprintf(fp,"%d",(poly_el>>k)&1);
                  fprintf(fp,"\n");
                  break;
}
}
/*******************************************************************************
* FUNC: int RowColCounter(FILE *Fp, unsigned *RowCount, unsigned *ColCount)
*
* DESC: Return by args RowCount and ColCount the number of rows and cols lines
*       of the matrix in file pointed by Fp.
*
* NOTE: On success return 0. On read error return -1. On size error return >0.
*******************************************************************************/
int RowColCounter(FILE *Fp, unsigned *RowCount, unsigned *ColCount)
{
char line[MAXLINESIZE];
unsigned register NumValues=0;
int Test=0;
double Value;

*RowCount=0;
*ColCount=0;

while (fgets(line,MAXLINESIZE-2,Fp))(*RowCount)++;

if(strlen(line)==1)(*RowCount)--;/* to discount the last \n in file
                                    for the case the user has typed it */
rewind(Fp);

while((Test=fscanf(Fp,"%lf",&Value))!=-1){
if(Test!=1)return 1;
NumValues++;
}

*ColCount=NumValues/(*RowCount);

rewind(Fp);

return NumValues%(*RowCount);
}
```

```
/************************************************************************
* FUNC: int RSDecode(int *data,int *s,int nn,int kk,int verbose,FILE *LogF)
*
* DESC: Reed-Solomon decoder.
*************************************************************************/
int RSDecode(int *data, int *s, int nn, int kk, int verbose, FILE *LogF)
{
int deg_lambda, el, deg_omega;
int i, j, r, no_p;
int u,q,tmp,num1,num2,den,discr_r;

int lambda[RS_NN_MAX],b[RS_NN_MAX],t[RS_NN_MAX],omega[RS_NN_MAX];
int reg[RS_NN_MAX],root[RS_NN_MAX],loc[RS_NN_MAX];

int syn_error, count;

if(gf_mm > 8) {
      fprintf(stderr,"M > 8 is not supported in the RS decoder,\n");
      fprintf(stderr,"increase the RS_NN_MAX constant !\n");
      exit(-1);
}

no_p = nn-kk; /* Number of parity symbols */
CLEAR(&lambda[1],no_p);
lambda[0] = 1;

for(i=0;i<(no_p+1);i++) b[i] = Index_of[lambda[i]];

/* Berlekamp-Massey algorithm to determine error+erasure locator polynomial: */

r = 0;
el = 0;
while (++r <= no_p) {   /* r is the step number */

/* Compute discrepancy at the r-th step in poly-form: */
discr_r = 0;

for (i = 0; i < r; i++){

if ((lambda[i] != 0) && (s[r - i] != A0)) {
discr_r ^= Alpha_to[Modnn(Index_of[lambda[i]] + s[r - i])];
}

}
discr_r = Index_of[discr_r];  /* Index form */

if (discr_r == A0) {
      /* 2 lines below: B(x) <-- x*B(x) */
      COPY(&b[1],b,no_p);
      b[0] = A0;
}
else
{
/* T(x) <-- lambda(x) - discr_r*x*b(x): */
t[0] = lambda[0];
```

```
for (i = 0 ; i < no_p; i++) {
if(b[i] != A0) t[i+1] = lambda[i+1] ^ Alpha_to[Modnn(discr_r + b[i])];
else  t[i+1] = lambda[i+1];
}

if (2 * el <= r - 1) {
el = r - el;

/* B(x) <-- inv(discr_r) lambda(x): */
for (i = 0; i <= no_p; i++)
b[i] = (lambda[i] == 0) ? A0 : Modnn(Index_of[lambda[i]] - discr_r + gf_nn_max);
}
else
{
/* B(x) <-- x*B(x): */
COPY(&b[1],b,no_p);
b[0] = A0;
}
COPY(lambda,t,no_p+1);
}
}

if(verbose > 2) {
fprintf(LogF,"Error Locator Polynomial Lambda:\n");
fprintf(stdout,"Dec\tBin\n");
for(i=0;i<=no_p;i++) PrintPolyEl(lambda[i],4,LogF);
fprintf(LogF,"\n");
}

/* Convert lambda to index form and compute deg(lambda(x)): */
deg_lambda = 0;
for(i=0;i<no_p+1;i++){
      lambda[i] = Index_of[lambda[i]];
      if(lambda[i] != A0)
            deg_lambda = i;
}
/* Find roots of the error+erasure locator polynomial by Chien Search: */

COPY(&reg[1],&lambda[1],no_p);
count = 0;        /* Number of roots of lambda(x) */

for (i = 1; i <= gf_nn_max; i++) {
      q = 1;
      for (j = deg_lambda; j > 0; j--)
            if (reg[j] != A0) {
                  reg[j] = Modnn(reg[j] + j);
                  q ^= Alpha_to[reg[j]];
            }
      if (!q) {
/* store root (index-form) and error location number: */
            root[count] = i;
            loc[count] = gf_nn_max - i;
            count++;
      }
}
```

```
if((verbose > 0) && (count > 0)) {
      fprintf(LogF,"Error Positions Before Mapping: ");
      for (i = 0; i < count; i++)
            fprintf(LogF,"%d ", loc[i]);
      fprintf(LogF,"\n");
}

/* map the error locations to the shortened array: */
/* if error in padded array part => decoder error */

for (i = 0; i < count; i++) {
      loc[i] = nn-1-loc[i];
      if(loc[i] < 0) return(-1);
}

if((verbose > 0) && (count > 0)) {
      fprintf(LogF,"Mapped Error Positions: ");
      for (i = 0; i < count; i++)
            fprintf(LogF,"%d ", loc[i]);
      fprintf(LogF,"\n");
}

/* Compute err+eras evaluator poly omega(x) = s(x)*lambda(x) (modulo
 x^(NN-KK)) in index form and find deg(omega) */

deg_omega = 0;
for (i = 0; i < no_p;i++){
      tmp = 0;
      j = (deg_lambda < i) ? deg_lambda : i;
      for(;j >= 0; j--){
            if ((s[i + 1 - j] != A0) && (lambda[j] != A0))
                  tmp ^= Alpha_to[Modnn(s[i + 1 - j] + lambda[j])];
      }
      if(tmp != 0)
            deg_omega = i;
      omega[i] = Index_of[tmp];
}
omega[no_p] = A0;

if(verbose > 3) {
      fprintf(LogF,"Error Evaluator Polynomial Omega:\n");
                fprintf(stdout,"Dec\tBin\n");
      for(i=0;i<=no_p;i++) PrintPolyEl(Alpha_to[omega[i]],4,LogF);
      fprintf(LogF,"\n");
}

/* Compute error values in poly-form. num1 = omega(inv(X(l))), num2 =
inv(X(l))^(B0-1) and den = lambda_pr(inv(X(l))) all in poly-form */

for (j = count-1; j >=0; j--) {
      num1 = 0;
      for (i = deg_omega; i >= 0; i--) {
            if (omega[i] != A0)
                  num1  ^= Alpha_to[Modnn(omega[i] + i * root[j])];
      }
      num2 = Alpha_to[Modnn(root[j] * (B0 - 1) + gf_nn_max)];
      den = 0;
```

```
/* lambda[i+1] for i even is the formal derivative lambda_pr of lambda[i]: */

      for (i = min(deg_lambda,no_p-1) & ~1; i >= 0; i -=2) {
            if(lambda[i+1] != A0)
                  den ^= Alpha_to[Modnn(lambda[i+1] + i * root[j])];
      }

      /* Apply error to data */
      if (num1 != 0) {
            data[loc[j]] ^= Alpha_to[Modnn(Index_of[num1] + Index_of[num2]\
+ gf_nn_max - Index_of[den])];
      }
}

if (deg_lambda != count) return -1;
else return count;

}
```

4.4 Códigos Convolucionais – Decodificador de Viterbi

Um código convolucional é gerado pela combinação linear em $\mathbf{GF}(2)$ das saídas de um *shift-register* de K estágios. A seqüência de bits a ser codificada é aplicada na entrada do *shift-register*, e este executa a convolução em $\mathbf{GF}(2)$ entre a seqüência de entrada e a resposta ao impulso da **<u>máquina de estado</u>** (*state machine*) representada pelo *shift-register*. A saída da máquina de estado constitui, portanto, a seqüência codificada.

O número de estados da máquina de estado de um codificador convolucional é 2^K, sendo K o número de estágios do *shift-register*. No contexto de códigos convolucionais K recebe o nome de *constraint length* [Taub]. A razão entre o número de entradas da máquina de estado e o número de saídas da mesma define a razão de codificação R_c (ver Capítulo I) do codificador.

Como uma máquina de estado construída a partir de um *shift-register* apresenta um conjunto finito de transições permitidas entre estados, quando a seqüência a ser codificada é a ela submetida, implicitamente ficarão restringidas as transições da seqüência codificada em sua saída. Se o receptor conhecer a tabela de transições permitidas, então os erros gerados por degradação do sinal no canal de comunicações poderão ser identificados e corrigidos.

A Figura 4.2 mostra um codificador convolucional com $K=2$ ($2^K=4$ estados) e $R_c=1/2$. A seqüência de bits a ser codificada é representada por u e a saída do codificador é a seqüência de bits v. Visto que $R_c=1/2$, para cada bit de u são gerados dois bits em v. O estágio D^0 transfere o valor lógico em sua entrada para a sua saída **<u>imediatamente após</u>** a ocorrência da borda de descida do pulso de *clock* (não representado na figura). De forma idêntica opera o estágio D^1. Representando a saída do estágio D^0 por D^0 e representando a

saída do estágio D^1 por D^1, então o par de bits $\mathsf{D}^0\mathsf{D}^1$ identifica um dos estados da máquina de estado.

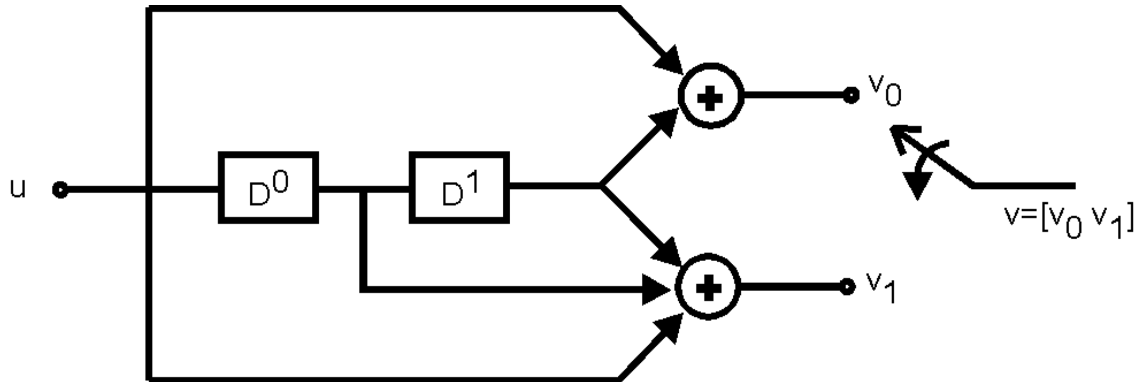


Figura 4.2: Codificador convolucional com $K = 2$ ($2^K = 4$ estados) e $R_c = 1/2$.

A Figura 4.3 mostra o diagrama de transição de estados do codificador da Figura 4.2. Cada círculo na Figura 4.3 representa um estado $\mathsf{D}^0\mathsf{D}^1$ dentre os $2^K = 4$ possíveis estados. O diagrama é construído a partir dos estados individuais considerando as **<u>transições permitidas</u>** a partir de cada estado como conseqüência do valor lógico de u. Por exemplo, suponhamos que a máquina de estado encontre-se no estado 10 (i.e., $\mathsf{D}^0 = 1$ e $\mathsf{D}^1 = 0$ na Figura 4.2). Se $\mathsf{u} = 1$ a saída resultante é $\mathsf{v} = 10$ e, após a borda de descida do *clock*, a máquina vai para o estado 11. Por outro lado, se $\mathsf{u} = 0$ a saída resultante é $\mathsf{v} = 01$ e, após a borda de descida do *clock*, a máquina vai para o estado 01.

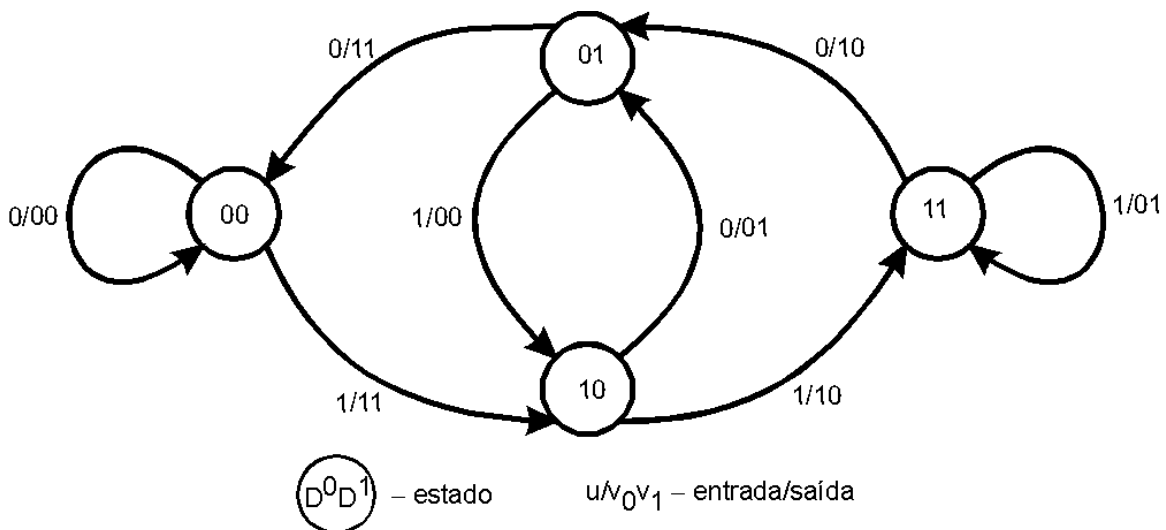


Figura 4.3: Diagrama de transição de estados para o codificador da Figura 4.2.

Dada uma seqüência u a ser codificada, a saída v no codificador de um transmissor digital é enviada ao receptor através do canal de transmissão, sendo recebida como uma seqüência r . Se nenhuma degradação de sinal ocorreu no canal de transmissão, $\mathsf{r}= \mathsf{v}$. A Tabela 4.8 mostra uma possível seqüência u e a resultante seqüência v para o codificador da Figura 4.2. É mostrado também a trajetória do estado $\mathsf{D}^0\mathsf{D}^1$ a medida que u é codificada,

partindo inicialmente do estado 00. Assumindo que v seja enviado através de um canal de transmissão com ruído/interferência, a Tabela 4.8 mostra uma possível seqüência r recebida com 2 erros.

Tabela 4.8: Exemplo de codificação para o codificador da Figura 4.2					
u =	0	1	1	0	0
D^0D^1 =	00	10	11	01	00
v =	00	11	10	10	11
r =	0***1***	11	1***1***	10	11

No receptor digital, o decodificador utiliza um algoritmo de decodificação baseado no princípio de mínima distância (MLSE – *maximum likelihood sequence detector*) denominado **Algoritmo de Viterbi**.

Vamos decodificar a seqüência r da Tabela 4.8 através do Algoritmo de Viterbi para testar a capacidade de correção de erros do mesmo. A Figura 4.4 mostra o **diagrama de treliça** utilizado pelo Decodificador de Viterbi adequado ao codificador convolucional da Figura 4.2.

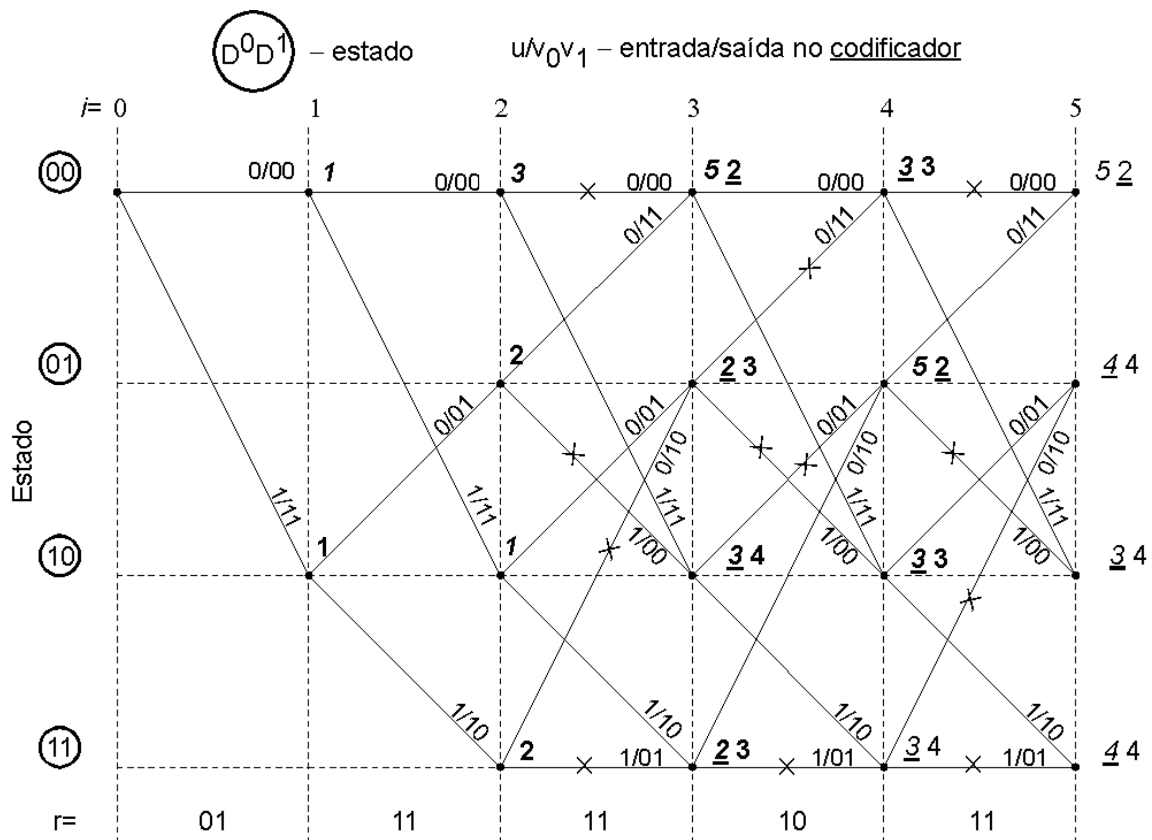


Figura 4.4: Diagrama de Treliça do Decodificador de Viterbi para o codificador convolucional da Figura 4.2.

O diagrama de treliça mostra todas as trajetórias (caminhos) das transições de estado da máquina de estado do codificador a cada instante *i* de codificação, a partir do estado 00. Cada ramo da treliça começa e termina em um estado, representando, assim, uma transição permitida. Cada ramo é identificado por $\mathsf{u}/\mathsf{v}_0\mathsf{v}_1$, isto é, a saída v do codificador quando, ao aplicarmos u em sua entrada, a máquina de estado executa a transição representada pelo ramo em questão.

A técnica de decodificação consiste em acumular em cada nó da treliça as Distâncias de Hamming entre a saída v do codificador e a seqüência r recebida a cada instante *i*. Se mais de um caminho chega a um nó "mata-se" aqueles de maior **métrica** (maior distância acumulada) − caminhos marcados com × na Figura 4.4 − ficando apenas aquele de menor métrica, denominado de **caminho sobrevivente**. A métrica acumulada de cada caminho encontra-se em negrito à direita de cada nó na Figura 4.4. Métricas sublinhadas representam métricas de caminhos sobreviventes. Métricas em itálico representam ramos que incidem no nó "por cima" e métricas em não-itálico representam ramos que incidem no nó "por baixo" , já que, no máximo 2 ramos incidem em um nó para este decodificador.

A decodificação final é iniciada a partir do caminho sobrevivente de menor métrica acumulada, identificando cada ramo sobrevivente da direita para a esquerda na treliça, conforma mostra a Figura 4.5.

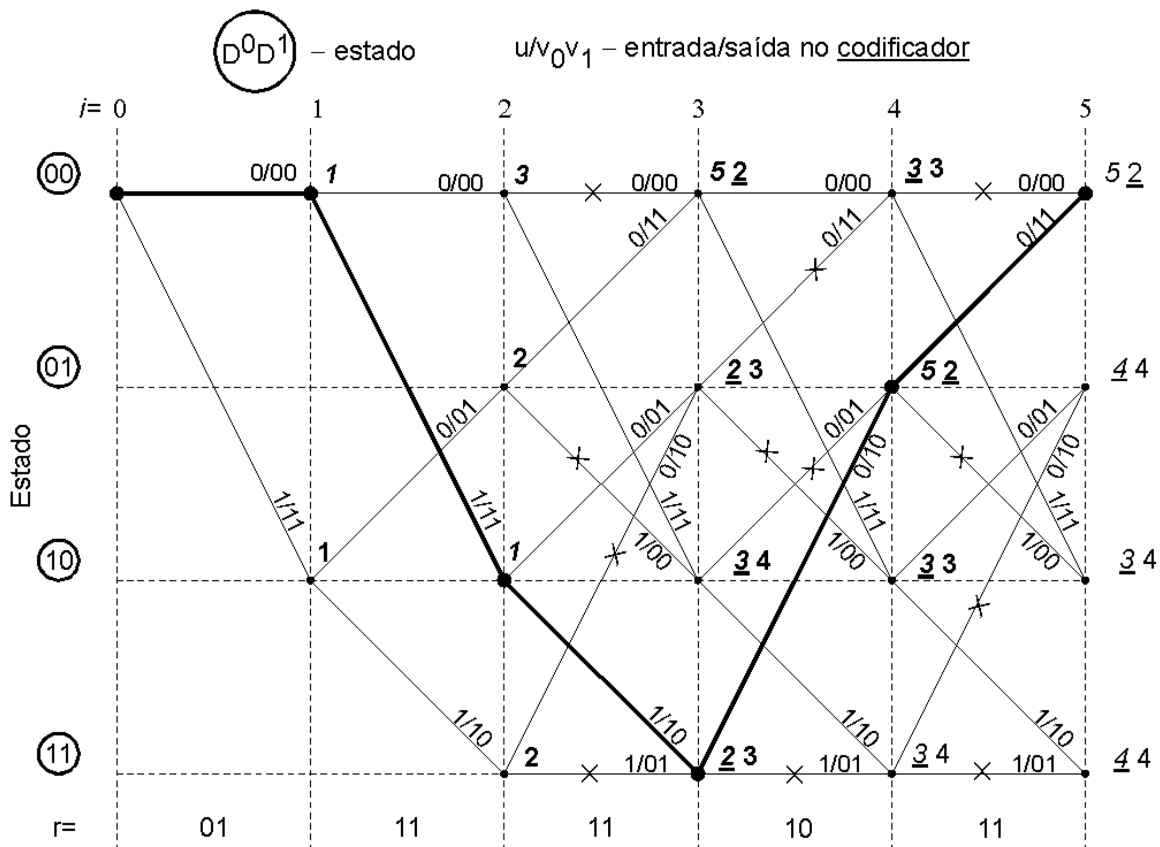


Figura 4.5: Decodificando a seqüência r da Tabela 4.8.

Ao lermos o valor de u nos identificadores u/v_0v_1 de cada ramo sobrevivente na Figura 4.5, verificamos que a seqüência originalmente transmitida foi $u = [01100]$,o que concorda com u mostrado na Tabela 4.8. Portanto, o decodificador identificou e corrigiu os 2 erros.

4.5 Códigos Concatenados – Estudo de Caso: DTV

Este estudo analisa a etapa de codificação/decodificação concatenada no contexto de transmissão de sinais para DTV (*Digital Television*) com modulação 8-VSB (8-*level Vestigial Sideband*), proposta pela ATSC (*Advanced Television Systems Committee*). São apresentados resultados de simulação quanto à capacidade de correção de erros desta etapa face à ruído Gaussiano aditivo e à surtos longos de ruído impulsivo.

4.5.1 Introdução

O sistema 8-VSB da ATSC [ATSC1] foi proposto em substituição ao veterano sistema NTSC para transmissão terrestre de sinais de televisão. O sistema 8-VSB é concebido para, ocupando os mesmos 6 MHz do sistema NTSC, apresentar superior desempenho face aos diversos tipos de degradação de sinal impostos pelo canal de transmissão. Como possíveis degradações citamos o desvanecimento de sinal, interferência, surtos de ruído e, principalmente, fantasmas (*multipath)*. A título de comparação de desempenho, o sistema NTSC apresenta qualidade de imagem considerada apenas marginal quando a relação sinal-ruído (*SNR – signal to noise ratio*) cai abaixo de 34 dB, enquanto que o sistema 8-VSB mantém qualidade de imagem constante até uma *SNR* tão baixa quanto 15 dB [Sgrignoli]. Em grande parte, este desempenho do sistema 8-VSB é devido à etapa de codificação/decodificação , ou etapa de processamento de dados, presente no transmissor/receptor, a qual é o escopo deste estudo.

Antes de analisarmos a etapa de processamento de dados, é instrutivo descrevermos brevemente as demais etapas envolvidas em um sistema 8-VSB. As Figuras 4.6 e 4.7 mostram respectivamente o diagrama geral de blocos do transmissor e receptor 8-VSB. Os blocos internos ao retângulo tracejado em ambas as figuras compreendem a etapa de processamento de dados. A informação de entrada do transmissor 8-VSB é constituída de pacotes de 188 bytes provenientes da camada de transporte de um codificador MPEG-2 [IEC1][IEC2], o qual previamente comprimiu o sinal de vídeo e áudio de forma a termos na entrada do transmissor uma taxa de informação de 19.39Mbps [ATSC1][ATSC2].

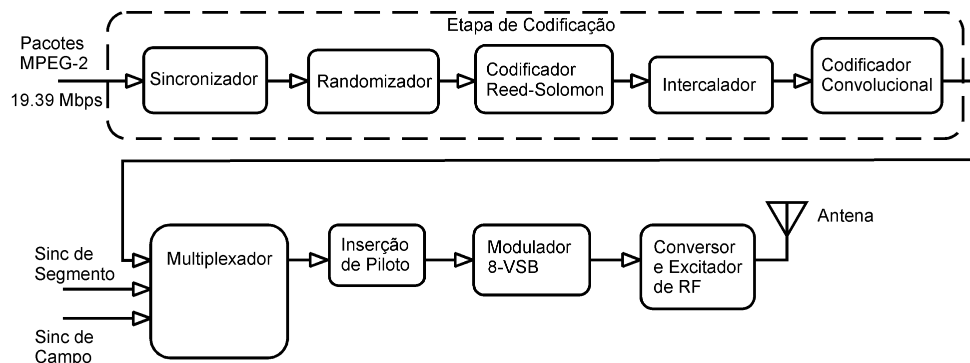


Figura 4.6: Diagrama de blocos do transmissor 8-VSB.

A saída da Etapa de Codificação é a saída do Codificador Convolucional, que é constituída de uma seqüência de símbolos 8-VSB, cada um deles com 3 bits, os quais podem assumir os valores {-7,-5,-3,-1,1,3,5,7}. A cada pacote MPEG-2 de 188 bytes que entra na etapa de codificação, é gerada na saída da mesma uma seqüência de 828 símbolos 8-VSB. Cada seqüência de 828 símbolos 8-VSB ao passar pelo Multiplexador é pré-acrescida da seqüência de símbolos [5,-5,-5,5], denominada de Sincronismo de Segmento. O Sincronismo de Segmento seguido pela seqüência de 828 símbolos 8-VSB é denominado Segmento de Dados, portanto cada Segmento de Dados é formado de 832 símbolos. Ao início de cada série de 312 Segmentos de Dados o Multiplexador acrescenta uma seqüência de 832 símbolos, cujos 4 símbolos iniciais representam o Sincronismo de Segmento, seguidos de uma seqüência de 828 símbolos especiais denominada Sincronismo de Campo. O Sincronismo de Segmento mais Sincronismo de Campo seguido dos 312 Segmentos de Dados é denominado de Campo, portanto cada Campo é constituído de 313 seqüências de 832 símbolos. O conjunto de dois Campos é denominado Quadro.

A Figura 4.8 é a representação bidimensional da seqüência unidimensional de $2\times313\times832$ símbolos 8-VSB compreendida em um Quadro. A representação é feita em duas dimensões apenas para fins descritivos, pois a nível operacional os símbolos de um Quadro ocorrem seqüencialmente ao longo do tempo. A Figura 4.9 mostra os valores dos 200 primeiros símbolos de um Segmento de Dado típico de um Quadro 8-VSB na saída da etapa de codificação.

A função do Sincronismo de Segmento é amarrar o intervalo de temporização (*clock*) do receptor com o do transmissor. No receptor, um filtro correlator no bloco Restauração de Sincronismo, utilizando a alta correlação proveniente da periodicidade do Sincronismo de Segmento, recupera o intervalo de *clock* original do transmissor. A função do Sincronismo de Campo é prover o Equalizador do receptor com seqüências de símbolos pré-conhecidas (PN511 e PN63 [ATSC1]), as quais são inseridas no Sincronismo de Campo gerado no transmissor. Estas seqüências são utilizadas no receptor como referência de erro para algum algoritmo adaptativo (em geral o LMS – *least mean squares*) aplicado aos coeficientes de um filtro usualmente FIR (*finite impulse response*) transversal, de tal forma ao filtro inverso assim obtido cancelar efeito de *multipath* [ATSC2].

Para que os sinais de Sincronismo de Campo e Sincronismo de Segmento possam ser extraídos no receptor, é necessário que alguma referência temporal (referência de fase) seja transmitida. Para tanto, a saída do Multiplexador passa pelo bloco Inserção de Piloto, o qual adiciona digitalmente o nível DC 1.25 a todos os símbolos. Note que o conjunto de níveis nominais dos símbolos 8-VSB na saída do Multiplexador é {-7,-5,-3,-1,1,3,5,7}. A adição deste nível DC, após o Modulador 8-VSB e o Conversor de RF, resulta na geração de uma pequena portadora na parte inferior do espectro do canal. No receptor, um PLL (*phase-lock loop*) no bloco Detetor Síncrono amarra o instante “zero” do receptor com o do transmissor através da informação de fase da portadora piloto.

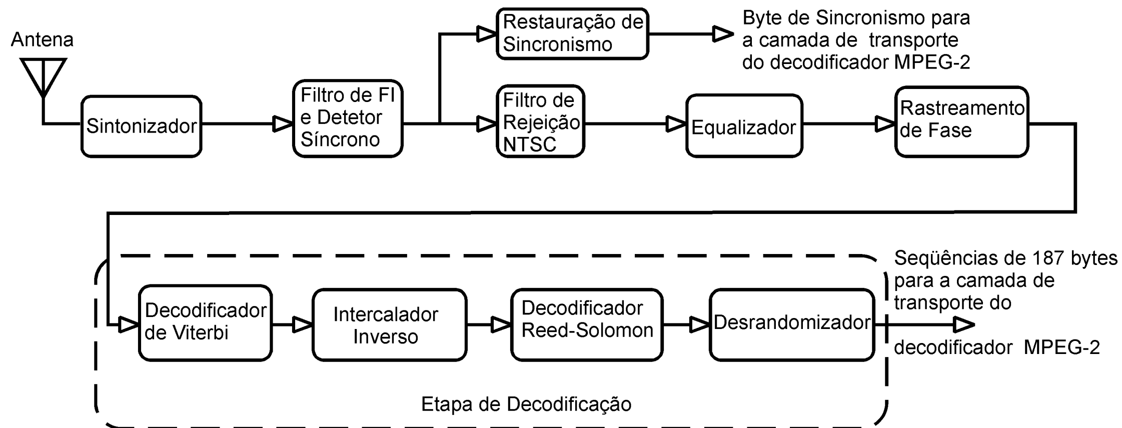


Figura 4.7: Diagrama de blocos do receptor 8-VSB. Segue ao receptor o decodificador MPEG-2, o qual reconstrói os sinais de vídeo e áudio [IEC2]. Os pacotes de 188 bytes da camada de transporte MPEG-2 [IEC1] são obtidos a partir da reinserção do byte de sincronismo às seqüências de 187 bytes na saída do Desrandomizador.

O Filtro de Rejeição NTSC é um recurso temporário no sistema 8-VSB [ATSC1][ATSC2]. Ele ficará ativo apenas durante o período de transição do sistema NTSC para o 8-VSB, devendo ser eliminado ao seu término [ATSC2]. Assim, este estudo assume esta fase como já ultrapassada, e fará a análise da etapa de processamento de dados considerando este recurso como inexistente.

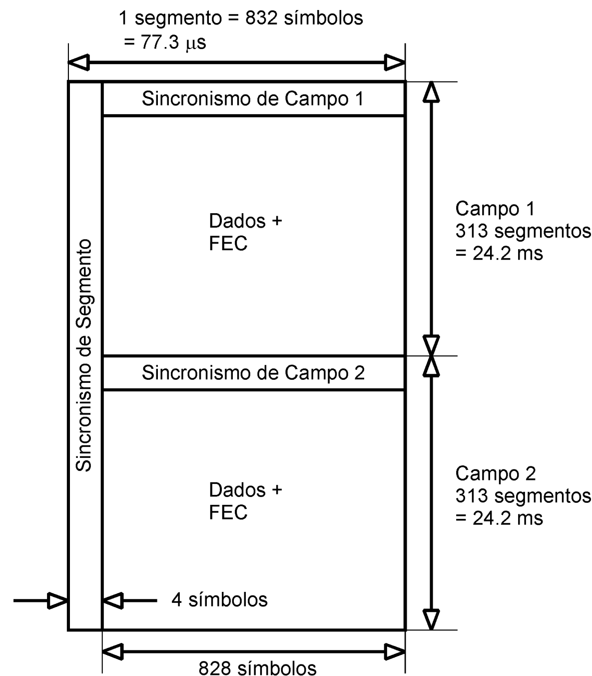


Figura 4.8: Organização de um Quadro 8-VSB. A seqüência de 828 símbolos que segue os 4 símbolos do Sincronismo de Segmento em cada Segmento de Dado contém informação dos pacotes de dados MPEG-2. Esta informação encontra-se acrescida de redundância propositalmente inserida pelos algoritmos de correção de erro (*FEC – forward error correction*) da etapa de codificação.

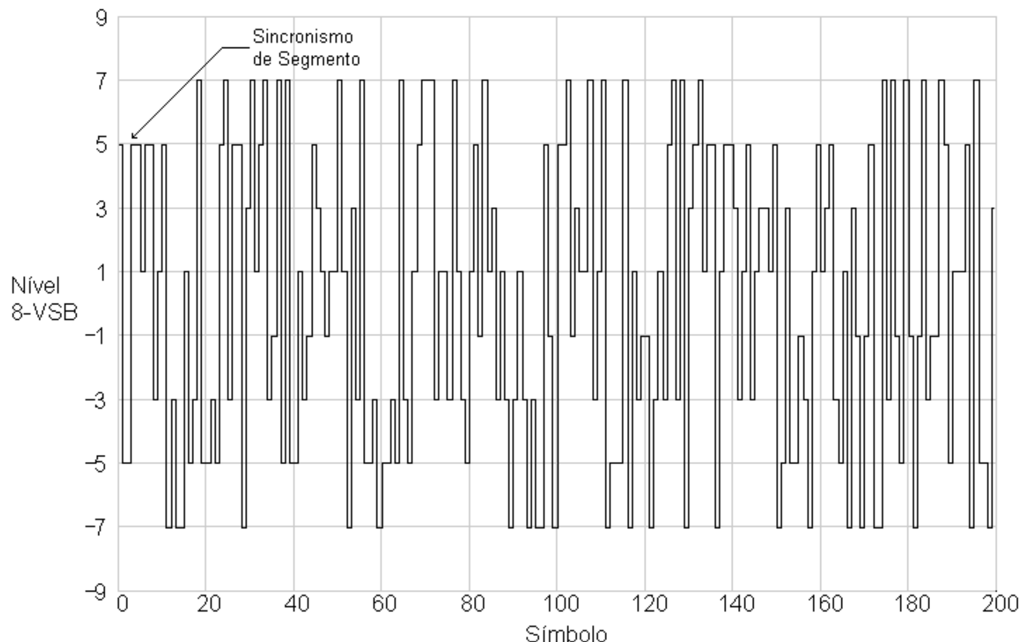


Figura 4.9: Segmento de Dado típico de um Quadro 8-VSB. Observe o caráter randômico dos níveis 8-VSB, o que contribui para a geração de um espectro plano dentro do canal de 6 MHz.

A entrada da etapa de decodificação do receptor é a entrada do Decodificador de Viterbi. Neste ponto do diagrama de blocos do receptor, a informação apresenta a mesma estrutura de dados mostrada nas Figuras 4.8 e 4.9. Ou seja, se nenhum erro incorrigível ocorreu, o fluxo de informação neste ponto é uma réplica daquele presente na saída do Codificador Convolucional no transmissor. Para cada seqüência de 828 símbolos 8-VSB que entra na etapa de decodificação, é gerada na saída da mesma uma seqüência de 187 bytes, os quais formarão o pacote de 188 bytes da camada de transporte MPEG-2 mediante a adição do byte de sincronismo.

4.5.2 Codificador

Nesta seção passamos a analisar a etapa de codificação do transmissor 8-VSB, conforme Figura 4.6. A análise que segue é subsidiada por simulador implementado pelos autores para fins de estudo do desempenho do sistema ATSC 8-VSB. Assim, a forma de estruturação e processamento da informação, nos aspectos que a norma ATSC [ATSC1] deixa a critério do projetista, seguirá a implementação concebida pelos autores deste texto.

Para cada pacote MPEG-2 de 188 bytes que entra na etapa de codificação, o Sincronizador extrai o byte de sincronismo (o primeiro byte) e armazena a resultante seqüência de 187 bytes em um *buffer* de 58344 bytes de capacidade. O byte de sincronismo será substituído no Multiplexador pelo Sincronismo de Segmento. O *buffer* estará cheio quando armazenar 312 seqüências de 187 bytes, perfazendo nesta situação o número de bytes necessários para que a etapa de codificação gere um Campo 8-VSB completo em sua saída.

Uma vez preenchido o *buffer*, o Randomizador faz a operação lógica ou-exclusivo (XOR) entre os bits de cada byte do *buffer* e os bits do byte de saída gerado pelo gerador de

seqüência pseudo-randômica mostrado na Figura 4.10. Esta operação garante um espectro de potência plano dentro do canal de 6 MHz, maximizando a eficiência da distribuição de potência na ocupação do canal de transmissão.

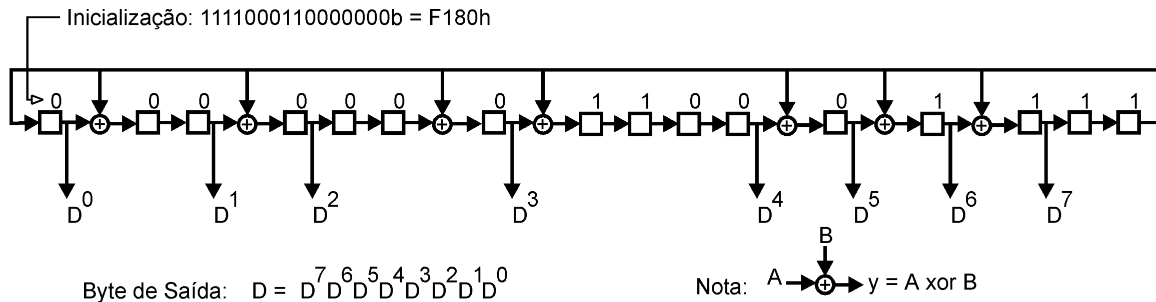


Figura 4.10: Gerador de seqüência pseudo-randômica através de um registrador de deslocamento (*shift register*) de 16 bits [Clark]. O *shift register* é inicializado com o valor hexadecimal F180h no início de cada Campo [ATSC1].

Cada uma das seqüências de 187 bytes armazenadas no *buffer* de 58344 bytes são processadas pelo Codificador Reed-Solomon. Para cada uma delas, o Codificador Reed-Solomon acrescenta uma seqüência de 20 bytes ao final, e armazena a resultante seqüência de 207 bytes em um *buffer* de 64584 bytes (a nível de implementação, apenas uma extensão do primeiro *buffer*). Este número de bytes corresponde a 312 seqüências de 207 bytes armazenadas, o que equivale a um Campo 8-VSB completo na saída da Etapa de Codificação. No contexto de códigos Reed-Solomon, cada seqüência de 187 bytes é denominada Mensagem e cada seqüência de 207 bytes é denominada Palavra-Código. Os 20 bytes acrescentados ao final de cada Mensagem são denominados de Paridade, a qual é a informação redundante adicionada à Mensagem para fins de correção de erro. Existe um mapeamento unívoco entre cada Mensagem e sua Paridade [Wicker], e como o decodificador "conhece" todas as possíveis palavras-códigos, quando uma Palavra-Código é corrompida no canal de transmissão o erro pode ser detectado e eventualmente corrigido no receptor [Clark].

Os códigos Reed-Solomon são uma sub-classe da classe de códigos cíclicos denominados BCH (*Bose-Chaudhuri-Hocquenghem*)[Costello]. Um código Reed-Solomon $\mathbf{RS}(n,k)$ é caracterizado pelo número de símbolos n por Palavra-Código, pelo número de símbolos k por Mensagem e pelo número de bits m por símbolo [Wicker]. Portanto, a etapa de codificação/decodificação do sistema DTV 8-VSB ATSC utiliza um código de bloco $\mathbf{RS}(207,187)$ com $m = 8$ (1 byte) por símbolo. Um código $\mathbf{RS}(n,k)$ é considerado um código sistemático [Costello] porque os símbolos da Mensagem não são transformados, mas apenas acrescidos dos $n-k$ símbolos de paridade. A taxa (*code rate* – medida da eficiência de transmissão de informação) para um código $\mathbf{RS}(n,k)$ é k/n, e o número máximo de símbolos passíveis de serem corrigidos em uma Palavra-Código recebida sob erro é $(n-k-1)/2$ para $n-k$ ímpar ou $(n-k)/2$ para $n-k$ par. Portanto o código $\mathbf{RS}(207,187)$ com $m = 8$ tem capacidade de corrigir até 10 bytes (10 símbolos) em uma Palavra-Código recebida sob erro, não importando quais dos 207 bytes tenha sido corrompido. Em um código de bloco cujos símbolos são apenas bits ($m = 1$), como nos códigos de Hamming binários por exemplo [Costello], quando o número de bits recebidos sob erro excede o número de bits que o código tem capacidade de corrigir, a Palavra

Código recebida é sumariamente "corrigida" para alguma outra totalmente diferente da que foi originalmente transmitida. Isto não acontece em um código Reed-Solomon. Para o código $\mathbf{RS}(207,187)$ com $m = 8$, se o número de erros exceder a 10 bytes, a Palavra-Código recebida não poderá ser corrigida, mas o algoritmo corretor de erros (Algoritmo de Berlekamp [Clark]) ainda assim identifica que a Palavra-Código recebida é incorrigível. Mas a grande vantagem do código Reed-Solomon torna-se aparente quando a informação a ser decodificada é proveniente de um fluxo de bits contínuo (sem delimitação de blocos), como aquele gerado por um Decodificador de Viterbi [Costello][Clark][INTEL]. Nesta situação, a capacidade de correção de erro do sistema concatenado Viterbi/Reed-Solomon é ainda maior porque o código Reed-Solomon corrigirá os símbolos como um todo, independentemente de qual bit nos símbolos em erro tenha sido corrompido.

O *buffer* de 64584 bytes na saída do codificador Reed-Solomon, quando preenchido com as 312 Palavras-Código de 207 bytes, é submetido a um processo de "embaralhamento" das posições de seus bytes através do Intercalador. Dois intercaladores são utilizados no sistema DTV 8-VSB ATSC. O primeiro é um Intercalador Convolucional (*convolutional interleaver* [Clark][Peterson]), mostrado na Figura 4.11, o qual intercala bytes associados a símbolos que podem pertencer a Segmentos de Dados distintos no Campo 8-VSB. O segundo é um Intercalador de Blocos (*block interleaver* [Clark][Peterson]) o qual intercala bytes associados a símbolos pertencentes ao mesmo Segmentos de Dados. Embora ambos os intercaladores precedam o Codificador Convolucional na Figura 4.6, veremos que a dinâmica do fluxo de dados entre eles não pode ser descrito de forma assim tão simples.

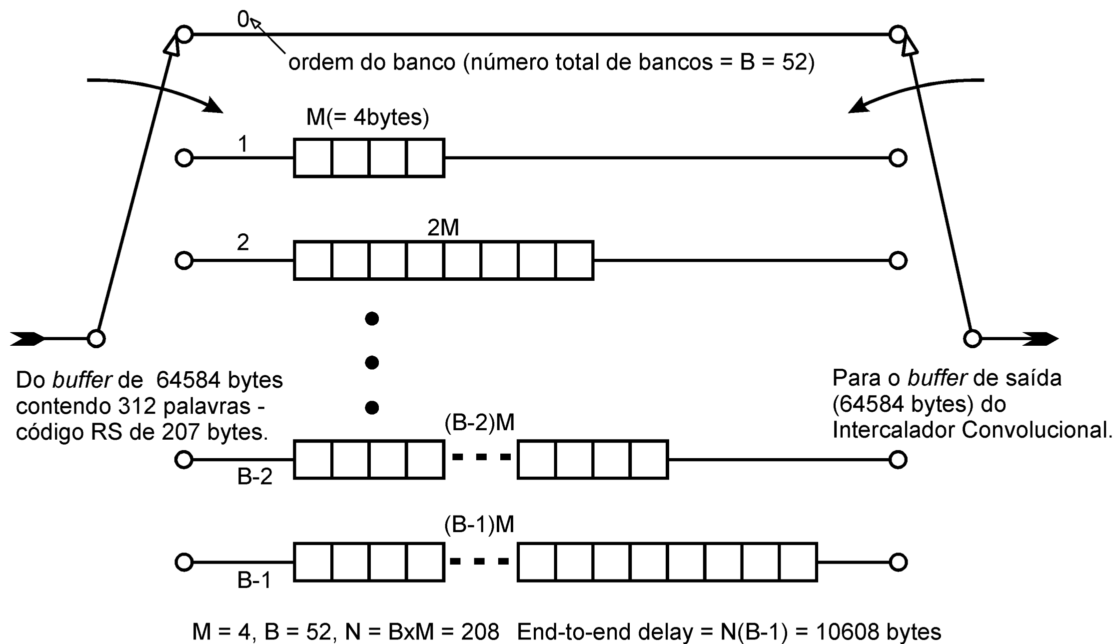


Figura 4.11: Intercalador Convolucional, formado por 52 bancos de *shift registers* operando sobre bytes. Este intercalador introduz um atraso de 10608 bytes denominado *end-to-end delay* [Clark] que deverá ser compensado no receptor. A posição das chaves com relação à

ordem do banco obedece à seqüência $[0,1,\cdots,51,0\cdots]$, sendo a posição 0 inicial sincronizada com o primeiro byte de dados do Campo.

A Figura 4.12 mostra a interação entre o Codificador Convolucional, o qual na realidade é composto por 12 codificadores em paralelo, e o Intercalador de Bloco. O diagrama interno de cada Codificador Convolucional é mostrado na Figura 4.13.

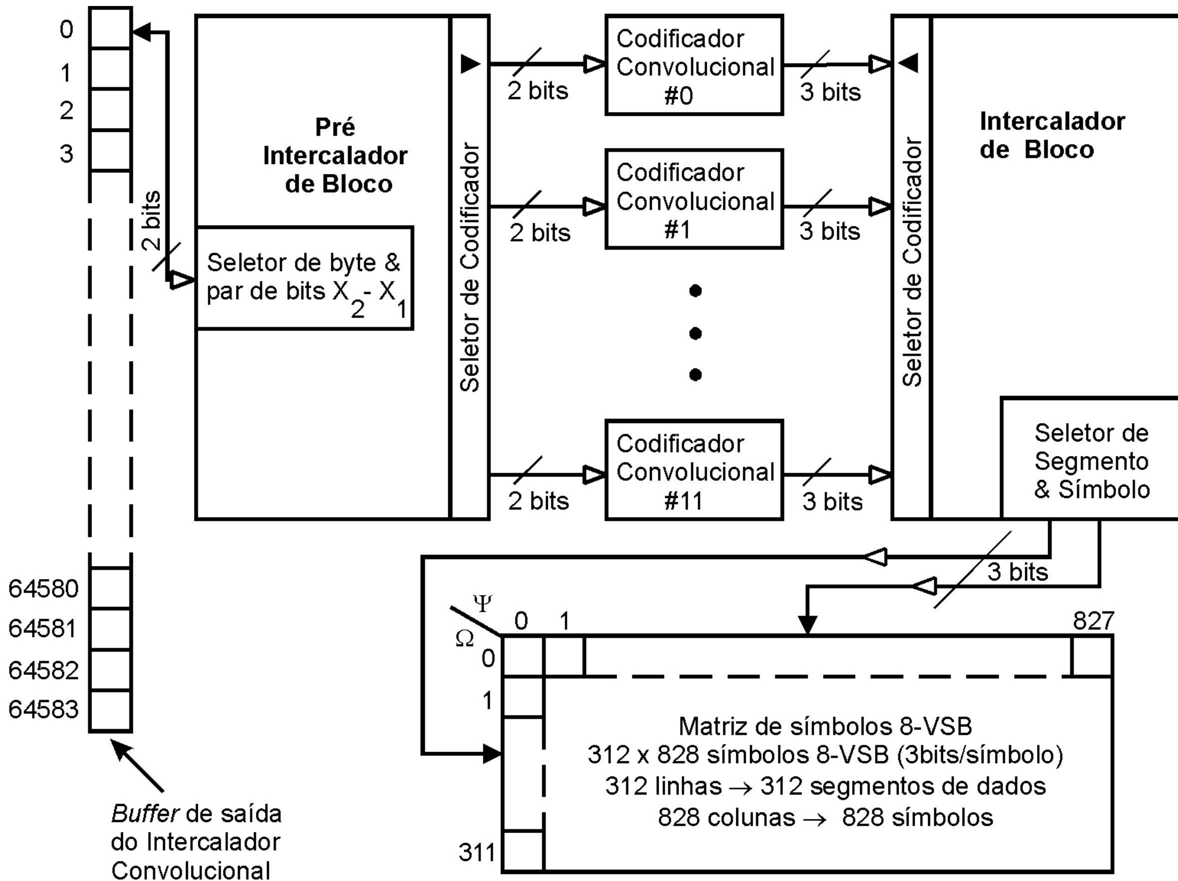


Figura 4.12: Interação do Intercalador de Blocos com o grupo de 12 Codificadores Convolucionais em paralelo.

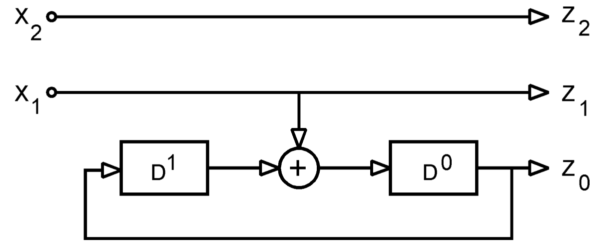


Figura 4.13: Diagrama interno de um dos 12 idênticos Codificadores Convolucionais mostrados na Figura 4.12, composto por um *shift register* de 2 bits com realimentação [Costello][Clark]. Para cada 2 bits aplicados respectivamente aos nós de entrada (X_2, X_1) o codificador gera 3 bits nos respectivos nós de saída (Z_2, Z_1, Z_0). O bit atribuído ao nó de entrada X_2 é transmitido sem codificação para o nó de saída Z_2. Durante o período de transição do sistema NTSC para o 8-VSB haverá o pré-codificador do filtro de interferência NTSC entre X_2 e Z_2 [ATSC1][ATSC2].

A operação do bloco Intercalador na Figura 4.6 pode ser descrito conforme segue. O Intercalador Convolucional processa as 312 palavras-código Reed-Solomon em sua entrada colocando o resultado em seu *buffer* de saída (Figura 4.11). Em função do símbolo Ψ que está sendo gerado na saída da etapa de codificação e a qual segmento Ω ele pertence (Figura 4.12–Matriz de Símbolos 8-VSB), o Pré-intercalador de Blocos seleciona o par de bits (b^u, b^v) $u, v = \{0,1,\cdots,11\}$ no byte *B* do *buffer* de saída do Intercalador Convolucional, $B = \{0,1,\cdots,64583\}$ e atribui (b^u, b^v) aos nós de entrada (X_2, X_1) do Codificador Convolucional *T*, $T = \{0,1,\cdots,11\}$. O codificador *T* gera então em sua saída a trinca de bits respectiva a (Z_2, Z_1, Z_0) a qual é atribuída pelo Intercalador de Blocos ao elemento da Matriz de Símbolos 8-VSB (Figura 4.12) na posição definida pela linha Ω coluna Ψ.

A relação analítica que define *B* como função de Ω e Ψ é determinada pelas equações (4.32) a (4.37).

$$B = 12\beta + (\lambda \bmod 48) \bmod 12 + \Delta \tag{4.32}$$

$$\beta = \left\lfloor \frac{\lambda}{48} \right\rfloor \tag{4.33}$$

$$\lambda = 828\,\Omega + \Psi \tag{4.34}$$

onde o operador $\lfloor . \rfloor$ retorna a parte inteira do argumento, o operador $p \bmod q$ retorna o resto da divisão entre *p* e *q*, e

$$\Delta = \begin{cases} 0, \; se \;\; \mathrm{f}(\beta) = \text{-}1 \;\text{ ou }\; \left\lfloor \dfrac{\lambda \bmod 48}{12} \right\rfloor < \mathrm{f}(\beta) \\ 4, \; se \;\; (\lambda \bmod 48) \bmod 12 < 8 \\ -8, \; se \;\; (\lambda \bmod 48) \bmod 12 \geq 8 \end{cases} \tag{4.35}$$

onde

$$f(\beta)=\begin{cases} t_1 = g((48\beta)\bmod 828\,,1), & se\ \ t_1 \neq -1 \\ t_2 = g((48\beta)\bmod 828\,,2), & se\ \ t_2 \neq -1 \\ t_3 = g((48\beta)\bmod 828\,,3), & se\ \ t_3 \neq -1 \\ -1, & se\ \ \text{nenhuma das condições acima for verdadeira} \end{cases} \tag{4.36}$$

e

$$g(l,p)=\begin{cases} p, & se\ \ (12p+l)\bmod 828 = 0 \\ -1, & se\ \ (12p+l)\bmod 828 \neq 0 \end{cases} \tag{4.37}$$

A relação analítica que define T como função de Ω e Ψ é dada pelas equações (4.38) e (4.39).

$$T = S_{[\Omega \bmod 3],[\lambda \bmod 12]} \tag{4.38}$$

onde λ é definido por (4.34) e $\boldsymbol{S}$ é a matriz 3x12 definida por

$$S=\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 0 & 1 & 2 & 3 \\ 8 & 9 & 10 & 11 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix} \tag{4.39}$$

A relação analítica que define u e v como função de Ω e Ψ é determinada pelas equações (4.40) a (4.42).

$$\mathsf{u} = \Phi_{\lfloor(\lambda \bmod 48)/12\rfloor,0} \tag{4.40}$$

$$\mathsf{v} = \Phi_{\lfloor(\lambda \bmod 48)/12\rfloor,1} \tag{4.41}$$

onde λ é definido por (4.34) e Φ é a matriz 4x2 definida por

$$\Phi=\begin{bmatrix} 7 & 6 \\ 5 & 4 \\ 3 & 2 \\ 1 & 0 \end{bmatrix} \tag{4.42}$$

Por exemplo, para gerar o símbolo Ψ=12 do segmento Ω=2 da Matriz de Símbolos 8-VSB (Figura 4.12), o Pré-intercalador de Blocos seleciona o par de bits (b^u, b^v), u=1,v=0, no byte B=412 do *buffer* de saída do Intercalador Convolucional, e atribui (b^u, b^v) aos nós de entrada (X_2, X_1) do Codificador Convolucional T=8. O Codificador Convolucional #8 gera então em sua saída a trinca de bits respectiva a (Z_2, Z_1, Z_0) a qual é atribuída pelo Intercalador de Blocos ao elemento $\mathbf{M}_{2,12}$ da Matriz de Símbolos 8-VSB, aqui denominada de **M**. Isto é, (Z_2, Z_1, Z_0) é armazenado na posição definida pela linha Ω=2 coluna Ψ=12 de **M**.

Para otimizar as propriedades de distância entre símbolos [Ungerboeck] representados pelos 8 níveis VSB, cada elemento (Z_2, Z_1, Z_0) da Matriz de Símbolos 8-VSB é transformado para o valor indicado pelo mapeamento da Tabela 4.9.

Z_2	Z_1	Z_0	Nível 8-VSB
0	0	0	-7
0	0	1	-5
0	1	0	-3
0	1	1	-1
1	0	0	+1
1	0	1	+3
1	1	0	+5
1	1	1	+7

Tabela 4.9: Mapeamento $(Z_2, Z_1, Z_0) \Rightarrow$ Nível 8-VSB

Uma vez transformada, a Matriz de Símbolos 8-VSB é enviada para o Multiplexador como um vetor unidimensional de 258336 símbolos 8-VSB. Denominando a Matriz de Símbolos 8-VSB de **M** e o vetor unidimensional de **V**, a correspondência entre os elementos de **M** e **V** é dada por

$$\mathbf{V}_i = \mathbf{M}_{\lfloor i/828 \rfloor,\, i \bmod 828} \ , \quad i = 0, 1, \cdots, 258335 \tag{4.43}$$

sendo os símbolos $\mathbf{V}_0$ e $\mathbf{V}_{258335}$ respectivamente o primeiro e o último símbolos a serem enviados ao Multiplexador. O Multiplexador insere as seqüências de sincronismo conforme descrito anteriormente gerando então um Campo completo de símbolos 8-VSB.

4.5.3 Decodificador

Nesta seção analisamos a etapa de decodificação do receptor 8-VSB, conforme Figura 4.7.

A saída do bloco de Rastreamento de Fase é colocada em um vetor **V** de 258336 símbolos cada um representado por um dos 8 níveis VSB. A Matriz de Símbolos 8-VSB **M** do receptor (Figura 4.14) é obtida de **V** obedecendo (4.43). A seguir, cada um dos elementos da matriz é transformado para a trinca de bits (Z_2, Z_1, Z_0) usando o mapeamento inverso da Tabela 4.9. O conjunto de 12 Decodificadores de Viterbi em conjunto com o Intercalador de Bloco converte os 312 x 828 símbolos da Matriz 8-VSB nos 64584 bytes do *buffer* de entrada do Intercalador Convolucional Inverso, conforme mostrado na Figura 4.14. O símbolo Ψ do segmento Ω da Matriz de Símbolos 8-VSB **M** é selecionado pelo Pré-intercalador Inverso de Blocos, o qual atribui os 3 bits do símbolo aos nós de entrada (Z_2, Z_1, Z_0) do Decodificador de Viterbi *T*. O Intercalador Inverso de Blocos atribui o valor dos nós de saída (X_2, X_1) do Decodificador de Viterbi *T* ao par de bits (b^u, b^v), no byte *B* do *buffer* de entrada do Intercalador Convolucional Inverso.

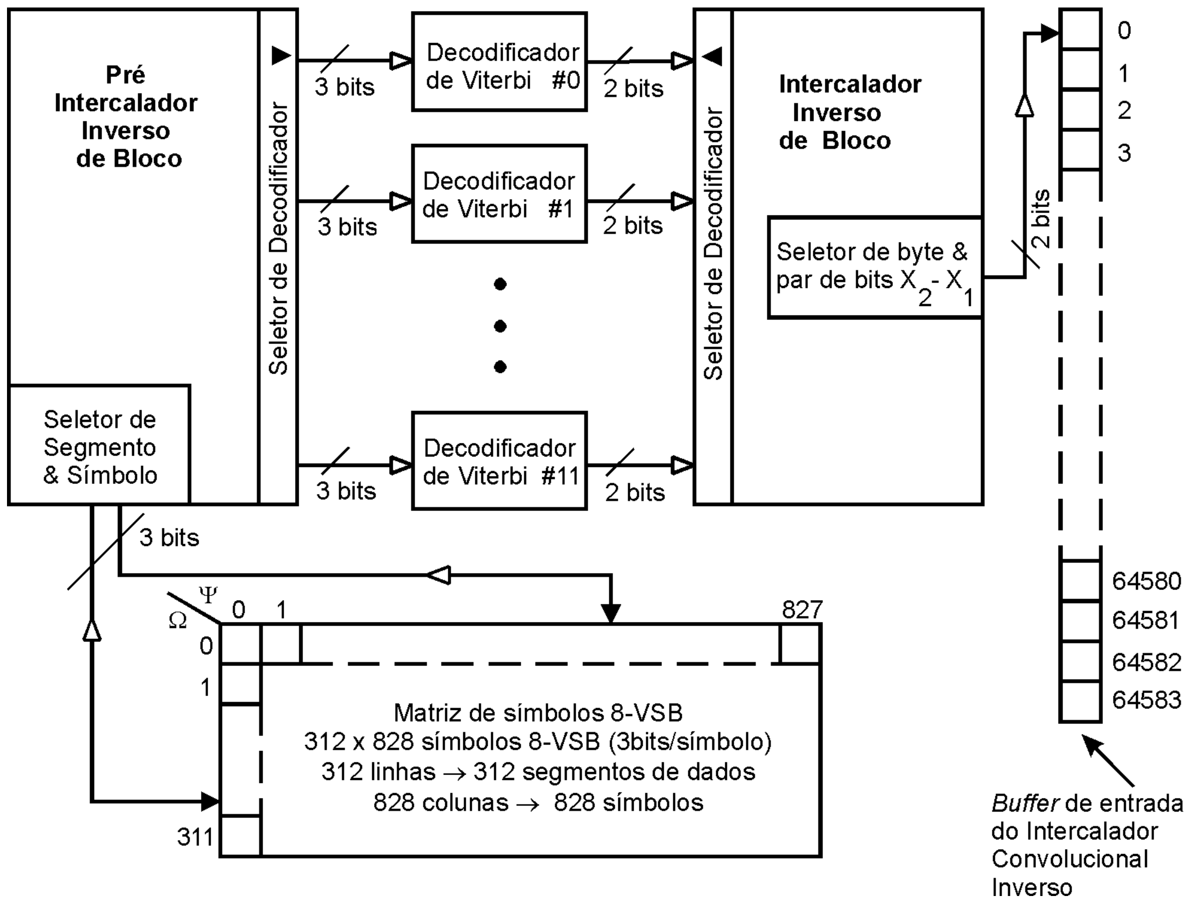
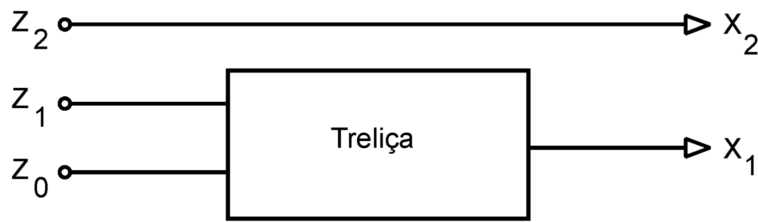
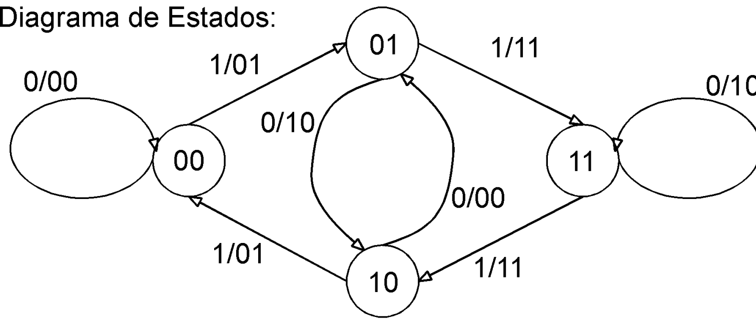


Figura 4.14: Intercalador Inverso de Blocos e grupo de 12 Decodificadores de Viterbi em paralelo.





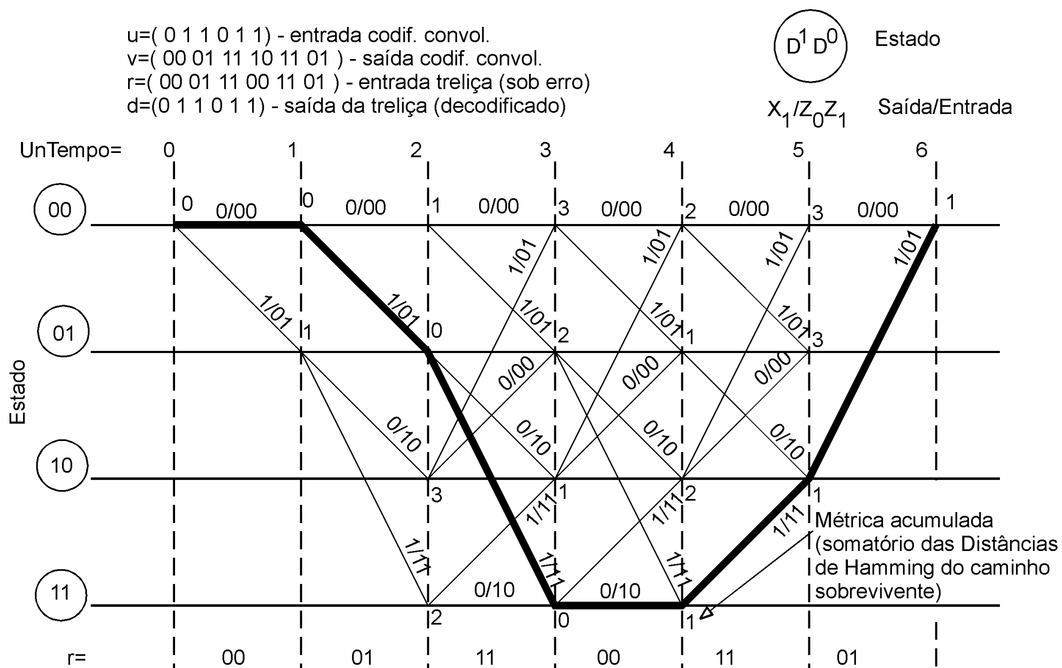


Figura 4.15: Diagrama interno de um dos 12 idênticos Decodificadores de Viterbi [Costello] mostrados na Figura 4.14. É mostrado o diagrama de estados que rege a treliça, o qual é associado ao Codificador Convolucional da Figura 4.13 [Clark]. No exemplo de operação mostrado, u é a seqüência na entrada X_1 do codificador da Figura 4.13, v é a saída do codificador (pares Z_0-Z_1), r é a seqüência v recebida com erro na entrada de treliça e d é a saída decodificada (e corrigida) pela treliça. A métrica utilizada é a Distância de Hamming.

A relação analítica que define *B*, *T,* u e v como função de Ω e Ψ é determinada pelas equações (4.32) a (4.42).

O Intercalador Convolucional Inverso é idêntico ao intercalador mostrado na Figura 4.11, exceto que a posição das chaves com relação à ordem do banco obedece à seqüência $[51,50,\cdots,0,51,\cdots]$, sendo a posição 51 inicial sincronizada com o primeiro byte do *buffer* de entrada. O *buffer* de saída do Intercalador Convolucional Inverso é uma fila (*queue*) com capacidade para 2×64584 bytes, ou seja, capacidade para 2 Campos de dados completos, conforme mostra a Figura 4.16. A saída é tomada 10608 bytes em atraso com relação ao início da fila de forma a compensar o atraso intrínseco ao Intercalador Convolucional.

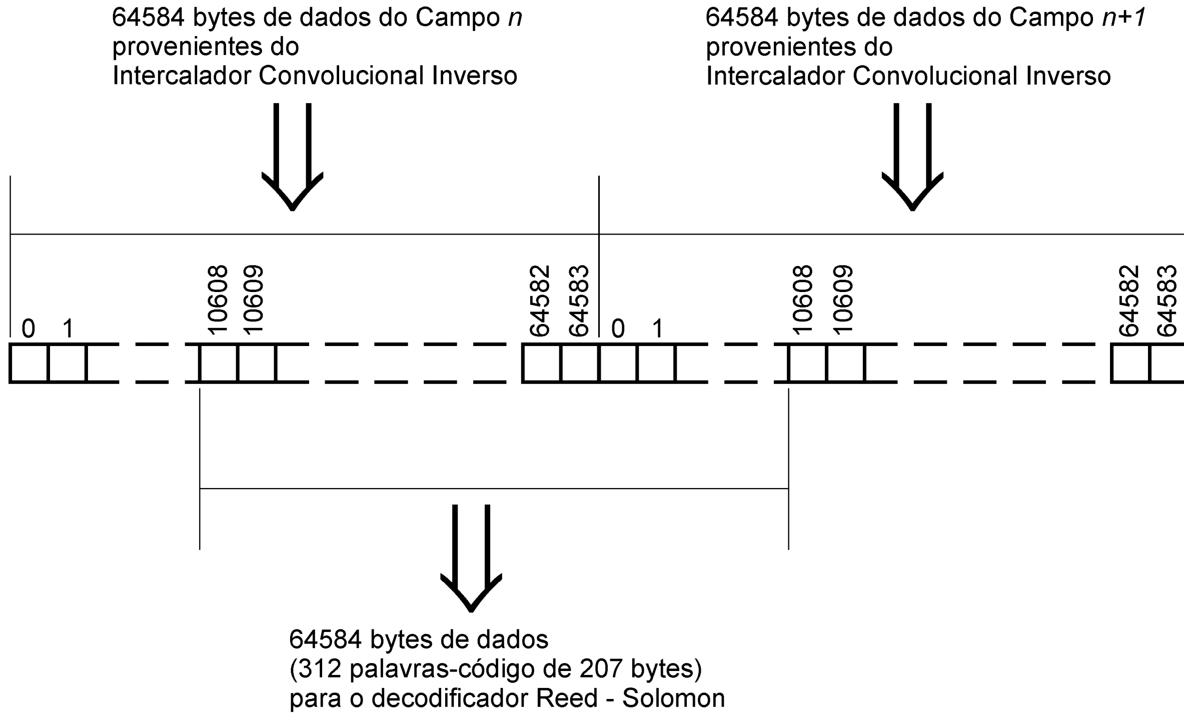


Figura 4.16: Compensação do atraso de 10608 bytes inserido pelo Intercalador Convolucional.

O Decodificador Reed-Solomon então decodifica e corrige (até 10 bytes sob erro) as 312 palavras-código provenientes do *buffer* de saída do Intercalador Convolucional Inverso gerando 312 mensagens de 187 bytes.

Finalmente, o Desrandomizador faz a operação lógica ou-exclusivo (XOR) entre os bits de cada byte das 312 mensagens e os bits do byte de saída gerado pelo gerador de seqüência pseudo-randômica mostrado na Figura 4.10. Cada uma das mensagens de 187 bytes é então enviada para processamento na camada de transporte do decodificador MPEG-2.

A decodificação concatenada Viterbi/Reed-Solomon com Intercalador intermediário exibe melhor capacidade de correção de erros em um canal com ruído gaussiano aditivo do que qualquer sistema corretor de erros de complexidade semelhante [Clark]. Isto ocorre basicamente devido às características de desempenho dos dois sistemas individuais serem aproximadamente complementares. Por exemplo, devido à natureza multi-bit de seus símbolos, os códigos Reed-Solomon tem sua eficiência de decodificação maximizada quando os erros a serem corrigidos ocorrem em rajadas (*bursts)* curtas de bits, mas perdem eficiência quando os erros nos bits ocorrem de maneira descorrelacionada no tempo.

No entanto esta deficiência é compensada pelo Decodificador de Viterbi, o qual é bastante adequado para erros deste tipo. Por outro lado, o Decodificador de Viterbi também falha quando tem a sua capacidade de correção de erros excedida, gerando nesta situação uma longa seqüência de erros em sua saída [Clark]. Esta seqüência de erros é em geral muito mais longa do que as curtas rajadas de bits que maximizam a eficiência do código Reed-Solomon, podendo inclusive exceder o número máximo de correção de símbolos. Ou seja, se nesta situação de falha do decodificador de Viterbi aplicarmos sua saída diretamente na entrada do decodificador Reed-Solomon este último muito provavelmente falhará na correção dos símbolos. Isto ocorre porque as palavras-código em sua entrada apresentarão alta correlação temporal entre os símbolos em erro [Michelson], apresentando grande chance de exceder a capacidade de correção do código Reed-Solomon. Este problema é solucionado pelo Intercalador Inverso inserido entre os dois decodificadores, o qual "embaralha" a seqüência de entrada do decodificador Reed-Solomon minimizando qualquer eventual correlação entre símbolos em erro originado por falha do Decodificador de Viterbi.

4.5.4 Resultados Experimentais

Nesta seção analisaremos o desempenho conjunto das etapas de codificação/decodificação através de simulação. A saída da Etapa de Codificação (Figura 4.6) é conectada à entrada da Etapa de Decodificação (Figura 4.7) através de um gerador de ruído Gaussiano aditivo. O simulador determina a variância E_s de cada Campo 8-VSB na saída da Etapa de Codificação e adiciona ruído Gaussiano de variância N_o de acordo com a razão E_s/N_o estipulada. Então é feita a contagem de quantas Palavras-Código Reed-Solomon não foram corrigidas com sucesso na Etapa de Decodificação e é determinada a taxa de erro de bits BER (*Bit Error Rate*). A BER é definida pela razão entre o número de bits decodificados erroneamente e o número total de bits decodificados.

O simulador obtém cada ponto da curva BER=f(E_s/N_o) mostrada na Figura 4.17 utilizando como critério de parada 5×10^6 bits sob erro ou 500×10^6 bits processados.

Na determinação da BER, esta seção segue a heurística proposta por Odenwalder [Odenwalder], a qual é adequada para códigos concatenados Viterbi/Reed-Solomon. São assumidos os seguintes eventos em caso falha do decodificador Reed-Solomon ao tentar corrigir uma Palavra-Código recebida sob erro:

1- São adicionados $(n-k)/2$ =10 bytes = 80 bits incorretos à Palavra-Código (pelo fato de o decodificador Reed-Solomon "corrigi-la" para uma Palavra-Código incorreta).

2- Todos os bits em um byte incorreto são também incorretos.

3- Todos os bytes incorretos em uma Palavra-Código ocorrem nos 187 bytes correspondentes à Mensagem.

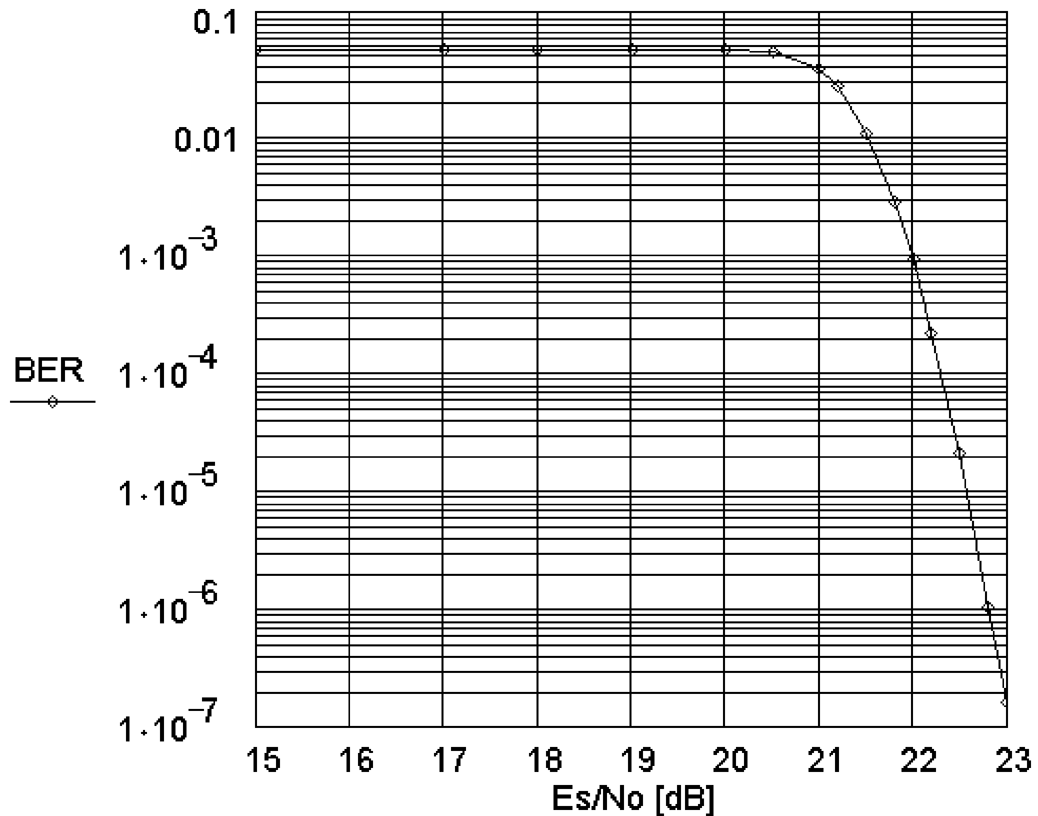


Figura 4.17: Desempenho normalizado do codec 8-VSB com um gerador de bytes aleatórios (distribuição uniforme) aplicado em sua entrada. Observe que a curva BER=f(E_s/N_o) é bastante íngreme se comparada com aquela resultante de um Decodificador de Viterbi não-concatenado com um Decodificador Reed-Solomon (*cliff effect*).

É importante lembrar que a capacidade de correção de erros de um Decodificador de Viterbi aumenta com o número de bits do *shift-register* do Codificador Convolucional (*constraint length*), mas a complexidade computacional do decodificador cresce exponencialmente com este parâmetro [Michelson]. Portanto, um Decodificador de Viterbi com performance semelhante à de um Decodificador Concatenado Viterbi/Reed-Solomon apresentará custo computacional bastante elevado visto que o *constraint length* necessário para equiparar as performances não será pequeno.

Note que o desempenho do codec 8-VSB é obtido em parte às custas de um atraso de 10608 bytes = 84864 bits que é intrínseco à operação do Intercalador Convolucional. Este atraso pode ser inaceitável para um sistema bidirecional de banda estreita para voz por exemplo, porque a cada novo fluxo de dados os bancos dos intercaladores devem ser totalmente preenchidos antes de enviarem informação útil. Para um sistema TDM (*Time Division Multiplex*) onde a taxa de bits enviados ao canal é bastante alta ou para um sistema de vídeo contínuo como o DTV 8-VSB da ATSC onde 10608 bytes equivalem a 4 ms, este atraso é perfeitamente admissível.

Foi investigado também neste estudo o desempenho do codec 8-VSB quando submetido à uma longa seqüência de ruído impulsivo. O objetivo é determinar quantos

Segmentos de Dados consecutivos (excluindo os sinais de sincronismo de segmento e campo) podem ser totalmente corrompidos por ruído sem que o codec falhe. O ruído impulsivo aplicado a cada símbolo de um segmento a ser corrompido foi aproximado mediante a seguinte heurística: se o símbolo 8-VSB for positivo o simulador o substitui pelo valor -7, caso contrário ele é substituído pelo valor 7.

Determinou-se experimentalmente que até 3 Segmentos de Dados completos e consecutivos podem ser corrompidos por ruído definido desta forma sem que ocorresse falha do codec.

Finalmente, é importante ressaltar que o desempenho aqui avaliado é apenas o da etapa de processamento de dados do sistema DTV 8-VSB da ATSC. A avaliação do sistema como um todo, incluindo a etapa de processamento de sinal e as etapas de RF, envolve metodologia distinta daquela aqui empregada.

4.5.5 Conclusão

Este estudo analisou e avaliou o desempenho da etapa de codificação e decodificação concatenada do sistema de transmissão DTV 8-VSB da ATSC. Os resultados mostraram que esta etapa do sistema é bastante robusta tanto quanto à ruído gaussiano aditivo como à ruído impulsivo.

4.6 Referências Bibliográficas

[IEC1] ISO/IEC IS 13818-1, International Standard (1994), *MPEG-2 Systems.*

[IEC2] ISO/IEC IS 13818-2, International Standard (1994), *MPEG-2 Video.*

[Ungerboeck] G. Ungerboeck. Channel coding with multilevel/phase signals. *IEEE Transactions on Information Theory,* pp. 55-67, vol. IT-28, No. 1, January 1982.

[INTEL] INTEL Application Note AP-269. *Using MMX Instructions to Implement Viterbi Decoding*. INTEL, 1996.

[Wicker] Wicker. *Error Control Systems for Digital Communication and Storage.* Prentice-Hall, 1995.

[Wilson] Wilson. *Digital Modulation and Coding*. Prentice-Hall, 1995.

[Sgrignoli] G. Sgrignoli. *ATSC Transmission System: VSB Tutorial*. Zenith Electronics Corporation. http://www.zenith.com/main/cool/tech/vsbtutor/
vsbtutor.htm

[Michelson] A.M. Michelson and A.H. Levesque. *Error Control Techniques For Digital Communication*. Wiley & Sons, 1985.

[Odenwalder] J. P. Odenwalder. *Error Control Coding Handbook (Final Report)*. Linkabit Corp. report for USAF, 1976.

[Ash] R. Ash, *Information Theory*, Interscience – John Wiley & Sons, 1967.

[Proakis] J. G. Proakis, *Digital Communications*, McGraw-Hill, 1995.

[Taub] H. Taub and D.L. Schilling, *Principles of Communications Systems*,

McGraw-Hill, 1986.

[Chen] Chi-Tsong Chen, *Linear System Theory and Design*, Harcourt Brace College Publishers, 1984.

[Costello] S. Lin and D. J. Costello Jr., *Error Control Coding*. Prentice-Hall, Englewood Cliff, 1983.

[Clark] G. C. Clark Jr and J. B. Cain. *Error-Correction Coding for Digital Communications.* Plenum Press, 1988.

[Peterson] W. W. Peterson and E. J. Weldon Jr. *Error-Correcting Codes.* MIT Press, 1990.

[Messerschmitt] E.A.Lee and D.G. Messerschmitt, *Digital Comunication*, Kluwer Academic Publihers, 1988.