Forward error correction in the ETSI standard for digital terrestrial television

Oome, R.O.J.

Award date:
1997

Technische Universiteit Eindhoven

Faculty of Electrical Engineering
Section of Information and Communication Systems

Master's Thesis:

Forward Error Correction in the ETSI Standard for Digital Terrestrial Television

R.O.J. Oome

Coach : Ir. W. Renirie
Supervisor : Prof. J. Jess
Period : March 1996 - February 1997

Abstract

In the ETSI-standard for digital terrestrial television, compatible with MPEG-2 coded TV signals, a baseline transmission system is specified for channel coding/modulation.
For Forward Error Correction (FEC) at the transmitter, a concatenated code is applied, consisting of a shortened Reed-Solomon code RS(n=204, k=188, t=8), and a variable-rate punctured convolutional code with constraint length 7. Performance is further improved by a convolutional Outer Interleaver (Forney approach), and a block-based Inner Interleaver.
In this paper, the signal processing for error correction at the receiver side is investigated, leading to a specification/algorithms for the different decoder-blocks, modeled in the C programming language, and some architecture considerations for a hardware implementation, that all meet the global performance requirements. The performances of the codes are investigated, to find optimal decoder settings.
The Reed-Solomon decoder is modeled, using the two most practically applied decoding-algorithms of this moment: the Euclidean algorithm and the Berlekamp-Massey algorithm.
The convolutional code is decoded using a Viterbi-algorithm, where test-results show that soft-decision must be used to meet the performance requirements.
Soft-decision is investigated up to 7 bit per coded bit, with special attention to quantization-borders in the different Gray-mapped OFDM constellations. Test-results show that a 3-bit soft-decision per coded bit suffices to give a satisfying performance.

Contents

1. Introduction

It is a virtual certainty that wireline and fiber communication will be fully digital by the end of the century. It is almost as likely that the same will be true for most wireless communication. Digital communication provides for excellent reproduction and greatest efficiency of transmission bandwidth and power through effective utilization of two fundamental techniques: the first consists of source compression coding to greatly reduce the transmission rate for a given degree of fidelity; the second is error control coding to further reduce signal-to-noise and bandwidth requirements.

The *European Television Standard Institute* (ETSI) has recently released the draft specification for digital terrestrial television [11]. This specification describes a complete transmission scheme based upon the *Coded Orthogonal Frequency Division Multiplexing* (COFDM) system, and allows for the use of either 1705 carriers (usually known as '2k') or 6817 carriers ('8k') per channel. These multiple carriers can be used to transmit large amounts of digital data representing video, audio, text, graphics or general data to receivers which may use roof-top aerials or simple indoor antennas. The system is designed to operate within the existing UHF spectrum, with 8 Mhz channel spacing.

The digital terrestrial specification is closely related to the existing ETSI standard specifications for satellite [9] and cable [10] systems, but uses flexible modulation and channel coding, and an additional signaling system.
The terrestrial channel capacity can be tailored to the needs of any particular broadcasting service, by the use of switchable error correction rates, modulation levels, and guard intervals; a receiver must automatically configure itself to the transmitted signal.
The first terrestrial broadcasts will most probably start in the United Kingdom as early as 1997, where the '2k' variant will be used. In other countries, e.g. Spain, the '8k' variant may be introduced later.

The subject I focused on is the functional block after demodulation where de-mapping and error-correction is performed, which I will call the 'FEC-decoder'.
My graduation work consists of specifying the algorithms for the FEC-decoder, that perform the required level of error correction, testing and optimizing these algorithms by modeling them functionally in the C programming language, and investigating some architecture structures.

For a good understanding of the hows-and-whies of the baseline system, I will give a brief description of the ETSI specifications with additional comments on what purpose these specifications serve, and omit details not concerning or irrelevant for the FEC-decoder. At this point it must be said that the reasons for particular choices in the baseline system specifications were not mentioned for commercial reasons of the participants of the DVB-group.

2. Baseline Transmission System Specifications

The ETSI specification [11] describes a ***baseline transmission system*** for digital terrestrial television broadcasting. It specifies the channel coding/modulation system intended for digital multi-programme Limited (LDTV) / Standard (SDTV) / Enhanced (EDTV) / High (HDTV) Definition TV terrestrial services.
The scope of the specification is as follows:

- It gives a general description of the Baseline System for digital terrestrial TV;
- It identifies the global performance requirements and features of the Baseline System, in order to meet the service quality targets;
- It specifies the digitally modulated signal, by describing in detail the signal processing at the modulator side, while the processing at the receiver side is left open to different implementation solutions. Still, certain aspects of the reception are referred to.

2.1 Baseline System

The system performs the adaption of the baseband TV signals from the output of the MPEG-2 transport multiplexer, to the terrestrial channel characteristics. Therefore, it is required that the system provides sufficient protection against high levels of Co-Channel Interference (CCI) and Adjacent-Channel Interference (ACI) emanating from existing PAL/SECAM services.
The system is directly compatible with MPEG-2 coded TV signals.

The following processes are applied to the data stream (see [11]):

1. Transport multiplex adaption and randomization for energy dispersal;
2. Outer coding (i.e. Reed-Solomon code);
3. Outer interleaving (i.e. convolutional interleaving);
4. Inner coding (i.e. punctured convolutional code);
5. Inner interleaving;
6. Mapping and modulation;
7. Orthogonal Frequency Division Multiplexing (OFDM) transmission.

The 7^{th} process is not part of our direct investigation, but must be examined for performance concerns.

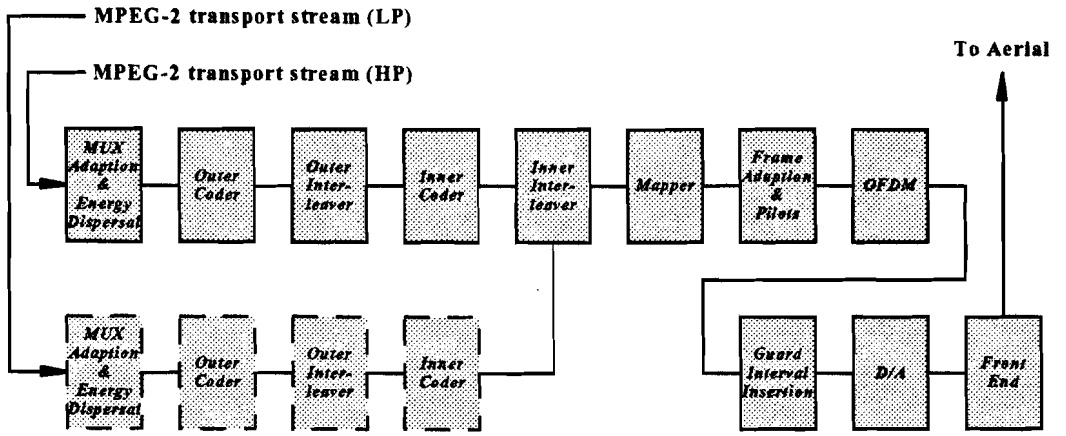


Figure 2-1: Functional Block Diagram of the Terrestrial Baseline System (Transmitter).

To maximize commonality of the Terrestrial Baseline System [11] with the Satellite [9] and Cable Baseline Specifications [10], the outer coding and outer interleaving of all three systems are common, and the inner coding is common with the Satellite Baseline Specification.

The system allows different levels of *Quadrature Amplitude Modulation* (QAM) and different inner code rates to adjust the bit rate to the current channel quality. The system also allows two-level hierarchical channel coding and modulation, including uniform and multi-resolution constellation, in which case the system must be expanded to include the modules shown dashed in figure 2-1. Then, two bitstreams, the *high-priority* (HP) and the *low-priority* (LP) bitstream, are fed to the inner interleaver.

The FEC-unit in the **receiver** however, requires only one set of the inverse elements:
1. Inner De-interleaver;
2. Inner Decoder;
3. Outer De-interleaver;
4. Outer Decoder;
5. Multiplex Adaption.

The only additional requirement thus placed on the receiver is the ability for the demodulator/de-mapper to produce one stream selected from those mapped at the sending end, i.e. HP or LP. The negative aspect of this receiver economy is that the decoding process cannot switch from LP to HP or v.v. while continuously decoding and presenting pictures and sound: a pause will be necessary (ETS estimation: video freeze frame for approximately 0.5 s, audio interruption for approximately 0.2 s) while the inner decoder and the various source decoders are suitably reconfigured and reacquire lock.

2.2 Channel Coding and Modulation

204

2.2.1 Transport Multiplex Adaption and Randomization

The Terrestrial Baseline System (see figure 2-1) receives fixed-length (188 bytes) MPEG-2 transport packets, including 1 sync-word byte (47_{HEX} = 01000111_b) as the first byte in each packet.
In order to ensure **adequate binary transitions**, the data of the transport packets are randomized in accordance with the configurations depicted in figure 2-2, where "⊕" performs a logical XOR, and "⊗" performs a logical AND. This diagram can as well be used for the *de-randomization*: process in the receiver: in that case the Data Input is randomized, so the Data Output would not be.

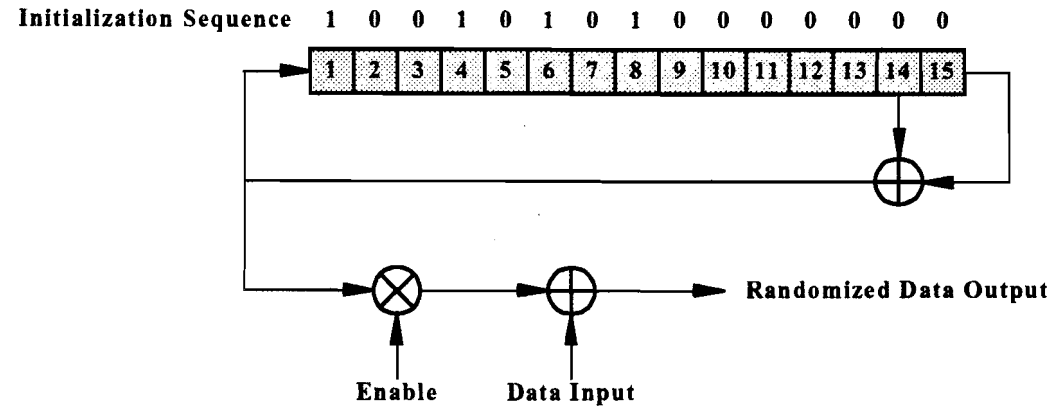


Figure 2-2: Scrambler/Descrambler Schematic Diagram

The polynomial for this *Pseudo Random Binary Sequence* (PRBS) generator equals $1 + x^{14} + x^{15}$, and the shift-registers are initially loaded with the sequence "100101010000000" at the start of every eight transport packets. The PRBS is XOR-ed with the transport packets, beginning with the second byte in every 8-packet (1504 bytes) period. The first (sync-word) byte is inverted to $B8_{HEX} = 10111000_b$, to provide an **initialization signal for the Descrambler**.
The 7 other sync-word bytes are preserved by temporarily disabling the addition (yet continue the generation) of the PRBS, to aid **other synchronization functions** further on in the signal processing, by control of the "Enable" signal. So, for initialization and generation of the enable signal, a simple counter will suffice.
The randomization process will be active also when the modulator input bit-stream is non-existent, or when it is non-compliant with the MPEG-2 transport stream format (i.e. 1 sync byte + 187 packet bytes).

2.2.2 Outer Coding: Reed-Solomon Code

The outer code is a Reed-Solomon RS(n=204, k=188, t=8) shortened code, and is performed on the input packet structure. All RS codes are *block-based codes*, meaning that for a fixed-length (k = 188 bytes) input message word (here: a transport packet), a fixed length (n=204 bytes) output code word is produced. Also, all RS codes are *systematic*, meaning that the original message is preserved in the codeword and $n-k$ parity-bytes are chained to each message word.
All RS-codes are *Minimum Distance Separable* (MDS) codes, meaning that the RS code allows to correct up to $t = (n-k)/2$ random erroneous bytes in a received word of n bytes.
Details of Reed-Solomon Codes will be discussed later on.

2.2.3 Outer Interleaving: Forney Interleaver

The convolutional interleaving process is based on the Forney approach which is compatible with the Ramsey type III approach. The interleaved data bytes are composed of error protected packets and are delimited by inverted or non-inverted MPEG-2 sync-word bytes, thereby preserving the periodicity of N=204 bytes.
The Interleaver may be composed of I=12 branches, cyclically connected to the input byte-stream by the input switch. Each branch j is a *First-in, First-out* (FIFO) shift register, with depth j·M cells where a cell contains one byte and M=N/I=17 bytes.
For synchronization purposes, the sync-word bytes is always routed in the branch j=0 of the Interleaver, corresponding to a null delay.
Of course, the input and output switches are synchronized. In the next figure 2-3 this is pictured.

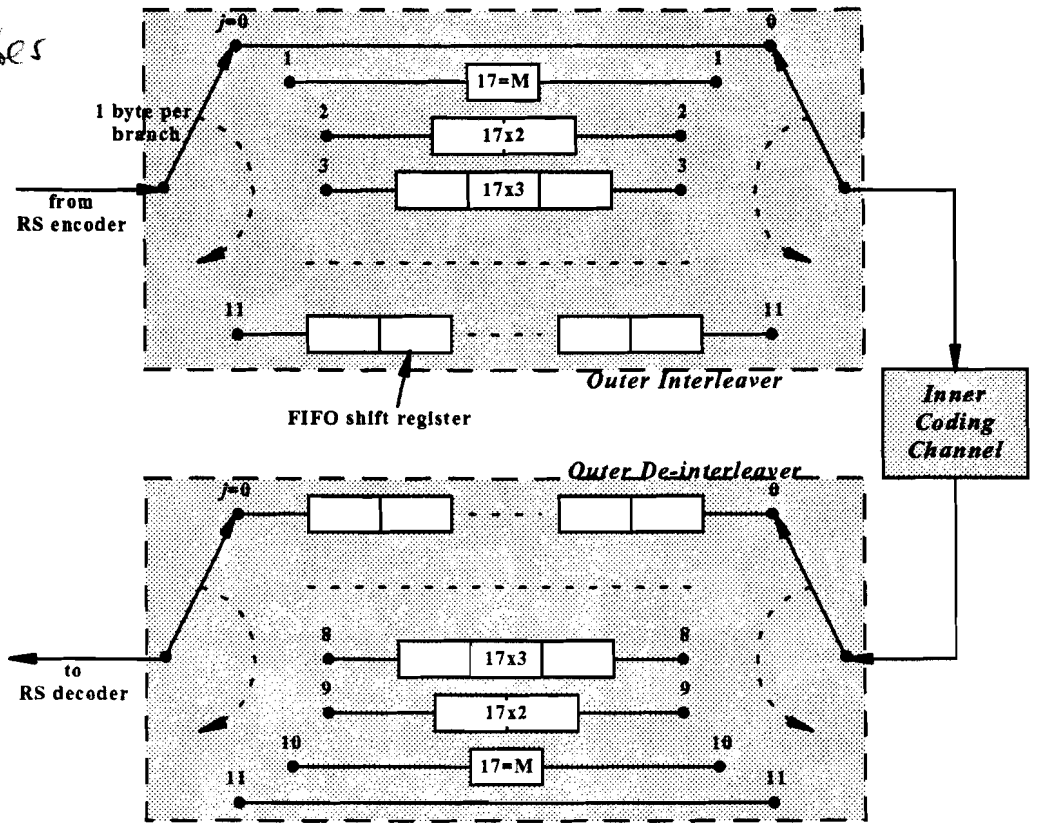


Figure 2-3: Conceptual Diagram of the Outer Interleaver and De-interleaver.

The Outer De-interleaver will be using at least
$0+1\cdot 17+2\cdot 17+\ldots+11\cdot 17 = 17(1+2+\ldots+11) = 1122$ bytes worth of memory, for storage of the data bytes. Branch j=0 is responsible for a delay of 11 packets of 204 bytes before the RS decoder can start decoding.

2.2.4 Inner Coding: Convolutional Code

The inner code comes from a range of punctured convolutional codes, based on a widely used [27] mother convolutional code of rate 1/2 with constraint length k=7 and 2^{k-1}=64 states. This will allow selection of the most appropriate level of error correction for a given service or data rate in either non-hierarchical or hierarchical transmission mode.
For convolutional codes, each output bit is a logical function of the "sliding window" over the k most recent input bits. For codes with rate $1/n$, each input bit results in n output bits, constructed by n, not necessarily all different, functions over the same k input bits. Because of the sliding window concept, convolutional codes are *not block-based.*
The used mother code is *non-systematic*, meaning that the original message is not literally included in the output. The other codes are constructed by omitting some bits in the output of the mother code, so they are non-systematic as well.
The possible code rates are 1/2, 2/3, 3/4, 5/6, and 7/8. The *minimum free distance* d_{free} for these code rates is 10, 6, 5, 4, and 3 respectively, and is a measure for the error-correction capability.
The mother code is a *transparent convolutional code*, which means that complements of codewords also are codewords. This property is especially useful for the Satellite Standard [9], where *π-phase ambiguity* tends to occur, meaning that bits are received inverted. The sync-word bytes show whether a phase reversal has occurred or not, to facilitate correction.
Convolutional codes are very well suited as inner codes in a concatenated coding scheme, because the bit-errors in the convolutional decoder output are very cluttered, resulting in less byte-errors than is the case with a same amount of randomly spread bit-errors.

2.2.5 Inner Interleaving

The Inner Interleaver actually consists of two separate interleavers: the Bit Interleaver and the Symbol Interleaver.

The input to the ***Bit-wise Interleaver*** consists of up to two bit streams: just HP, or HP *and* LP.
In the next figure 2-4, a block diagram of the inner interleaving and mapping-process for non-hierarchical transmission modes is given[1].

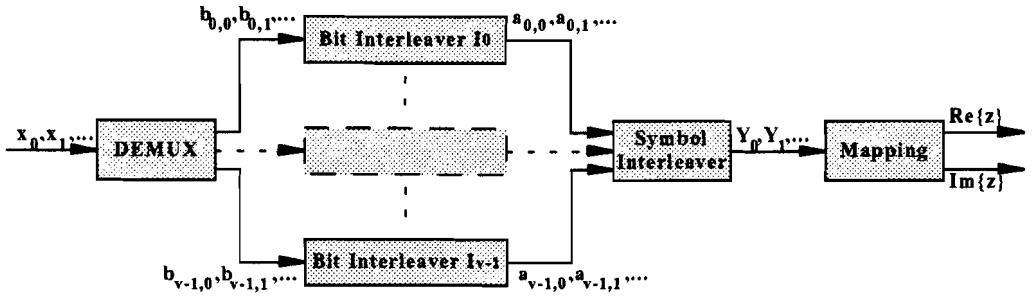


Figure 2-4: Mapping of the input bits onto output modulation symbols, for non-hierarchical transmission modes.

Demultiplexing.
First, the input is demultiplexed into v sub-streams, where v=2 for QPSK, v=4 for 16-QAM, and v=6 for 64-QAM. Demultiplexing is defined as a mapping of the input bits, x_{di} onto the output bits $b_{e,do}$, where di is the input bit number, $e \in \{0,\ldots,v-1\}$ is the demultiplexed bit stream number, do is the bit number of a given stream at the output of the Demultiplexer.
In *non-hierarchical mode*, the single input stream x_{di} is demultiplexed into v sub-streams $b_{e,do}$, with

$$x_{di} = b_{[di \bmod v]\mathrm{div}(v/2)+2[di \bmod (v/2)],\ di \mathrm{div} v}$$

In *hierarchical mode*, the HP stream x'_{di} is demultiplexed into 2 sub-streams, with

$$x'_{di} = b_{di \bmod 2,\ di \mathrm{div} 2},$$

the LP stream x''_{di} is demultiplexed into v-2 sub-streams, with

$$x''_{di} = b_{[di \bmod v-2]\mathrm{div}((v-2)/2)+2[di \bmod ((v-2)/2)]+2,\ di \mathrm{div}(v-2)}$$

This applies in both uniform and non-uniform QAM modes.
In the above formulas, **mod** is the integer modulo operator, and **div** is the integer division operator.

Bit interleaving.
Each of the v substreams is processed by a separate bit interleaver, labeled I_0 to I_{v-1}. I_0 and I_1 are used for QPSK, I_0 to I_3 are used for 16-QAM, and I_0 to I_5 are used for 64-QAM. The block size is the same for each bit interleaver (126 bits), but the interleaving sequence is different in each case. For each bit interleaver I_e, the input bit vector is defined by:

$$B(e) = \left(b_{e,0}, b_{e,1}, b_{e,2}, \ldots, b_{e,125}\right), \quad e \in \{0,\ldots,v-1\}$$

The interleaved output vector for each bit interleaver I_e is

[1] In *hierarchical transmission modes*, we have two parallel Demultiplexers instead of one.

$$A(e) = \left(a_{e,0}, a_{e,1}, a_{e,2}, \ldots, a_{e,125}\right).$$

where

$$a_{e,w} = b_{e,H_e(w)}, \quad w = 0,\ldots,125$$

and $H_e(w)$ is a permutation function which is different for each bit interleaver:

$$\begin{aligned} H_0(w) &= w \\ H_1(w) &= (w+63) \bmod 126 \\ H_2(w) &= (w+105) \bmod 126 \\ H_3(w) &= (w+42) \bmod 126 \\ H_4(w) &= (w+21) \bmod 126 \\ H_5(w) &= (w+84) \bmod 126 \end{aligned}$$

Hence, the output from the bit interleavers is a sequence of v-bit words or *data symbols*.
The 6 bit interleavers will be using 6·126=756 bits for storage, and the delay will be 126 bits.

Symbol interleaving.
Depending on the transmission mode, the symbol interleaver acts on blocks of N_{MAX} data symbols, where N_{MAX}=1512 for 2k mode, and N_{MAX}=6048 for 8k mode. Thus in the 2k mode, 12 groups of 126 data symbols from the bit interleavers are read sequentially into a vector Y', and in the 8k mode 48 groups of 126 data symbols from the bit interleavers are read sequentially into a vector Y', where $Y' = \left(Y'_0, Y'_1, Y'_2, \ldots, Y'_{N_{max}-1}\right)$, and $Y'_q = \left(a_{0,q}, a_{1,q}, \ldots, a_{v-1,q}\right)$ for $q = 0,\ldots, N_{MAX} - 1$.
The *interleaved vector* $Y = \left(Y_0, Y_1, Y_2, \ldots, Y_{N_{max}-1}\right)$ is part of an *OFDM symbol*, which contains one interleaved vector and some signaling information.

An *OFDM frame* consists of 68 OFDM symbols, numbered from 0 to 67. Four OFDM frames constitute one *OFDM super-frame*. The concept of a super-frame is chosen because an OFDM super-frame always contains an integer number of Reed-Solomon code words, no matter what code rate or constellation is used. This avoids the need for stuffing. Further, it is convened that the first data byte transmitted in an OFDM super-frame is a sync word byte: this gives the Viterbi decoder in the receiver a handle for synchronization.

The interleaved vector $Y = \left(Y_0, Y_1, Y_2, \ldots, Y_{N_{max}-1}\right)$ is defined by using a mapping permutation function $H(q)$ on the data symbol indices $q = 0,\ldots, N_{MAX} - 1$:
For *even* numbered OFDM symbols $Y_{H(q)} = Y'_q$; for *odd* numbered OFDM symbols $Y_q = Y'_{H(q)}$.
The fact that the permutation is inverted for each pair of consecutive OFDM symbols, makes in-place scheduling of the memory, containing the data symbols, possible. The same applies for the symbol de-interleaver in the receiving decoder. The amount of memory needed for storage of the data symbols thus equals $\mathrm{MAX}(v \cdot N_{MAX})\,[\mathrm{bit}] = 6 \cdot 6048\,[\mathrm{bit}] = 4536\,[\mathrm{byte}]$.
The permutation function H(q) is defined by a (hard-wired) bit permutation on the contents of a feedback shift register. This shift register "spreads" consecutive data symbols, and is different for the 2k and 8k mode. Details are given in [11].

The purpose that symbol interleaving serves is that burst errors introduced by noise and typical channel characteristics are spread as randomly as possible within an OFDM symbol: this improves the performance of the Viterbi decoder.

2.2.6 Signal Constellations and Mapping

The symbol interleaver delivers interleaved vectors $Y = \left(Y_0, Y_1, Y_2, \ldots, Y_{N_{max}-1}\right)$, consisting of v-bit data symbols, to the mapper. The system uses *Orthogonal Frequency Division Multiplex* (OFDM) *Transmission*, with Gray mapping. Each data symbol y_q is mapped to one data carrier.
In this OFDM schematic, a fixed set of $M=2^v$ complex signals, called a *signaling constellation* (or a *modulation alphabet*), has been chosen to transmit the v-bit data symbols, where v=2, 4 or 6. For the modulation method *Quaternary Phase Shift Keying* (QPSK) is chosen for 2-bit data symbols, 16-ary *Quadrature Amplitude Modulation* (16-QAM) is chosen for 4-bit data symbols, and 64-QAM is chosen for 6-bit data symbols. There are 2 types of M-ary QAM: uniform QAM and non-uniform QAM. The uniform constellations are shown below in figure 2-5, where the dots denote the tips of signal phasors, i.e. they represent all possible data symbol values in the complex field.

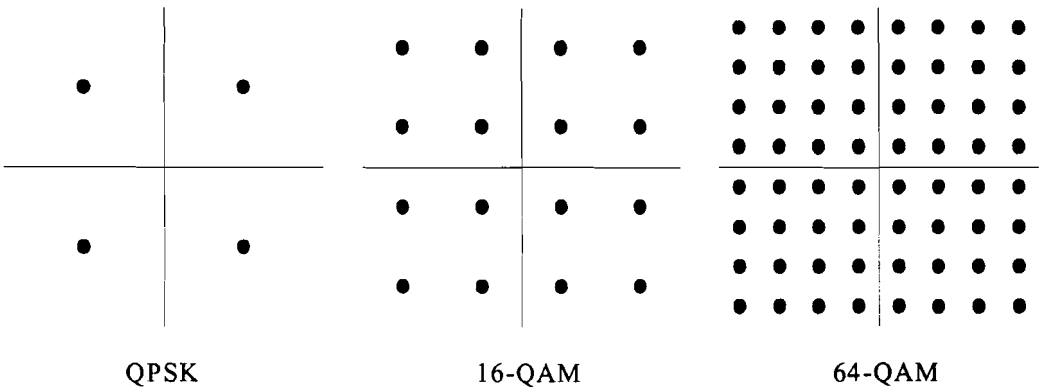


Figure 2-5: Uniform OFDM signal constellations.

The horizontal (real) position $\mathrm{Re}\{z\}$ of a constellation point z is called the *In-Phase Component* (I), and is constructed by using the even numbered bits in the data symbol (y_0,y_2,y_4). The vertical (imaginary) position $\mathrm{Im}\{z\}$ of a constellation point z is called the *Quaternary Component* (Q), and is constructed by using the odd numbered bits in the data symbol (y_1,y_3,y_5).
In uniform M-ary QAM, the relative distance between an axis and the nearest points is defined as 1, so in accordance the distance between 2 adjacent rows (or 2 adjacent columns) of constellation points is 2.
In non-uniform 16-QAM and 64-QAM, each quadrant in the constellation keeps the same uniform spacing internally (=2), but the distance between an axis and the nearest points now can take a value α, where α can take the values 1 (uniform!), 2 and 4.

The v-bit symbols are mapped to the appropriate constellations using *Gray-mapping*, which has an interesting property: the *Hamming distance* between 2 (vertical or horizontal) adjacent constellation points is always 1 (i.e. 2 complex adjacent v-bit data symbols differ in only one of the v bits).
This has important practical implications: for instance, if the complex mapping of a received data symbol lies in-between 2 constellation points, the value of only 1 bit is uncertain, whereas the other v–1 bits are probably correct. If the complex mapping of a received data symbol lies in the center of a square composed by 4 adjacent constellation points, the values of 2 bits are uncertain, whereas the other v-2 bits are probably correct.
So in any received v-bit data symbol within the constellation boundary, at most 2 bit can be pointed out to be uncertain. This can be shown in the next figure:

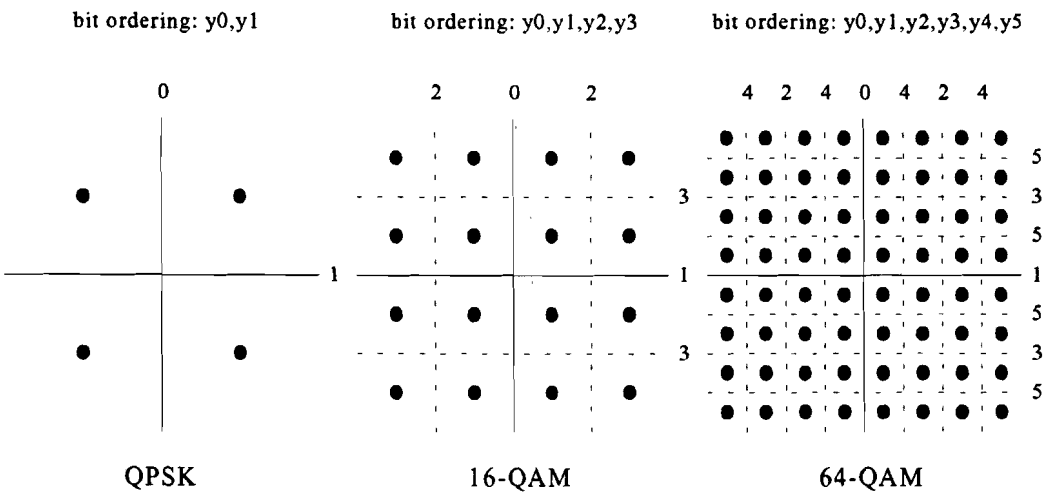


Figure 2-6: Uniform constellation diagrams with Gray coding.

The top-right point in each of these constellation diagrams is the all-zero data symbol[2]:

- QPSK: $Y_j = [y_0,y_1] = [0,0]$;
- 16-QAM: $Y_j = [y_0,y_1,y_2,y_3] = [0,0,0,0]$;
- 64-QAM: $Y_j = [y_0,y_1,y_2,y_3,y_4,y_5] = [0,0,0,0,0,0]$.

For each horizontal or vertical, dotted or straight line that is crossed, one of the v bits is flipped with the index belonging to the line as shown in the figure.

Example:
At the top-right position of the 64-QAM constellation, a step to the left (crossing a "4"-line) results in $Y_j = [0,0,0,0,1,0]$, another step to the left (crossing a "2"-line) results in $Y_j = [0,0,1,0,1,0]$, and then a step down (crossing a "5"-line) results in $Y_j = [0,0,1,0,1,1]$.

2.2.7 Reference Signals

At top-level, a transmission consists of a sequence of *OFDM superframes*. An OFDM superframe corresponds to four consecutive *OFDM frames*, which in turn consist of 68 consecutive *OFDM symbols* each. An OFDM symbol constituted by a set of K=1705 carriers (2K mode) or K=6817 carriers (8K mode).
Various cells within the OFDM frame are modulated with reference information whose transmitted value is known to the receiver. Cells containing reference information are transmitted at "boosted" power level. The information transmitted in these cells are *scattered* or *continual pilot cells*. Each continual pilot coincides with a scattered pilot every fourth symbol; the number of *useful data carriers* is constant from symbol to symbol: 1512 useful carriers in 2k mode and 6048 useful carriers in 8k mode.
Further, *Transmission Parameter Signalling* (TPS) carriers are used for the purpose of signalling parameters related to the transmission scheme, i.e. to channel coding and modulation. The TPS is transmitted at the "normal" power level, in parallel on 17 TPS carriers for the 2k mode and on 68 carriers for the 8k mode. The TPS is defined over one OFDM frame, and each OFDM symbol conveys one TPS bit.
The carrier indices for continual pilots, scattered pilots and TPS carriers in OFDM frames are given in tables in [11].

Relevant TPS includes:

- (current) Frame number [0..3];
- Constellation (QPSK, 16-QAM or 64-QAM);

[2] this is not consistent with ETSI's Cable Standard! [10]

- Hierarchy information (Non-hierarchical, hierarchical with constellation-parameter α=1, 2 or 4);
- Code rate of the high priority or the single non-hierarchical stream (1/2, 2/3, 3/4, 5/6, 7/8);
- Code rate of the low priority stream (1/2, 2/3, 3/4, 5/6, 7/8);
- Transmission mode (2K, 8K).

After reception of the frame, an error correction included in the TPS sequence (with a BCH code) must take place, to determine the correct signalling. A new OFDM frame can be recognised by its changing frame number.

The receiver must wait a half OFDM frame on average, and one OFDM frame worst-case, to receive the first TPS bit. Receiving and decoding all TPS information lasts another OFDM frame duration. An OFDM frame lasts T_F seconds, an OFDM symbol lasts T_S seconds, where T_S=280 µs for 2k mode and T_S=1120 µs for 8k mode worst-case, see [11]. Since $T_F = 68 \cdot T_S$, the TPS information is known after $T = 1.5 \cdot T_F$=28.56 ms (2k) or 114.24 ms (8k) on average, and worst-case after $T = 2 \cdot T_F$=38.08 ms (2k) or 152.32 ms (8k).

The reason for 8k mode transmissions is that because of the longer symbol duration, there is a larger guard interval ("silent" interval inbetween two consecutive symbols). Especially in Single Frequency Networks this results in less problems with echoes.

3. Reed-Solomon Codes

In the coding for *Digital Terrestrial Television* (DTT), a so-called *Reed-Solomon* (RS) code, a member of Bose-Chaudhuri-Hocquenghem (BCH) codes, is used.
This class of codes aim to make detecting and recovering errors, due to transmission, possible at the decoding stage. The main advantage of Reed-Solomon codes constitutes of the high error-correcting capability in particular when dealing with burst-errors, which property makes RS codes highly suitable as the 'outer code' in combination with a *Convolutional* (Trellis) code as the 'inner code'. It so happens that the output coming from a convolutional (Viterbi) decoder used for the 'inner decoding' has a "bursty" nature.
This combination of coding-techniques, which are called *concatenated codes*, has proved itself in practice, as can be seen e.g. from the Voyager expeditions to Uranus and to Neptune.

3.1 Mathematical Basis

Reed-Solomon codes are codes which consist of a sequence of data-bits to whom a sequence of parity-bits are added, in order to make error-correction of the data-bits possible. Algebraic mathematics involving 'finite fields' are used for computing these parity-bits, where in particular 'Galois Fields', discovered by the French mathematician Evariste Galois, play a keyrole in these coding-principles. For clarification I will begin with explaining the development of these mathematics bottom-up from bit to Galois Field.

A bit $b \in \{'0','1'\}$ can be seen (by means of an isomorphism) as an element of a *field* $Z_2 = \{[0],[1]\}$, with the following arithmetic rules:

Table 3-I: **Arithmetic rules for Z_2**

	Addition ⊕ (XOR)		**Multiplication ⊗ (AND)**	
	[0]	[1]	[0]	[1]
[0]	[0]	[1]	[0]	[0]
[1]	[1]	[0]	[0]	[1]

This field can be extended to $Z_2[x]$, the ring of polynomials in indeterminate x over Z_2.

Accordingly, a polynomial $a(x) \in Z_2[x]$ can be denoted as

$$a(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = \sum_{i=0}^{n} a_i x^i$$

where the coefficients $a_0, a_1, \ldots, a_n$ are elements in Z_2. If $a_n \neq [0]$, then this is the leading coefficient and *n is the degree of the polynomial*, denoted $\deg(a(x)) = n$. Further, define $\deg(0) = -\infty$.

Example:

$$a(x) = [1] + [0]x + [1]x^2 = 1 + x^2; \quad \deg(a(x)) = 2$$

A polynomial is said to be *monic* if its leading coefficient is unity. The grammar rules with polynomials are the following. Let

$$p(x) = p_0 + p_1x + \cdots + p_mx^m$$

and

$$q(x) = q_0 + q_1x + \cdots + q_nx^n$$

be polynomials over $Z_2[x]$, then addition is defined as

$$p(x) + q(x) = (p_0 \oplus q_0) + (p_1 \oplus q_1)x + \cdots + (p_n \oplus q_n)x^n + p_{n+1}x^{n+1} + \cdots + p_mx^m$$

for $m \geq n$ (whenever $m < n$, a similar rule applies), and for multiplication we have

$$p(x)q(x) = (p_0 \otimes q_0) + ((p_0 \otimes q_1) \oplus (p_1 \otimes q_0))x + \cdots + (p_m \otimes q_n)x^{m+n}$$

If we consider $\left(j > m \rightarrow p_j = 0\right) \wedge \left(j > n \rightarrow q_j = 0\right)$, then the coefficient a_k of x^k is

$$a_k = \sum_{i=0}^{k} \oplus \left(p_i \otimes q_{k-i}\right)$$

We can construct a *finite field* by taking a field extension of the field Z_2 with the help of an irreducible polynomial $p(x) \in Z_2[x]$. Let $\deg(p(x)) = m$. By definition, an irreducible polynomial is not divisible by any other polynomial with smaller degree, bigger than 1[1].
Then, the quotient ring $Z_2[x]/(p(x))$ is a finite field of *order* 2^m, meaning that the ring contains 2^m different polynomials. This field is better known as the *Galois field* of order 2^m, or $GF(2^m)$ for short. All non-zero elements in this field can be represented as powers of some primitive element λ. In literature, the polynomial $p(x)$ is referred to as the *Field's Generator Polynomial* (FGP) or the *primitive polynomial.*
An element $a(x) \in GF(2^m)$ can be written as

$$a(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{m-1}x^{m-1},$$

where the coefficients $a_0, a_1, \ldots, a_{m-1}$ are elements in Z_2, and any non-zero element $a(x) \in GF(2^m)$ can be written as $a(x) = \lambda^s, \quad 0 \leq s < 2^m - 1$.

[1] Note that this is similar to the definition of primes in the infinite field N, the field of natural numbers.

Example:
Let's take FGP $p(x) = 1 + x^2 + x^3 + x^4 + x^8$, $p(x) \in Z_2[x]$, then
$Z_2[x]/(p(x)) = GF(2^8)$ is a Galois Field of order 2^8.

When adding two polynomials $a(x), b(x) \in GF(2^m)$ as defined like in $Z_2[x]$, the result $r(x) = a(x) \oplus_m b(x) = (a(x) + b(x)) \bmod p(x) = a(x) \oplus b(x)$ is automatically in $GF(2^m)$, however, with multiplication in $Z_2[x]$ this is not always the case when $\deg(a(x)b(x)) \geq m$. Then the (intermediate) solution $a(x)b(x) \in Z_2[x]$ must be taken modulo $p(x)$ in order to get the final result

$$r(x) = a(x) \otimes_m b(x) = (a(x)b(x)) \bmod p(x)$$

to be in the Galois Field.

Example:
Let's take the multiplication $a(x)b(x) = x^3 + x^9 \in Z_2[x]$, then the result in $GF(2^8)$, with $p(x) = 1 + x^2 + x^3 + x^4 + x^8$, $p(x) \in Z_2[x]$, equals $r(x) = x^5 + x^4 + x$, because $x^9 + x^3 = (x^8 + x^4 + x^3 + x^2 + 1)x + (x^5 + x^4 + x) = p(x)x + r(x)$.

Properties of Galois Fields:

- Let's take any $a(x) \in GF(2^m)$. In $GF(2^m)$ there exists a unique *'0'-element* with the properties $a(x) \oplus_m 0 = 0 \oplus_m a(x) = a(x) \in GF(2^m)$ and $a(x) \otimes_m 0 = 0 \otimes_m a(x) = 0 \in GF(2^m)$;
- Also, in $GF(2^m)$ there exists a unique *'1'-element* with the property $a(x) \otimes_m 1 = 1 \otimes_m a(x) = a(x) \in GF(2^m)$;
- Each element has its *inverse element* included in the field except for the '0'-element, which has none. The relation between an element $a(x) \in GF(2^m)$ and its inverse $a^{-1}(x) \in GF(2^m)$ is expressed as $a(x) \otimes_m a^{-1}(x) = 1$.

Example:
In $GF(2^8)$ the '0'-element equals the polynomial 0 and the '1'-element equals the polynomial 1;

To make things even more complicated, a Galois Field itself can be extended by means of $GF(2^m)[y]$, the ring of polynomials with the coefficients being elements in $GF(2^m)$, so

$$b(y) = b_0(x) + b_1(x)y + b_2(x)y^2 + \cdots + b_{n-1}(x)y^{n-1} = \sum_{i=0}^{n-1} b_i(x)y^i$$

where

$$b_i(x) = a_{i,0} + a_{i,1}x + a_{i,2}x^2 + \cdots + a_{i,m-1}x^{m-1} \in GF(2^m), \quad i = 0, \ldots, n-1$$

Example:

Let's take the field $GF(2^8)$ as used earlier, and an extension of $n = 204$, then we have, for polynomial $b(y) \in GF(2^8)[y]$ representing a word $\mathbf{b} = (b_0, b_1, \cdots, b_{203})$

$$b(y) = b_0(x) + b_1(x)y + b_2(x)y^2 + \cdots + b_{203}(x)y^{203} = \sum_{i=0}^{203} b_i(x)y^i$$

This element can be viewed as a sequence of 204 bytes $b_0, b_1, b_2, \cdots, b_{203} \in GF(2^8)$.

From now on, normal plus and minus signs will be used to denote "⊕", and multiplication notation for "⊗".

3.2 Reed-Solomon Codes

A *Reed-Solomon codeword* is a sequence of n characters which can be viewed as the coefficients of a *codeword polynomial* [2: p.869]:

$$C(x) = \sum_{i=0}^{n-1} c_i x^i$$

The characters $c_{n-1}, c_{n-2}, \cdots, c_1, c_0$ are elements in a finite field. When dealing with RS codes, this finite field has order 2^m, and it is called a Galois Field $GF(2^m)$, where m might be any (positive) integer.

A sequence of n characters, $\mathbf{c} = (c_0, c_1, \cdots, c_{n-1})$, is a *RS codeword* of length n if and only if its corresponding polynomial, $C(x)$, is a multiple of the *Code's Generator Polynomial* (CGP). Let this polynomial be monic with $\deg(g(x)) = r$, and be defined as

$$g(x) = \prod_{i=b}^{b+r-1}(x - \lambda^i) = \sum_{i=0}^{r} g_i x^i, \quad g_r = 1$$

with $\lambda, g_i \in GF(2^m)$. The generator polynomial must have as roots $2t$ consecutive powers [20: p.118], the 'check' positions, which are given by λ^{b+i}, $i = 0,1,\cdots,r-1$, and some non-negative constant b. Actually the roots of $g(x)$ are all roots of any RS codeword generated by this CGP.

The original approach to constructing Reed-Solomon codes can be explained as follows: Suppose that we have a packet of k information characters, $m_0, m_1, \cdots, m_{k-1}$, taken from the finite field $GF(2^m)$. With these characters we construct the *message polynomial*

$$M(x) = m_0 x^r + m_1 x^{r+1} + \cdots + m_{k-1} x^{r+k-1} = \sum_{i=0}^{k-1} m_i x^{r+i}$$

Now, we divide $M(x)$ by $g(x)$, and call the remainder of this division the *parity polynomial* $P(x)$. Typically, $\deg(P(x)) < \deg(g(x))$, so we can write

$$P(x) = p_0 + p_1 x + \cdots + p_{r-1} x^{r-1} = \sum_{i=0}^{r-1} p_i x^i$$

It can be seen easily that subtracting $P(x)$ from $M(x)$ doesn't affect the information characters in the latter term at all. As subtraction of finite field elements in $GF(2^m)$ is the same as addition, we conclude that $\big(M(x)+P(x)\big)$ is a multiple of $g(x)$ and thus have constructed a codeword of length *r+k* by computing the parity polynomial and chaining it to the corresponding message polynomial! The reason for this curious procedure is that in this way, RS codes are *systematic codes*, i.e. that the message bits are contained unchanged in the codeword.

A *t-error correcting (n,k) RS code*, also denoted as RS(*n,k,t*) has the following properties:

- **Length**: *n*, the total number of characters of the RS code, where $n = 2^m$ for *acyclic RS codes*, $n = 2^m - 1$ for *cyclic RS codes* and $n < 2^m - 1$ for *shortened RS codes* in $GF(2^m)$;

A code is said to be *cyclic* if, for any codeword $\mathbf{c} = (c_0, c_1, \cdots, c_{n-1})$ in this code, the cyclically shifted word $\mathbf{c}' = (c_1, c_2, \cdots, c_{n-1}, c_0)$ is also a codeword.

- **Dimension**: *k*, the number of information characters of the RS code;

Since the *k* information characters are selected from $GF(2^m)$, each one can take on 2^m different values, so in this RS code there are $(2^m)^k = 2^{mk}$ codewords.

- **Redundancy**: $r = n-k$;
- **Rate**: k/n;
- **Distance**: $d = r+1$;
- **Error correction capability**: $t = \left\lfloor \frac{d}{2} \right\rfloor = \left\lfloor \frac{n-k+1}{2} \right\rfloor$, the maximum number of correctable errors;

In 1964 Singleton showed that this was the best possible error correction capability for any code of the same length and dimension [39: p.4], and therefore belong to the class of *Maximum Distance Separable* (MDS) codes.

3.3 Constructing RS Codes

As stated earlier,

$$\begin{aligned} C(x) &= M(x) + P(x) \\ &= g(x) \times Q(x) + \sum_{i=0}^{r-1} c_i x^i + P(x) \\ &= g(x) \times Q(x), \end{aligned}$$

holds for any codeword, guaranteeing that $M(x) + P(x)$ is a multiple of $g(x)$ and an irrelevant $Q(x)$. Seen in this light, encoding is merely a matter of computing the parity polynomial $P(x)$.

As described in [2: p.869], [17: p.76], the most straightforward method of obtaining the remainder of a division by a monic polynomial $g(x)$ is by a shift register wired according to $g(x)$ as shown in figure 3-1:

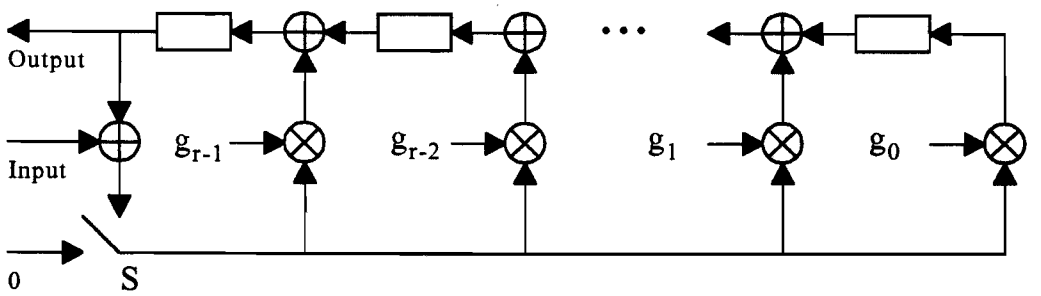


Figure 3-1: Parity generator with shift register wired according to $g(x)$.

All lines shown in this figure represent m-bit wide buses. Each ⊕ represents an exclusive-or of two m-bit characters, each ⊗ represents a multiplication in the Galois field, $GF(2^m)$, and the rectangles are m-bit wide shift registers. Initially, switch S connects 0, until the first information character appears at the Input-line. Then, during k information characters, S connects (Input ⊕ Output) for feedback purposes. After this, the r parity characters can be read at the Output-line, as the Input remains 0 and S connects 0, in the same time initializing the shift registers at 0 before receiving a next message.

3.4 Reception of RS Codes

A decoder that requires its input to be from the same alphabet as the channel input is called a *Hard-Decision Decoder* (HDD).
In reality, however, channel noise is almost always a continuous phenomenon. What is transmitted may be selected from a discrete set, but what is received comes from a continuum of values. This viewpoint leads to the *Soft-Decision Decoder* (SDD), which accepts a sequence of real samples of the noisy channel output and estimates the sequence of channel input symbols that was transmitted. It is now well known that soft-decision decoding techniques can provide approximately 2 dB more *coding gain* for the white Gaussian channel [7: p.108].
Coding gain [27] is defined as the saving in energy per source bit of information for a coded system relative to an uncoded system, both operating at the same delivered *Bit Error Rate* (BER).

In practical systems, there are two types of *errata* [1: p.229]: *erasures*, whose locations are known, but whose values are unknown, and *errors* whose locations and values are both unknown.

RS codes can use 1 bit of soft-decision per symbol [17: p.62] to mark erasures. This information, called an *erasure indicator*, thus is an indication that the received symbol is incorrect. For now, I will only treat the hard-decision RS decoder, because the decoding algorithms are basically the same, except that the soft-decision decoding is very time consuming and not required by the DTT requirements.

A received word is a polynomial $V(x)$ that consists of the original codeword polynomial $C(x)$ plus (i.e. XOR) an *error polynomial* $E(x)$:

$$E(x) = e_0 + e_1 x + \cdots + e_{n-1} x^{n-1} = \sum_{i=0}^{n-1} e_i x^i$$

$$V(x) = C(x) + E(x) = \sum_{i=0}^{n-1} c_i x^i + \sum_{i=0}^{n-1} e_i x^i = \sum_{i=0}^{n-1} (c_i \oplus e_i) x^i = \sum_{i=0}^{n-1} v_i x^i$$

The RS code can correct up to t errors en ν erasures, as long as $2t + \nu \leq n - k$.
If the number of errata in one word exceed the error-correcting capability, the RS decoder has two possible outputs [15: p.263]. Either a *decoder failure* occurs, meaning that the RS decoder recognizes that the number of errors exceeds the code's capabilities and fails to find any codeword at all, or a *decoder error* occurs, meaning that the RS decoder outputs a codeword different from the transmitted codeword, without recognizing its mistake.

3.5 The Search for Error Locations

Decoding of RS codes can be done in several ways, using different algorithms. There are three main classes of decoding algorithms:

1. Syndrome-based decoding, where one can distinguish algebraic decoding algorithms (sometimes called hybrid decoding algorithms) and transform decoding algorithms;
2. Remainder-based decoding;
3. Time-domain decoding.

According to [40], Syndrome-based algebraic decoding has the most practical significance, so I will concentrate on this decoding option.

The received polynomial at the input of the decoder $V(x)$ can be evaluated at the $r = 2t$ roots of the FGP $p(x)$, which are $1, \lambda, \lambda^2, \cdots, \lambda^{r-1}$. Since the codeword polynomial $C(x)$ is a multiple of the CGP $g(x)$, and $g(\lambda^j) = 0$ for $j = 0,1,\cdots,r-1$, we have

$$\begin{aligned} V(\lambda^j) &= C(\lambda^j) + E(\lambda^j) \\ &= E(\lambda^j) = \sum_{i=0}^{n-1} e_i \lambda^{ji}, \quad j = 0,1,\cdots,r-1 \end{aligned}$$

This final set of r equations involves only components of the error pattern, not those of the codeword. They can be used to compute *syndromes* S_j, $j = 0,1,\cdots,r-1$, which are defined as

$$S_j \triangleq V(\lambda^j) = \sum_{i=0}^{n-1} v_i \lambda^{ij}, \quad j = 0,1,\cdots,r-1$$

Following evaluation of the syndromes, the error pattern e_i, $i = 0,1,\cdots,n-1$, can be determined. Suppose that ν errors, $0 \leq \nu \leq t$, occur in unknown locations $i_1, i_2, \cdots, i_\nu$. Then the error polynomial can be written as

$$E(x) = e_{i_1} x^{i_1} + e_{i_2} x^{i_2} + \cdots + e_{i_\nu} x^{i_\nu}$$

where e_{i_l} is the value of the lth error. We do not know the error locations $i_1, i_2, \cdots, i_\nu$, nor do we know the error values $e_{i_1}, e_{i_2}, \cdots, e_{i_\nu}$. In fact, we do not even know the number of errors ν. All these values must be computed in order to correct the errors.
Let us evaluate the received polynomial at $x = 1, \lambda$ to obtain the syndromes S_0, S_1:

$$\begin{aligned} S_0 &= V(1) \\ &= E(1) = e_{i_1} + e_{i_2} + \cdots + e_{i_\nu} \end{aligned}$$

$$\begin{aligned} S_1 &= V(\lambda) \\ &= E(\lambda) = e_{i_1}\lambda^{i_1} + e_{i_2}\lambda^{i_2} + \cdots + e_{i_\nu}\lambda^{i_\nu} \end{aligned}$$

Changing the notation, we define the error values $Y_l = e_{i_l}$, for $l = 1,2,\cdots,\nu$, and the error field locations $X_l = \lambda^{i_l}$, for $l = 1,2,\cdots,\nu$, where i_l is the actual location of the lth error and X_l is the field element associated with this location. Using this notation, the syndromes S_0, S_1 are given by

$$\begin{aligned} S_0 &= Y_1 + Y_2 + \cdots + Y_\nu \\ S_1 &= Y_1 X_1 + Y_2 X_2 + \cdots + Y_\nu X_\nu \end{aligned}$$

Similarly[2], we can evaluate the received polynomial at each of the powers of λ, which are roots of $g(x)$. We then have the following set of r equations in ν unknown error field locations $X_1, X_2, \cdots, X_\nu$ and ν unknown error values $Y_1, Y_2, \cdots, Y_\nu$:

$$\begin{aligned} S_j &= Y_1 X_1^{\,j} + Y_2 X_2^{\,j} + \cdots + Y_\nu X_\nu^{\,j}, \quad j = 0,1,\cdots,2t-1 \\ &= \sum_{l=1}^{\nu} Y_l X_l^{\,j}, \quad j = 0,1,\cdots,2t-1 \end{aligned}$$

This set of equations must have at least one solution because of the way in which the syndromes are defined. It can be shown that the solution is unique for $0 \le \nu \le t$.

The direct solution of a system of nonlinear equations is difficult except for small values of t. A better approach requires intermediate steps. For this purpose, the *error locator polynomial* $\Lambda(x)$ is introduced as follows:

$$\Lambda(x) = 1 + \Lambda_1 x + \cdots + \Lambda_\nu x^\nu$$

This polynomial is defined to have as roots the inverse error field location numbers X_l^{-1}, $l = 1,2,\cdots,\nu$. That is,

$$\Lambda(x) = \prod_{l=1}^{\nu} (1 - xX_l), \quad X_l = \lambda^{i_l}$$

[2] Note that we would only need S_0, S_1 if at most one error (X_1, Y_1) had occurred, resulting in a straightforward (linear!) correction-mechanism [19].

Let's evaluate this polynomial by equation:

$$1+\Lambda_1 x+\cdots+\Lambda_\nu x^\nu=\prod_{l=1}^{\nu}(1-xX_l),\quad X_l=\lambda^{i_l},$$

substitute the right side's roots, $x=X_l^{-1},\quad l=1,\cdots,\nu$, to get the following set of ν equations:

$$1+\Lambda_1 X_l^{-1}+\cdots+\Lambda_\nu X_l^{-1}=0,\quad l=1,\cdots,\nu,$$

multiply each equation with $Y_l X_l^{j+\nu}$ to obtain

$$Y_l\left(X_l^{j+\nu}+\Lambda_1 X_l^{j+\nu-1}+\cdots+\Lambda_\nu X_l^{j}\right)=0,\quad l=1,\cdots,\nu,$$

where j can be any integer. We can summate these equations over l, to get one equation

$$\sum_{l=1}^{\nu}Y_l X_l^{j+\nu}+\Lambda_1\sum_{l=1}^{\nu}Y_l X_l^{j+\nu-1}+\cdots+\Lambda_\nu\sum_{l=1}^{\nu}Y_l X_l^{j}=0$$

Now we have to find appropriate values for j to get convenient equations. Remember the definition of the syndromes earlier on, by which we can get restrictions on j, and rewrite:

$$S_{j+\nu}+\Lambda_1 S_{j+\nu-1}+\cdots+\Lambda_\nu S_j=0$$
$$\Rightarrow \Lambda_1 S_{j+\nu-1}+\cdots+\Lambda_\nu S_j=-S_{j+\nu},\quad j=0,1,\cdots,\nu-1$$

which can be written in the following matrix equation:

$$\begin{bmatrix} S_0 & S_1 & \cdots & S_{\nu-1} \\ S_1 & S_2 & \cdots & S_\nu \\ \vdots & \vdots & \ddots & \vdots \\ S_{\nu-1} & S_\nu & \cdots & S_{2\nu-2} \end{bmatrix}\begin{bmatrix} \Lambda_\nu \\ \Lambda_{\nu-1} \\ \vdots \\ \Lambda_1 \end{bmatrix}=\begin{bmatrix} -S_\nu \\ -S_{\nu+1} \\ \vdots \\ -S_{2\nu-1} \end{bmatrix}$$

The polynomial $\Lambda(x)$ of smallest degree will have degree ν, and there is only one $\Lambda(x)$ of degree ν because the ν×ν matrix above is invertible.
For moderate ν, the obvious method of solving by using matrix inversion is not unreasonable. However, the number of computations necessary to invert a ν×ν matrix is proportional to ν^3 [4: p.176], which for larger ν is not very efficient.

Instead, Berlekamp-Massey [17: p.85] write the matrix equation, with dummy change $j=j-\nu$ as

$$S_j=-\sum_{i=1}^{\nu}\Lambda_i S_{j-i},\quad j=\nu,\nu+1,\cdots,2\nu-1$$

For fixed Λ, this is the equation of an autoregressive filter, which can be implemented as a linear-feedback shift register with taps given by Λ, like in figure 3-2 (initialize with j=ν).

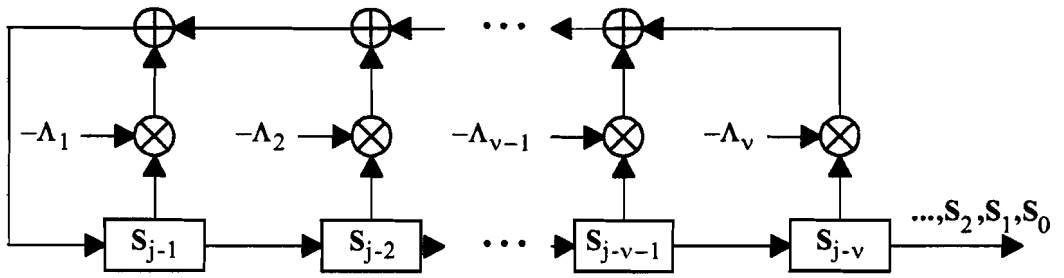


Figure 3-2: Error locator polynomial as a shift register circuit.

Any procedure for designing this autoregressive filter is also a method for solving the matrix equation for the Λ-vector.

Once we have found $\Lambda(x)$, we only have to find its roots to know the inverse field locations of the errors X_l^{-1}, $l = 1,2,\cdots,\nu$ (and thus the error locations i_l, $l = 1,2,\cdots,\nu$). This can be done by just trying all the possible values for x, being all the possible locations at which an error could have occurred, so whenever

$$\Lambda(x) = 1 + \Lambda_1 x + \cdots + \Lambda_\nu x^\nu = 0,$$

we have determined an error with inverse field location λ^{-i_l}. The common, systematic way in which this can be done is known as the *Chien Search Method.*

3.6 The Search for the Error Values

Finding the error values is a process that, in many decoding algorithms, can be combined with the search process for the error locations.
Thus, for evaluation of the error locator polynomial $\Lambda(x)$ we define the *syndrome polynomial*

$$S(x) = \sum_{i=0}^{2t-1} S_i x^i$$

and we define the *error evaluator polynomial* $\Omega(x)$ by means of *Berlekamp's Key Equation*:

$$\Omega(x) = S(x)\Lambda(x) \bmod x^{2t}$$

There are several polynomials $\Lambda(x)$ and $\Omega(x)$ that satisfy this equation, but it can be shown that only the solution with $\Lambda(x)$ of lowest degree is the right one [4], which can be seen as the minimal number of errors able to mess up a codeword in this particular way.
We can find $\Lambda(x)$ and $\Omega(x)$ with help of the (extended) *Euclidean Algorithm* for polynomials or the *Berlekamp-Massey Algorithm*, which both will be treated later on.

The error evaluator polynomial is related to the error field locations and error values as

$$\Omega(X_l^{-1}) = Y_l \prod_{i \neq l} (1 - X_i X_l^{-1})$$

and the error values are given by the *Forney algorithm*:

$$Y_l = -X_l \frac{\Omega(X_l^{-1})}{\dot{\Lambda}(X_l^{-1})}, \quad l = 1,2,\cdots,\nu$$

where $\dot{\Lambda}(x)$ is the *formal derivative* of $\Lambda(x)$, defined as

$$\dot{\Lambda}(x) \hat{=} \sum_{j=0}^{(\nu-1) \text{ div } 2} \Lambda_{1+2j} x^{2j}$$

After evaluation of the error values, the received vector can be corrected by subtracting the error values from this received vector. In figure 3-3 below this whole procedure is shown schematically.

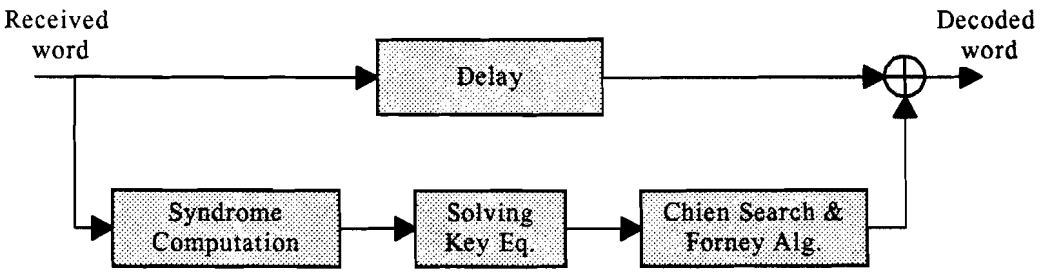


Figure 3-3: Block diagram for Syndrome-based Algebraic Decoding.

3.7 The Berlekamp-Massey Algorithm

The *Berlekamp-Massey algorithm*, is an inductive design procedure to find the smallest linear feedback shift register, as defined in chapter 3.5.
At the i^{th} iteration, the *shift register length* is L_i, with taps according to $\Lambda^{(i)}(x)$, which will produce the sequence S_0, ..., S_{i-1}. Although the i^{th} shift register may not be unique, all alternatives have the same length L_i.
At iteration i, the *discrepancy* Δ_i is computed as being the desired output S_i minus (= XOR) the output by the intermediate shift register:

$$\Delta_i = S_i - \sum_{j=1}^{i} \Lambda_j^{(i)} S_{i-j} = \sum_{j=0}^{i} \Lambda_j^{(i)} S_{i-j},$$

Adaption of the shift register length L_i plays an important role in updating $\Lambda^{(i)}(x)$. Because the proof (see [4: p.180]) is rather lengthy, the other recursive equations are given below.

$$\delta = \begin{cases} 1, & (\Delta_i \neq 0) \wedge (2L_i \leq i) \\ 0, & otherwise \end{cases},$$

$$L_{i+1} = \delta(i - L_i) + (1-\delta)L_i,$$

$$\begin{bmatrix} \Lambda^{(i+1)}(x) \\ B^{(i+1)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_i \\ \delta x/\Delta_i & (1-\delta)x \end{bmatrix} \begin{bmatrix} \Lambda^{(i)}(x) \\ B^{(i)}(x) \end{bmatrix},$$

$$\begin{bmatrix} \Omega^{(i+1)}(x) \\ A^{(i+1)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_i \\ \delta x/\Delta_i & (1-\delta)x \end{bmatrix} \begin{bmatrix} \Omega^{(i)}(x) \\ A^{(i)}(x) \end{bmatrix}$$

for $i = 0,1,\cdots,2t-1$.
The initial conditions are $\Lambda^{(0)}(x) = 1$, $B^{(0)}(x) = x$, $L_0 = 0, \Omega^{(0)}(x) = 0$, $A^{(0)}(x) = 1$.

With these recursive equations, we can obtain the error locator $\Lambda^{(2t)}(x)$ and the error evaluator $\Omega^{(2t)}(x)$ polynomials, and thus solve Berlekamp's Key Equation for t or less errors. Notice that the matrices update requires at most $4t$ multiplications and divisions per iteration, and that the calculation of Δ_i requires no more than t multiplications per iteration averaged. There are $2t$ iterations, and thus at most $10t^2$ multiplications and divisions per received word.
Note: I had to slightly adjust the equations and initial conditions in [17: p.85], because originally the syndrome polynomial was defined as

$$S(x) = \sum_{i=1}^{2t} S_i x^i,$$

causing a somewhat different key equation. With the adjustments made, we still obtain the proper error locator and evaluator polynomials.

For convenience of the C-model, I rewrote the above matrix equations as:

$$\begin{bmatrix} \Lambda^{(i+1)}(x) \\ \Omega^{(i+1)}(x) \end{bmatrix} = \begin{bmatrix} \Lambda^{(i)}(x) \\ \Omega^{(i)}(x) \end{bmatrix} + \Delta_i \begin{bmatrix} B^{(i)}(x) \\ A^{(i)}(x) \end{bmatrix},$$

$$\begin{bmatrix} B^{(i+1)}(x) \\ A^{(i+1)}(x) \end{bmatrix} = x \left\{ \begin{bmatrix} B^{(i)}(x) \\ A^{(i)}(x) \end{bmatrix} + \frac{\delta}{\Delta_i} \begin{bmatrix} \Lambda^{(i+1)}(x) \\ \Omega^{(i+1)}(x) \end{bmatrix} \right\},$$

maintaining the given equations for $\Delta_i, \delta, L_{i+1}$, and the same initial conditions.

3.8 The Euclidean Algorithm

The error locator as well as the error evaluator polynomials can also be obtained using the *Euclidean algorithm*, a method for finding the *Greatest Common Divisor* (GCD) of two polynomials. The algorithm used here is a slightly expanded version, which will produce the polynomials a(x) and b(x) satisfying

$$GCD[R(x), T(x)] = a(x)R(x) + b(x)T(x)$$

where the algorithm is repeated in a convenient matrix form using the notation

$$R(x) = \left\lfloor \frac{R(x)}{T(x)} \right\rfloor T(x) + Rem(x)$$

To represent the division algorithm.
The Euclidean algorithm can be explained using the following set of recursive equations [17: p.86]:

$$Q^{(i)}(x) = R^{(i)}(x) \operatorname{div} T^{(i)}(x),$$

$$\mathbf{A}^{(i+1)}(x) = \begin{bmatrix} 0 & 1 \\ 1 & Q^{(i)}(x) \end{bmatrix} \mathbf{A}^{(i)}(x),$$

$$\begin{bmatrix} R^{(i+1)}(x) \\ T^{(i+1)}(x) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & Q^{(i)}(x) \end{bmatrix} \begin{bmatrix} R^{(i)}(x) \\ T^{(i)}(x) \end{bmatrix},$$

The initial conditions are $R^{(0)}(x) = x^{2t}$, $T^{(0)}(x) = \sum_{j=0}^{2t-1} S_j x^j$, $\mathbf{A}^{(0)}(x) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.
The algorithm terminates when $\deg(T^{(i)}(x)) < t$. Suppose this occurs when $i = i'$. Then

$$\Omega(x) = \Delta^{-1} T^{(i')}(x)$$

$$\Lambda(x) = \Delta^{-1} A_{22}^{(i')}(x)$$

where $\Delta = A_{22}^{(i')}(0)$. The solutions of these equations uniquely satisfy Berlekamp's Key Equation (for the proof I refer to [4]), satisfying

$$\left(\deg(\Omega(x)) < t\right) \wedge \left(\deg(\Lambda(x)) < t+1\right) \wedge \left(\Lambda_0 = 1\right)$$

Again, I made some minor modifications to the above equations to avoid unnecessary variable storage in the C-program. Therefore I have chosen to introduce a new polynomial $Rem^{(i)}(x)$, to avoid storage of two other ones: $R^{(i)}(x)$, $T^{(i)}(x)$ to be exact. I define

$$Rem^{(i)}(x) = R^{(i)}(x) \bmod T^{(i)}(x),$$

which can be obtained easily from the same computation used to compute $Q^{(i)}(x)$, and with this we can substitute another computation with a simpler

$$\begin{bmatrix} R^{(i+1)}(x) \\ T^{(i+1)}(x) \end{bmatrix} = \begin{bmatrix} T^{(i)}(x) \\ Rem^{(i)}(x) \end{bmatrix}.$$

3.9 Some Decoder-part Architecture Examples

Basic computations are addition, multiplication, and inversion in the Galois Field. Addition is simple: this can be done by a bitwise XOR of 2 operands (in $GF(2^m)$ these are m-bit wide words) in a serial or parallel way. Multiplication in the Galois Field is harder, because it is a modulo operation. The multiplication may be implemented directly (combinatorial), but might require as many as $m^3 - m$ two-input adders [1]. Another, rather direct solution is given by [1] and will be explained using FGP $p(x) = 1 + x^2 + x^3 + x^4 + x^8, p(x) \in Z_2[x]$. We want to compute $r(x) = a(x) \otimes_8 b(x) = a(x)b(x) \bmod p(x)$, and do so by storing the multiplicand $a(x)$ in a feedback shift register which is wired to replace $a(x)$ by $x \otimes_8 a(x)$. The multiplier $b(x)$ is stored in a register which is cyclically shifted to the right. The recursively computed product is $r(x)$. The multiplier will need as much as $m = 8$ iterative steps to compute the multiplication as can be seen in figure 3-4 below.
The recursive equations are:

$$r^{(i)}(x) = r^{(i-1)}(x) \oplus_8 \left(b_{i-1} \otimes a^{(i-1)}(x)\right)$$

$$a^{(i)}(x) = \left(x \otimes a^{(i-1)}(x)\right) \bmod p(x) = x \otimes_8 a^{(i-1)}(x)$$

which are initialized by $r^{(0)} = 0, a^{(0)}(x) = a(x)$, and after m=8 iterations we find $r(x)$ to be

$$r^{(8)}(x) = \left(\left(b_0 \oplus x\left(b_1 \oplus \cdots x\left(b_6 \oplus x b_7\right)\cdots\right)\right) \otimes a^{(0)}(x)\right) \bmod p(x) = b(x) \otimes_8 a(x)$$

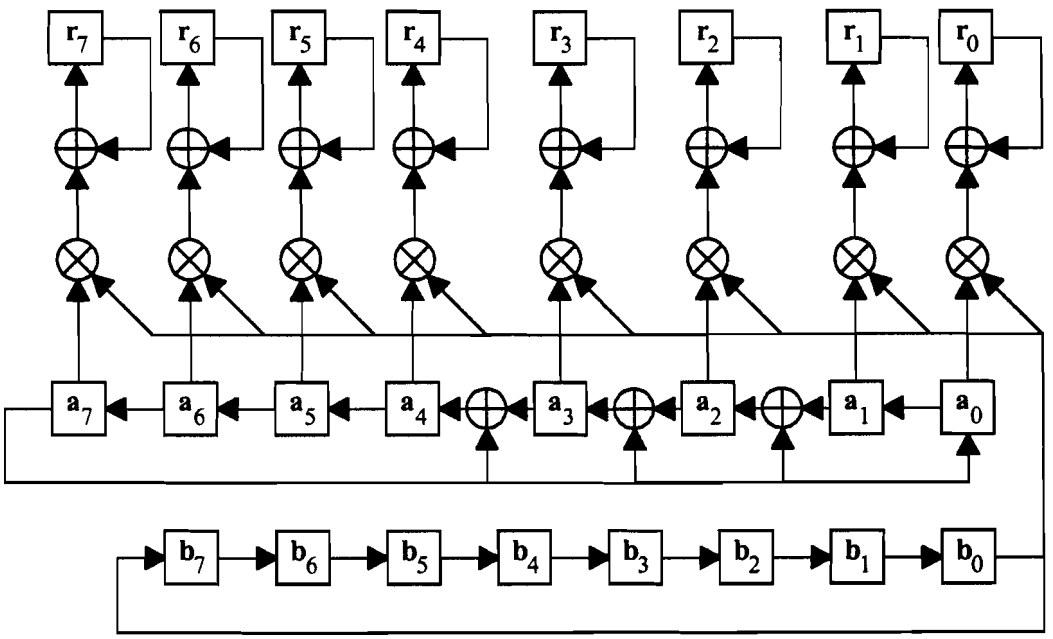


Figure 3-4: Berlekamp's Galois Field Multiplier.

In the next figure 3-5, I have designed a multiplier somewhat similar to Berlekamp's. Instead of storing $a(x)$ in a feedback shift register, this is done with the intermediate product $r(x)$. Now we do not have to rotate $a(x)$, but we have to trade three 3-input adders (XORs) for six 2-input adders (XORs). Note that $b(x)$ is shifted cyclically left.
There is one recursive equation:

$$\begin{aligned} r^{(i)}(x) &= \left(x \otimes r^{(i-1)}(x)\right) \bmod p(x) \oplus_8 \left(b_{8-i} \otimes a(x)\right) \\ &= \left(x \otimes r^{(i-1)}(x)\right) \bmod p(x) \oplus_8 \left(b_{8-i} \otimes a(x)\right) \bmod p(x) \\ &= \left(\left(x \otimes r^{(i-1)}(x)\right) \oplus_8 \left(b_{8-i} \otimes a(x)\right)\right) \bmod p(x) \end{aligned}$$

which is initialized by $r^{(0)} = 0$, and after m=8 iterations we find $r(x)$ to be

$$r^{(8)}(x) = \left(\left(b_0 \oplus x\left(b_1 \oplus \cdots x\left(b_6 \oplus x b_7\right)\cdots\right)\right) \otimes a(x)\right) \bmod p(x) = b(x) \otimes_8 a(x)$$

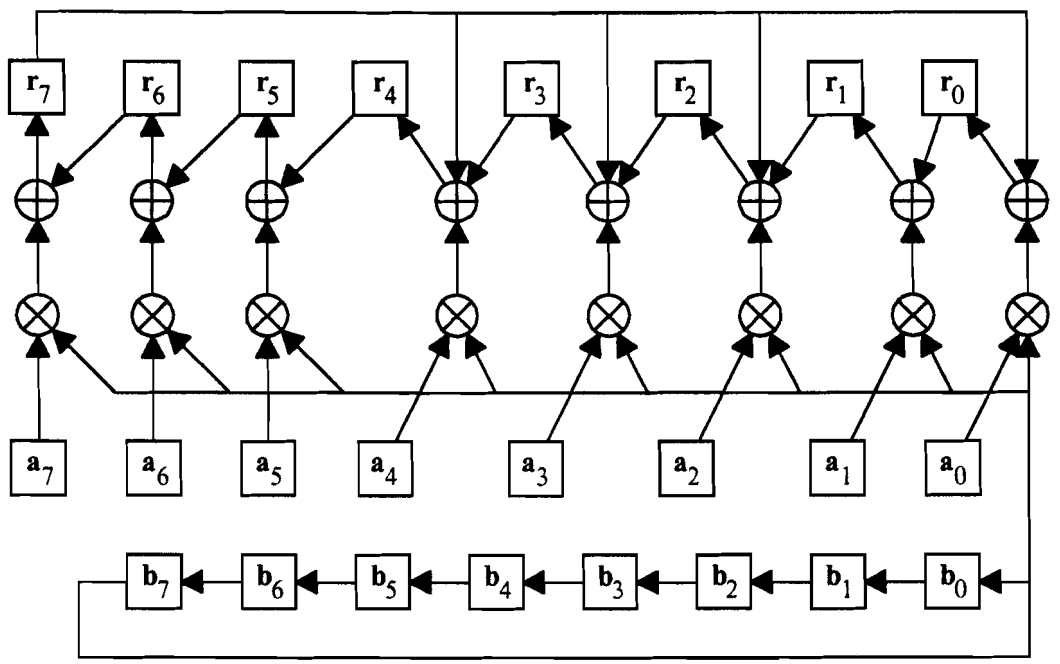


Figure 3-5: Alternative Galois Field Multiplier.

There are several other methods to perform multiplication in the Galois Field, such as with log/antilog tables (not practical for circuitry implementations) [1], the dual basis multiplication algorithm [2], or the Massey-Omura normal basis multiplication algorithm [17]. The last two algorithms require that the operands are given relative to another basis, which of course complicates things. Secondly, these are modular or/and universal designs, which are, practically by definition, not area-efficient. Since the RS code, and in particular the FGP, is not bound to be changed to another FGP, we can be sure that a direct, uni-functional, and thus area-efficient multiplier implementation is the right solution. In this stage however, we have not yet made a choice for a multiplier.

The last note on multipliers is that, as can be seen further on in this chapter, for example the Syndromes computation and Chien Search can be designed using only constant multipliers, i.e. one operand always has the same constant value. This type of multiplication can be implemented most efficient, but also most static, with combinatorial logic circuits.

The last operator in the Galois Field is division, which is one of the most complicated operations in $GF(2^m)$. It is most commonly implemented [17] by multiplication with the inverted operand, where this (non-zero) inverted operand is obtained from a ROM-table. For $GF(2^m)$, the size of the ROM is on the order of $m \cdot 2^m$ bits.

The first step for algebraic decoding RS codes is to compute the syndromes. These can be conveniently written as:

$$S_j = \left[\cdots\left(v_{n-1}\lambda^j + v_{n-2}\right)\lambda^j + \cdots + v_1\right]\lambda^j + v_0, \quad j = 0,1,\cdots,r-1$$

This can be realized in the following structure of cells, where we initialize $S_{j,0}$ as 0, and after receiving n symbols ($v_{n-1},\ldots,v_0$), the final syndromes are $S_{j,n}$.

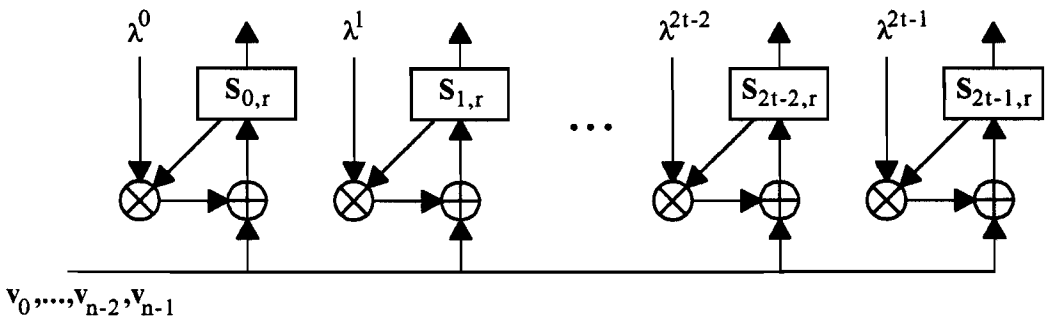


Figure 3-6: Architecture for computation of the Syndromes

The number of multiplications, when already possessing $\lambda^0,\ldots,\lambda^{2t-1}$, is $2tn$ per received word. When we omit the multiplication by $\lambda^0 = 1$, there remain $n(2t-1)$ multiplications.

Now, we must try to solve Berlekamp's Key Equation by either the Berlekamp-Massey algorithm or the Euclidean Algorithm. Here, the designer has a wide variety of options, because the number of different implementations is merely the same as the number of articles on Reed-Solomon decoding.

The next step is the Chien Search Method, where we find an error location if $\Lambda(\lambda^i) = 0$ for any $i = 0,1,\ldots,n-1$. This can be written as

$$\begin{aligned}
\Lambda(\lambda^i) &= \sum_{j=0}^{2t}\Lambda_j\lambda^{ij} = \sum_{j=0}^{2t}\Lambda_j\lambda^{(j-1)i}\lambda^i \\
&= \lambda^i\sum_{j=0}^{2t}\Lambda_j\lambda^{(j-1)i} \\
&= \lambda^i\sum_{j=\text{odd}}\Lambda_j\lambda^{(j-1)i} + \lambda^i\sum_{j=\text{even}}\Lambda_j\lambda^{(j-1)i} \\
&= \lambda^i\dot{\Lambda}(\lambda^i) + \lambda^i\sum_{j=\text{even}}\Lambda_j\lambda^{(j-1)i}
\end{aligned}$$

This shows that the circuit can be designed to compute the derived error values for all roots, needed in the Forney algorithm. Further, we only need constant multipliers and a register for each term.

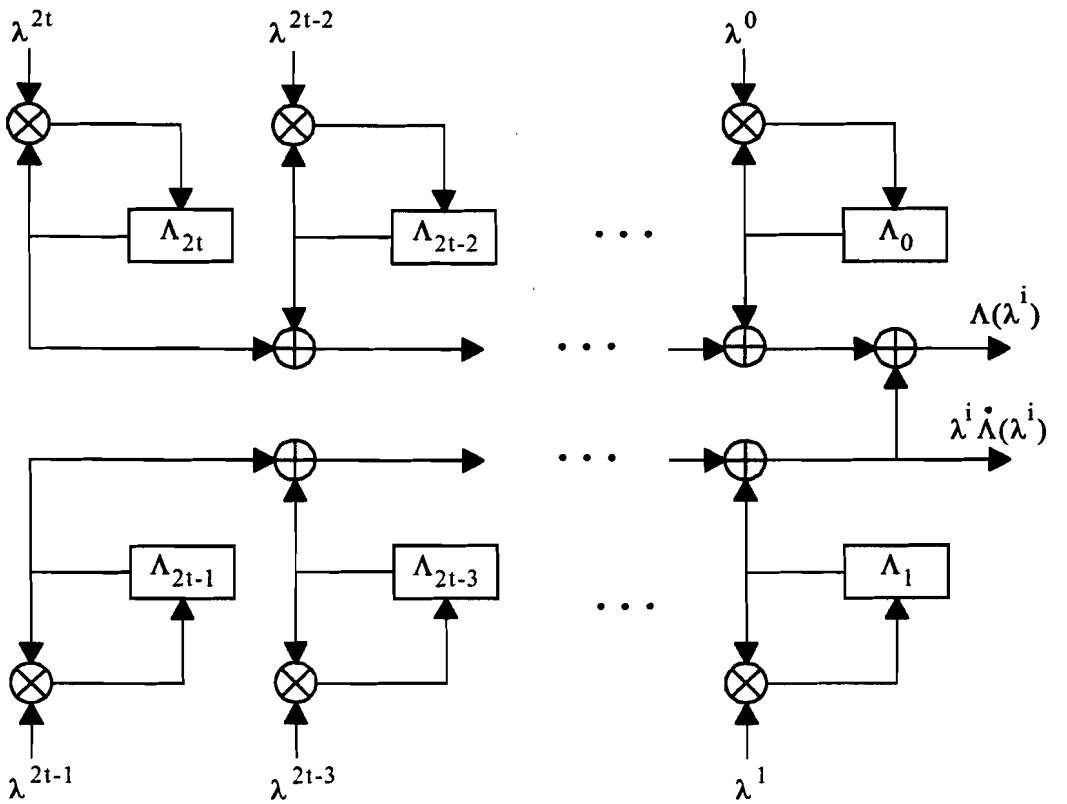


Figure 3-7: A structure for the Chien Search.

If, after $i = 0,1,\ldots,n-1$ iterations, this structure delivers $\Lambda(\lambda^i) = 0$, an error has been detected with error field location λ^{-i}, and subsequently the error location is (n-i) mod n (take length n of the non-shortened RS code). To check all n positions, we'll need $2t$ multiplications per iteration, so in total $2nt$ multiplications per received word.

The same circuit structure can be used to compute $\Omega(\lambda^{-l})$ as well, so we only need an additional division operator and a multiplier to obtain the error values. When we only compute values for the error locations, a maximum of $2t^2 + t$ multiplications and t divisions per received word are needed.

If we take a division to be a multiplication by an inverse, we can add up the multiplications and divisions which counts up to

$n(2t-1) + 10t^2 + 2nt + 2t^2 + t = 12t^2 + 4nt - n + t$.

This gives a good idea of the complexity of decoding a received word, using the Berlekamp-Massey algorithm.

For our shortened RS code, we can fill in $n = 204$, $t = 8$, resulting in 7100 multiplications per received word at most. RS-packets arrive at a maximum rate of 21.1 kHz (for 64-QAM with guard-interval = 1/32. See [11]), which means that our RS decoder must be able to perform 21.1k × 7100 = 149.5 million Galois Field multiplications per second.

3.10 Performance of RS Codes

The Reed-Solomon decoder is an *incomplete decoder*, meaning that it only decodes those received words lying in a decoding sphere about a codeword [4: p.10].

As stated in chapter 3.4, some error patterns lead to a *decoding error*, meaning that the received word lies within another codewords' sphere than the decoding sphere of the transmitted codeword; the probability for a decoding error is P_E.

Some error patterns lead to a *decoding failure*, meaning that the received word doesn't lie within any codewords' sphere and is declared by the decoder to be unrecognizable. Such error patterns are called *incorrigible error patterns*; the probability for a decoding failure is P_F.
Finally we have the error patterns that lead to a *decoding success*, meaning that the received word lies within the decoding sphere of the transmitted codeword: the probability for a decoding success is $1-P_E-P_F$.

Because any RS(n,k,t) code is an MDS code, meaning that the minimum distance $d = 2t+1$, it allows the correction of all error patterns of at most t symbol errors in a received word of length n [symbols], where $2t = n - k$.
In the next figure these parameters are shown in relation to codewords and their decoding spheres in the space of n-tuples. In fact, there are $2^{m \cdot n}$ different n-tuples, and $2^{m \cdot k}$ different codewords.

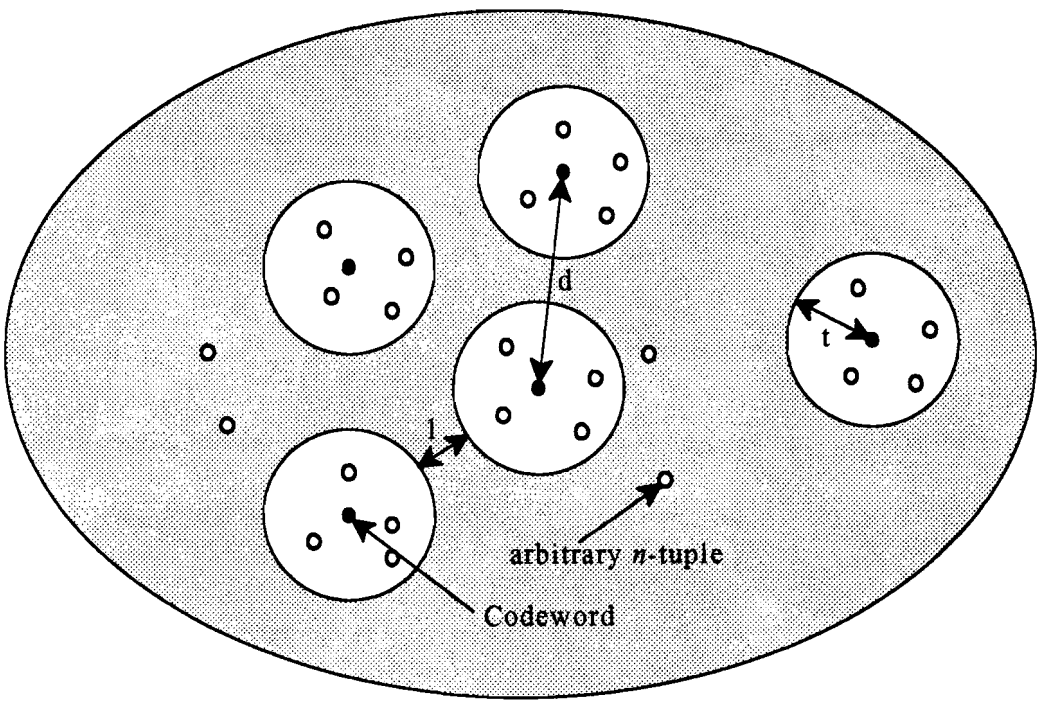


Figure 3-8: Codewords and their decoding spheres.

Let the RS(n,k,t) code be defined over $GF(2^m)$, so each symbol consists of m bits. If we assume the *bit error rate* BER to be independent and spread randomly then we can define the *symbol error rate* as P_S, the probability for a symbol to have e_b erroneous bits, where $1 \le e_b \le m$:

$$\begin{aligned} P_S &= P(1 \le e_b \le m) \\ &= 1 - P(e_b = 0) \\ &= 1 - \binom{m}{0}(1-\mathrm{BER})^m(\mathrm{BER})^0 \\ &= 1 - (1-\mathrm{BER})^m \end{aligned}$$

If we define the *message error rate* as the probability for no decoding success, $P_{NS} = P_E+P_F$, then the message error rate is given by the probability for more than t erroneous symbols per received word

$$\begin{aligned} P_{NS} &= P(e_s > t) \\ &= 1 - P(0 \le e_s \le t) \\ &= 1 - \sum_{i=0}^{t} \binom{n}{i} (P_S)^i (1 - P_S)^{n-i} \end{aligned}$$

If we consider the RS(n=204, k=188, t=8) code over $GF(2^8)$, and take BER=$2 \cdot 10^{-4}$, then P_S =$1.6 \cdot 10^{-3}$. In that case P_{NS} = $7.1 \cdot 10^{-11}$. In **Appendix A**, the performance results are shown for a range of different input bit-error rates.

4. Convolutional Coding & Viterbi Decoding

Convolutional codes are extremely practical codes and are widely used. E.g. they have been adopted by both NASA and ESA for ensuring that communications during space missions are reliable. They are widely used in conjunction with Reed-Solomon codes: each message is first encoded with a Reed-Solomon code as the 'outer code', and the resulting codeword is then encoded with a convolutional code as the 'inner code'. So is the case in ETSI's Standard for *Digital Terrestrial Television* (DTT), where the transmissions are allowed to take on different levels of error correction for a given service or data rate, using different (punctured) rates.
The Viterbi decoding algorithm is a *complete decoding algorithm* for convolutional codes, meaning that *every* received word is assigned to a nearby codeword. An incomplete decoding algorithm, on the other hand, assigns every received word to a codeword within distance t, if there is one, and otherwise refuses to decode [4: p.52].
Because a Viterbi decoder is complete, the probability of decoding failure is zero, therefore is preferred above an incomplete decoder as the inner decoder in a concatenated system.

4.1 Encoding using Shift Registers

One reason cyclic codes are so useful is that polynomial encoding and decoding can be implemented easily and efficiently by hardware devices known as *shift registers*. Briefly, these devices consist of $m+1$, $m \geq 0$ concatenated single-bit registers (or delay elements) and a "clock" which controls the movement or shifting of the data contained in the registers. In every timeslot, an input bit is received and the new contents of the registers are combined, using binary addition (XOR), to form the output. A code that is obtained in this way is called a *convolutional code*, of which I will give the formal definition later.

Example:
Take a look at the next shift register:

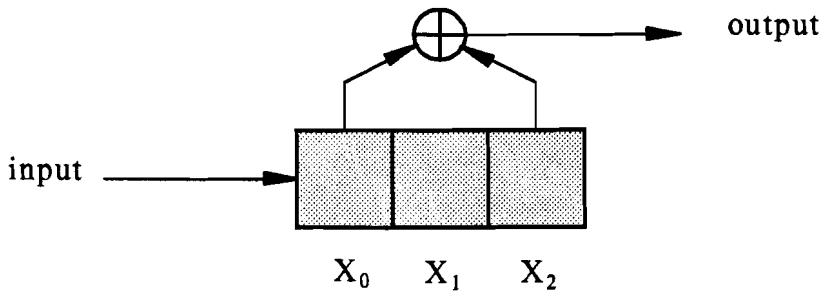


Figure 4-1: A shift register for generating a single output stream.

The shifting of the register can be caught in formula:

$$X_2(t) = X_1(t-1), \quad X_1(t) = X_0(t-1), \quad X_0(t) = input(t).$$

The computation of the output sum is described by

$$output(t) = X_0(t) \oplus X_2(t) = input(t) \oplus input(t-2)$$

We can keep track of the input, output and contents of the registers in every timeslot by means of a table like below (Table). Here we set the initial contents of the registers to X_0=X_1=X_2=0, and have an arbitrary input stream that is, for example, 101000. The output becomes valid after reception of the first input bit, so the "initial" output (0) is discarded.

Table 4-I: Shift register of figure 4-1, generating output for an arbitrary input.

timeslot	input	$X_0 X_1 X_2$	output = $(X_0 \oplus X_2)$
-1	-	0 0 0	(0)
0	1	1 0 0	1
1	0	0 1 0	0
2	1	1 0 1	0
3	0	0 1 0	0
4	0	0 0 1	1
5	0	0 0 0	0

It is obvious that the input sequence cannot be determined uniquely when only looking at the output-sequence and not knowing the initial contents of the registers[1].

To overcome this problem, another output sum is computed in the system as well, as shown in figure 4-2 below:

[1] Verify this by assuming another initial value for the shift register in timeslot -1. The input can be chosen in such a way that the same output is produced as in Table .

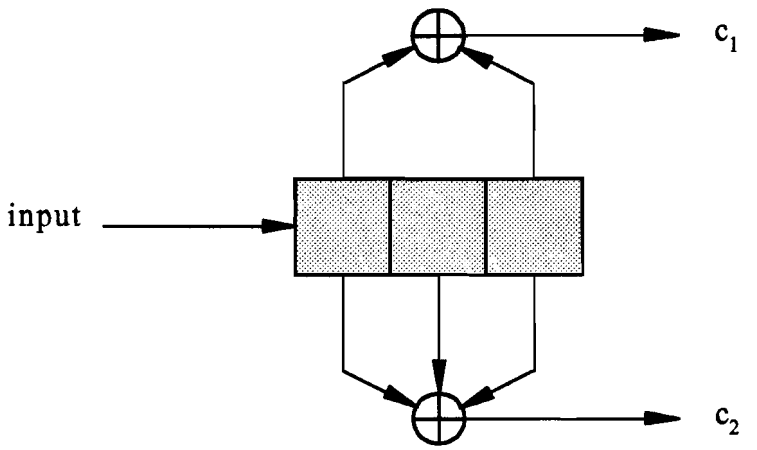


Figure 4-2: A shift register with 2 output streams for generating a code (c_1,c_2) (Moore).

For this shift register, Table is listed below. Now it appears possible to trace back the exact input sequence when given both output sequences.

Table 4-II: ***Shift register of figure 4-2, outputs for an arbitrary input.***

timeslot	input	$X_0 X_1 X_2$	$c_1 c_2$
-1	-	0 0 0	- -
0	1	1 0 0	1 1
1	0	0 1 0	0 1
2	1	1 0 1	0 0
3	0	0 1 0	0 1
4	0	0 0 1	1 1
5	0	0 0 0	0 0

Since we want to use a relatively simple code capable of also correcting errors we must find a suitable shift register. We can view a shift register as a *finite state machine* (FSM) [27: p. 276] with the same properties. In fact, with $m+1$ registers we can construct a *Moore machine* with 2^{m+1} states, because the output is a function of the present state only.
However, since we also have input, we'd rather use a *Mealy machine*: then the states for a shift register consisting of $m+1$ registers, are represented by the m leftmost registers, so there are 2^m different states.

Example:
Let's take the shift register from figure 4-2 (a Moore machine), then the Mealy machine looks like follows:

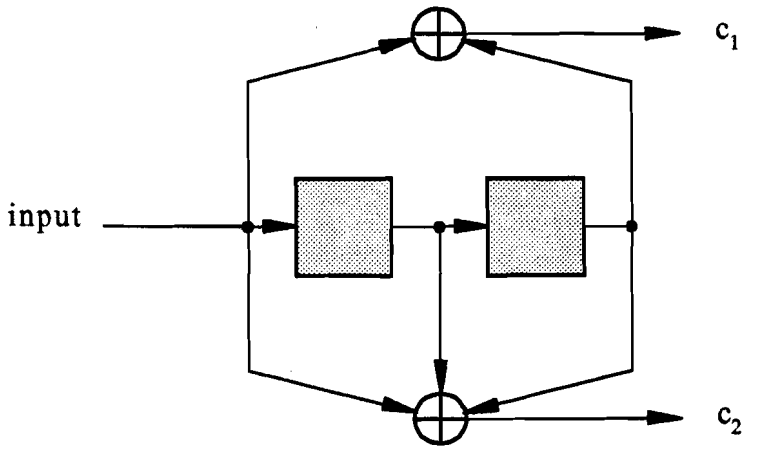


Figure 4-3: Another shift register with 2 output streams for generating a code (c_1,c_2) (Mealy).

If we specify *x(t)*, *y(t)* and *S(t)* to be the input, output and state, known at timeslot *t* respectively, and call λ and δ the output function and state transition function respectively, the Mealy machine is summarized by:

$$y(t) = \lambda[S(t), x(t)]$$
$$S(t+1) = \delta[S(t), x(t)]$$

If the shift register's FSM is currently in state
$S(t) = (s_0, s_1, \cdots, s_{m-1}) = (x(t-1), x(t-2), \cdots, x(t-m))$ then, with input bit equal to $x(t)=b$, the next state reached is $S(t+1) = \delta[(s_0, s_1, \cdots, s_{m-1}), b] = (b, s_0, s_1, \cdots, s_{m-2})$.
The initial state is the *zero state*, the state where each of the first *m* registers contains 0.

This information is often shown graphically: The *state diagram* of this code is a directed graph in which the vertices, or states, are all binary words of length *m*. For each state $(s_0, s_1, \cdots, s_{m-1})$ there is an edge directed from this state to $(b, s_0, s_1, \cdots, s_{m-2})$, $b = 0,1$. The edges are labeled with the respective output belonging to a shift register's contents $(b, s_0, s_1, \cdots, s_{m-1})$.

Example:
The state table of the shift register in figure 4-2 can be found in Table :

Table 4-III: ***State table for the code of figure 4-2.***

present state S(t) $X_0 X_1$	next state S(t+1), output y(t)	
	input=0	input=1
0 0	0 0, 0 0	1 0, 1 1
1 0	0 1, 0 1	1 1, 1 0
0 1	0 0, 1 1	1 0, 0 0
1 1	0 1, 1 0	1 1, 0 1

For this code, the output is given by:

$$y(t) = (c_1(t), c_2(t)) = (x(t) \oplus X_1, x(t) \oplus X_0 \oplus X_1)$$

The state diagram is represented in figure 4-4:

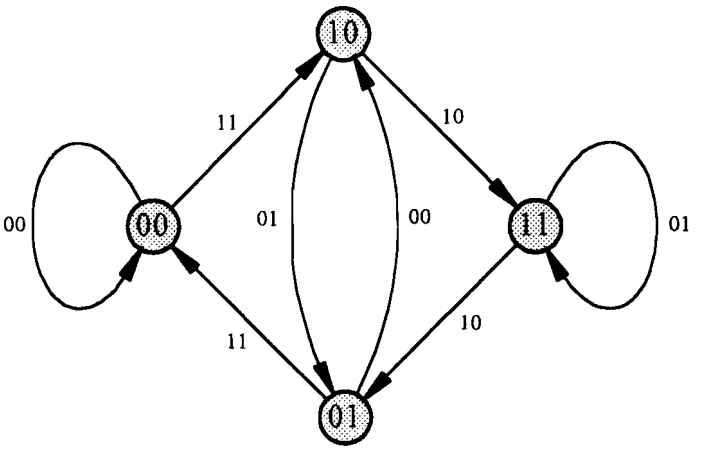


Figure 4-4: State diagram for the code of figure 4-2.

Notice that the left binary digit of each state represents the value of the message bit that led to that state. So, beginning at the zero state, a *(directed) walk* can be made along the edges to adjacent states, meanwhile recovering the message bits.

Another very useful way of viewing the code structure is with help of the *(code) trellis*. A trellis is a kind of state diagram, where the vertices represent the states at certain (equidistant) points in time. In fact, the graph is directed, but due to the causality principle, the graph is acyclic in time: the direction of any edge in the graph is from left to right. In this way, each possible input sequence corresponds to a particular path through the trellis. The convention is that an input 0 corresponds to the selection of the upper branch and an input 1 to the lower branch. For convenience, we label the edges with the corresponding output.

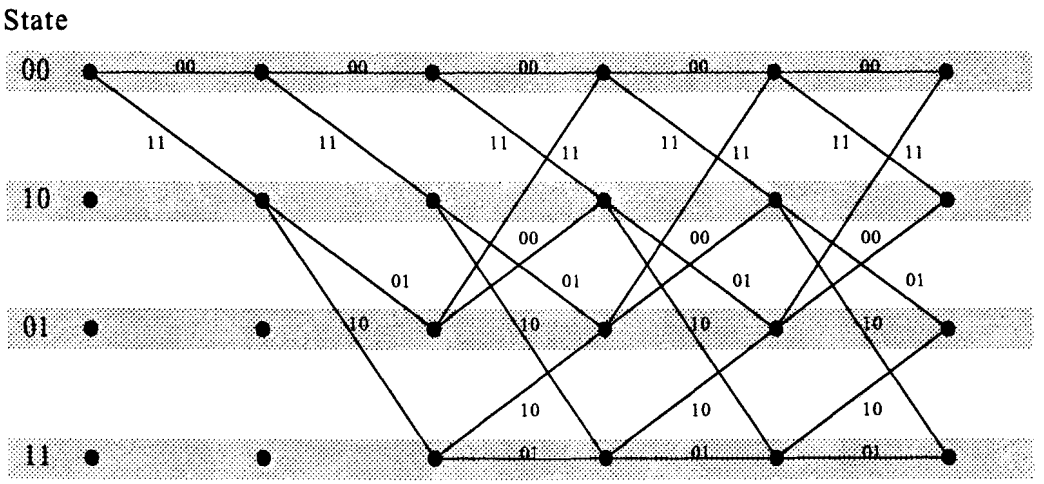


Figure 4-5: Trellis for the code of figure 4-2.

An interesting property is that 2 states, when only differing in the last binary digit, lead to the same next state when receiving the same input bit b. Or, grammatically:

$$\left.\begin{array}{l}(s_0,s_1,\cdots,s_{m-2},0)\xrightarrow{b}\\(s_0,s_1,\cdots,s_{m-2},1)\xrightarrow{b}\end{array}\right\}\ (b,s_0,s_1,\cdots,s_{m-2}),\quad b\in\{0,1\}.$$

where the arrow denotes a transition on input b. Observe that precisely 2 states can be reached from any other state in one step.

This subset of the trellis is commonly referred to as the *butterfly*, because of its typical appearance in a trellis, as shown in figure 4-6 below, where $(n=2^{m-1})\wedge(0\le i<n)$.

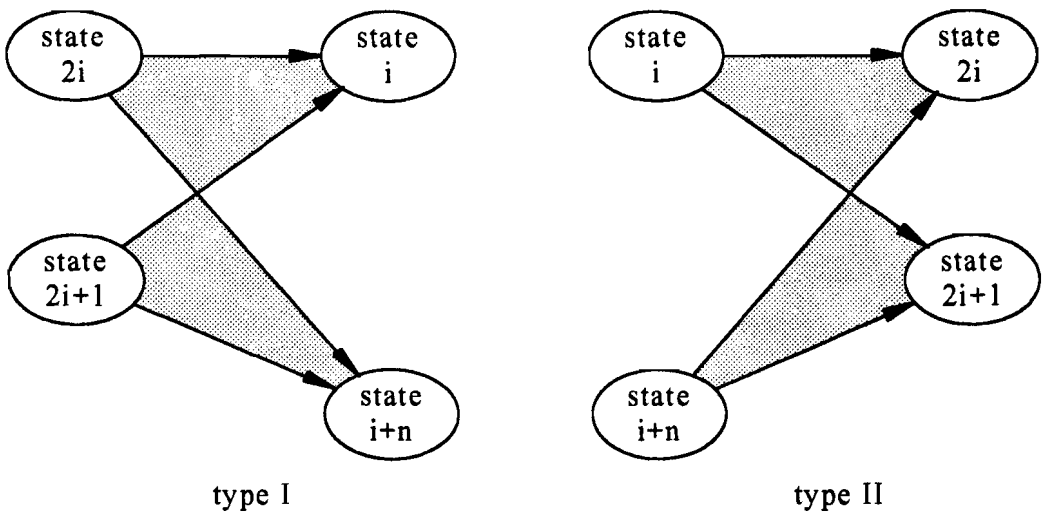


Figure 4-6: Butterflies for convolutional codes.

The two types of butterflies differ in the way the states are numbered: In type I, state $(s_0,s_1,\cdots,s_{m-1})$ has the most recent input bit s_0 as the MSB, in type II the MSB is oldest registered input bit s_{m-1}.
Notice that I used the type I notation to refer to the states, thusfar.

The Viterbi decoding algorithm, which will be discussed later, makes use of the trellis of a code to trace back in time the most likely path along visited states.
However, since the convolutional code used in DTT is a more complex extension of ordinary convolutional codes, I will first go deeper into this material.

4.2 Theory of Convolutional Codes

Let's move to a more formal definition of what exactly a convolutional code is [18: p.192]. For this, we'll regard convolutional code generating machines as Moore machines.
Consider the following so-called *generator polynomials* (sometimes called *check polynomials*) $G_1(D),\cdots,G_n(D)$, where D (sometimes also called z) is the delay operator, and

$$G_i(D) = g_{i,0} + g_{i,1}D + \cdots + g_{i,m}D^m, \quad g_{i,j} \in Z_2,$$

and a sequence of message bits $m_0, m_1, m_2, \ldots$, used to define the message polynomial
$m(D) = m_0 + m_1 D + m_2 D^2 + \ldots, \quad m_j \in Z_2$.
An (n,k=m+1) (binary) *convolutional code* is the code consisting of all codewords

$$C(D) = \left(c_1(D), c_2(D), \cdots, c_n(D)\right),$$

where $c_i(D) = m(D)G_i(D)$.
The *constraint length* of a code is defined as k = m+1 = the number of (previous) message bits where the next code symbol depends upon.

Example:
Recall the Moore machine of figure 4-2. This FSM has generator polynomials
$G_1(D) = 1 + D^2, \quad G_2(D) = 1 + D + D^2$.
The used input sequence 101000... constitutes the message polynomial
$m(D) = 1 + 0 \cdot D + 1 \cdot D^2 + 0 \cdot D^3 + 0 \cdot D^4 + 0 \cdot D^5 + \ldots = 1 + D^2$.
The convolutional code $C(D) = \left(C_1(D), C_2(D)\right)$, with
$C_1(D) = m(D)G_1(D) = \left(1 + D^2\right)\left(1 + D^2\right) = 1 + D^4$, and
$C_2(D) = m(D)G_2(D) = \left(1 + D^2\right)\left(1 + D + D^2\right) = 1 + D + D^3 + D^4$.
This solution can be verified in Table .

The resulting convolutional codeword $C(D) = \left(m(D)G_1(D), m(D)G_2(D), \cdots, m(D)G_n(D)\right)$ can be made into a single bitstream of digits, instead of the n streams we have originally, by *interlacing* $c_1(D), c_2(D), \cdots, c_n(D)$. Interlacing here means the same as time multiplexing.

As the rate measures the fraction of information that each code digit carries, and we have n generated bits for each message bit, the *(code) rate* is defined as $r = 1/n$.

The punctured (this term will be defined further on) convolutional code that will be used in DTT, is based on a mother convolutional code of rate 1/2 with 64 states, i.e. using a shift register with k=7 registers. The 2 generator polynomials of the mother code are

$$G_1(D) = 1 + D + D^2 + D^3 + D^6 \quad (\equiv 171_{OCT})$$
$$G_2(D) = 1 + D^2 + D^3 + D^5 + D^6 \quad (\equiv 133_{OCT})'$$

generating a code $C(D) = (X_{output}, Y_{output}) = (m(D)G_1(D), m(D)G_2(D))$ for message $m(D)$. The encoding, using shift registers, is given in figure 4-7 below (here, the interlacing process of X_{output} and Y_{output} is performed by a double clock speed shift register).

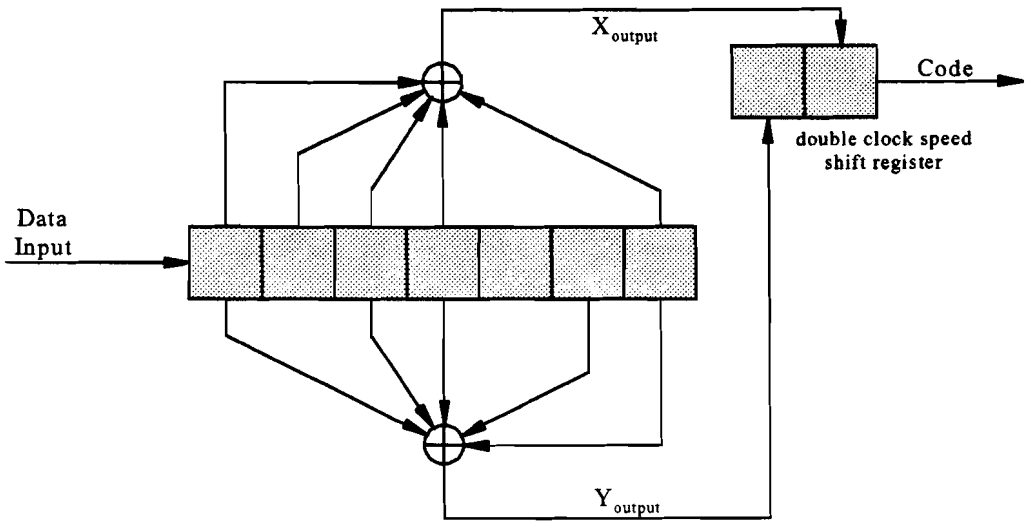


Figure 4-7: The mother convolutional code of rate 1/2.

Note that two codeword bits are generated for each single input message bit, therefore the mother code rate is 1/2.

A high-rate $n/(n+1)$, $n \geq 2$, *punctured convolutional code* is defined [18: p.810] by a set of $(n+1)$ generator polynomials $\{G_1(D), G_2(D), \cdots, G_{n+1}(D)\}$. It is noted that all the generators in this set need not be different. The punctured code can be generated by periodically deleting bits from a low-rate $1/b$, $2 \leq b \leq n+1$, convolutional encoder where b is the number of generators used in the convolutional code.
The pattern in which the coded bits are discarded is called the *perforation pattern* of the punctured code [29: p.80] and is conventionally described by a *perforation matrix*.

In addition to the mother code with rate 1/2, the DTT system allows punctured rates of $n/(n+1)$ = 2/3, 3/4, 5/6 and 7/8. The punctured convolutional code shall be used as given in Table below. In this table, X and Y refer to the pairs of outputs of the convolutional encoder. The mentioned free distance will be discussed later.

Table 4-IV: ***Puncturing pattern and transmitted sequence after parallel to serial conversion for the possible code rates.***

Code rate r	Puncturing pattern / perforation pattern	Free distance d_{free}	Transmitted sequence	Corresponding generator polynomials
1/2	X: 1 Y: 1	10	$X_1\ Y_1$	$G_1\ G_2$
2/3	X: 1 0 Y: 1 1	6	$X_1\ Y_1\ Y_2$	$G_1\ G_2\ G_2$
3/4	X: 1 0 1 Y: 1 1 0	5	$X_1\ Y_1\ Y_2\ X_3$	$G_1\ G_2\ G_2\ G_1$
5/6	X: 1 0 1 0 1 Y: 1 1 0 1 0	4	$X_1\ Y_1\ Y_2\ X_3\ Y_4\ X_5$	$G_1\ G_2\ G_2\ G_1\ G_2\ G_1$
7/8	X: 1 0 0 0 1 0 1 Y: 1 1 1 1 0 1 0	3	$X_1\ Y_1\ Y_2\ Y_3\ Y_4\ X_5\ Y_6\ X_7$	$G_1\ G_2\ G_2\ G_2\ G_2\ G_1\ G_2\ G_1$

Example:
Consider rate 3/4: output bits from the rate 1/2 encoder, corresponding to blocks of n=3 input bits are divided in blocks of $2n$=6 bits. Every third and sixth bit is discarded as specified in the table above. There remain n+1=4 codeword bits which are interlaced, hence a convolutional code of rate 3/4 is generated:

$$m_1\ m_2\ m_3 \overset{conv.coding,\ r=1/2}{\Rightarrow} \begin{matrix} X_1 X_2 X_3 \\ Y_1 Y_2 Y_3 \end{matrix} \overset{puncturing}{\Rightarrow} \begin{matrix} X_1 - X_3 \\ Y_1 Y_2 - \end{matrix} \overset{interlacing}{\Rightarrow} X_1\ Y_1\ Y_2\ X_3$$

Suppose we'd start initially from the zero state S(-1) = 0 0 0 0 0 0 0, and would encode the message 1 0 1 1 1 0 0 0 0 ... with rate 3/4:

$$101|110|000|\ldots \overset{conv.coding,\ r=1/2}{\Rightarrow} \begin{matrix} 110|111|111| \\ 100|001|100| \end{matrix}\ldots$$

$$\overset{puncturing}{\Rightarrow} \begin{matrix} 1-0|1-1|1-1| \\ 10-|00-|10-| \end{matrix}\ldots \overset{interlacing}{\Rightarrow} 1100|1001|1101|\ldots$$

To examine the quality of convolutional codes, we will use some helpful definitions:
The *Hamming distance* d(a,b) between two sequences a and b of length n is defined [4: p.9] as the number of places in which they differ.
The *Minimum distance* d_{min} of a code C is defined [4: p.9] as the Hamming distance of the pair (a,b) of distinct codewords with smallest Hamming distance.
The *Weight* of a convolutional codeword, infinitely long, is defined [4: p.442] as the number of non-zero components (= non-zero bits) it has. In every convolutional code there is a loop on the zero state with weight 0.
The *Free distance* d_{free} of a code is defined [4: p.442] as the weight of the non-zero codeword of smallest weight.
For a convolutional code C with code rate $1/V$ and constraint length k, we consider $L>k$ message bits, where L is the *decoding constraint length*, or *decoding depth*, to generate $L \cdot V$ bit long codeword paths, then we can define the *Minimum decoding distance* d_L as the smallest Hamming distance between any two codewords of length $L \cdot V$, originating from the same initial state, but with a distinct first bit.
If we now let L go to infinity, d_L becomes d_{free}, so $d_{free} \geq d_L \geq d_{min}$.

With the free distance, a statement can be made about the error-correcting capabilities of a code. If at most t errors satisfying $2t+1 \leq d_{free}$ occur in the first $l=m+1$ output bits then the code is able to correct these t errors.

For the DTT punctured convolutional codes, the free distances are known [9: p.14], and given in the above Table .

Example:
The code C of figure 4-2 has $d_{free}(C)=5$, since this is the weight along the path through states (00, 10, 01, 00), corresponding to codeword "11 01 11", whereas all other different paths through a non-zero state have larger weights. For this simple code, the prove can quickly be obtained by an exhaustive search along all possible paths, where a path is rejected as soon as its weight exceeds 5.

If it is possible for a single decoding-error event to induce an infinite number of additional errors in the codeword, then the decoder is said to be subject to *error propagation*. The existence of error propagation may be endemic in the choice of a catastrophic generator polynomial for the convolutional code, in which case it is called *catastrophic error propagation*.
A convolutional code is called *catastrophic* if its state diagram contains a zero weight cycle different from the loop on the zero state [18: p.203].

Example:
Consider the code C generated by the shift register in figure 4-1, having generator polynomial $G(D)=1+D^2$. Suppose we want to send a message

$m(D)=1+D^2+D^4+\ldots=\sum_{i=0}^{\infty}D^{2i}$, encoded as

$$C(D)=m(D)G(D)=\left(1+D^2+D^4+\ldots\right)\cdot\left(1+D^2\right)=1+D^2+D^2+D^4+D^4+\ldots=1.$$

The absence of higher order terms in $C(D)$ indicates a zero weight cycle on a non-zero message, hence we have a catastrophic code.

It can be proved that a (2,1,m) convolutional code is catastrophic precisely when

$$gcd\left(G_1(D),G_2(D)\right)\neq 1$$

It can be shown (using the Euclidean algorithm for division of polynomials) that the code, used in the DTT system, is non-catastrophic for rate 1/2. However, error propagation may also be inherent in the choice of decoding algorithm, in which case it is called *ordinary error propagation*. Especially for the other punctured rates this should be taken seriously when developing the decoder: an important parameter here is the decoding constraint length, as defined above and illustrated in the next chapter.

4.3 Decoding Convolutional Codes

When a convolutional codeword is passed through a channel, errors are made from time to time in the codeword symbols. The decoder must correct these errors by processing the codeword. The convolutional codeword however, is so long that the decoder can only remember a part of it at a time. Although the codeword is infinite in length, all decoding decisions are made on codeword segments of finite length. But because of the structure of the code, no matter how one chops out a part of the received word for the decoder to work with, there is some interaction with other parts of the received word that the decoder does not see. Thus there might be useful information available that the decoder does not use.
The number of symbols that the decoder can store is called the *decoding-window width* W. In general, increasing W increases the performance of the code, but one eventually reaches a point of diminishing return.
The theoretically best way to decode convolutional codes for a given W is to compute all codewords of length W and compare the received word to each of them. Select that codeword closest to the received word, and take the first (oldest) frame of the estimated codeword. This frame is re-encoded and subtracted from the received word. The received word is altered by discarding the oldest frame (the oldest bit(s) still in memory) and adding the new frame at the other end, after which the procedure is

repeated. Unfortunately, the problem with implementing this decoder, known as the *minimum-distance decoder*, is too complex [4: p.378].
Although several other decoding algorithms are known, like the Fano algorithm [4: p.383], the Viterbi algorithm is widely accepted as the most practical one: it is a maximum likelihood decoder for memoryless binary noise, and strongly recommended in [11].

4.4 The Viterbi Algorithm

The Viterbi algorithm makes use of weighting factors, or *metrics*, to find the most likely sequence of message bits that was sent. The algorithm traces through a trellis identical to that used by the encoder in an attempt to emulate the encoder's behaviour. The edges in the trellis are called *branches*, where each branch has a weighting factor called the *branch-metric*. This branch-metric is an estimation value, computed by the decoder when receiving the next code symbol, of how likely the encoder followed the same branch in its own trellis. If we walk along a path through the decoder's trellis and sum up all branch-metrics along this path (the *path-metric*), this gives a measure of how likely the encoder took the same path in its trellis: the sequence of states along the most likely path reveals the most likely message part thusfar, in other words the path that results in the smallest distance to the expected codeword.

4.4.1 Metric Calculations

Except for the additive structure used in the measure of distance, the nature of distance is unimportant. An obvious solution is to define the branch-metric as the Hamming distance between the trellis output belonging to that branch and the received codeword bits. However, in the oncoming investigation of the Viterbi algorithm I will try to keep the definition open for other suggestions.

In the decoder's trellis, state i has trellis output (c_1,c_2) when receiving an input b. Further, suppose in time-slot j the Viterbi decoder receives a symbol (h_1,h_2) coming from the channel, then we can compute all branch-metrics $BM_{i,b}(j)$ as

$$BM_{i,b}(j) = m_1 + m_2 \quad \text{with} \quad m_l = \mathrm{d}(h_l, c_l), \quad l = 0,1$$

Next, we can calculate the metric of a series of branches: the path-metric or *discrepancy*. Whenever two paths coincide at a state in the trellis, we choose the path with the minimum resulting path-metric (the *survivor*), and with equal path-metrics an arbitrary path is chosen. Recalling the properties of butterflies in a trellis, we can be sure that in timeslot j each state in the trellis is always reached by two branches from states in timeslot j-1, so for every state we must compare two path-metrics.

For type II butterflies we'll find branches from state i to $2i$, from state i to $2i$+1, from state i+n to $2i$, and from state i+n to $2i$+1, as depicted in the following figure.

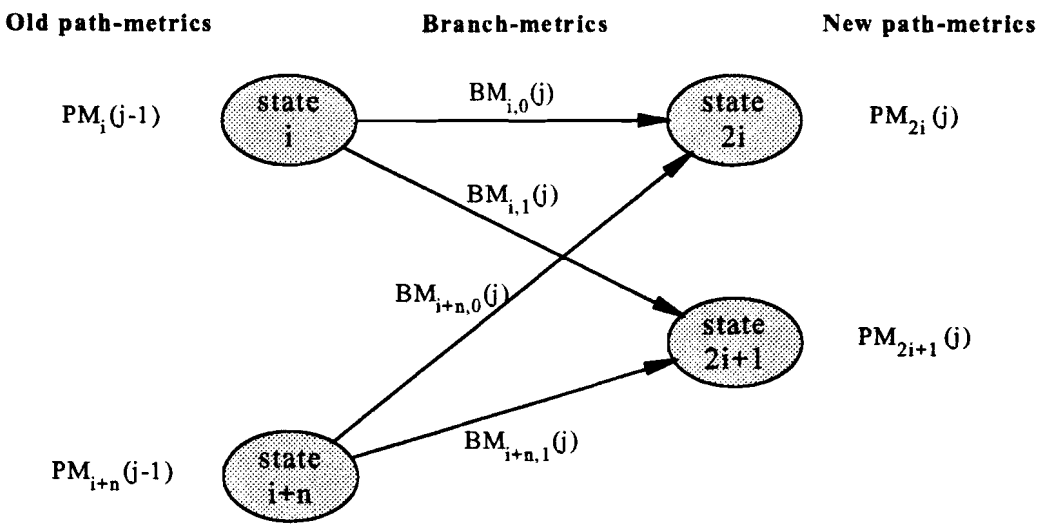


Figure 4-8: Metrics in a type II butterfly.

In each butterfly we choose the 2 branches which result in the smallest new path-metrics $PM_{2i}(j)$, $PM_{2i+1}(j)$:

$$PM_{2i}(j) = \min\left(PM_i(j-1) + BM_{i,0}(j), PM_{i+n}(j-1) + BM_{i+n,0}(j)\right)$$
$$PM_{2i+1}(j) = \min\left(PM_i(j-1) + BM_{i,1}(j), PM_{i+n}(j-1) + BM_{i+n,1}(j)\right)$$

Example:
We'll use the shift register from figure 4-2 to show how the metrics-calculation with Hamming distance is done. Suppose the input sequence 0 1 0 1 1 1 0 0 0... was encoded starting in state 00, resulting in the encoder output sequence 00 11 01 00 10 01 10 11... . Suppose the channel mutated this sequence by flipping 2 bits at positions 4 and 5, resulting in 00 10 11 00 10 01 10 11... as seen at the receiver. The decoder's trellis, having a decoding-window width W, evolves as can be seen in figure 4-9 below:

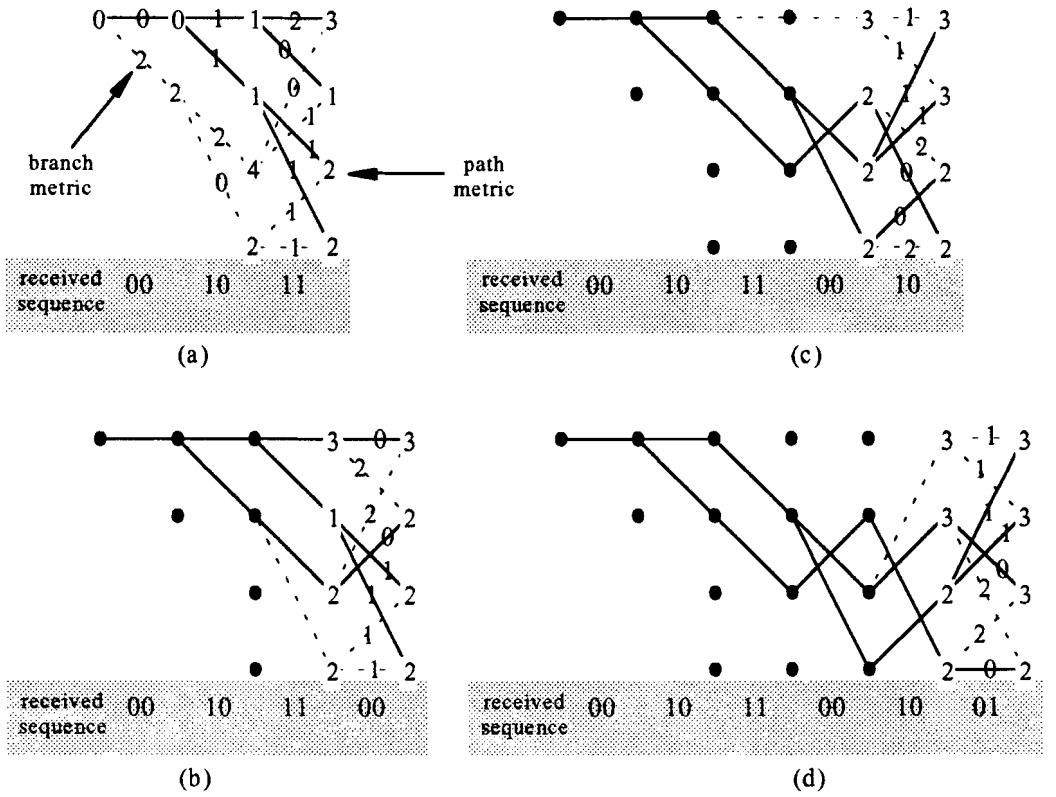


Figure 4-9: Path-metric calculations. (a) after 3 timeslots; (b) 4; (c) 5; (d) 6.

The dotted lines/branches are "dead-ends", i.e. they are not part of a candidate path anymore, as a result of either

1. a larger path-metric than its rival path in the butterfly;
2. a decoder-decision in favour of its rival path with equally good path-metric;
3. both its successor branches in the next timeslot are "dead-ends" (back heritage).

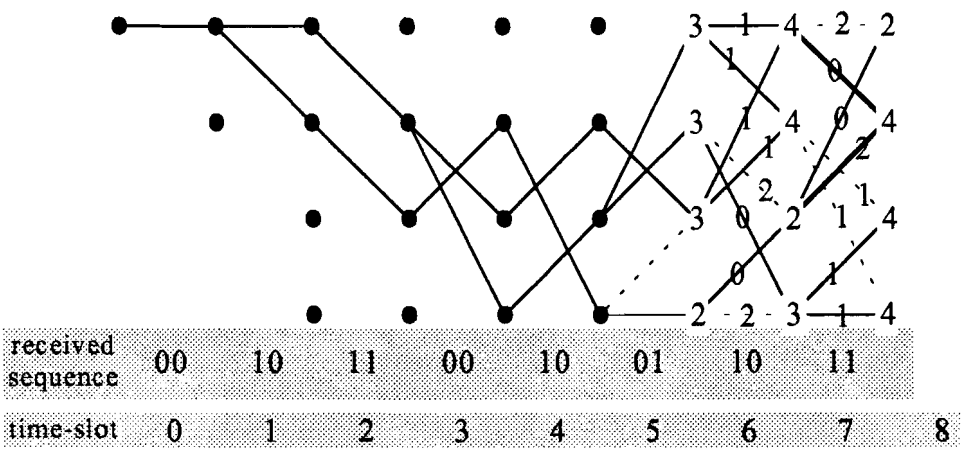


Figure 4-10: Path-metric calculation of 2 more timeslots.

4.4.2 Branch Decision Strategies

A question that now arises (see point 2 mentioned above) is that based on what grounds the decoder can make a decision between two paths with equal path-metrics? Look for example at the 2 thick printed branches in timeslot 7 in figure 4-10. Since these two branches result in the same Hamming

distance to the received bit-sequence and we have no further knowledge about the correctness of this sequence, the decoder can choose either one.
A reasonable strategy is to create "dead-ends" when facing such a decoding dilemma, which results in paths merging very soon when looking back in time. When applied in the above example, we must decide in favour of the lower branch from state 01 to state 10, resulting in the upper branch becoming a "dead-end".
The second strategy is to keep both possibilities open until these branches reach the end of the decoding window. However, this is not very practical because we do not know the number of such dilemma's, and we may even have to make a choice for one of the paths after all.
The third strategy is simply a "coin-flip" to reach a decision.

The first strategy mentioned above cannot always be applied: for example look at the butterfly in figure 4-11, timeslot 6. A reasonable choice would be to choose the predecessor of both branch-receiving states to be the same (arbitrary) state.
Coincidentally choosing either one has the same result: in timeslot 7 the dead-end strategy leads to the elimination of all four of the optional branches, as depicted below.

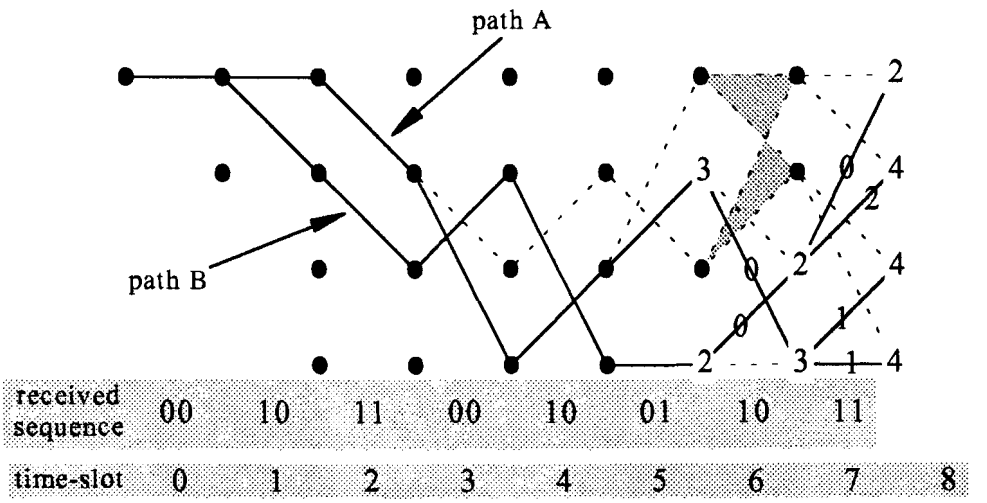


Figure 4-11: Result of applying dead-end strategy.

From the above examples it again becomes clear that every decoding-decision always concerns only one butterfly, not influenced by the other butterflies at this point.

Until timeslot 8, we postponed to make a decision in favour of path A or path B, although the path-metric for path B suggests to be the best. Recovering the message bits for both paths up to timeslot 7 results in the following table VI:

Table 4-V: ***Message decoding for two candidate paths A and B.***

original message	next state (start =00)	encoder output	decoder input	received in timeslot:	states on path A	message path A	states on path B	message path B
0	00	00	00	0	00	0	00	0
1	10	11	10	1	00	0	10	1
0	01	01	11	2	10	1	01	0
1	10	00	00	3	11	1	10	1
1	11	10	10	4	01	0	11	1
1	11	01	01	5	10	1	11	1
0	01	10	10	6	11	1	01	0
0	00	11	11	7	01 / 11	0 / 1	00 / 10	0 / 1

The table reveals the result we hoped for: path B, having the smallest path-metric, is the path of the correct message! Verification shows that the path-metric of path B equals the number of errors in the received sequence. However, if a false path has the best path-metric then the correct path apparently has a larger distance to the expected correct path, so the path-metric is an underbound on the number of errors on the "best" path.

4.4.3 Path Decision Strategies

It is clear that if we would wait long enough before actually deciding for a path, that would give the most likely solution. The question then arises: how and when are these output bit decisions to be made, and how large must we choose the decoding window width W?

Part of this question is answered when employing the following strategy, called the *maximum likelihood output rule* [27: p.317]. After processing each pair of received parity bits (i.e. branch-metric computation) and updating path metrics and path histories, the best path metric is found: the according path has the largest path likelihood. Then the oldest bit on this path will be chosen as the output bit decision.
Note that the k most recent bits on a particular path are defined by the state associated with that path, and thus, when path histories are stored as bit sequences, that storage may begin with the bit decision associated with the $(k+1)^{th}$ previous node.
Also, this rule doesn't pin itself down to one single path throughout: if, at a certain point in the future, another path has a better path-metric, the next output bit is simply chosen to be from this latter path.
Since no good theoretical performance bounds can be deduced [1,4,27,etc.], we must rely on simulation results of the decoder. Simulations [27: p.317-330] for convolutional codes with rate $1/b$ show that for a decoding window width W larger than[2] $4k$, results essentially identical to those for $W=4k$ bits were obtained, indicating that no further significant performance gains could be realized for the unquantized channel when the maximum likelihood output rule is used.
Other sources [18: p.206] mention W to be somewhere in-between $4m$ and $6m$, used in practice.

Another technique, used to make output bit decisions, is the *majority-voting* decision on the oldest bits in the decoding window, where a decoding window width of $5k$ gives comparable performance to the maximum likelihood output rule.
A third option is to make a decision by just picking a path at random, *random-pick*, where simulations show that, for a comparable performance, the size of the decoding window must be approximately doubled, leading to a decoding window width of about $8k$ bit [27: p.325].

4.4.4 Remarks and Features

At this point a remark of practical interest can be made: because we know that the Reed-Solomon code works on whole bytes, we could recover the decoded message byte by byte, instead of for every bit. The advantage of this strategy is that only once in every 8 output bits a traceback has to be performed instead of for each bit separately, saving us a lot of memory accesses. For a comparable performance, the decoding window width must be extended by 1 byte.

Some very attractive additional features can be included in a Viterbi decoder design for a modest additional cost. For example, during normal operation on a good channel, the decoder typically sees the path metrics for the correct (best) path grow at a fairly steady rate while the other path metrics lay behind more or less as a group. The rate of increase of the correct path metric is a function of the received *Signal-to-Noise Ratio* (S/N), and after averaging the rate of increase of the metric along the path favored by the decoder, an *estimate of the channel quality* can be produced. In addition, should the channel S/N degrade, the Viterbi decoder can readily recognize the poorer link transmission quality.

[2] Recall that $k=m+1$ is the constraint length of a convolutional code.

Another useful feature of Viterbi decoding is that it can be made *self-synchronizing*. That is, it is not necessary to know where a message starts, to be able to decode. One may simply try a code rate and an arbitrary puncturing block starting point and attempt decoding with all path metrics initially set to the same value. If the chosen code rate and puncturing block starting point are correct, the path metrics will grow slowly compared to incorrect decoding. If the path metrics indicate incorrect decoding, the start of the puncturing block is changed and decoding is restarted, and the procedure continues. In the DTT system self-synchronization is not necessary, however in the ETSI/DVB standards for Satellite and Cable systems the lack of frame synchronization makes this is a very useful option.

A remark should me made on 180-degree (or π-) phase ambiguity, which can occur in the satellite standard. This phenomena is a typical transmission uncertainty about what the sign of the received symbols in the signal are. The used code in DTT happens to be a *transparent convolutional code* [27: p.325], which has the property that complements of codewords are codewords as well. A code will be transparent if each parity generator has an odd number of taps, which is true in our case. The correct code can be obtained by decoding the MPEG-2 sync byte delimiting the interleaved frame [9: p.13].

As a final remark, we could increase performance of the Viterbi decoder specifically for DTT by making use of foreknowledge about the message. Since we know that, at the transmitter side, the synchronization bytes are infected by scrambling, RS-encoding nor outer interleaving, they enter the convolutional encoder in their original form. So, we may expect them to reappear this way when leaving the Viterbi decoder at the receiver side. In fact, we can already spot them (by counting 203 bytes after the last received sync byte) when they just enter the decoding window! Knowing this, we can adjust the according path-metric(s) to the best value, and the other path-metrics to the worst value to be sure that this specific path will be chosen.
Observe that this option can only be used when there already is a lock on the sync bytes.

4.5 Soft-Decision Decoding

In the examples for explaining the Viterbi algorithm thusfar, the branch-metric was regarded as the *Hamming distance*. Since no additional channel information was used to question the reliability of the received symbols this is called *Hard-Decision Decoding* (HDD).

Several sources [4, 27] report a performance increase of about 1 to 2.2 dB when using *Soft-Decision Decoding* (SDD) instead of HDD: additional channel information is used to estimate the reliability of the received symbols.
Error-control codes give the best results if an intelligent interaction exists between the modulator/demodulator (modem) design and the encoder/decoder (codec) design. To relate this to DTT, a simplified block diagram is given below:

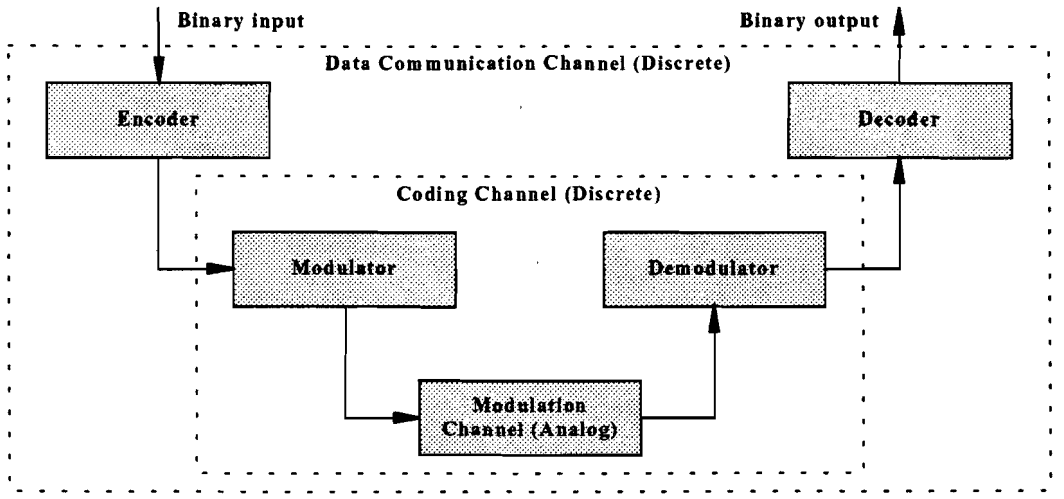


Figure 4-12: Simplified Block-diagram of the DTT system.

The transmission side (encoder and modulator) has already been specified in [9,10,11,12], we will need to examine/design the other blocks to obtain a receiver that meets the performance requirements. Of a fixed digital communication system that transmits digital data through an additive Gaussian noise channel the quality is judged by the performance at low transmitted power [4: p.461].
The requirements are derived from simulated system performance results anticipating perfect channel estimation and without phase noise, and are given in [12]. The following modulation channel models have been used in the simulations:

- Gaussian channel: for transmission disturbed by white noise;
- Ricean channel: for fixed reception and echoes;
- Rayleigh channel: for portable reception and echoes.

For simplicity, I have concentrated on the Gaussian channel.

4.5.1 The Modulation Channel

In the DTT transmitter, v-bit data symbols $Y_i = [y_0,...,y_{v-1}]$ coming from the Inner Interleaver are mapped on 2^v complex symbols $z_i = z_{i,I} + i \cdot z_{i,Q}$ in a constellation, where v=2, 4 or 6 for QPSK, 16-QAM and 64-QAM respectively (see chapter 2.2.6.).
Since the bits in $y_0,...,y_{v-1}$ are randomly picked, every possible combination of these v bits is equally likely to happen: $P(Y_i) = 2^{-v}$. As a result, the values $z_{i,I}$ and $z_{i,Q}$ are not correlated, so we have a discrete rectangular distribution on the constellation: $P(z_{i,I}) = P(z_{i,Q}) = 2^{-v/2}$.
To have equal transmission costs, in DTT transmission is performed with the same energy for all modulation schemes. Therefore the complex symbols z_i are scaled down to normalized modulation values $c_i = c_{i,I} + i \cdot c_{i,Q}$ with a normalization factor as shown in Table 4-VI, yielding an average signal energy $C = E\left[c_i \cdot c_i^*\right] = 1$.

Table 4-VI: *Normalization factors for data symbols.*		
Modulation Scheme	α	Normalization factor
QPSK		$c_i = z_i / \sqrt{2}$
16-QAM	1	$c_i = z_i / \sqrt{10}$
	2	$c_i = z_i / \sqrt{20}$
	4	$c_i = z_i / \sqrt{52}$
64-QAM	1	$c_i = z_i / \sqrt{42}$
	2	$c_i = z_i / \sqrt{60}$
	4	$c_i = z_i / \sqrt{108}$

The modulation values c_i are send on data carriers, which are mingled with additional information-carrying pilots in an OFDM symbol: the OFDM symbols constitute a juxtaposition of equally-spaced orthogonal carriers. The amplitudes and phases of the data cell carriers are varying symbol by symbol according to the mapping process described in chapter 2.2.6.

When these OFDM symbols are transmitted over an *Additive White Gaussian Noise* (AWGN) Channel, introducing no energy losses and adding a sample of Gaussian noise $n_i = n_{i,I} + i \cdot n_{i,Q}$ with *Noise Variance* $N = \sigma^2$ to each symbol $c_i = c_{i,I} + j \cdot c_{i,Q}$, this symbol is received as $r_i = r_{i,I} + j \cdot r_{i,Q} = c_i + n_i = c_{i,I} + n_{i,I} + j \cdot (c_{i,Q} + n_{i,Q})$.
The input c_i to the AWGN channel can have any of 2^v discrete values, so the conditional *Probability Density Function* (PDF) of the channel output r, given an input c_i, is given by [27]

$$p(r|c = c_i) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(r-c_i)^2/2\sigma^2},$$

where σ is the *Standard Deviation* of the signal r.
The overall PDF for an output r is given by a weighted summation over all the PDF's, for all possible inputs $c_i \in C$:

$$p(r) = \sum_{c_i \in C} p(r|c = c_i) p(c = c_i) = 2^{-v} \cdot \sum_{c_i \in C} \frac{1}{\sqrt{2\pi}\sigma} e^{-(r-c_i)^2/2\sigma^2}.$$

Example:
This function is given for QPSK modulation, in a mesh-plot in **Appendix B**. For the *Carrier-to-Noise Ratio* (C/N), we took 3.1 dB. Since $\frac{C}{N}[\text{dB}] = 10 \cdot log\left(\frac{C}{N}\right)$, and we know that $C = E[c_i \cdot c_i^*] = 1$, the noise is computed as $N = 10^{-0.31C} = 0.4898$. For the standard deviation we have $\sigma = \sqrt{N} = 0.6998$.
On the second page of **Appendix B** we plotted the bit error rates (BER) of the different modulation schemes on the AWGN channel, which can be computed from the PDF's.

The Gaussian noise on the channel thus changes the discrete signal c to a continuous signal r. Since a digital system must quantize received signals r to discrete values d in order to perform computations on them, we will think of the modulation channel as an M-input, Q-output *Discrete Memoryless Channel* (DMC), as depicted in the following figure[3]:

[3] Here, the index of c distincts between all possible different values for a discrete signal c.

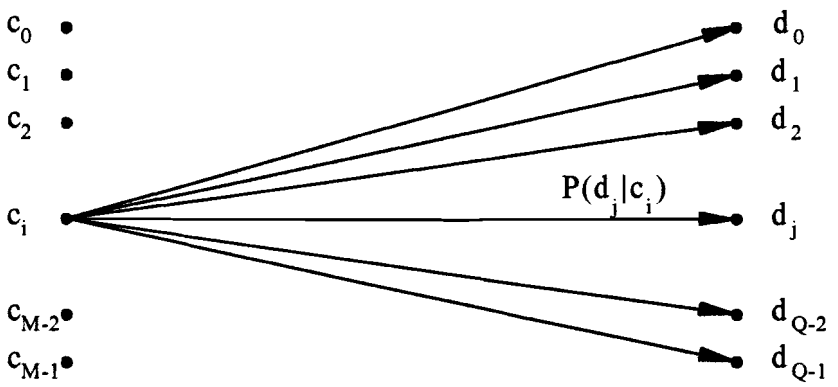


Figure 4-13: Model of an M-input, Q-output Discrete Memoryless Channel.

For a hard-decision decoder M = Q, whereas for a soft-decision decoder M < Q. In the easiest form of a soft-decision decoder Q = M+1, where d_{Q-1} denotes an erasure.
In our case $M=2^v$, and in general we quantize every received continuous signal r to one of $Q=2^{v+SD}$ values, where *SD* is the number of additional soft-decision bits for each constellation point. It is obvious that, for economical reasons, we must keep the number of soft-decision bits low, but high enough to be able to meet the performance requirements. The question is how big to choose *SD*, how we must quantize the continuous values r to discrete values d, and how the Viterbi decoder can operate on soft-decisions.

This problem has been tackled in a number of sources [4,7,15,27] and concentrates on relating a distance or reliability value to each received bit, called the log-likelihood, which will be explained in the following chapter.

4.5.2 Metric Calculations for Soft-decision Decoding

First, we must state that, since we know that the Gaussian noise has no influence on the independence of the orthogonal components, we can divide the noise $N = N_I + N_Q$ evenly between the In-Phase and Quadrature component: $N_I = N_Q = N/2$.
The problem of defining a soft-decision value for a bit in a constellation of Gray encoded square QAM, is solved by simply defining separate sub-channels for each bit position in the constellation, and compute the bit error rate on each sub-channel. Although there is a dependency between the separate bits in one component in the constellation, this hardly influences the outcome [42: p.20].

We'll now formulate the concept of the log-likelihood or algebraic value $L(u)$ [15: p.248] of a binary random variable $u \in GF(2) = \{+1, -1\}$:

$$L(u) = ln\frac{P(u=+1)}{P(u=-1)}$$

We subsequently call $L(u)$ the "soft" value or L-value of a binary random variable.
An important property of L-values as defined above is the following:

$$L(-u) = ln\frac{P(-u=+1)}{P(-u=-1)} = ln\frac{P(u=-1)}{P(u=+1)} = -ln\frac{P(u=+1)}{P(u=-1)} = -L(u)$$

So the sign of $L(u)$ corresponds to the hard decision, and the magnitude $|L(u)|$ is the reliability of this decision. Suppose we receive a continuous value r from the channel denoting a bit u, then

$$\begin{aligned} L(u|r) &= ln\frac{P(u=+1|r)}{P(u=-1|r)} \\ &= ln\frac{P(r|u=+1)\,P(u=+1)/P(r)}{P(r|u=-1)\,P(u=-1)/P(r)} = ln\frac{P(r|u=+1)}{P(r|u=-1)} \\ &= ln\frac{PDF(r|u=+1)}{PDF(r|u=-1)} \end{aligned}$$

These conditional PDF's can easily be derived from the constellations for the respective types of modulation[4].

Example:
In **Appendix C**, the conditional PDF's and corresponding log-likelihood magnitudes $|L(u)|$ are given for the In-Phase component of all modulation types in DTT's non-hierarchical transmission. The dashed lines denote the conditional PDF's of the "0"-bit, and the solid lines denote the conditional PDF's of the "1"-bit. Note that here the In-Phase component is not yet normalized.

Now, suppose we receive a sequence of continuous values $R = r_1,r_2,r_3,...$ and want to know the probability that a transmitted sequence $U = u_1,u_2,u_3,...$ corresponds to R. Since we assume that the received symbols are not correlated, we can say that, for any sequence R and U

$$\begin{aligned} P(U|R) &= P\left(u_1 \wedge u_2 \wedge u_3 \wedge \ldots \middle| r_1 \wedge r_2 \wedge r_3 \wedge \ldots\right) \\ &= P\left(u_1|r_1\right)\cdot P\left(u_2|r_2\right)\cdot P\left(u_3|r_3\right)\cdot\ldots \end{aligned}$$

With this knowledge, we can compute the log-likelihood of the sequence U, when given R:

$$\begin{aligned} L(U|R) &= ln\frac{P(U|R)}{P(-U|R)} = ln\frac{P\left(u_1|r_1\right)\cdot P\left(u_2|r_2\right)\cdot P\left(u_3|r_3\right)\cdot\ldots}{P\left(-u_1|r_1\right)\cdot P\left(-u_2|r_2\right)\cdot P\left(-u_3|r_3\right)\cdot\ldots} \\ &= ln\frac{P\left(u_1|r_1\right)}{P\left(-u_1|r_1\right)} + ln\frac{P\left(u_2|r_2\right)}{P\left(-u_2|r_2\right)} + ln\frac{P\left(u_3|r_3\right)}{P\left(-u_3|r_3\right)} + \ldots \\ &= L\left(u_1|r_1\right) + L\left(u_2|r_2\right) + L\left(u_3|r_3\right) + \ldots \end{aligned}$$

The sum of the log-likelihoods along a path is a measure for the likelihood of that path: the path with *maximum cumulative log-likelihood* is the most likely. Because the Viterbi algorithm originally chooses the *minimum* of cumulated branch-metrics as the most likely sequence, the negated L-value can be used to define the branch-metric.

Log-likelihoods tend to keep growing in magnitude $|L(u|r)|$ for larger $|r|$, so at a certain value we must have a virtual threshold T>0 to keep the log-likelihoods inbound. At the same time we can define a set of normalized reliability measures:

[4] In practice, on many soft-decision channels that are not Gaussian, $L(u)$ is unknown, and the Euclidean distance is used.

$$\alpha_j = \begin{cases} 1 & if \left|L(u_j|r_j)\right| \geq T \\ \dfrac{\left|L(u_j|r_j)\right|}{T} & if \left|L(u_j|r_j)\right| \leq T \end{cases}$$

There is still a problem with these reliability measures: we must quantize them to get a finite set of (discrete) values, say with an accuracy of *sd* bit per *L*-value (and, of course, 1 bit for the hard-decision). The simplest way to do this is to define the quantized reliability measures $\alpha_{j,q}$ uniform as integers

$$\alpha_{j,q} = \left\lfloor \alpha_j \cdot (2^{sd} - 1) \right\rfloor$$

The value of *sd* must be kept low, so this is a point of investigation: in the DTT receiver every bit of log-likelihood must be transported through the Inner De-Interleaver between the demodulator output and the Viterbi decoder input ports. Each extra soft-decision bit, in worst-case of 64-QAM and 8k transmission mode, gives an additional need for 4536 byte of memory, so in total we'll need (1+*sd*)·4536 byte of memory for the Inner De-Interleaver alone!

Initially, the path-metrics are all set to zero, since we do not know the state in which the convolutional encoder was when generating the first received bit.
The bits are received sequentially, so with these we can fill up the empty gaps in the puncturing matrix from left to right. While doing this, for each butterfly the branch-metrics and path-metrics are calculated for the first received pair of bits (see also figure 4-8). The branch-metrics calculation involving erasures is a bit tricky: since we do not know whether a "0" or "1" was sent, we cannot make a hard decision. However, the probability is the same for both options, so we could define the soft value as a constant value.

4.6 Performance of the Viterbi Decoder

As stated earlier, we must rely on simulations to find out the performance of the Viterbi decoder. The parameters that are still open for determination are the decoding window width (or pathlength) W, the number of soft-decision bits *sd*, and the threshold value T. The tests must be performed at all performance bounds as given in [11], before a statement can be made about standard compliance. This means that for every possible code rate (1/2, 2/3, 3/4, 5/6, and 7/8) and type of modulation (QPSK, 16-QAM, 64-QAM; hierarchical and non-hierarchical), we must examine the performance when trying all combinations of the parameters. The tests must be performed on a substantial amount of message bits, say one million output bits, to get an impression of the performance.
It must be noted that, as can be seen in **Appendix A**, a post-Viterbi decoder **byte** error rate of $1.25 \cdot 10^{-3}$ gives a post-RS decoder **message** error rate of $2.0 \cdot 10^{-4}$. This is the measure against which all performance results are held.

The first series of tests (see **Appendix D**) is performed to find an optimum value for the threshold T and number of soft-decision bits *sd*. The tests are done with a constant pathlength of 200, a value that is large enough to exclude performance gain when increased. Each plot contains the results for one code rate (1/2) because here T is of most influence, and only non-hierarchical transmission is examined.
It is clear that the number of soft-decision bits *sd* is of influence on the optimum threshold value, however this influence declines if *sd* gets bigger. If *sd*>2, a choice for T=2 seems legitimate. Because the large pathlength flatters the results, we will continue the tests with *sd*=3. Note also that for 16-QAM the performance is worse than for QPSK, and for 64-QAM it is better than for QPSK. This can be explained when the input bit error rates to the Viterbi-decoder are compared: for QPSK this is $7.6 \cdot 10^{-2}$ at 3.1 dB, for 16-QAM this is $8.2 \cdot 10^{-2}$ at 8.8 dB, and for 64-QAM this is $7.4 \cdot 10^{-2}$ at 14.4 dB. So, following the specified requirements, 16-QAM has a worse starting point to begin with.

The second series of tests (see **Appendix E**) is performed to find the smallest path-length or decoding window width W, just good enough to meet the specified requirements. The test-results show that for larger code rate $n/n+1$ a larger path-length is needed to reach the same or better performance. Again, the difference between the 3 modulation types attracts attention: the 16-QAM performs worse than the other two. Even the performance bound is not reached for code rate 1/2, and barely for code rate 7/8. Altogether, we could settle for path-lengths of about W=60 bit for rate 1/2, W=75 for 2/3, W=90 for 3/4, and W=140 bit for 5/6 and 7/8. For this choice all QPSK and 64-QAM rates perform according to the requirements, and for 16-QAM we probably must choose for sd=4 bit of soft-decision.

4.7 Memory Management for Viterbi Decoding

After metrics calculation, two things must be stored: the new path-metrics and, of course, the survivors that led to them. The trellis-structure suggests that each survivor be stored at a position indicated by the state it led to. Since each state can be reached by only 2 known different states, 1 bit of storage information suffices to be able to trace back to a previous state and along a path. The trace back calculations depend on the type of butterfly that is being used (see figure 4-6):

- Type I: Suppose being in state i or i+n. If the predecessor is state 2i, a '0' is stored, and if the predecessor is state 2i+1, a '1' is stored. Trace-back is done by loading the state-number (i or i+n) in a register with the MSB at the left, retrieving the earlier stored survivor-bit and shifting it in *from the right*. Now the register contains the predecessing state, indicating the next survivor.
- Type II: Suppose being in state 2i or 2i+1. If the predecessor is state i, a '0' is stored, and if the predecessor is state i+n, a '1' is stored. Trace-back is done by loading the state (2i or 2i+1) in a register with the MSB at the left, retrieving the earlier stored survivor-bit and shifting it in *from the left*. Now the register contains the predecessing state, indicating the next survivor.

This procedure for type I and type II can be repeated as long as survivors are stored in the path-history. Note that while tracing back, loading does not have to be repeated.

For a decoding window width $W \leq 2^s$, in every timeslot 2^m survivors are stored, which makes a grand total of $W \cdot 2^m$ bit of storage. A practical survivor-storage technique is the following:
The memory is divided into W blocks of 2^m states, each block thus stands for all survivors in one timeslot, and the position in a block is associated to the state-number. The fact that the number of states is a power of 2 simplifies tracing back: if we store the blocks after another for consecutive timeslots, then the survivor addresses can be divided into two parts. The least significant part (m bit wide) is used to denote the survivor position, and the more significant part (s bit wide) denotes the timeslot modulo W. The previous timeslot is found by subtracting 1 (modulo W) from the more significant address-part.

The path-metrics also occupy memory: since we would like to have only the latest path-metrics in store, and these metrics grow steadily with the received amount of bits, every now and then we will have to reduce all path-metrics by the same value V to keep them inbound. A smart way of keeping the number of reductions low is to invert all soft-decision bits before handing them to the Viterbi decoder and keep track of the **minimum path-metrics** instead of the maximum path-metrics. Secondly, we do not want to be needing too much memory to store each metric, so perhaps we can find some properties, concerning path-metrics, which can be used for minimizing the amount of memory.

The following procedure, described for sd=0, with minor modification is also applicable for sd>0:
In the code used by us, the 2 candidate predecessor-states for any next state differ exactly in one bit-position, and this position contributes to both trellis output-bits (see figure 4-7). As a consequence, the trellis output-bits for one state are the opposite of the other state's trellis output-bits. With the same decoder input, the branch-metrics leading to the next state are either 0 and 2, or both 1.

We want to know if there is an upperbound for the difference between the smallest path-metric and the largest path-metric at a given point in time. Define the smallest path-metric PM(j) as a (discrete) function over time (timeslot j).

Let's examine a state s_i at timeslot j with smallest path-metric p_i=PM(j):

- PM(j) is non-descending, because the branch-metrics are equal or greater than 0.
- In the next timeslot there are at most 2 possible states with predecessor s_i. If the branch-metrics are 0 and 2 respectively, then PM(j+1)=PM(j) because the zero-weight branch then lies on a smallest path-metric path. If the branch-metrics are both 1, then PM(j+1)=PM(j)+1 if s_i was the only state with path-metric p_i in timeslot j. Else, possibly PM(j+1)=PM(j) if there was at least one other state with path-metric p_i in timeslot j and an outgoing zero-weight branch. Fitted in a formula:

$$\forall_{j\geq 0} PM(j) \leq PM(j+1) \leq PM(j)+1$$

- In the trellis, all possible states can be reached from state s_i in at most *but also exactly* 6 steps, where the largest path-metric will be at most p_i+ 2·6 and the smallest path-metric, according to the above formula, will be at least p_i and at most p_i+6, so an upperbound for the maximum difference *after 6 steps* is 12. Since PM(j) is non-descending in time (in other words: could remain p_i), at every step between the first and the sixth, 12+PM(j) gives an upperbound for the maximum path-metric in timeslot j+6, so then the maximum difference is still bounded by 12:

$$\forall_{j\geq 0, 0\leq i<2^m} PM_i(j+6) \leq PM(j)+12 \leq PM(j+6)+12,$$

and, for the first steps after initialization:

$$\forall_{0\leq j\leq 6. 0\leq i<2^m} PM_i(j) \leq 12$$

So we can conclude that if, at a given point in time, we decide to subtract the smallest path-metric p_i from all path-metrics, the resulting largest path-metric is at most 12. Also, if the largest path-metric is p_i +12, p_i >0, the smallest path-metric exceeds 0, therefore subtracting p_i from all path-metrics at this point is allowed to keep all values positive.

A suggestion about at which point to subtract the constant from all path-metrics can be made. Of course the smallest path-metric must be known, but this is no extra effort because we must know the according state to start the trace-back anyhow. Algorithmically the easiest way to subtract here is by flipping a single 1-bit at the same MSB-position in every path-metric, say at position n from the LSB. Therefore, the smallest path-metric may not be less than 2^n , and the largest path-metric may not exceed $2^{n+1}-1$.
Fortunately, we know the upperbound for the maximum difference between these extreme path-metrics (i.e. 12), so for the possible n we have the inequality:

$$2^n + 12 \leq 2^{n+1} - 1$$

which holds for a smallest n=4 (2^4+12=28 ≤ 2^5–1=31), so a 5-bit wide memory will be enough to store the path-metrics, where the simultaneous bit-flip can be done as soon as the smallest path-metric equals 16.

After calculation of all new path-metrics, a trace-back is performed. If necessary, this can be done while the next new path-metrics are calculated, and avoid addressing conflicts by, for example, memory-interleaving. However, it would be very tricky to have more than one trace-back running at the same time, because they both search through the same memory. At some input speeds, the trace-back could be too slow to be performed in one timeslot: to solve this, we could use one trace-back run to obtain more than 1 bit, because it is likely that this particular path would also be minimal for the next few trace-backs. To be on the save side, we could extend the path-history by the same number of bits that we obtain through one trace-back. Since the Viterbi-decoder output is delivered to a byte-wise de-Interleaver, gathering 8 bit per trace-back would then seem appropriate.

5. Conclusions

Throughout this paper, the Digital Terrestrial Television Baseline System for Forward Error Correction has revealed itself as a strong combination of coding techniques, to achieve an excellent performance. Unfortunately, the DVB group that proposed the standard to the ETSI does not reveal how they came to this particular choice of encoding. In spite of this our goal, development of a functional model for the Forward Error Correction unit in the receiver, is reached. This model has been implemented in the C programming language, and most of the necessary performance tests have been done.

Concluding evaluation of the standard

Advantages of digital over analog terrestrial transmission include:

- Quasi Error Free (QEF) system performance, which can be reached for bad channel characteristics;
- Frequency efficiency: more channels can be transmitted using the same bandwidth, where the required bandwidth for a channel is adjustable to the amount of information at the transmitter;
- Other services (e.g. interactive ones) are possible, yet are still under investigation of the Digital Video Broadcasting (DVB) group;
- Transmission by Single Frequency Networks is possible, which maximizes frequency efficiency;
- The Outer Coding and Outer Interleaving, used by the Satellite [9] and Cable [10] systems are common to Terrestrial, and the Inner Coding of the Satellite system is common as well. This facilitates integration of these systems in one physical decoder.

Disadvantages:

- Switching between channels causes a considerable waiting-time or pause, due to re-initialization of the decoding pipeline for FEC and MPEG-2, and synchronization of the analogue front-end. Though less severe, this is also the case for switching between layers in hierarchical transmission;
- Early demultiplexing in the receiving tuner causes that only one stream is processed by the FEC-decoder, and as a consequence the decoder back-end can deliver only one programme at a time. This implies that e.g. simultaneous watching and (analogue) recording two different programmes can only be done when the decoder is carried out two times;
- There is need for a serious amount of memory, estimation in the order of 10 to 20 Kbytes;
- High-speed computations are required, which makes parallel implementations of for example the Reed-Solomon Decoder necessary. As a result the decoder will occupy a larger area on the IC. The need for more hardware implemented algorithms also makes the decoder less flexible with regard to modification to eventual future expansions of the standard.

Achieved results

Most of the work was demanded by the Reed-Solomon Decoder and the Viterbi Decoder.

- The **RS Decoder** concept is not as transparent as we'd like it to be, yet the performance can be pinpointed very neatly. Two decoding algorithms, i.e. the Berlekamp-Massey algorithm and the Euclidean algorithm, have been simulated in C and globally tested. In literature, the emphasis is put on RS decoders which are able to cope with variable code lengths, generator polynomials, and the order of the Galois Field. Because of the speed by which the computations must be performed (up to 150 million Galois Field multiplications per second; decoding speed at least 47 μs per received word), we must restrict ourselves to a more hardware-based solution, thus less flexible, and rely on the consistency of the ETSI standard. Though performance can be improved by using erasure information, this is too time and space consuming, and is not really necessary to obtain the required performance. Several parts in the RS decoding process have been examined for hardware implementations, except for the main algorithm, which was not feasible at this point.
- The **Viterbi algorithm** for decoding convolutional codes is conceptually easier, but no tight theoretical performance bounds can be given and had to be obtained by functional simulations. These simulations reveal the need for soft-decision decoding, in order to achieve the desired performance: testing with uniform quantization shows that for QPSK and 64-QAM three bits of soft-decision information suffices per received bit, whereas for 16-QAM we will be needing four soft-decision bits. Further, we will need a decoding window width W, depending on the used code rate and type of modulation, of $60 \leq W \leq 140$ bit.
- Of practical interest are the different properties of the decoding stages and how this relates to resource needs. Here follows an estimation of the amount of memory storage needed, the minimal delay and an indication of the required computation capacity needed, per decoding stage. Only significant and/or relevant figures are given.

Table 5-I: ***Properties of decoding stages.***

Decoding Stage	Memory storage	Delay	Computation Capacity
Descrambler	–	–	–
RS Decoder	≥ 236 bytes	0.30 ms	150 million GF multiplications / s
Forney De-Interleaver	1.122 Kbytes	3.32 ms	–
Viterbi Decoder (W=140)	≥ 1.200 Kbytes	0.03 ms	550 million butterflies / s
Inner De-Interleaver (*sd*=3)	18.522 Kbytes	1.14 ms	–
FEC Decoder Total	≥ 21.080 Kbytes	4.79 ms	–

This table shows that most memory is needed by the Inner De-Interleaver. In fact, this amount is a linear function of the number of soft decision bits *sd*. The RS Decoder needs to do a great deal of Galois Field multiplications, so we will need small and fast multipliers, and an error locating algorithm that can work in parallel. The Viterbi Decoder must perform 4 additions, 2 comparisons, 2 pathmetric value storage's and 2 survivor bit storage's per butterfly. The huge number of butterflies per second indicates that an efficient memory connection and storage is required, and that some butterflies must operate in parallel.
These figures show, to which direction the hardware implementation of the FEC Decoder will lead.

Future investigations

As we only made a functional model of the FEC Decoder, and performed tests to gain insight in the Viterbi algorithm, we must further investigate the hardware possibilities that would/could contribute to a single-chip implementation of the entire FEC Decoder. This bold ambition requests a minute use of area, while on the other hand we must still reach the required performance. Secondly, the FEC decoder will be part of a settop-box which involves additional requirements. For these reasons we must do some more research:

- The **quantization method** in the demodulator as assumed in the functional model uses uniform quantization levels. Another possibility would be to use *non-uniform quantization*, e.g. a companding-like method (*com*pression and ex*panding*). The idea is that more important areas have a finer quantization scale than less important areas, which could reduce the number of soft-decision bits needed for the likelihood. We can even go a step further by *dynamically altering* the quantization levels, depending on the channel quality at that moment, to get a tailor-made likelihood.
- We can reduce area and delay by **shifting decoding stages into each other** (e.g. de-randomization during the delay in the RS decoder).
- We can speed-up the trace-back in Viterbi decoder by separate storing the last trace-back path: it will be likely that the next trace-back path will coincide with this path in an early stage, so after comparison we will not have to do the whole trace-back again.
- We can reduce channel switching delays by using **presetted channel information** (e.g. code rate and type of modulation) to get a lock sooner than with trial-and-error.
- To minimize area when **supporting satellite and cable reception**, we must make sure the same resources (i.e. corresponding decoding stages) can be used. The satellite standard uses convolutional coding as well, but in the decoder needs self-synchronization and a way to cope with pi-phase ambiguity. In the cable standard, besides 16-QAM and 64-QAM, 32-QAM is used as well. Therefore, these subjects need further study.
- As a feature, we can deduce a channel quality estimation in the Viterbi decoder, by monitoring the metrics-progression. This is especially useful for aiming at satellites in the satellite standard.

References

1. Berlekamp, E.R.
ALGEBRAIC CODING THEORY.
New York, NY: McGraw-Hill, 1968, p. 466.

2. Berlekamp, E.R.
BIT-SERIAL REED-SOLOMON ENCODERS.
IEEE Tr. on Inf. Theory, Vol. 28 (1982), NO. 6, p. 869-874.

3. Berlekamp, E.R., G. Seroussi, and Po Tong.
A HYPERSYSTOLIC REED-SOLOMON DECODER.
In: Wicker, S.B. and V.K. Bhargava
REED-SOLOMON CODES AND THEIR APPLICATIONS.
Ch. 10, p. 205-241.
Piscataway, NJ: IEEE Press, 1994, p. 322.

4. Blahut, R.E.
THEORY AND PRACTICE OF ERROR CONTROL CODES.
Reading, Mass.: Addison-Wesley, 1983, p. 500.

5. Chen, J., and P. Owsley
A BURST-ERROR-CORRECTING ALGORITHM FOR REED-SOLOMON CODES.
IEEE Tr. on Inf. Theory, Vol. 38 (1992), NO. 6, p. 1807-1812.

6. Cideciyan, R.D., and E. Eleftheriou
CONCATENATED REED-SOLOMON/CONVOLUTIONAL CODING SCHEME FOR DATA TRANSMISION IN CDMA CELLULAR SYSTEMS.
IEEE 44th Vehicular Technology Conference (Cat. No. 94CH3438-9), Vol. 2, p. 1369-1373.
Conf. Date: 8-10 June 1994.

7. Cooper III, A.B.
SOFT DECISION DECODING OF REED-SOLOMON CODES.
In: Wicker, S.B. and V.K. Bhargava
REED-SOLOMON CODES AND THEIR APPLICATIONS.
Ch. 6, p. 108-124
Piscataway, NJ: IEEE Press, 1994, p. 322.

8. Dabiri, D., and I.F. Blake
FAST PARALLEL ALGORITHMS FOR DECODING REED-SOLOMON CODES BASED ON REMAINDER POLYNOMIALS.
IEEE Tr. on Inf. Theory, Vol. 41 (1995), NO. 4, p. 873-885.

9. ETS 300421 STANDARD: DIGITAL BROADCASTING SYSTEMS FOR TELEVISION, SOUND AND DATA SERVICES; FRAMING STRUCTURE, CHANNEL CODING AND MODULATION FOR 11-12 GHZ SATELLITE SERVICES. (REFERENCE: DE/JTC-DVB-6)
Route des Lucioles - Sophia Antipolis - Valbonne - France: European Telecommunications Standards Institute (ETSI), 1994, p. 27.

10. ETS 300429 STANDARD: DIGITAL BROADCASTING SYSTEMS FOR TELEVISION, SOUND AND DATA SERVICES; FRAMING STRUCTURE, CHANNEL CODING AND MODULATION FOR CABLE SYSTEMS. (REFERENCE: DE/JTC-DVB-7)
France: ETSI, 1994, p. 21.

11. DRAFT PREPRINT ETS 300744, VER. 0.0.3: DIGITAL BROADCASTING SYSTEMS FOR TELEVISION, SOUND AND DATA SERVICES; FRAMING STRUCTURE, CHANNEL CODING AND MODULATION FOR DIGITAL TERRESTRIAL TELEVISION. (REFERENCE: DE/JTC-DVB-8)
France: ETSI, 1996, p. 41.

12. ETSI DRAFT SPECIFICATION FOR DIGITAL TERRESTRIAL TELEVISION. (REFERENCE: TM 1545, REV. 2)
France: ETSI, 1996, p. 42.

13. Forney Jr., G.D.
BURST-CORRECTING CODES FOR THE CLASSIC BURSTY CHANNEL.
IEEE Tr. on Comm. Techn., Vol. Com-19 (1971), NO. 5, p. 772-781.

14. Ghosh, M.
ERROR CORRECTING SCHEMES FOR DIGITAL TELEVISION BROADCASTING.
IEEE Tr. on Cons. El., Vol. 41 (1995), NO. 3, p. 400-404.

15. Hagenauer, J., E. Offer, and L. Papke
MATCHING VITERBI DECODERS AND REED-SOLOMON DECODERS IN A CONCATENATED SYSTEM.
In: Wicker, S.B. and V.K. Bhargava
REED-SOLOMON CODES AND THEIR APPLICATIONS.
Ch. 11, p.242-271
Piscataway, NJ: IEEE Press, 1994, p. 322.

16. Hasan, M.A., and V.K. Bhargava
BIT-SERIAL SYSTOLIC DIVIDER AND MULTIPLIER FOR FINITE FIELDS GF(2^M).
IEEE Tr. on Comp., Vol. 41 (1992), NO. 8, p. 972-980.

17. Hasan, A., V.K. Bhargava, and T. Le-Ngoc
ALGORITHMS AND ARCHITECTURES FOR THE DESIGN OF A VLSI REED-SOLOMON CODEC.
In: Wicker, S.B. and V.K. Bhargava
REED-SOLOMON CODES AND THEIR APPLICATIONS.
Ch. 5, p. 60-107.
Piscataway, NJ: IEEE Press, 1994, p. 322.

18. Hoffman, D.G., D.A. Leonard, C.C. Lindner, K.T. Phelps, C.A. Rodger, and J.R. Wall
CODING THEORY, THE ESSENTIALS.
New York: MARCEL DEKKER, INC., 1992, p. 277.

19. Hooijen, O.G.
IMPLEMENTING A FAST 16-BIT REED-SOLOMON DECODER IN A DIGITAL SIGNAL PROCESSOR.
Master Thesis.
Eindhoven: Technische Universiteit Eindhoven, Faculteit der Electrotechniek, 1993, p. 69.

20. Keiichi Iwamura, Yasunori Dohi, and Hideki Imai
A DESIGN OF REED-SOLOMON DECODER WITH SYSTOLIC-ARRAY STRUCTURE.
IEEE Tr. on Comp., Vol. 44 (1995), NO. 1, p. 118-122.

21. Kuang Yung Liu
ARCHITECTURE FOR VLSI DESIGN OF REED-SOLOMON DECODERS.
IEEE Tr. on Comp., Vol. C-33 (1984), NO. 2, p. 178-189.

22. Kuang Yung Liu, and Jun-Ji Lee
RECENT RESULTS ON THE USE OF CONCATENATED REED-SOLOMON/VITERBI CHANNEL CODING AND DATA COMPRESSION FOR SPACE COMMUNICATIONS.
IEEE Tr. on Comm., Vol. Com-32 (1984), NO. 5, p. 518-523.

23. Masakatu Morii, and Masao Kasahara
GENERALIZED KEY-EQUATION OF REMAINDER DECODING ALGORITHM FOR REED-SOLOMON CODES.
IEEE Tr. on Inf. Theory, Vol. 38 (1992), NO. 6, p. 1801-1807.

24. McEliece, R.J., and L. Swanson
ON THE DECODER ERROR PROBABILITY FOR REED-SOLOMON CODES.
IEEE Tr. on Inf. Theory, Vol. IT-32 (1986), NO. 5, p. 701-703.

25. Mester, R.
FEHLERKORREKTUR IN HOCHGESCHWINDIGKEIT, 1. TEIL, CHIPSATZ FUER DEN REED-SOLOMON-CODE: 160 MBIT/S KEIN PROBLEM.
Elektonik, NO. 25 (1992), p. 46-52.

26. Mester, R.
FEHLERKORREKTUR IN HOCHGESCHWINDIGKEIT, 2. TEIL, CHIPSATZ FUER DEN REED-SOLOMON-CODE: 160 MBIT/S KEIN PROBLEM.
Elektonik, NO. 26 (1992), p. 40-44.

27. Michelson, A.M., and A.H. Levesque
ERROR-CONTROL TECHNIQUES FOR DIGITAL COMMUNICATION.
New York: John Wiley & Sons , 1985, p. 463.

28. Moon Ho Lee, Seung Bae Choi, and Jin Su Chang
A HIGH SPEED REED-SOLOMON DECODER.
IEEE Tr. on Cons. El., Vol. 41 (1995), NO. 4, p. 1142-1149.

29. O'Reilly, J.J., Y.Bian, A. Popplewell, S. Fragiacomo, and R. Blake
FORWARD ERROR CONTROL FOR FUTURE TRANS-OCEANIC OPTICAL SYSTEMS.
Fifth IEE Conference on 'Telecommunications' (Conf. Publ. No. 404), p. 78-82.
Conf. Date: 26-29 March 1995.

30. Poli, A., and L. Huguet
ERROR CORRECTING CODES, THEORY AND APPLICATIONS.
Translated by I. Craig from: *Codes Correcteurs.* Paris, France: Masson, 1989.
Hertfordshire, UK: Prentice Hall International, 1992, p. 512.

31. Reed, I.S., and G. Solomon
REED-SOLOMON CODES: A HISTORICAL OVERVIEW.
In: Wicker, S.B. and V.K. Bhargava
REED-SOLOMON CODES AND THEIR APPLICATIONS.
Ch. 2, p. 17-24.
Piscataway, NJ: IEEE Press, 1994, p. 322.

32. Shao, H.M., and I.S. Reed
ON THE VLSI DESIGN OF A PIPELINE REED-SOLOMON DECODER USING SYSTOLIC ARRAYS.
IEEE Tr. on Comp., Vol. 37 (1988), NO. 10, p. 1273-1280.

33. Shao, H.M., T.K. Truong, L.J. Deutsch, J.H. Yuen, I.S. Reed
A VLSI DESIGN OF A PIPELINE REED-SOLOMON DECODER.
IEEE Tr. on Comp., Vol. C-34 (1985), NO. 5, p. 393-403.

34. Shyue-Win Wei, and Che-Ho Wei
HIGH-SPEED DECODER OF REED-SOLOMON CODES.
IEEE Tr. on Comm., Vol. 41 (1993), NO. 11, p. 1588-1593.

35. Sorger, U.K.
A NEW REED-SOLOMON CODE DECODING ALGORITHM BASED ON NEWTON'S INTERPOLATION.
IEEE Tr. on Inf. Theory, Vol. 39 (1993), NO. 2, p. 358-365.

36. Taipale, D.J., and M.J. Seo
AN EFFICIENT SOFT-DECISION REED-SOLOMON DECODING ALGORITHM.
IEEE Tr. on Inf. Theory, Vol. 40 (1994), NO. 4, p. 1130-1139.

37. Truong, T.K., W.L. Eastman, I.S. Reed, and I.S. Hsu
SIMPLIFIED PROCEDURE FOR CORRECTING BOTH ERRORS AND ERASURES OF REED-SOLOMON CODE USING EUCLIDEAN ALGORITHM.
IEE Proc., Vol. 135, Pt. E (1988), NO. 6, p. 318-324.

38. Whitaker, S.R., J.A. Canaris, and K.B. Cameron
REED SOLOMON VLSI CODEC FOR ADVANCED TELEVISION.
IEEE Tr. on Circ. and Syst. for Video Techn., Vol. 1 (1991), NO. 2, p. 230-236.

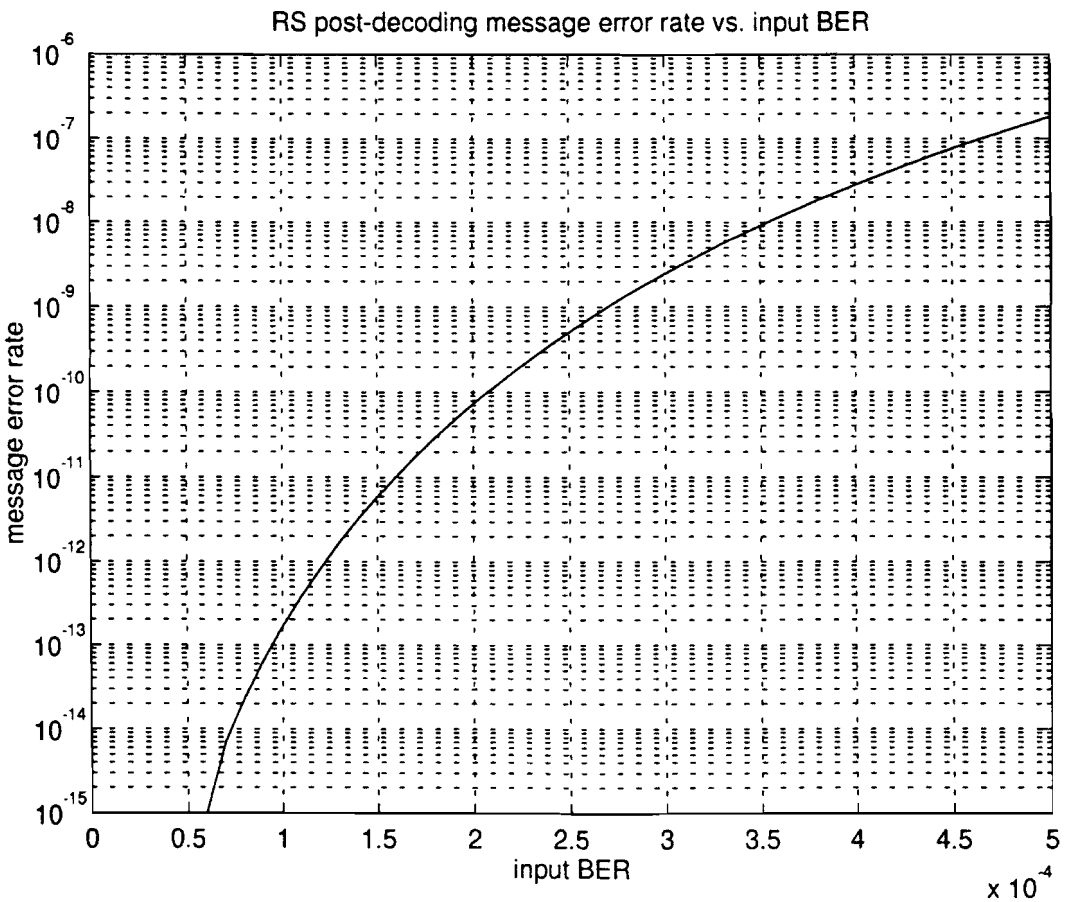
39. Wicker, S.B. and V.K. Bhargava
REED-SOLOMON CODES AND THEIR APPLICATIONS.
Piscataway, NJ: IEEE Press, 1994, p. 322.

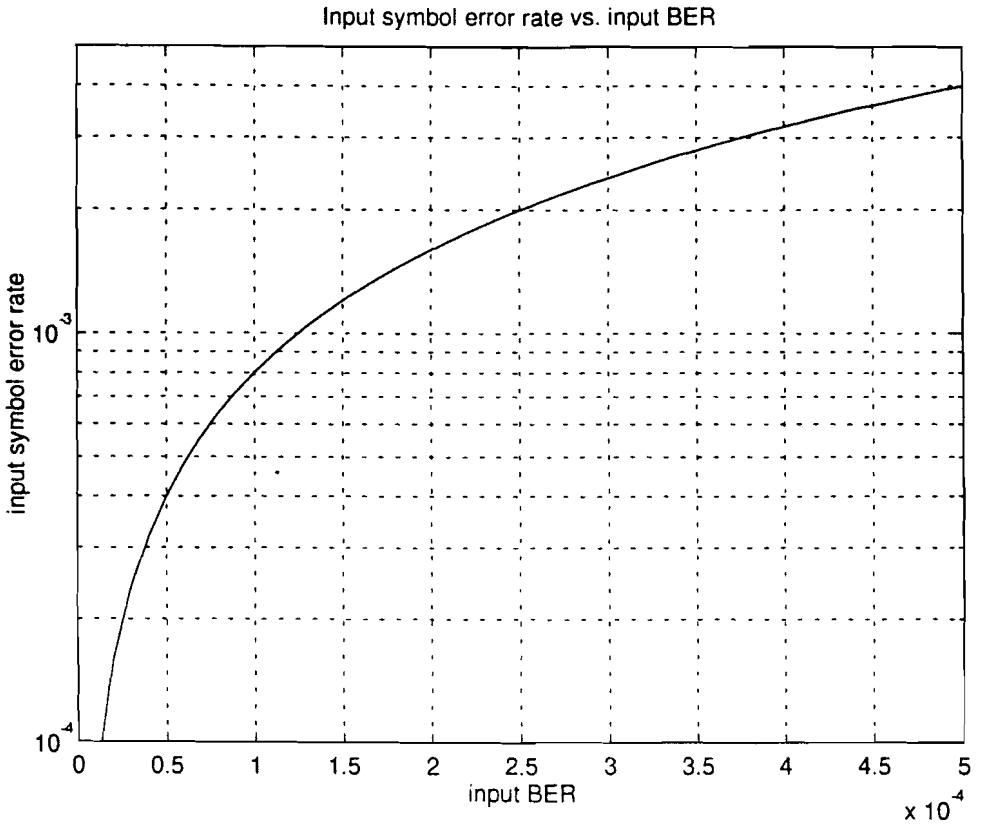
40. Wicker, S.B. and V.K. Bhargava
AN INTRODUCTION TO REED-SOLOMON CODES.
In: Wicker, S.B. and V.K. Bhargava
REED-SOLOMON CODES AND THEIR APPLICATIONS.
Ch. 1, p. 1-16.
Piscataway, NJ: IEEE Press, 1994, p. 322.

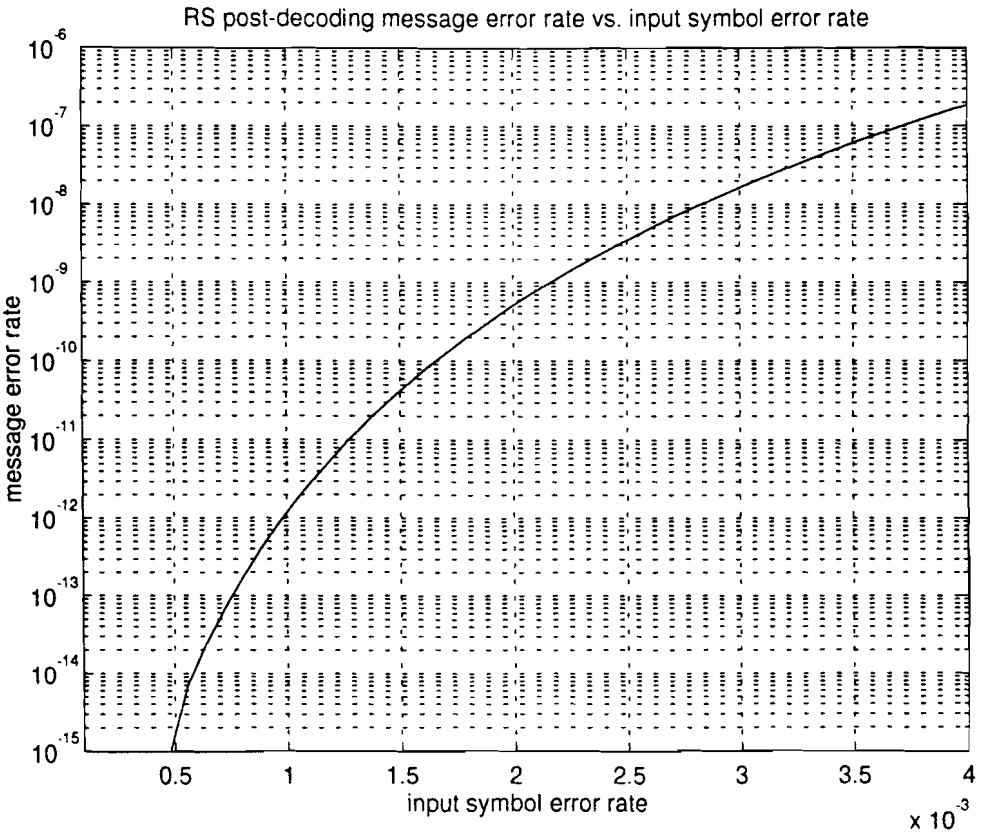
41. Zhou, B.B.
A NEW BIT-SERIAL SYSTOLIC MULTIPLIER OVER GF(2^M).
IEEE Tr. on Comp., Vol. 37 (1988), NO. 6, p. 749-751.

42. Webb, W.T. and L. Hanzo
MODERN QUADRATURE AMPLITUDE MODULATION, PRINCIPLES AND APPLICATIONS FOR FIXED AND WIRELESS CHANNELS.
New York, NY: IEEE Press, 1994, p. 557.

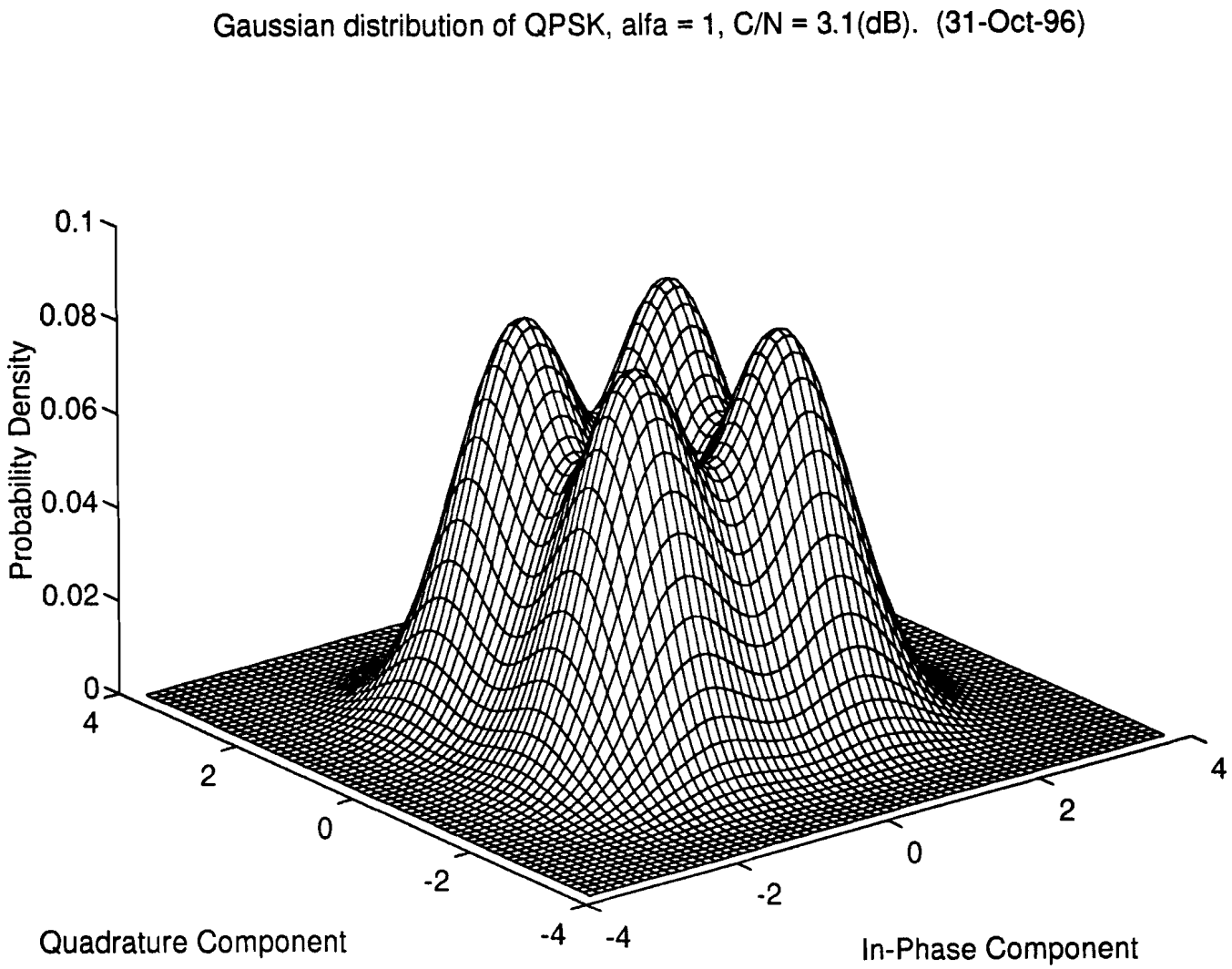
Appendix A: Theoretical Performance of Reed-Solomon Code RS(204,188,8)



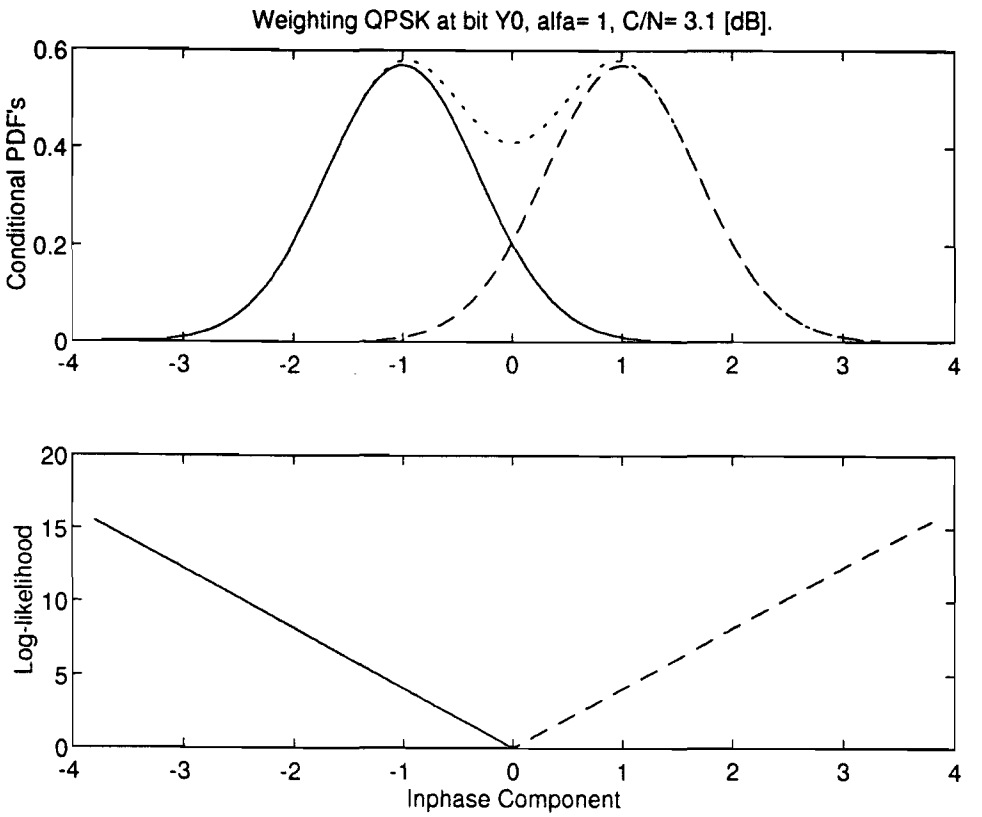

Input symbol error rate vs. input BER
input symbol error rate
10^{-3}
10^{-4}
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
input BER
x 10^{-4}

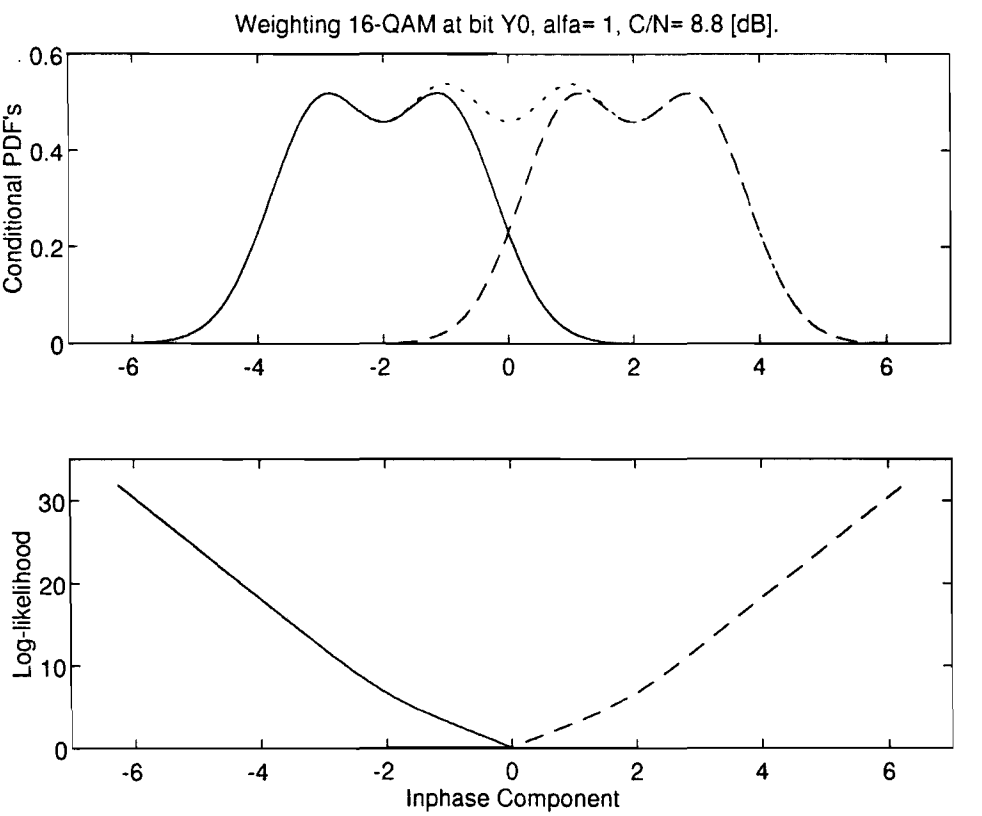

RS post-decoding message error rate vs. input symbol error rate
message error rate
10^{-6}
10^{-7}
10^{-8}
10^{-9}
10^{-10}
10^{-11}
10^{-12}
10^{-13}
10^{-14}
10^{-15}
0.5
1
1.5
2
2.5
3
3.5
4
input symbol error rate
x 10^{-3}

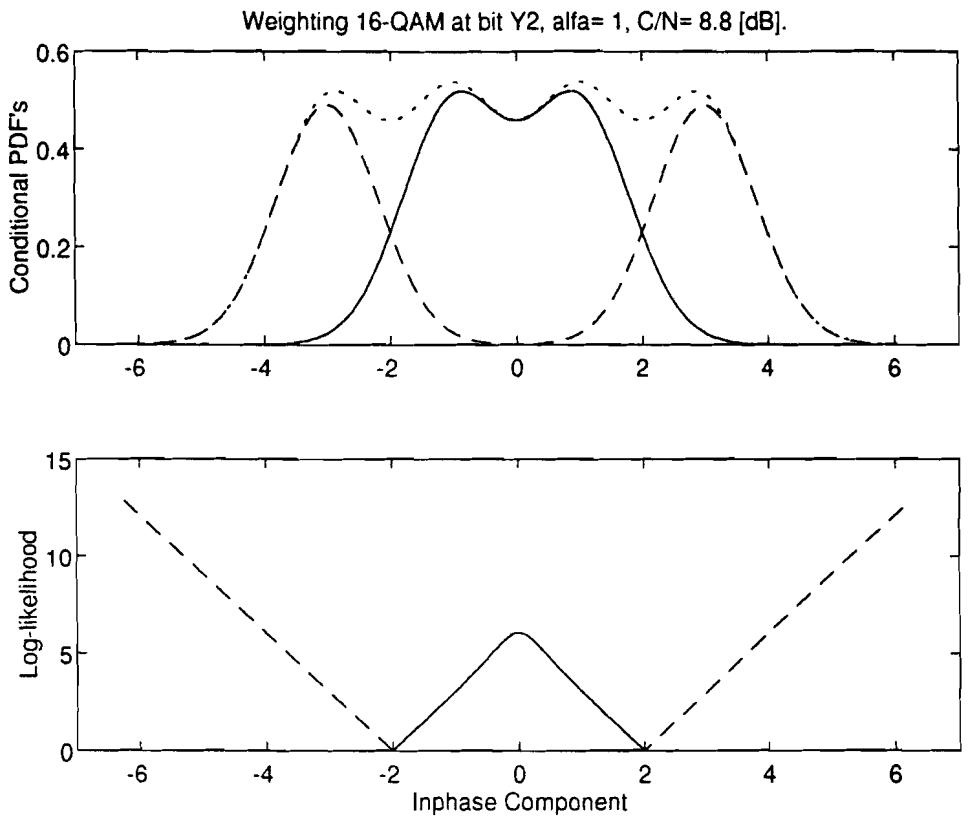
Appendix B: Overall PDF of QPSK

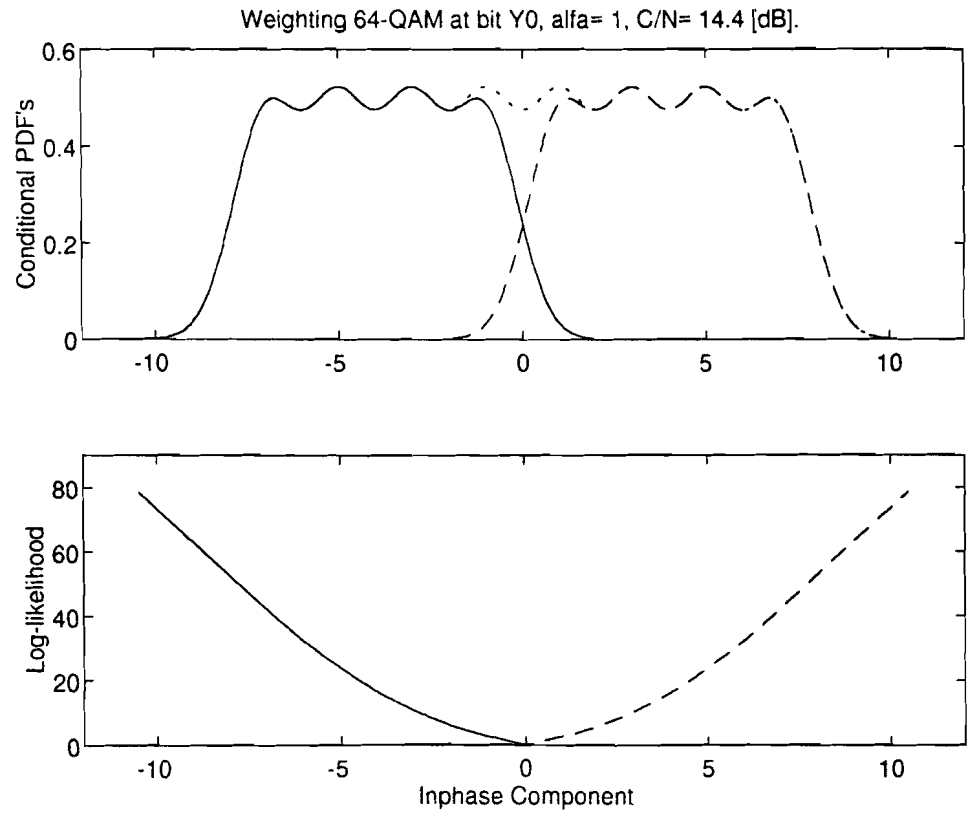


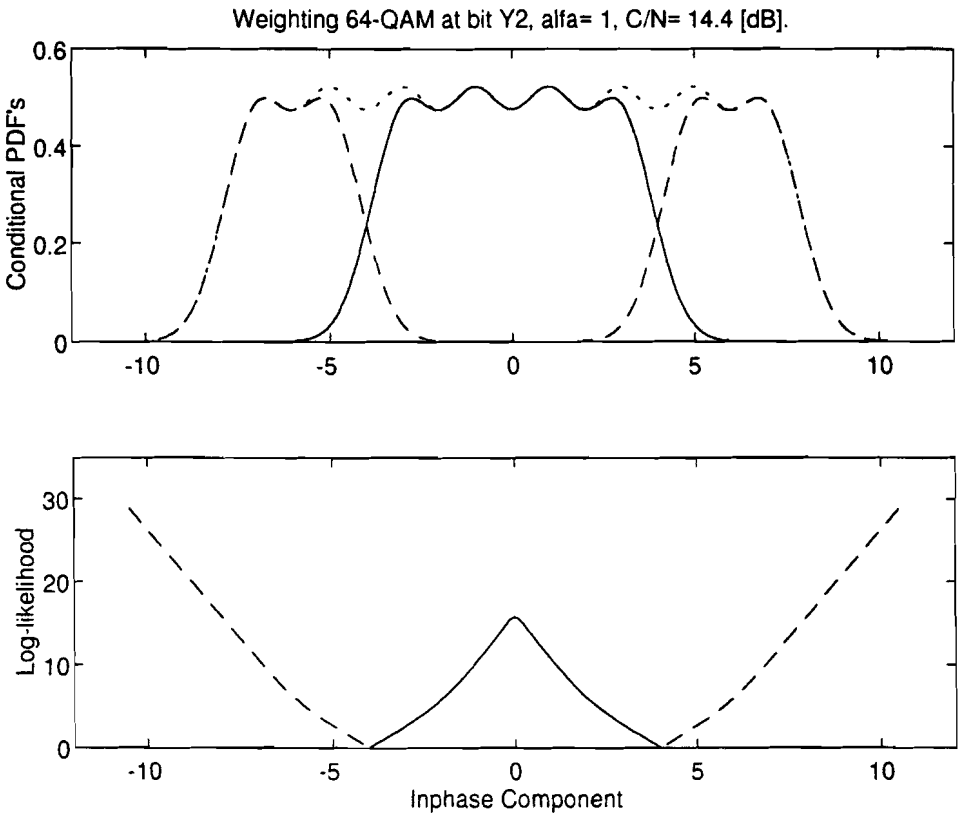
Appendix C: Log-likelihoods for non-hierarchical transmission schemes

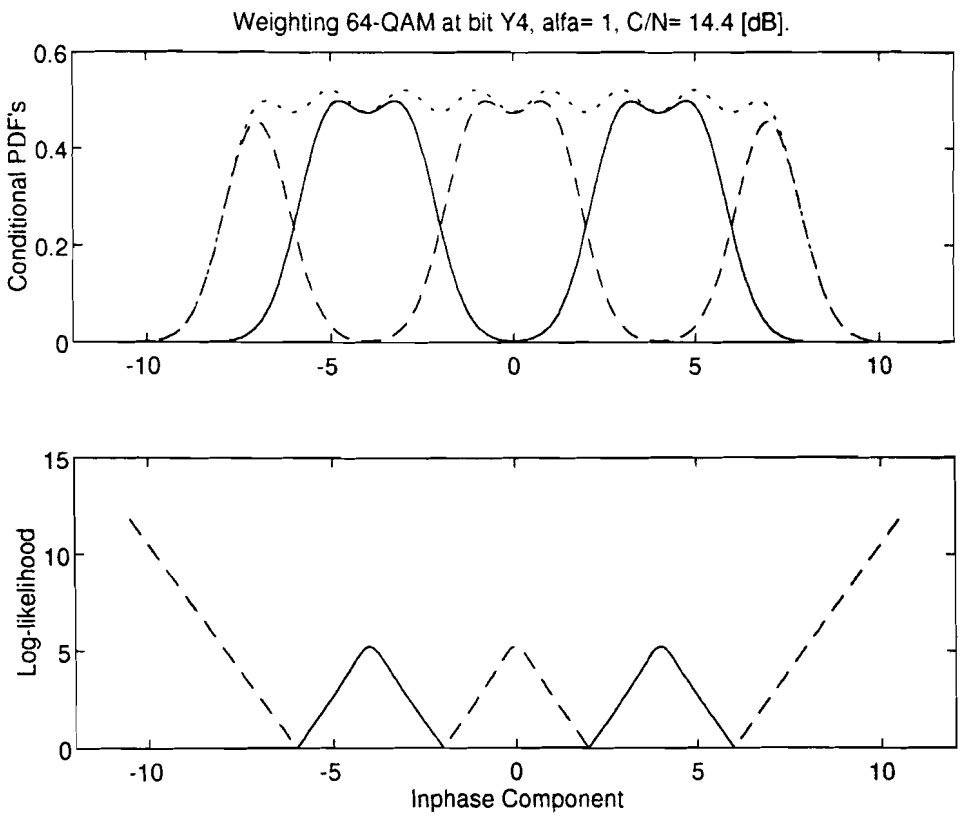




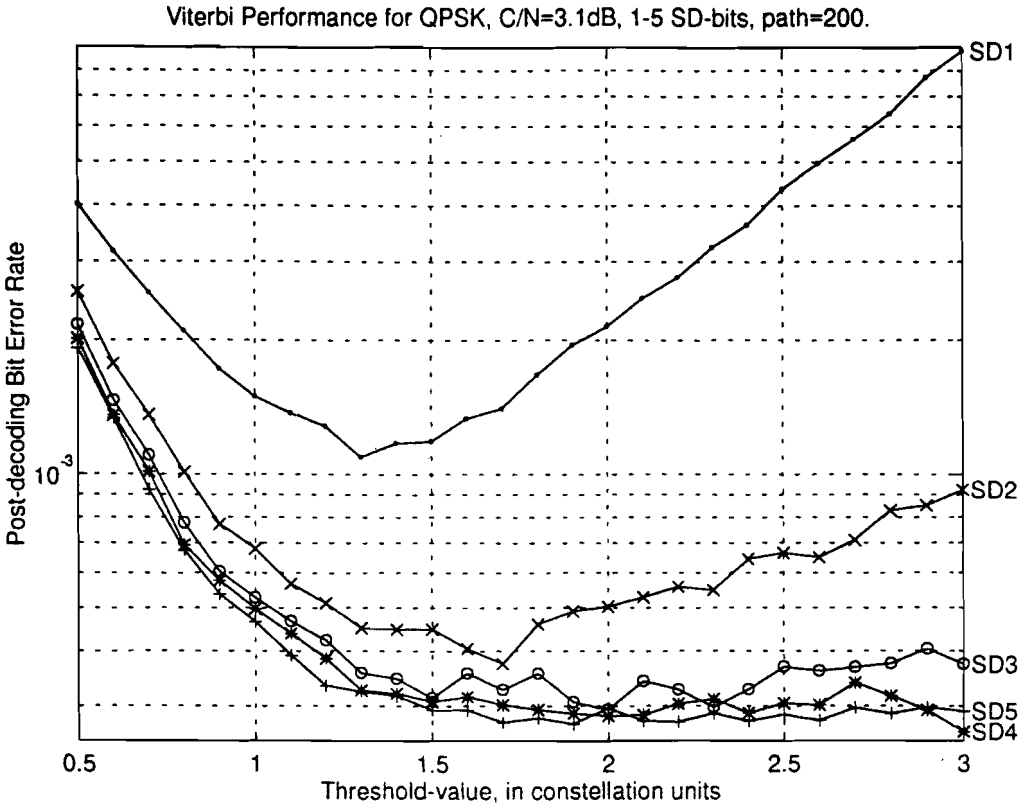

Weighting 16-QAM at bit Y2, alfa= 1, C/N= 8.8 [dB].
Conditional PDF's
0.6
0.4
0.2
0
-6
-4
-2
0
2
4
6
Log-likelihood
15
10
5
0
-6
-4
-2
0
2
4
6
Inphase Component

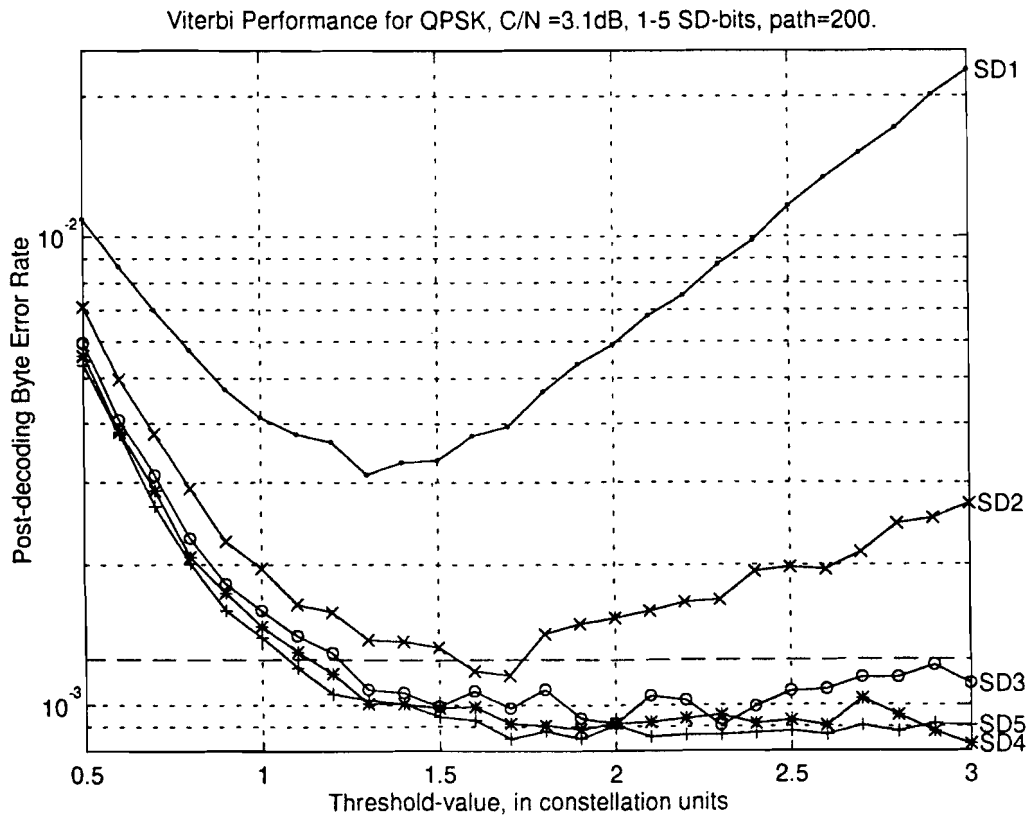

Weighting 64-QAM at bit Y0, alfa= 1, C/N= 14.4 [dB].
Conditional PDF's
0.6
0.4
0.2
0
-10
-5
0
5
10
Log-likelihood
80
60
40
20
0
-10
-5
0
5
10
Inphase Component

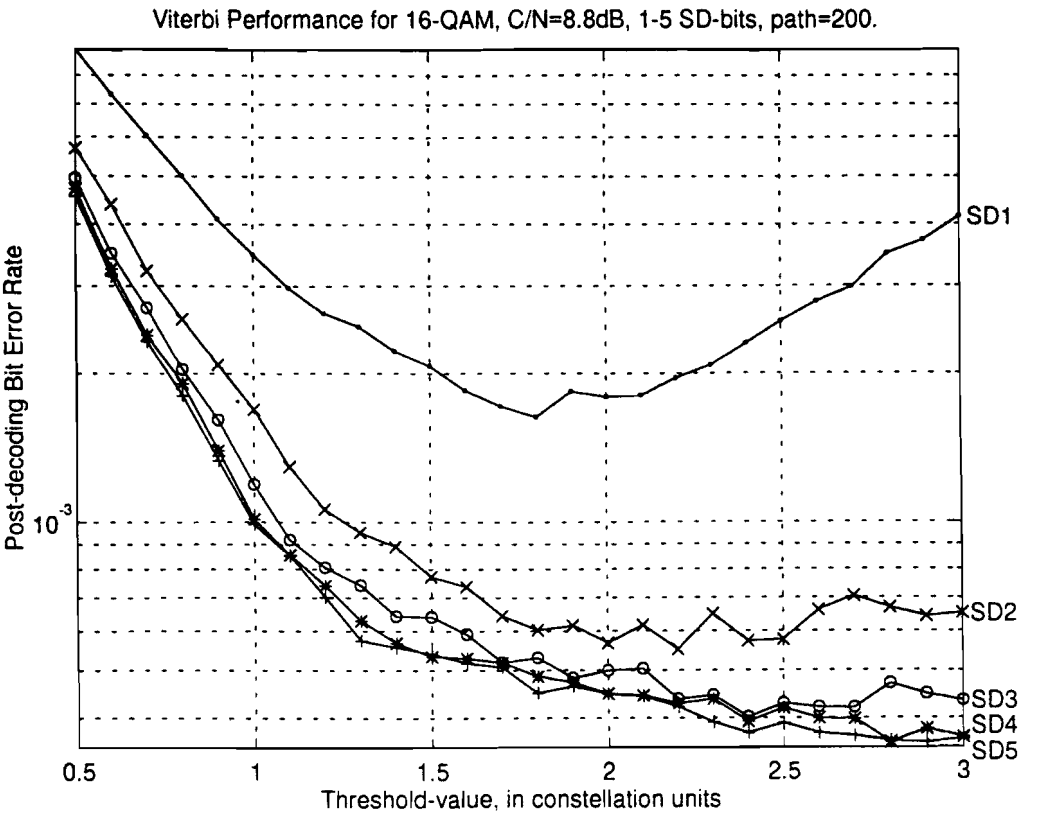

Weighting 64-QAM at bit Y2, alfa= 1, C/N= 14.4 [dB].
Conditional PDF's
0.6
0.4
0.2
0
-10
-5
0
5
10
Log-likelihood
30
20
10
0
-10
-5
0
5
10
Inphase Component

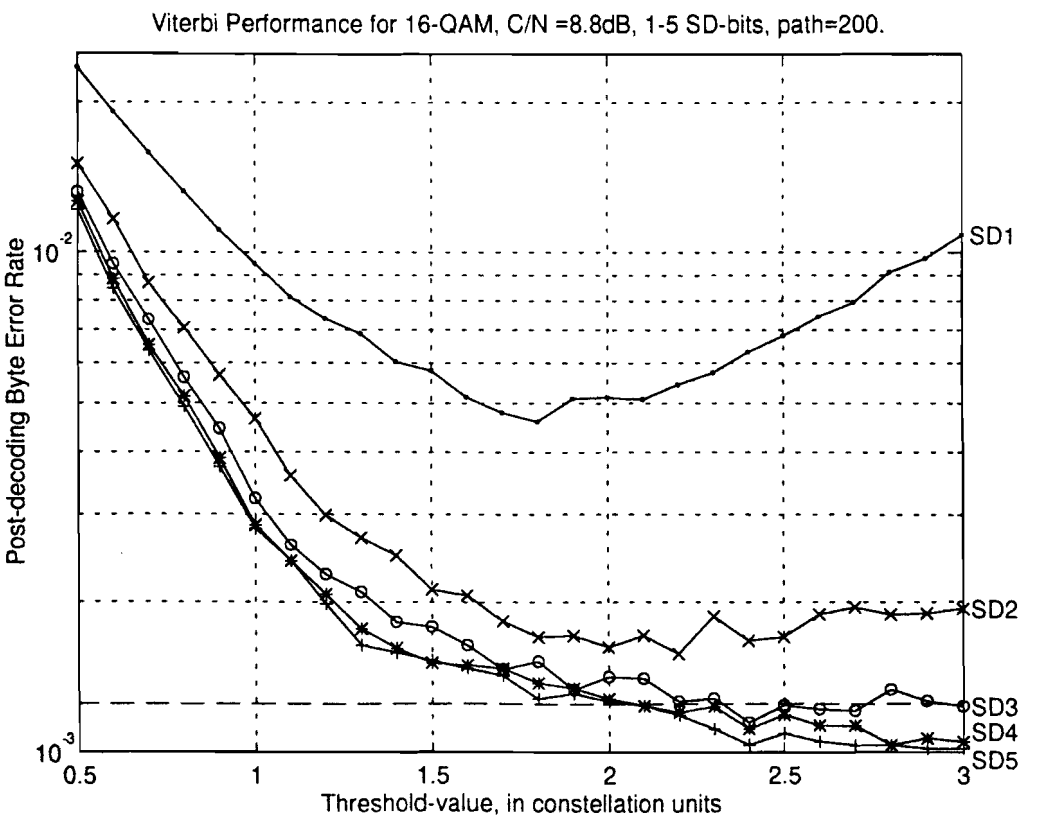

Weighting 64-QAM at bit Y4, alfa= 1, C/N= 14.4 [dB].
Conditional PDF's
0.6
0.4
0.2
0
-10
-5
0
5
10
Log-likelihood
15
10
5
0
-10
-5
0
5
10
Inphase Component

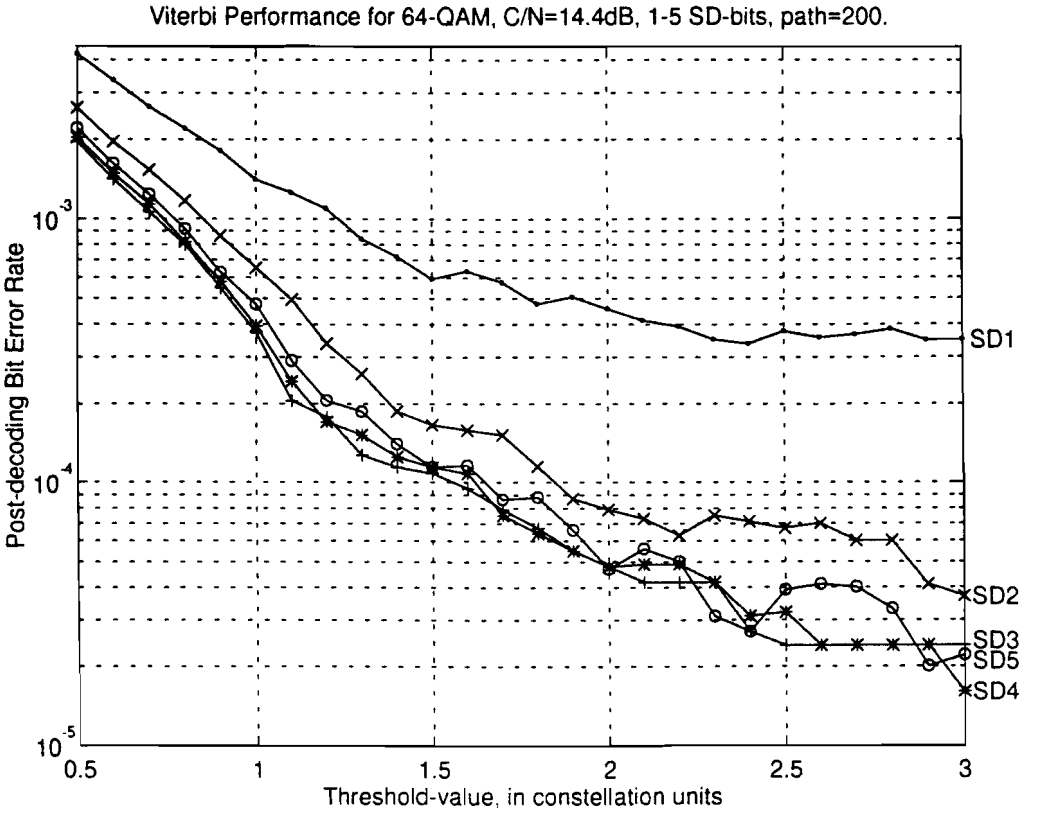
Appendix D: Test-results for Soft-Decision Parameters

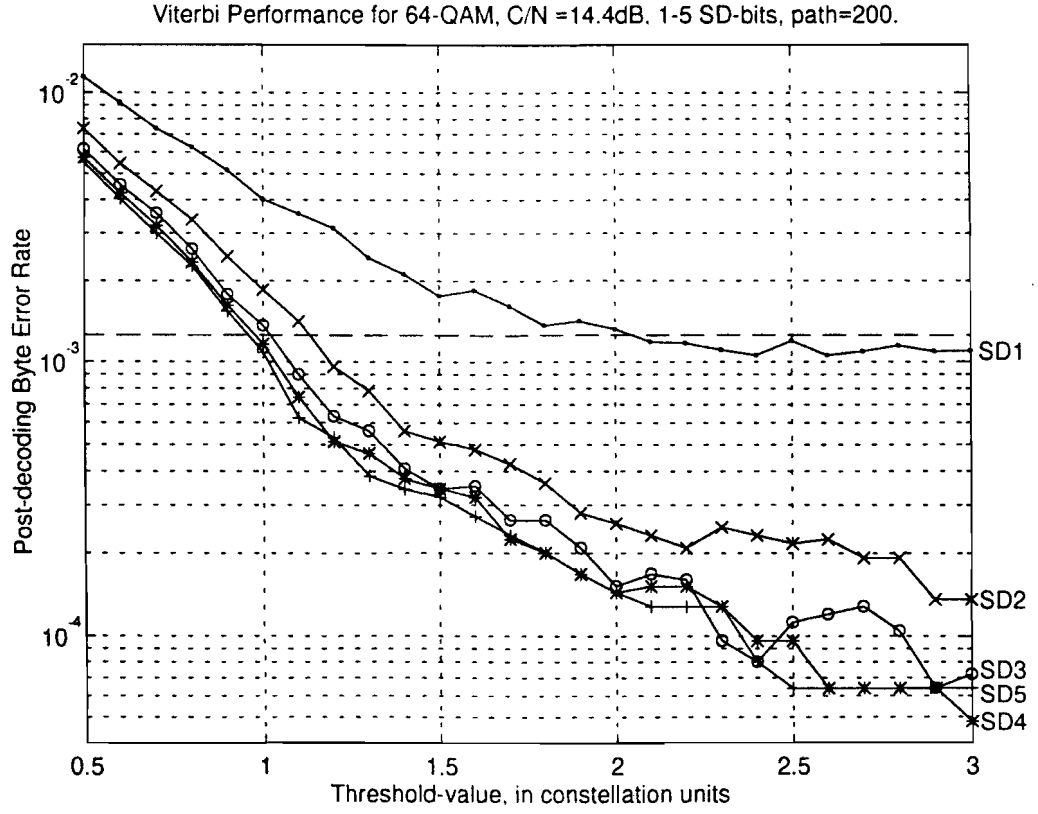




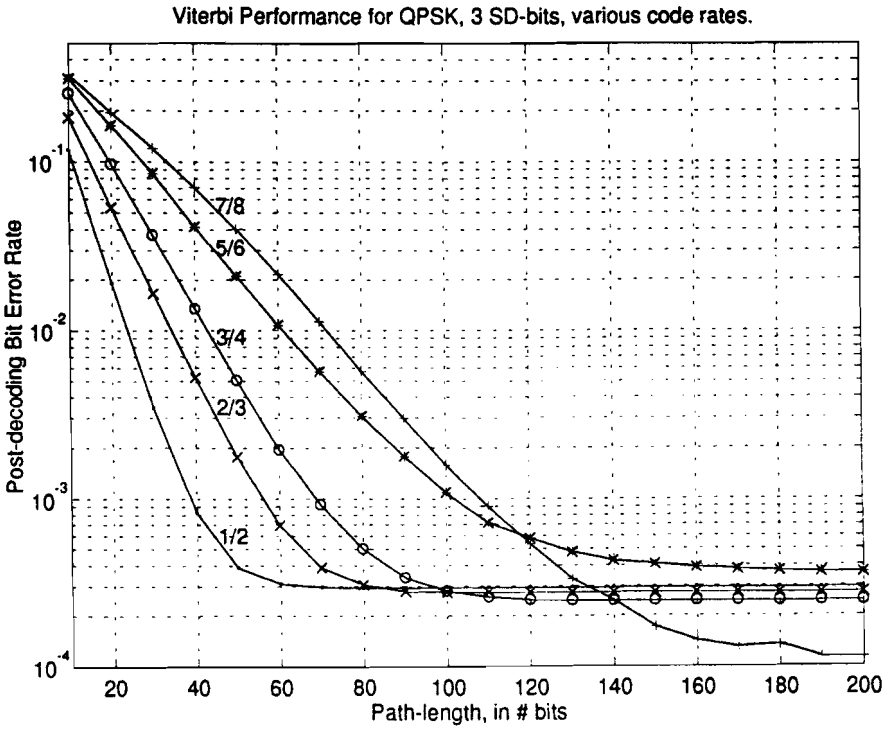

Viterbi Performance for 16-QAM, C/N=8.8dB, 1-5 SD-bits, path=200.
Post-decoding Bit Error Rate
10-3
SD1
SD2
SD3
SD4
SD5
0.5
1
1.5
2
2.5
3
Threshold-value, in constellation units

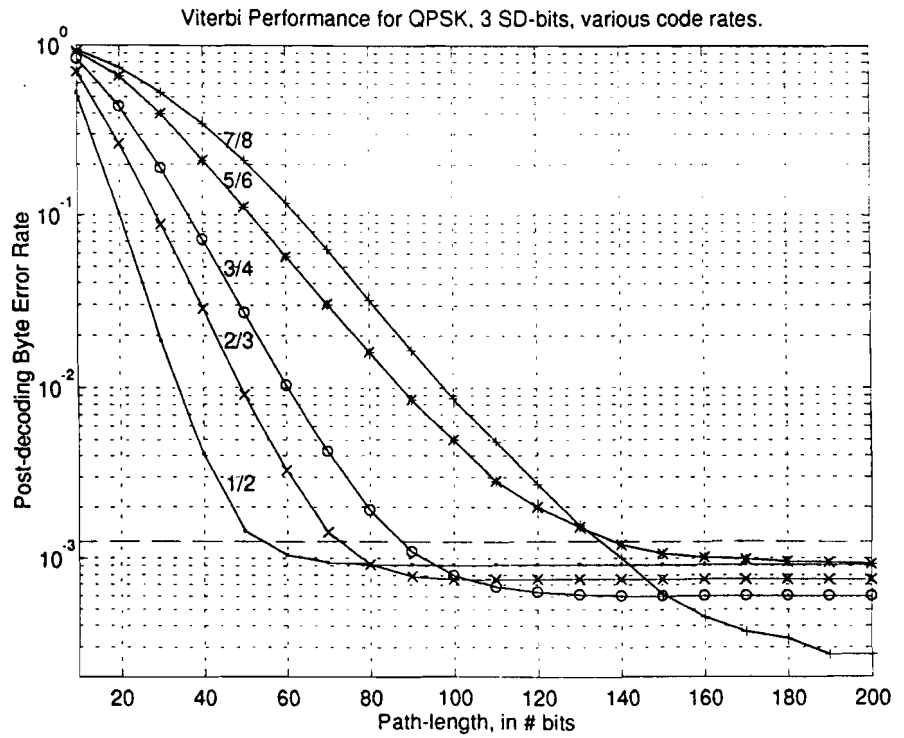

Viterbi Performance for 16-QAM, C/N =8.8dB, 1-5 SD-bits, path=200.
Post-decoding Byte Error Rate
10-2
10-3
SD1
SD2
SD3
SD4
SD5
0.5
1
1.5
2
2.5
3
Threshold-value, in constellation units

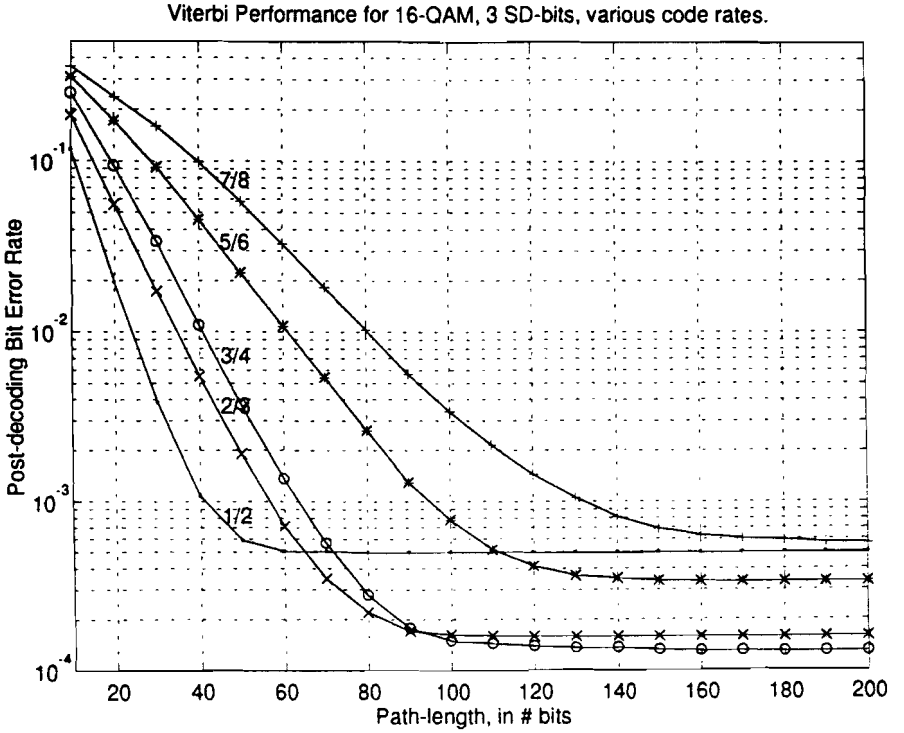

Viterbi Performance for 64-QAM, C/N=14.4dB, 1-5 SD-bits, path=200.
Post-decoding Bit Error Rate
10^-3
10^-4
10^-5
0.5
1
1.5
2
2.5
3
Threshold-value, in constellation units
SD1
SD2
SD3
SD5
SD4

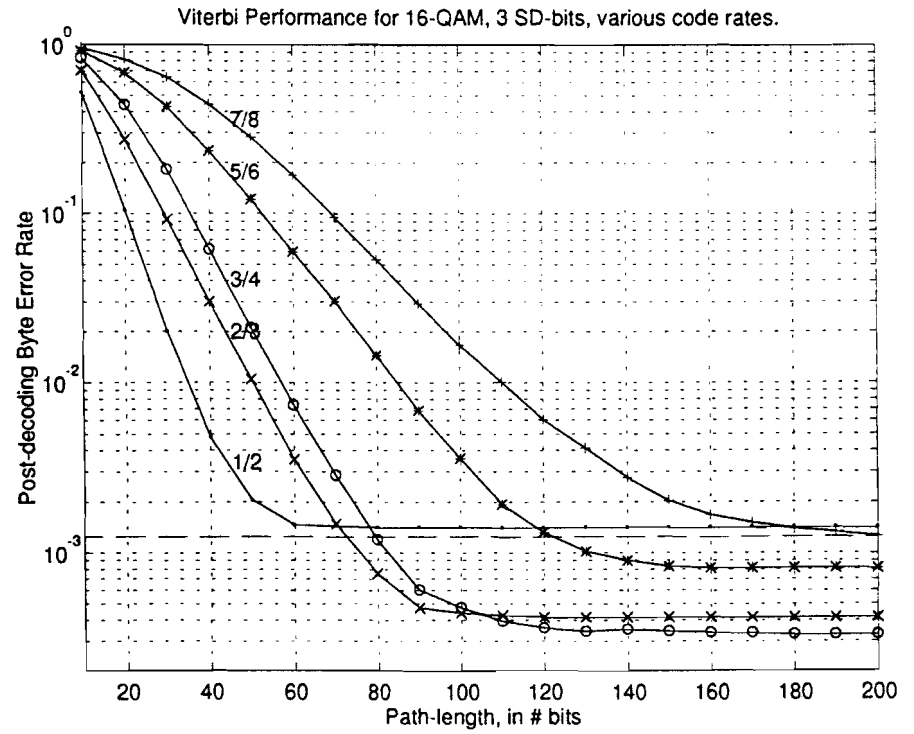

Viterbi Performance for 64-QAM, C/N =14.4dB, 1-5 SD-bits, path=200.
Post-decoding Byte Error Rate
10^-2
10^-3
10^-4
0.5
1
1.5
2
2.5
3
Threshold-value, in constellation units
SD1
SD2
SD3
SD5
SD4

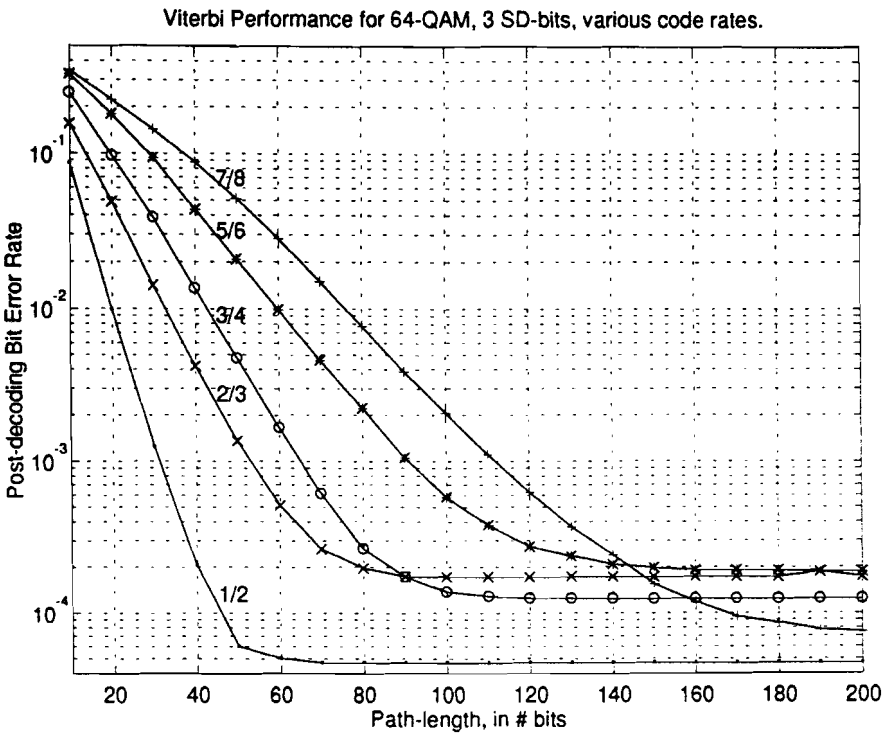
Appendix E: Test-results for Pathlength Investigation in the Viterbi-Decoder

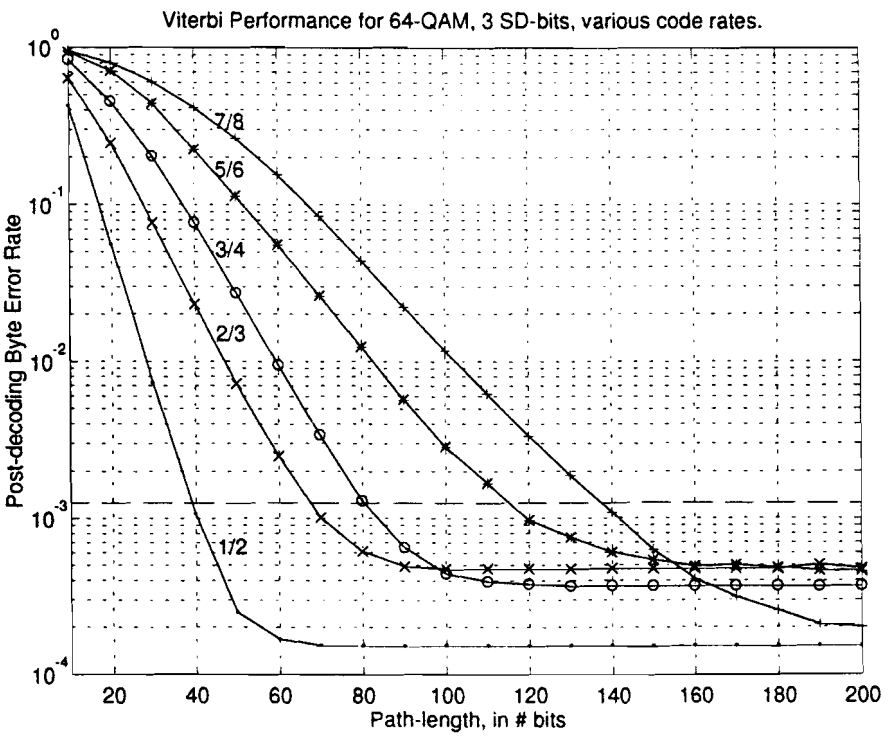





Viterbi Performance for 16-QAM, 3 SD-bits, various code rates.
Post-decoding Bit Error Rate
Path-length, in # bits
7/8
5/6
3/4
2/3
1/2
10^-1
10^-2
10^-3
10^-4
20
40
60
80
100
120
140
160
180
200


Viterbi Performance for 16-QAM, 3 SD-bits, various code rates.
Post-decoding Byte Error Rate
Path-length, in # bits
7/8
5/6
3/4
2/3
1/2
10^0
10^-1
10^-2
10^-3
20
40
60
80
100
120
140
160
180
200


Viterbi Performance for 64-QAM, 3 SD-bits, various code rates.
Post-decoding Bit Error Rate
Path-length, in # bits
7/8
5/6
3/4
2/3
1/2
10^-1
10^-2
10^-3
10^-4
20
40
60
80
100
120
140
160
180
200


Viterbi Performance for 64-QAM, 3 SD-bits, various code rates.
Post-decoding Byte Error Rate
Path-length, in # bits
7/8
5/6
3/4
2/3
1/2
10^0
10^-1
10^-2
10^-3
10^-4
20
40
60
80
100
120
140
160
180
200

Appendix F: Bit Error Rates for non-hierarchical transmission schemes