TEXAS INSTRUMENTS

V.34 Transmitter and Receiver Implementation on the TMS320C50 DSP

Application Report

TEXAS
INSTRUMENTS

V.34 Transmitter and Receiver Implementation on the TMS320C50 DSP

Dr. Steven A. Tretter, Faculty Advisor

Christopher J. Buehler, Jonas Keating, Hayden Metz,
Carl J. Nuzman, and Hemanth Sampath
University of Maryland, Department of Electrical Engineering

PRINTED WITH
SOY INK™

TEXAS INSTRUMENTS

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Contents

List of Figures

V.34 Transmitter and Receiver Implementation on the TMS320C50 DSP

ABSTRACT

This application report presents the design of an efficient V.34 transmitter and receiver pair. The algorithms behind the advanced encoding and decoding schemes of the V.34 recommendation are described, and the assembly language functions that implement these algorithms are referenced. The entire assembly language source code of the project is provided with full documentation of the details of the implementation. C source code from V.34 modem simulations is also provided. It is found that the TMS320C50 digital signal processor (DSP) is exceptionally well-suited to the task of encoding and decoding V.34 modem signals.

1 Introduction

The International Telecommunications Union (ITU) modem recommendation V.34 represents the state of the art in modern modem design [1]. Through the use of advanced coding techniques, modems conforming to the V.34 standard can achieve data communication rates previously thought unattainable on standard general-switched telephone networks (GSTNs). Such modems are rapidly becoming the method of choice for transmitting data quickly and reliably over standard telephone lines. Because of this popularity, the practical design and implementation of a V.34 class modem using conventional digital signal processing hardware is of prime importance.

This project submission details the implementation of a V.34 transmitter-and-receiver pair using standard TMS320C50 evaluation modules (EVMs). The project was designed using two unmodified EVMs on identical Hewlett Packard Vectra 486 PCs. The project consists of two components: the V.34 transmitter and the V.34 receiver.

The transmitter has two parts: an EVM-compatible object file called tran.out and a PC-based front-end called transmit.exe.To run the transmitter, one must first load and run the object file on the EVM, and then run the PC executable file on the host PC.

The receiver has two analogous parts: recv.out and receive.exe. The procedure for running them is the same as for the transmitter. To demonstrate their operation, the output of the EVM running the transmitter should be connected to the input of the EVM running the receiver.

The programs for both the transmitter and receiver are written entirely in TMS320C50 assembly language. They implement a large subset of the ITU recommendation V.34. The transmitter operates at 9.6-, 14.4-, and 19.2-kbps data rates only. However, this speed limitation is imposed only by the choice of a 2400-baud symbol rate. The routines developed in this project are fully general and can be used to transmit at up to the full speed of the V.34 specification (if the maximum 3200-baud symbol rate is utilized). The PC front-end programs are written in C and provide a rudimentary user interface to the DSP programs. In particular, they display the transmitted and received point constellations on the PC graphics screen. These displays are especially useful for debugging purposes, but they also illustrate the operation of the transmitter-receiver pair nicely. Data typed at the terminal also can be transmitted over the data channel once a connection has been made.

The bulk of this paper documents the algorithms used in the two programs. The assembly language routines that implement the particular algorithms are referenced throughout the text. The source files contain explanations of the 'C50 implementation particulars. Therefore, these source files should be examined in addition to the text of the project submission.
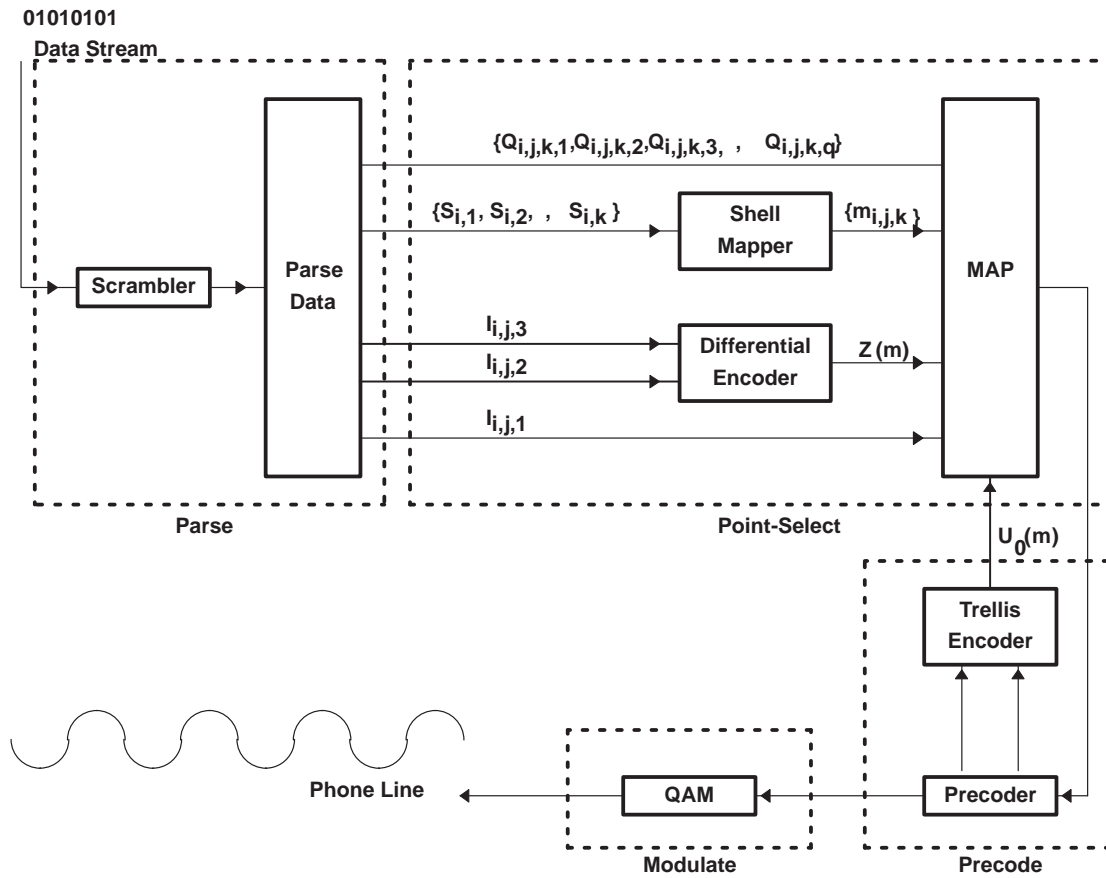
01010101
Data Stream


Figure 1. Block Diagram of the Transmitter

2 V.34 Transmitter

The transmitter consists of four logical units that correspond to the stages through which the data flows as it is encoded for transmission. A block diagram of the basic V.34 transmitter is shown in Figure 1. The first of these units, called *parse*, accepts a stream of binary input data, scrambles it, and then partitions these scrambled bits into different groups to be passed to the next unit. The second logical unit, *point-select*, uses the parsed bits to select signal points from a constellation of 2-dimensional (2D) points that has been specified for use in V.34. The third logical unit, *precode*, applies a precoding filter to the signal points to compensate for the noise-whitening filter present in the V.34 receiver. This unit also contains the trellis encoder, connected in a feedback configuration, which ensures that the transmitted points correspond to a proper trellis sequence. The final unit, *modulate*, performs standard quadrature amplitude modulation (QAM) of the signal points to construct the final output waveform.

2.1 General Overview

The goal of the V.34 transmitter is to map binary input data to an output sequence of 2D signal points. These points are modulated using QAM at a specified carrier frequency for transmission over an analog channel. The frequency at which the modem outputs these points, or symbols, is called the symbol rate. The speed of a modem, or data-transmission rate, is a function of its symbol rate and the method by which the binary data is mapped to the symbols. As an example, a very simple encoding scheme might map 4 bits of data to one of 16 different 2D points. Using a typical symbol rate of 2400 symbols/sec would allow such a modem to transmit at 9600 bits/sec. This general procedure is used in most modern high-speed modems. While the V.34 modem uses the same basic ideas, it employs more advanced techniques to select particular sequences of 2D points that simultaneously minimize the transmitted signal power and maximize the probability of correct decoding in the receiver.

In the V.34 modem, the output sequence of 2D points is divided into sub-sequences of eight points called mapping frames. These mapping frames also can be viewed as a sequence of four 4D points or as a single 16D point. These interpretations are useful when considering the operation of the trellis encoder, which operates on 4D points, and the shell mapper, which operates on 16D points. In this paper, mapping frames are referred to by time index i, 4D points are referred to by time index $m=4i+j$ ($j=0,1,2,3$), and 2D points are referred to by time index $n=2m+k$ ($k=0,1$). These relationships are illustrated in Figure 2.

Output Stream

i–2	i–1	i	i+1	i+2	i+3	i+4

16D: j =

0	1	2	3

4D: k =

0	1

2D Symbols
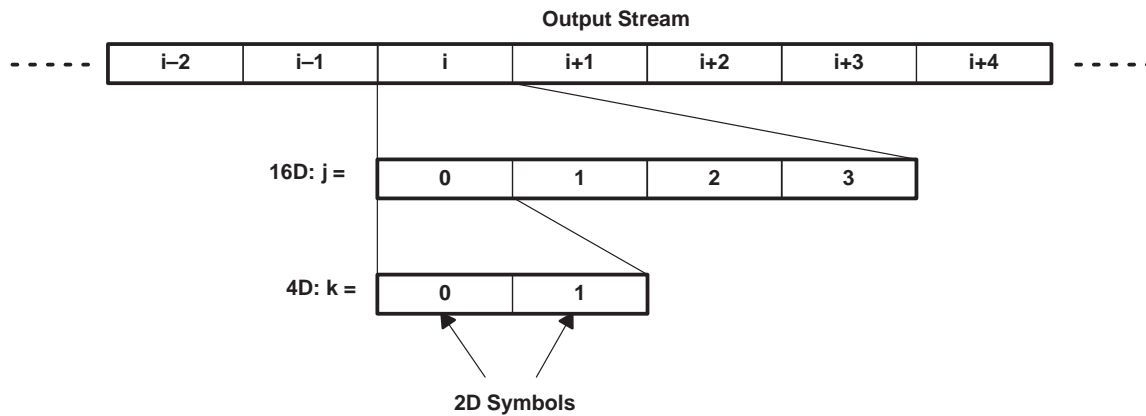
Figure 2. Mapping Frame Conventions

A mapping frame is the smallest unit of output from the transmitter. Each iteration of the main loop of the transmitter program transmits a sequence of eight output points corresponding to one mapping frame. The amount of data encoded in one mapping frame varies according to the data-transmission rate and the symbol rate. For example, at its maximum data-transmission rate of 28800 bits/sec and a symbol rate of 3200 symbols/sec, the V.34 modem encodes 72 bits of data in one mapping frame (28800/3200/8 = 72). This simplified implementation of the transmitter operates at 2400 symbols/sec with data-transmission rates of 9600, 14400, and 19200 bits/sec. For these rates, the transmitter encodes 32, 48, and 64 bits per mapping frame, respectively. Although the number of bits encoded per mapping frame varies, the algorithms used to map this data to signal points are the same regardless of either symbol rate or data-transmission rate. Because of this generality, the algorithms developed in this project can be used for data transmission at any speed, supported by the V.34 recommendation with little or no modification.

The algorithms used by the V.34 transmitter to encode a block of data for a single mapping frame are presented in the following sections. The assembly language functions that implement these algorithms on the TMS320C50 DSP are referenced and described. Refer to Figure 1 for an illustration of how the four different logical units of the transmitter interact and to Appendix A for assembly language source code listings.

2.2 Parse

The parse logical unit serves two major purposes in the transmitter. First, it scrambles the input data to eliminate long, repetitive sequences of bits. Such sequences could cause a periodic output signal, eliciting loss of symbol clock tracking, or unstable behavior in the adaptive filters of the V.34 receiver. Second, it groups the scrambled bits into discrete packets for further processing by the point-select stage of the transmitter.
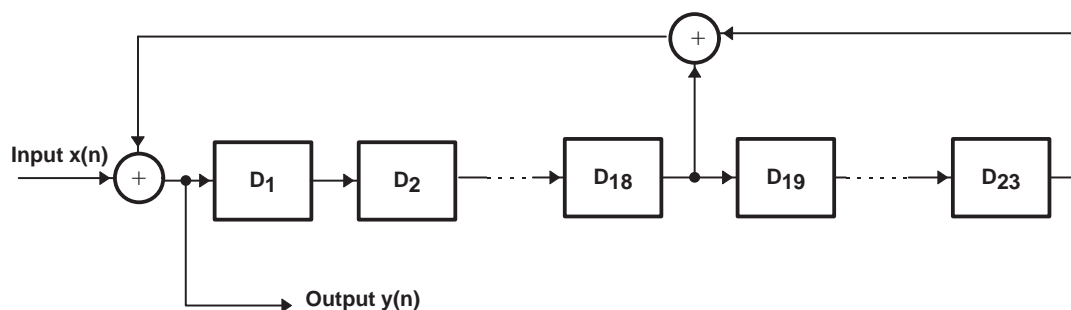
Figure 3. Scrambler Diagram

2.2.1 Scrambler

The data scrambler specified for V.34 uses a linear-shift feedback register with the following generating polynomial (GP):

$$(GP) = 1 + x^{-18} + x^{-23}$$

A diagram of this scrambler, viewed in terms of a 23-tap delay line, is shown in Figure 3. This particular scrambler has been used in many recent modem designs because of its desirable properties. Among other things, the scrambler is self-synchronizing, and the generating polynomial has a maximal period of $2^{23} \pm 1$.

The scrambling function of the parse unit is performed by the subroutine scramble16 found in the source file scram.asm. The subroutine maintains the delay line in a 32-bit state variable (of which only 23 bits are significant) throughout the execution of the program. Using the fact that the first non-zero coefficient of the generating polynomial occurs at the 18th position of the delay line, the routine efficiently scrambles 16 bits in parallel.

2.2.2 Parser

The parser partitions the block of binary data for one mapping frame into different groups of bits for processing by subsequent stages of the transmitter. One group of bits goes to the shell mapper, four groups go to the differential encoder, and the others pass uncoded into the point-select stage. These groups of bits are arranged in the following manner:

$(S_{i,1},S_{i,2},...,S_{i,K})$

$(I_{1i,0},I_{2i,0},I_{3i,0}),(Q_{i,0,0,1},Q_{i,0,0,2},...,Q_{i,0,0,q}),(Q_{i,0,1,1},Q_{i,0,1,2},...,Q_{i,0,1,q}),$

$(I_{1i,1},I_{2i,1},I_{3i,1}),(Q_{i,1,0,1},Q_{i,1,0,2},...,Q_{i(,1,0,q)}),(Q_{i,1,1,1},Q_{i,1,1,2},...,Q_{i,1,1,q}),$

$(I_{1i,2},I_{2i,2},I_{3i,2}),(Q_{i,2,0,1},Q_{i,2,0,2},...,Q_{i,2,0,q}),(Q_{i,2,1,1},Q_{i,2,1,2},...,Q_{i,2,1,q}),$

$(I_{1i,3},I_{2i,3},I_{3i,3}),(Q_{i,3,0,1},Q_{i,3,0,2},...,Q_{i,3,0,q}),(Q_{i,3,1,1},Q_{i,3,1,2},...,Q_{i,3,1,q}).$

As mentioned previously, the subscript *i* refers to the time index of the mapping frame. The *K* bits labeled S are the inputs to the shell mapper, the four groups of three bits labeled I are the inputs to the differential encoder, and the eight groups of *q* bits labeled Q are the uncoded bits. The values of the constants *K* and *q* vary depending on the data transmission rate, while there are always four groups of three I bits. These constants determine the size of a block of bits for one mapping frame. Complete details are given in Table 10/V.34 in reference [1]. For 9600 bits/sec, *K*=20 and *q*=0. Therefore, one mapping frame consists of 20+12=32 bits. For 14400 bits/sec, the constants are *K*=28 and *q*=1 (28+12+8=48 bits per mapping frame), and for 19200 bits/sec, they are *K*=28 and *q*=3 (28+12+24=64 bits per mapping frame).

The parsing operation is carried out by the parsedata function found in the file parse.asm. This function of the transmitter differs for each set of constants *K* and *q*. Therefore, the actual parsing of data is carried out by one of three sub-functions: parsedata96, parsedata144, or parsedata192. The choice of which parsing function to use is determined at the beginning of the execution of the program and is dynamically executed by using the bacc assembly language instruction.

2.3   Point-Select

The unique method of 2D point selection used by the V.34 transmitter is one of its main advancements over previous generations of modems. It is because of these advancements that data-transmission rates twice those of older modems are now achievable.

The V.34 transmitter selects 2D points from a subset of the $2Z^2+(1,1)$ lattice. This subset is called the superconstellation, and it consists of 960 total points. A 240-point subset of the superconstellation, on the $4Z^2+(1,1)$ lattice, is shown in Figure 4. The full superconstellation can be obtained from this subset by rotating it by $0°$, $90°$, $180°$, and $270°$ degrees. Each combination of symbol rate and data rate uses a particular subset of the superconstellation; rarely is the full superconstellation utilized. In this report, these subsets of the superconstellation are referred to as transmission constellations.
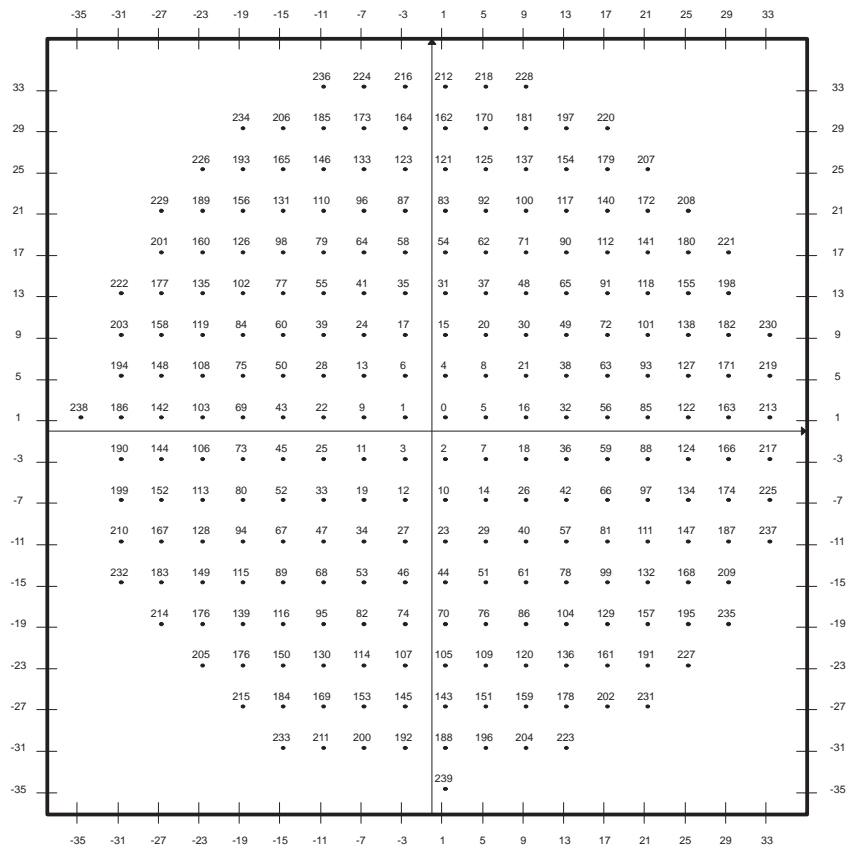
Figure 4. 240-Point Quarter Superconstellation

The points of the transmission constellations are organized into concentric rings of increasing distance from the origin. Each ring contains an equal number of constellation points. The distance metric is defined as the squared Euclidean norm, and it is proportional to the power of the point; consequently, points farther from the origin have greater power. Since one goal of the V.34 transmitter is to minimize the average power of the transmitted signal, it is advantageous to use a point-selection scheme that chooses points from inner rings more often than those from outer rings. This aim is accomplished by the shell mapper.

The V.34 point-selection scheme has three parts that correspond to the three sets of bits generated in the parse unit. First, the shell mapper uses the K S bits to generate a sequence of eight rings. Second, the eight groups of q uncoded Q bits are used to select a point within each ring from the one-quarter superconstellation (see Figure 4). Third, the four groups of I bits (along with output bits U_0 from the trellis encoder) are used by the differential encoder to rotate the four pairs of 2D points. The 90° rotations applied by the differential encoder result in points from the full superconstellation.

2.3.1 Shell Mapper

Shell mapping is a technique used to achieve shaping gain while minimizing the power of the transmitted signal [2],[3]. The shell mapper algorithm selects a sequence of eight rings for the eight 2D points in a single mapping frame.

All transmission constellations are divided into M rings labeled 0 to M±1. Each of the rings is assigned a weight defined to be equal to its label (other weight assignments are possible, but are not used in V.34). In one mapping frame, there are M^8 possible orderings of M rings, and the algorithm seeks to map blocks of K bits to the 2^K sequences of eight rings with the least energy. The energy of a single block of rings is measured in terms of the total weight of the block, which is the sum of the weights of its constituent rings. The M^8 sequences of rings are ordered from least weight to greatest, with each sequence assigned a label from 0 to M^8±1. Runs of sequences that have the same weight form a shell (from which the algorithm derives its name), and the sequences are ordered lexicographically within shells. The first 2^K sequences with the smallest weights are those selected by the shell mapper algorithm. Therefore, given a K-bit number as input, the shell mapper algorithm outputs the sequence of rings which has that number as its label.

As an example, consider the sequence of rings {0,0,0,0,0,0,0,0}. It has weight 0 and is assigned label 0 since there are no sequences of less weight. Therefore, given an input of 0, the shell mapper will return the sequence {0,0,0,0,0,0,0,0}. For weight 1, there are eight possible sequences. These sequences are ordered {0,0,0,0,0,0,0,1}, {0,0,0,0,0,0,1,0}, ..., {1,0,0,0,0,0,0,0} with labels 1, 2, ..., 8. Together, they form the shell of weight 1. Given these labels as input, the shell mapper will select the corresponding sequence of rings. At the other extreme, the sequence {5,5,5,5,5,5,5,5} has weight 40 but is never selected by the algorithm since $2^K < M^8$, leaving this sequence with a label greater than 2^K±1. Clearly, this inequality must hold for the shell mapping to be possible.

The easiest and fastest way to implement shell mapping is by table lookup. However, the memory needed to store an 8×2^{28}-word lookup table far exceeds that of most modem systems (as well as the 64K of SRAM on the 'C50 evaluation module that was used for this project). In light of this fact, a divide-and-conquer approach, which combines smaller lookup tables with some computational effort, is used. The ITU recommendation V.34 provides the actual algorithm that was used in this implementation of a transmitter (see Appendix A). A very brief description of how the algorithm works follows.

Assume that the input to the shell mapper is a *K*-bit number R_0. Three lookup tables are used by the algorithm. The table $g_2(p)$ is the number of ring sequences of length two with weight p. Likewise, table $g_4(p)$ is the number of ring sequences of length four with weight p. Finally, the table $z_8(p)$ is the number of ring sequences of length eight with weight less than p. The first step in the algorithm is to determine the value p such that $z_8(p) < R_0 < z_8(p+1)$. This value of p is the weight of the sequence of eight rings corresponding to R_0, and the difference $R_0 - z_8(p)$ is its offset into the shell of equal-weight sequences. Using the value of the offset, the algorithm then uses the g_4 table to determine the weights and offsets of the two sub-sequences of length four, which make up the final sequence. In a similar fashion, the g_2 table is used to find the weights of the four sub-sequences of length two that constitute the final output sequence. Given the weight of a length-two sequence, the two rings of the sequence can be determined by a simple conditional statement. After finding the two rings in each of the four sub-sequences, the final sequence is found by concatenation. A more complete description of the algorithm is beyond the scope of this paper; refer to Appendix A for a more detailed explanation.

The shell mapping operation is accomplished by the assembly language function called shell_map in the source file named shell.asm. It accepts an input in the accumulator and returns a sequence of eight shells in the array mjk. Although the shell mapping algorithm remains invariant for any number of shells, the lookup tables change with the number of shells. In this implementation, the 9.6-kbps data rate requires M=6 shells, while the 14.4- and 19.2-kbps data rates require M=12 shells. Two sets of tables are stored to accommodate both configurations. The function shell_map dynamically loads the addresses of all tables, so it can function correctly with any number of shells (provided that the proper tables are pre-computed).

2.3.2 *Mapper*

Mapping the eight sets of uncoded Q bits to a point within a shell is a straightforward operation. The points in Figure 4 are each labeled with a number ranging from 0 to 239. They are numbered in order of increasing power, and these numbers are the point indices into a lookup table. To select a point within shell m with a given set of q Q bits = $\{Q_1,Q_2,...,Q_q\}$, the following equation is used:

$$point\ index = Q_1 + 2^1 Q_2 + ,\ldots, 2^{q-1} Q_q + 2^q m$$

In essence, the Q bits are taken as a binary number, and they are added to the index of the first point within the shell. This addition results in the index of a point within the quarter superconstellation. This point can then be retrieved from the lookup table for further processing by the differential encoder.

The mapping operation is carried out by the assembly language function get_initial_point found in the source file getpoint.asm. It loads the constellation point from a table called coords in the file data.asm.

2.3.3 *Differential Encoder*

After the initial points from the quarter superconstellation have been selected by the shell mapper and the mapper portions of the point-select unit, they are differentially encoded to ensure that the transmitted sequence of symbols is 90º rotationally invariant. The differential encoder takes four bits as inputs: one set of three I bits, and the output bit U_0 from the trellis encoder (described later). It operates on one 4D symbol (two 2D points) at a time. Therefore, for one mapping frame, the differential encoder is applied four times. In contrast, the shell mapper is called once per mapping frame, and the simple mapper is used eight times.

The purpose of the differential encoder is to make the sequence of symbols generated by the transmitter invariant to 90º rotations. Since the receiver cannot detect a phase offset of 90º (i.e., the constellation looks the same rotated 90º), the data to be transmitted is encoded according to differences in phase rather than absolute phases. This encoding is accomplished by the following procedure.

Assume that the input to the differential encoder is $\{I_3, I_2, I_1, U_0\}$, and that the encoder has a pair of initial points (i.e., one 4D symbol) to encode. Then, by considering the bit pair (I_3,I_2) to be a 2-digit binary number, the differential encoder calculates the modulo-4 sum

$$Z_m = [(I_3, I_2) + Z_{m-1}] \bmod 4,$$

where $Z_{m\pm1}$ is the previously calculated value of Z_m. The 2-bit number Z_m can assume the values (0,1,2,3) and represents a rotation of Z_m*90^0 clockwise. The first point of the 4D pair then is rotated by the amount specified by Z_m. The second point in the 4D pair is rotated in a similar manner, with the rotation factor computed as

$$W_m = [(Z_m + (I_1, U_0)] \bmod 4,$$

where (I_1, U_0) is considered a 2-digit binary number. The second point then is rotated W^m*90^0 clockwise.

The functions of the differential encoder are performed by three assembly language routines:

- Compute_rotation_factorZ
- Compute_rotation_factorW
- Rotate90_cw

These routines are found in the source file rotate.asm and provide straightforward implementations of the operations described previously.

2.4 Precode

The precode unit consists of the combined nonlinear precoder and trellis encoder. The innovative combination of the two is another feature unique to the V.34 modem. By including the trellis encoder in a feedback loop (as shown in Figure 1), it is possible to compensate for the noise-whitening prediction-error filter of the V.34 receiver without destroying the shaping gain provided by the shell mapper.

2.4.1 Nonlinear Precoder

The V.34 receiver contains a 4-tap prediction-error filter that is used for noise-whitening purposes. This finite impulse response (FIR) filter has complex coefficients and is given by the transfer function

$$H(z) = 1 + h_1 z^{-1} + h_2 z^{-2} + h_3 z^{-3}.$$

The V.34 transmitter contains a local replica of this filter, the inverse of which is used to precode the transmitted signal points. This precoding has the effect of pre-emphasizing the points in a certain way so that they will be properly decoded at the receiver after the noise-whitening process. The downside of this precoding is that it destroys the trellis sequence (see next section) that is transmitted. To remedy this situation, V.34 provides for a correction bit to be computed by the nonlinear precoder to preserve the validity of the trellis sequence. This correction bit, C_0, is computed once for every 4D symbol. It is then exclusive-ORed with the output Y_0 of the trellis encoder to form the bit U_0, one of the input bits to the differential encoder.

The functions of the nonlinear precoder are performed by a variety of short assembly routines in the transmitter. The filtering process is done by a pair of routines: precoder_filter_real and precoder_filter_imag. The strict rounding procedures dictated by the V.34 recommendation are carried out by two functions: round_precoder_output and quantize. The function quantize is actually a wrapper for three other functions, quantize96, quantize144, and quantize192, each of which executes slightly different rounding techniques for the different speeds of the transmitter. The computation of the correction bit is done by the routine compute_C0. These routines are found in the source files pre_filt.asm, quantize.asm, and conv_enc.asm.

2.4.2   *Trellis Encoder*

In short, trellis encoding is a method of achieving coding gain by increasing the density of the constellation while keeping the minimum distance between points the same. One important task of the trellis encoder is to ensure that the transmitted sequence of points conforms to a trellis sequence. A valid trellis sequence is essential for proper decoding by the receiver. Explaining the details of trellis encoding could easily fill this entire paper, so the interested reader is referred to L-F Wei's paper "Trellis-Coded Modulation with Multidimensional Constellations" [4]. It is instructive to note that while trellis encoding has been used in previous modem designs, the unique feedback configuration is particular to the V.34.

There are three 4D trellis encoders specified for use with V.34: a 16-state code, a 32-state code, and a 64-state code. In this transmitter, however, only the 16-state 4D code invented by L-F Wei is implemented. The heart of the trellis encoder is the 16-state convolutional encoder shown in Figure 5. In a typical configuration, the convolutional encoder's two inputs would be taken from the differentially encoded I bits. However, when configured as in the V.34 transmitter, the inputs are computed from the output points after the nonlinear precoding has been applied. The algorithm for computing these inputs to the convolutional encoder from the precoded output points is given in Appendix A. This algorithm yields the inputs for all three types of encoders for V.34. Modifying the transmitter to use one of the other codes would require only inserting a new state table to implement the convolutional encoder.
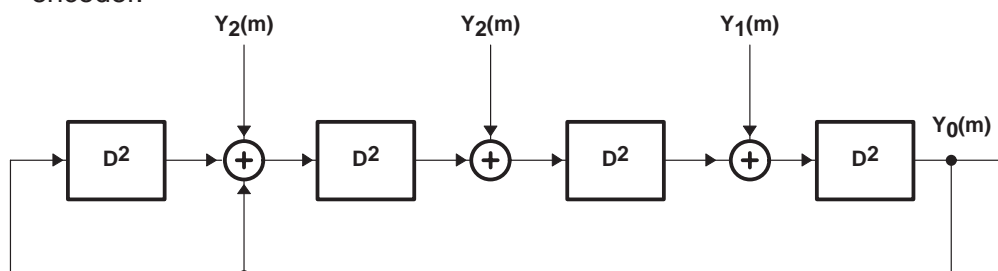
Figure 5. 16-State Convolutional Encoder

The output Y_0 of the convolutional encoder is computed once for every 4D symbol. After modification by the correction bit C_0, the resultant bit U_0 is used to select the next 4D point. It is important to note that the output of the convolutional encoder depends only on past 4D symbols. This fact makes it possible to connect it in such a feedback loop.

The trellis encoder is implemented in the program with two lookup tables by the function do_convolutional_encoder. One lookup table is used by the algorithm in Appendix A to compute the inputs to the convolutional encoder, and the other is used to compute the next state and output of the encoder given the current state and the inputs. The U_0 bit is calculated by the function compute_U0. Both of these routines are found in the file conv_enc.asm.

2.5   Modulate

The QAM of the transmitter is done in an efficient manner. The digital-to-analog (D/A) converter on the EVM is configured to operate at 9600 samples per second. With a symbol rate of 2400 symbols/sec, this results in four samples per symbol. A window of eight 2D symbols is stored at any one time, so the modulator uses a baseband-shaping filter 32-samples wide whose output is modulated up to the passband around the carrier frequency of 1800 Hz. The shaping filter used is a raised cosine filter with excess bandwidth factor equal to 0.12. However, to save computation, two passband-shaping filters are used: one for the in-phase component, and the other for the quadrature component, both of which have been pre-modulated by the carrier frequency. The 2D symbols must be modulated into the passband before application of the filters. Because of the particular choice of symbol rate and carrier frequency (2400 symbols/second and 1800 Hz), this modulation of the symbols simplifies to a rotation by a multiple of 90º. This operation is performed by a short-jump table routine.

To save further computation, the passband-shaping filters are implemented as four banks of 8-tap interpolation filters. Instead of 256 multiply-and-accumulate instructions per output symbol (32 taps $\times$ 4 samples $\times$ 2 channels), only 64 (8 taps $\times$ 4 samples $\times$ 2 channels) are required.

The QAM in the transmitter is performed by two functions: QAM_mod and QAM_wait, located in the qam_mod.asm source file. QAM_mod performs the actual filter calculations using repeated mac and macd instructions, while QAM_wait simply pauses the execution of the program so that the interrupt-driven output procedure can "sync up" with the main transmitter routines. Since the 'C50 DSP is idle over 95% of the time that the transmitter is executing, the QAM_wait function is essential for synchronizing the program to the symbol rate.

Demodulate

Analog Signal

| Passband Equalizer | → | Demod | x(n) → | Prediction Error Filter | $\mathbf{y}(n)$ |

Decode

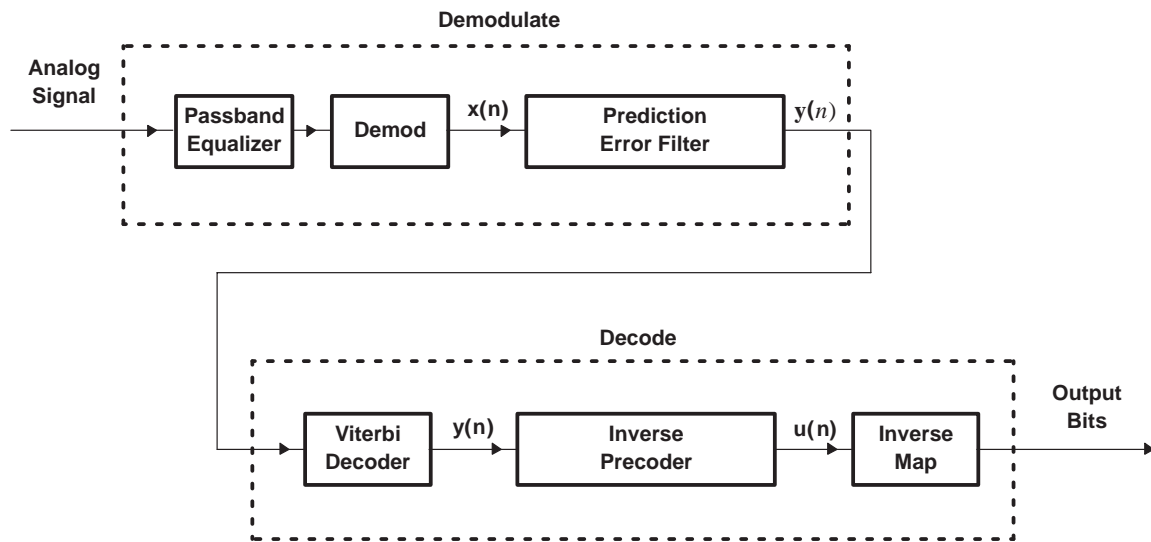| Viterbi Decoder | y(n) → | Inverse Precoder | u(n) → | Inverse Map | → **Output Bits** |

Figure 6. Block Diagram of Receiver

3   V.34 Receiver

A block diagram of this implementation of a typical V.34 receiver is shown in Figure 6. The receiver consists of two logical units that correspond to the two major functions of a modem receiver. The first unit, *demodulate*, accepts an analog waveform as input and demodulates it from a passband QAM signal to a series of baseband signal points. Much of the demodulate unit is based on algorithms found in reference [5]. The second unit, *decode*, takes the raw 2D points output by the demodulate unit, determines the most likely trellis sequence of constellation points, and decodes each mapping frame into the original sequence of bits.

3.1   General Overview

The goal of the V.34 receiver is to sample the sequence of 2D points that was transmitted by the V.34 transmitter and to perform the inverse of the transmitter's encoder operations on this sequence. This goal is simple in theory but difficult to achieve in practice because the transmitted QAM signal is distorted by a variety of nonlinearities during its journey to the receiver. In typical modem connections, these distortions result from traversing numerous consumer phone lines and switching stations. For the transmitter-receiver pair of this project, the distortions arise as the transmitted signal travels through a coaxial cable, into an oscilloscope, and finally to the receiver. Clearly, the distortions experienced by this signal are (presumably) less than those found on a standard phone line, but they are significant nonetheless.

The ideal situation for any modem receiver is to sample the exact transmitted analog signal at the precise symbol rate. By doing so, the sequence of 2D symbols can be retrieved exactly. There are two primary problems that prevent the receiver from doing this directly: first, the sampling rate of the receiver may not match exactly the symbol rate of the transmitter; and second, the received signal may not have the same "shape" as the transmitted signal (i.e., even if the sampling rate were exact, the sampled points would not be exactly those transmitted). The first problem is caused by small, but significant differences in the environment of the transmitter and the receiver. Temperature differences, clock crystal differences, and numerous other factors contribute to slight mismatches in the symbol rates between two modems. The second problem is due to the channel distortions mentioned above. These variations are unpredictable characteristics of the channel through which the signal is transmitted.

The front-end (demodulate unit) of the V.34 receiver is designed to combat these sources of error in sampling signal points. The first problem is handled by the symbol clock recovery mechanism. This process adjusts the sampling rate of the receiver to match the symbol rate of the received signal. The receiver's nominal sampling rate is 7200 Hz, or about three samples per symbol. The symbol clock recovery mechanism continuously corrects this sampling rate based on a computed error estimate. A 144-tap adaptive equalizer compensates for the second problem of channel distortion. The adaptive equalizer is well suited to correcting problems due to unpredictable channel conditions.

The back-end (decoder unit) of the V.34 receiver operates on the sequence of 2D points that is returned by the demodulate unit. The main workhorse of this portion of the receiver is the Viterbi decoder. The Viterbi decoder is an algorithm that is used to determine the most likely sequence of points that was received by the front-end. Once the most likely sequence has been ascertained, the inverse-mapping and unshell-mapping routines transform the points back into the original stream of bits that produced them. Like the transmitter's encoder algorithms, the receiver's decoding algorithms work on one mapping frame (eight 2D symbols) at a time. Brief descriptions of the main units of the V.34 receiver follow.

3.2   Demodulate

Given an out-of-sync and distorted analog signal, it is the responsibility of the demodulate unit to produce solid, accurate data for the decoder unit. This is accomplished in two separate steps. First, the demodulate unit must compensate for any errors in the signal's phase. This is accomplished by the symbol clock recovery section of the demodulator. This section of the demodulator continuously modifies the sampling rate of the analog interface chip (AIC) to ensure that the samples taken by the analog-to-digital (A/D) converter will be as accurate as possible. The second problem, that of channel distortion, is handled in the adaptive equalizer section of the demodulator. This section maintains an array of coefficients which, when convolved with the analog input signal, enables the demodulator to undo the effects of channel distortion. Between these two tools, the demodulator is able to transform an analog input into digital points for use by later parts of the receiver.

3.2.1   *Symbol Clock Recovery*

With a symbol rate of 2400 symbols/sec and a sampling rate of three samples per symbol, the AIC is set to sample the received signal at a frequency of 7200 Hz. Given this, the sampling instants are $1/7200 + t$, where t represents the clock phase. This phase varies due to offsets between the transmitter and receiver, and the goal of the symbol clock recovery loop is to adjust the AIC sampling frequency to ensure that t remains as close to zero as possible. Any error in the clock recovery will cause the receiver timing reference to drift away from the center of the delay line. This would lead to inaccuracy and instability in the receiver.

Each time through the main loop, the main receiver program calls the sym_clock routine in the file sc.asm. The main loop is executed every sampling instant (every time a transmit interrupt occurs), as shown in Figure 7. The sym_clock routine applies two bandpass filters to the sampled data in the input buffer, inpbuffer. One of the filters is tuned to the upper Nyquist frequency of 3000 Hz, while the other is tuned to the lower Nyquist frequency of 600 Hz. The intermediate real variables, eta_l(n) and eta_h(n), correspond to the upper and lower bandpass filter outputs, respectively. They are computed every sampling instant. On the other hand, the imaginary part of the product $Gu(1/7200 + t)*Gl(1/7200 + t)'$ is computed only once per baud. After extraneous terms of the cross-correlation are removed by DC and AC filtering, the timing error $v(n)$ is hard-limited to 2, -2, or 0 depending on its sign. Then, $v(n)$ is passed through a random walk filter to reduce clock jitter, and the output is stored in a threshold flag. This threshold flag, thresh, can have values of 1, -1, or 0. At this point, the AIC sampling phase is either advanced or retarded depending on the value of thresh. If thresh is negative, the AIC phase is retarded; if thresh is positive, the AIC phase is advanced; and when thresh=0, the AIC phase is not adjusted. The overall effect of this loop is to ensure that the AIC samples the analog signal at the most accurate possible time.
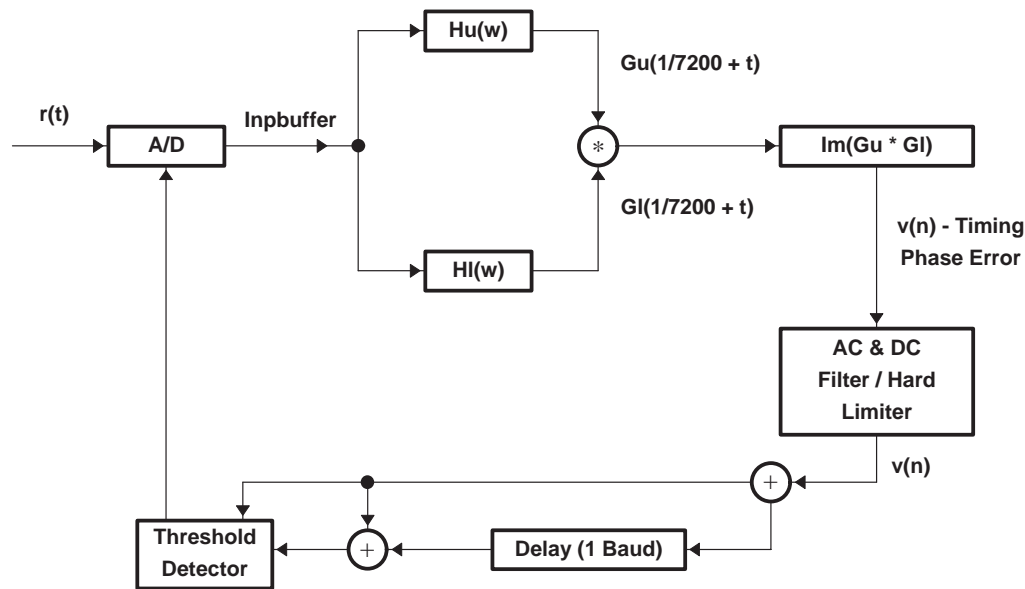
Figure 7. Symbol Clock Recovery Block Diagram

3.2.2 *Phase-Splitting Fractionally-Spaced Equalizer*

The purpose of the equalizer is to compensate for the amplitude and phase distortions of the channel. Since channel frequency and noise statistics cannot be known exactly, it is necessary to implement the equalizer using a least-mean square (LMS) adaptive-tap-adjustment algorithm. The process is illustrated in Figure 8. A general introduction to adaptive equalization can be found in reference [6]. First, the baseband error is computed by comparing the output from the demodulated filter to an ideal constellation point. The goal is to iteratively minimize the baseband error (B) by incrementing the complex tap values, h(m,n) = (hr1(m,n) + j*hi1(m,n)) by small amounts in the directions opposite to the gradient. The gradient is calculated from the following derivatives:

$$\frac{dB}{dh} = \frac{dB}{dhr1} + j^* \frac{dB}{dhi1} = -2 * E\left\{ e(nT) * inpbuffer\left(nT - m\frac{T}{3} \right) \right\}$$

where *e(nT)* is the passband error, inpbuffer is the 144-word buffer where the sampled received data are stored, *n* is the time instant, *T* is one over the baud rate, and *m* runs from 0 to 143.

In the equalize routine, found in the file eq.asm, the real filter hr1 is initialized as a unit impulse function, so that its coefficients are zero everywhere except for the center filter element, which is a one. The imaginary filter, hi1, is initialized to be the Hilbert transform of hr1. First, the data in the input buffer (inpbuffer) is convolved with real and imaginary filter taps. The convolved output, represented in the program by two arrays sigR and sigI such that the output is equal to (sigR + j*sigI), is then demodulated. This is accomplished by performing a complex multiplication with the angle supplied by the carrier tracking loop. The demodulated output (modR + j*modI) is then quantized to the nearest ideal constellation point (cnR+j*cnI) in the slicer routine. Next, the baseband error (diffR+ j*diffI) is computed by finding the difference between the actual and the ideal points. Finally, this error is remodulated, by again using the output from the carrier tracking loop to form the passband error (errorR +j*errorI).

The routine eq_adapt, also found in eq.asm, uses the computed passband error to update the filter taps (hr1 + j*hi1) in double precision. A block diagram of the tap update process is shown in Figure 8. During any given baud, only 36 real and imaginary filter taps of the 144 total real and imaginary filter taps are updated. This is to ensure that the routine never runs for more than one-third of a baud (2777 cycles). After testing the modem, it was noted that by the time the initial training period is finished, the taps have become very stable, and the updating can be done even less frequently.
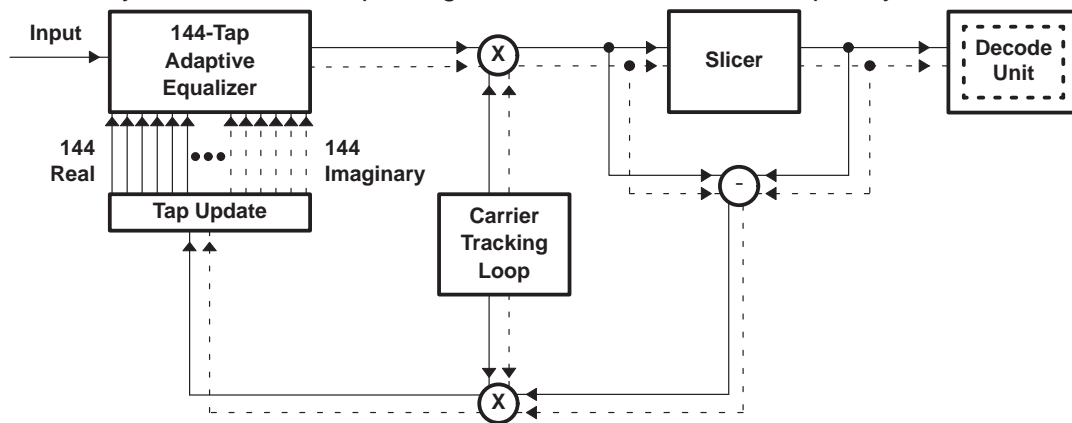
Figure 8. Equalizer Adaptation-Loop Block Diagram

Finally, the slicer routine, found in slicer.asm, quantizes (modR,modI) to the ideal constellation point, (cnR,cnI). First, the routine zeros out the lowest n bits of the incoming data (modR,modI). It then adds a one into the n-th bit, resulting in an odd constellation point, and stores the upper 16 bits of the modified data into (cnR, cnI). This number is hard-limited to a maximum value and stored into cnR and cnI. Based on these values, the inverse magnitude square of the point is computed, to be used in the carrier routine.

3.2.3 Fast Equalizer

The receiver employs a modified version of the training mechanism described in "Rapid Training of a Voiceband Data-Modem Receiver Employing an Equalizer with Fractional-T Spaced Coefficients" by Chevillat et al. [7]. A simple block diagram can be found in Figure 9. The fast equalizer routine first traps a periodic training signal in the delay line by counting off a specified number of bauds after the carrier-detect routine has detected a signal. The routine computes the equalizer coefficients by spectral division and then performs coefficient centering of the computed data. The routine thus performs cyclic equalization on a periodic training sequence (a constant amplitude zero autocorrelation sequence) whose period is equal to the length of the equalizer delay line.
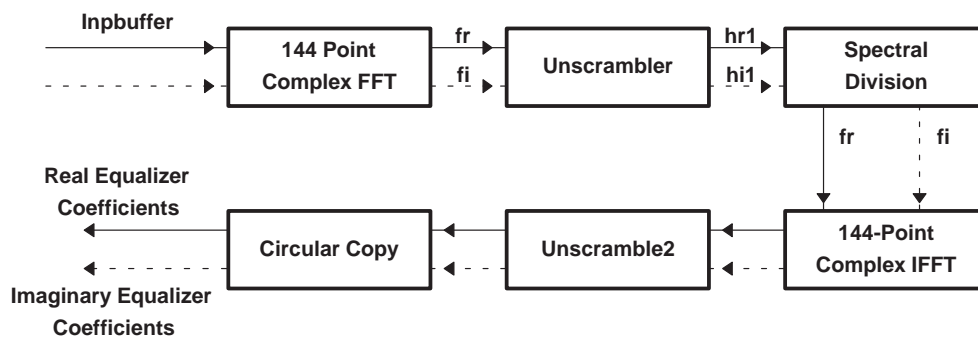
Figure 9. Fast Equalizer Block Diagram

The implementation of this algorithm can be found in the fast_equalize routine in the assembly language file fast.asm, which works as follows. First, the input buffer, inpbuffer, is taken as the real part of the data, and the imaginary part is set to all zeros. Next, a fast Fourier transform (FFT) is performed on the data and the result is passed through the descramble routine (found in the file descram.asm). Then, the following equation is implemented to obtain the frequency components of the equalizer coefficients, *C(i)*:

$$C(i) = \frac{B(i \bmod 48) * D(i)'}{D(i \bmod 48)^2 + D(48 + i \bmod 48)^2 + D(96 + i \bmod 48)^2}$$

where *i* runs from 0 to 143, and *B(i)* is the FFT of the periodic sequence transmitted by the transmitter. Next, the FFT routine is used to compute the inverse Fourier transform of C to get the equalizer coefficients in time domain. Then, unscramble2 is called to unscramble the FFT output, fr. The output is placed into gr, which is copied into hr1 in a circular fashion, placing the maximum value from gr into the center of hr1. Finally, unscramble2 is called again on fi, and the result, gr, is copied circularly into hi1 such that it is rotated by the same amount. Through this process, the fast equalizer is able to completely fill all real and imaginary coefficients in a single pass.

3.3   Decoder

The output of the demodulate unit is a sequence of noise-corrupted 2D points. The first step of the decoder unit is to decide to which ideal constellation points these distorted points correspond. This task is accomplished by the Viterbi decoder. Next, groups of eight points (i.e., one mapping frame) are passed through the inverse precoder and the inverse mapper to be decoded into the output data stream.

3.3.1   *Viterbi Decoder*

The Viterbi maximum-likelihood algorithm is a procedure used for determining the most likely sequence of transmitted points. The key concept behind the operation of the Viterbi algorithm is the idea of a trellis sequence that is generated in the transmitter by the trellis encoder.

Recall that the selection of points in the transmitter is governed in part by the operation of a 16-state convolutional encoder (the trellis encoder). The convolutional encoder takes only two input bits, implying that there are state transitions out of each state to only four different subsequent states. It can similarly be reasoned that there are four different transitions into each state from four previous states. The encoder also has one output bit that is used to select the 4D point from which the current two input bits are derived. These three bits—the two input bits and the single output bit—taken together as a binary number, specify in which of eight different 4D subsets the current 4D symbol is found. Since this subset is specified in part by the trellis encoder (and only a limited number of different state transitions are possible), if the receiver can deduce which state the trellis encoder is in, it can eliminate certain subsets of points from consideration when trying to determine the most likely sequence.

The Viterbi algorithm is one way of implementing this point elimination technique. It maintains a large table in memory to store all possible state transitions and their probabilities of occurring for a finite window of 4D symbols. This particular implementation keeps track of a history of 16 4D symbols, and, for each 4D symbol, it stores 16 entries corresponding to the 16 states in the convolutional code. Each state entry contains a pointer to the most likely previous state as well as a pointer to the ideal 4D constellation points which correspond to the path to that state. The previous path pointers in the table for the most current 4D symbol point back through 16 states in the Viterbi table. These pointers form what is called a trellis path. At each iteration of the Viterbi algorithm, the most likely trellis path is selected, and the oldest 4D symbol in the path is output as part of the transmitted sequence of symbols.

The Viterbi algorithm is initiated once every time the receiver has demodulated two noise-corrupted 2D symbols (i.e., once every 4D symbol period). It is assumed that the receiver has been operating for some time, and that the Viterbi table has already been filled with valid data. The first step in the algorithm is to quantize the noise-corrupted points to the closest point in each of the eight possible 4D subsets. The squared errors for each of these eight subsets, called the branch errors, are recorded. Next, the algorithm iterates through each of the 16 state entries in the Viterbi table that corresponds to the current 4D symbol. For each state, it seeks to update the cumulative path metric, described in the following paragraph.

Each state has a previous cumulative path metric and a current cumulative path metric (which is currently being computed). These metrics are the total sum of all the individual branch errors that make up the trellis path terminating at that state. A small cumulative path metric indicates that the branches composing the path are very likely to be the correct ones. A large path metric indicates just the opposite. Since the path branches correspond directly to certain 4D symbols, finding the path with the smallest cumulative path metric will result in the most likely sequence of points.

To compute the current cumulative path metric for a single state, the Viterbi algorithm checks the four possible state transitions into that state. Each of these transitions, or branches, has a branch error associated with it that was calculated at the beginning of the algorithm. The current cumulative path metric is chosen to be the minimum of the sums of the four previous path metrics and their associated branch errors. The Viterbi algorithm updates the current path metrics for all 16 states in this manner.

Once the current cumulative path metrics have been computed, the algorithm can determine a 4D output symbol. It does this by following the trellis path starting at the state with the smallest cumulative path metric back until it reaches the oldest state in the table. The 4D symbol located at this state then is output for further processing by the later stages of the decode unit.

The assembly language routines for implementing the Viterbi algorithm are found in the source file viterbi.asm. Quantizing to the 4D subsets and calculating the branch errors are accomplished by the routines quantize4, calculate_errors, and calc_branch_errors. Updating the cumulative path metrics is done in two passes, eight states each, by the routines update_path_metrics and upm_finish. Tracing back the most likely trellis path is done by the function trace_back.

3.3.2   Inverse Precoder

The outputs of the Viterbi decoder correspond to the estimated trellis sequence that was transmitted. However, because of the transmitter's nonlinear precoder, this trellis sequence is not necessarily the actual sequence of points that was selected by the transmitter's point-select unit. To account for this possible discrepancy, the trellis sequence from the Viterbi decoder must be inversely precoded by the inverse of the receiver's noise whitening filter. Performing this operation in the receiver is almost identical to precoding the points in the transmitter.

The same routines for performing the precoding operation in the V.34 transmitter are used for the inverse precoder in the receiver with little modification. They are found in the pre_filt.asm assembly language file.

3.3.3   Inverse Mapper

The goal of the inverse mapper is to recover the input bits (the S, Q, and I bit groups) from one entire mapping frame. Ideally, the input to the inverse mapper is the exact sequence of points that was output from the point-select stage of the transmitter. To retrieve the original sequence of bits from these points, it is necessary to invert the operation of the differential encoder, the shell mapper, the simple mapper, the scrambler, and the parser. Undoing these operations is very similar to the actual encoding methods themselves, so the descriptions here will be brief.

Extracting the I bits from the received points is accomplished by reversing the equations of the differential encoder. The rotation factor of the received point within its ring is easy to obtain. With this information, it is possible to solve the equations

$$(I_3, I_2) = [Z_{m-1} - Z_m] \bmod 4,$$
$$(I_1, I_0) = [Z_m - W_m] \bmod 4,$$

to determine the bits (I_3, I_2, I_1) for each received 4D point.

The Q bits and the ring index of each point are found through the use of an inverse constellation point lookup table. This two-dimensional table is indexed by the x and y coordinates of a constellation point, and its entries contain the point indices of the point table in the transmitter. After consulting the lookup table, the lower q bits of the index are masked off to obtain the q Q bits. The remaining high-order bits after the first q constitute the ring index of the point, and they are stored in an array until a block of eight ring indices has been obtained. When eight ring indices have been stored, the end of a mapping frame has been reached, and the unshell-mapper can be called to compute the final K S bits of the original bit sequence. At this point, the eight groups of Q bits have been retrieved.

The unshell-mapper is the most complex routine in the inverse mapper. In essence, it precisely reverses the process of shell mapping to build the index of the received 8-ring sequence. The unshell-mapper algorithm uses the same tables g_2, g_4, and z_8 as the shell mapper. It starts with the four pairs of 2-ring indices. The length-2 shell mapper indices of these pairs are easily determined by way of conditional statements. Next, the algorithm combines the two pairs of length-2 sequences to determine the indices of the two sequences of length 4. Finally, the algorithm combines the indices of the two length-4 sequences to retrieve the index of the final length-8 sequence. For a more detailed explanation of the inverse shell mapping algorithm, refer to Appendix A.

Upon decoding the S, I, and Q bits, the inverse mapper reverses the operation of the parser to combine these groups of bits into one final output array. This array is then unscrambled using a descrambler algorithm with the same generator polynomial as that found in the transmitter. If all went well, the unscrambled output is the same sequence of bits that was transmitted.

The inverse mapper routines are found in the source files invmap.asm and unshell.asm. The simple decoding of the differential encoder is done using the routine get_binary_subset_label to determine the rotation factor of the points and the routine compute_ibits to retrieve the I bits. The function find_point_index returns the index of the unrotated 2D constellation point as well as its uncoded Q bits and ring index. The routine unshell_map performs the inverse shell mapping algorithm to determine the S bits of the mapping frame.

4 Summary

This project is implemented on two TMS320C50 EVM systems. It consists of two parts: a V.34 transmitter and a V.34 receiver. One-way communication links at speeds of 9.6, 14.4, and 19.2 kbps can be established, and the performance of the 'C50 DSP has been evaluated.

Four distinct operational units comprise the V.34 transmitter. The parse unit transforms the input data stream into a form usable by the rest of the transmitter. The point-select unit chooses a sequence of constellation points in a manner that minimizes transmitted signal power and maximizes the probability of correct decoding. The precode unit precodes the transmitted points to compensate for a prediction error filter in the receiver and ensures that these precoded points still conform to a valid trellis sequence. The final unit, modulate, uses QAM to construct the final output waveform.

The V.34 receiver has been similarly divided into distinct units: the demodulate front-end and the decode back-end. The demodulate unit is responsible for correctly sampling the input analog waveform at the symbol instants. It contains functions for symbol clock recovery as well as adaptive channel equalization. The decode unit performs the inverse of the transmitter's encoding operations. It uses the Viterbi algorithm to determine the most likely trellis sequence.

The operations required to perform the encoding and decoding functions for V.34 are ideally suited to fast DSPs such as the 'C50. By implementing the algorithms entirely in assembly language, it is possible to exploit the unique hardware features of the 'C50. Simulations of the transmitter program indicate that the DSP is idle (i.e., waiting for interrupts to occur) over 95% of the execution time. Although the task of the receiver is more complex computationally, even the most conservative estimates show that the DSP is working less than 42% of the time. These numbers demonstrate that the next phase of the implementation of a V.34 modem, the combination of the transmitter and receiver functions onto one DSP, is easily achievable on a single 'C50 DSP. Additional features of V.34, such as channel separation by echo cancellation techniques, could be accommodated with the excess processor time.

References

1. ITU Study Group 14, "Draft Recommendation V.34 for a Modem Operating at Data Signalling Rates of up to 28800 bit/s for Use on the General Switched Telephone Network and on Leased Point-to-Point 2-Wire Telephone-Type Circuits", Document 57-E, June 1994.

2. S. A. Tretter, "Fundamentals of Trellis Shaping and Precoding", unpublished notes.

3. R. Laroia, N. Farvardin, and S. A. Tretter, "On Optimal Shaping of Multidimensional Constellations", *IEEE Trans. Inform. Theory*, Vol. IT-40, pp. 1044-1056, July 1994.

4. L. F. Wei, "Trellis Coded Modulation with Multidimensional Constellations", *IEEE Trans. Inform. Theory*, vol. IT-33, pp. 483-501, July 1987.

5. S. A. Tretter, *Communication System Design Using DSP Algorithms: With Laboratory Experiments for the TMS320C30*, Plenum Publishing Corp., 1995.

6. R. D. Gitlin, J. H. Hayes, S. B. Weinstein, *Data Communication Principles*, Plenum Publishing Corp, 1992.

7. P. R. Chevillat, D. M. Maiwald, and G. Ungerboeck, "Rapid Training of a Voiceband Data-Modem Receiver Employing an Equalizer with Fractional-T Spaced Coefficients", *IEEE Trans. Commun.*, vol. COM-35, pp. 869±876, Sept. 1987.

8. v.34 transmitter and receiver code at:
   ftp://ftp.ti.com/pub/tms320bbs/c5xfiles/apprep34.exe

Appendix A.   VIII.   Constellation Shaping by Shell Mapping

Selected excerpts from Dr. Steven A. Tretter's "Fundamentals of Trellis Shaping and Precoding" on the subject of shell mapping. Used with permission.

A technique known as shell mapping or SVQ shaping has recently been proposed for constellation shaping in V.fast modems. It achieves higher shaping gains with comparable or less computational complexity than trellis shaping or precoding. Shell mapping can be easily combined with trellis channel coding. Constraints on constellation expansion ratio (CER_s) and peak-to-average ratio (PAR_2) are easily included. Shell mapping achieves shaping gains close to that of N-sphere shaping if there are no PAR_2 or CER_s constraints. Also, LTF precoders can be cascaded with shell mapped constellations to perform channel equalization at the transmitter without destroying shaping gain.

The initial use of shell mapping appears to be in a commercial modem manufactured by ESE[1] of Canada to map data blocks to 24-dimensional constellation points selected from the Leech lattice. This mapping method was also suggested in a Ph. D. thesis by Frank Robert Kschischang[2] of the University of Toronto and he also thoroughly analyzed the problem of CER_s and PAR[2] constraints using multidimensional truncated polydisks. Khandani and Kabal[3,4], also independently studied the properties of truncated polydisks but did not give them a name and they discuss some constellation addressing schemes that are not the same as the shell mapping method proposed for V.fast. Also independently, Laroia, Farvardin, and Tretter[5] at the University of Maryland discovered the same shell mapping and truncated polydisk ideas of Kschischang which they called structured vector quantizer (SVQ) shaping. In addition, they suggest grouping the shells into rings as suggested by Calderbank and Ozarow[6] to simplify the mapping complexity yet retain most of the possible shaping gain. In May 1992 Motorola Information Systems[7] (Codex) presented a paper to the CCITT V.fast committee also proposing the use of shell mapping and rings to achieve shaping gain relatively easily and stated that "Trellis shaping is no longer required."

1. G. Lang and F. Longstaff, "A Leech Lattice Modem," *Journal on Selected Areas in Communications*, Aug. 1989.

2. Frank Robert Kschischang, "Shaping and Coding Gain Criteria in Signal Constellation Design," Ph.D. Thesis, University of Toronto, Canada, Department of Electrical Engineering, Communications Group Technical Report, June 1991.

3. A.K. Khandani and P. Kabal, "Shaping Multi-dimensional Signal Spaces — Part I: Optimum Shaping Shell Mapping," Submitted to *IEEE Trans. Information Theory*, July 5, 1991, Revised April 1, 1992. Presented in part at the IEEE Int. Symp. Inform. Theory, June 24–28, 1991.

4. A.K. Khandani and P. Kabal, "Shaping Multi-dimensional Signal Spaces — Part II: Shell-addressed Constellations," Submitted to *IEEE Trans. Information Theory*, July 5, 1991, Revised April 1, 1992.

5. Rajiv Laroia, Nariman Farvardin, and Steven A. Tretter. "On SVQ Shaping of Multidimensional Constellations — High-Rate Large-Dimensional Constellations," Proceedings of the Princeton Conference on Information Sciences and Systems, March 1992. Also submitted to the *IEEE Trans. on Information Theory*, January 1992.

6. A.R. Calderbank and L.H. Ozarow. "Nonequiprobable Signaling on the GAussian Channel." *IEEE Transactions on Information Theory*, Vol. 36, No. 4, July 1990, pp. 726–740.

7. Motorola Information Systems, "Signal mapping and shaping for V.fast." CCITT working paper, Question 3/XVII, WP XVII/1, May 1992.

A. System Description

The block diagram of the transmitter for a system using shell mapping for constellation shaping is shown in Figure VIII–1. The diagram shows the Wei 16-state 4D channel code but can easily be generalized to use any channel code. As described in Section IV, the transmitted 2D constellation is a subset of $z^2+(1/2,1/2)$ and this constellation is partitioned into four 2D subsets A, B, C, and D. The Wei encoder takes in 3 bits every 4D symbol and generates 4 output bits per 4D symbol. The first and second pairs of output bits specify the pair of 2D symbols that form the 4D subset selected by the Wei encoder.

K Bits per Shaping Block of N 2D Symbols → **Shell Mapping Ring Sequence Algorithm** → **Blocks of N Ring Indices $(r_1, r_2, \ldots, r_N)$**

u Uncoded Bits per 2D Symbol → **Select Point Within Ring and Subset** → **2D Symbol Selector** →

$k_c = 3$ Bits per 4D Symbol → **WEI 4D Encoder With the Differential Encoding** → **$n_c = 4$ Bits per 4D Symbol** / **Sequences of 2D Subsets**
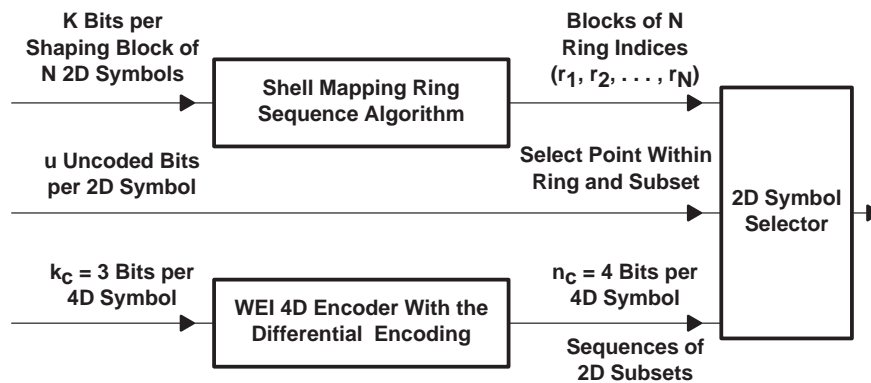
Figure VIII–1.   Constellation Shaping With Shell Mapping

The 2D constellation is partitioned into M "rings" so that each ring contains the same number of points according to the approach of Calderbank and Ozarow. Furthermore, if the transmitter accepts u uncoded bits per 2D symbol, each ring must contain 2^u points from each of the four 2D subsets. Thus the total number of points in the 2D constellation is

$$L = 4M\,2^u = M\,2^{u+2} \tag{VIII–1}$$

The ring closest to the origin should contain the 2^{u+2} constellation points of least energy. The next ring should contain the next 2^{u+2} points in order of energy, etc.

The sequence of 2D rings is determined on a block basis by the shell mapping algorithm. The transmitter takes in K bits every N 2D symbols to select a block of N rings. The shell mapping algorithm for selecting the ring sequences will be described in detail in the following sections. Basically, the algorithm maps blocks of K bits to the 2^K least energy blocks of N rings out of the M^N possible ring blocks. The algorithm requires a reasonable amount of computation and memory for tables.

A relationship between the number of rings M and the number of shaping bits K will now be determined. The K shaping bits specify 2^K blocks of N rings. Each 2D constellation is partitioned into M rings, so there are M^N possible ring blocks. Therefore, it is necessary that

$$2^K \leq M^N \text{ or } 2^{K/N} \leq M \tag{VIII–2}$$

To achieve shaping gain, the constellation size must be expanded. Forney and Wei[8] show that most of the available shaping gain can be obtained with a constellation expansion ratio of $CER_s = 1.5$. Each shaping block the transmitter accepts Nu uncoded bits, K shaping bits, and 3(N/2) coded bits which the Wei encoder converts to 4(N/2) bits. Thus the total number of bits for constellation point selection per block is

$$B = Nu + K + 4(N/2) = N(u + 2) + K \tag{VIII–3}$$

In an unshaped system this requires a 2D constellation of size

$$[2^{N(u+2)+K}]^{1/N} = 2^{u+2+K/N} \tag{VIII–4}$$

According to (VIII–1), the number of points in the 2D constellation for the shell mapped system is $L = M\,2^{u+2}$ so the constellation expansion ratio is

$$CER_s = M\,2^{u+2}/2^{u+2+K/N} = M2^{-K/N} \tag{VIII–5}$$

If CER_s is required to be no greater than 1.5, this upper bounds M by

$$M \leq 1.5\ 2^{K/N} \tag{VIII–6}$$

Combining (VIII–2) and (VIII–6), we see that M must be limited to the range

$$2^{K/N} \leq M \leq 1.5\ 2^{K/N} \tag{VIII–7}$$

8. G.D. Forney and L-F. Wei, "Multidimensional Constellations, Part I," *IEEE J. SAC*, Vol. 7, No. 6, August 1989, p. 887.

The selection of the number of rings M allows a tradeoff between constellation expansion and shaping gain. Using the smallest M gives the smallest constellation expansion and smallest shaping gain, while the largest M gives the largest constellation expansion and shaping gain.

The Motorola CCITT paper[7] on page 7 has some additional formulas relating M, L, and K. We will now see where they come from. First, the number of shaping bits K can be divided by the shaping block length N to give a quotient p and remainder k. Thus K can be expressed as

$$K = Np + k \text{ for } 0 \leq k \leq N\text{--}1 \tag{VIII--8}$$

Substituting this form for K into (VIII–3) gives

$$B = N(u + 2) + K = N(u + 2) + (Np + k) = N(u + 2 + p) + k \tag{VIII--9}$$
$$= nN + k \text{ where } n = u + 2 + p \text{ or } u + 2 = n\text{--}p$$

According to (VIII–1) the number of points in the 2D constellation is

$$L = M\ 2^{u+2} = M\ 2^{n-p} \tag{VIII--10}$$

The number of uncoded bits must be positive, so from (VIII–9) we find that

$$u = n - 2 - p \geq 0$$
$$\text{or } 0 \leq p \leq n\text{--}2 \tag{VIII--11}$$
$$\text{and } n \geq 2$$

For a fixed 2D constellation size L, increasing p forces the number of rings M to increase and results in greater complexity. For a block size of N=8, Motorola recommends using $p \leq 3$.

B. Weights of Ring Blocks, the Basic Concept of Shell Mapping, and Some Data Tables Required for Efficient Implementations

Suppose that the rings are labelled from 0 to M–1 with ring 0 being closest to and ring M–1 furthest from the origin. Let a block of ring indices be denoted by

$$r = [r_1, r_2, \cdots, r_N] \tag{VIII--12}$$

with $r_i \in \{0, 1, \ldots, M–1\}$ for $i=1, \ldots, N$. Each ring must be assigned an integer weight $W_1(r_i)$ where the subscript 1 indicates that the argument is a one-dimensional vector. For example, assuming that the constellation point coordinates are integers, the ring weight could be the average squared distance of points in the ring from the origin. Motorola[7] suggests using the weight function $w_1(i) = i$ for $i = 0, \ldots, M–1$. Justification is given in Appendix VIII–A. The ring weights must form a nondecreasing sequence, that is,

$$w_1(0) \leq w_1(1) \leq \cdots \leq w_1(M–1) \tag{VIII–13}$$

The weight of a sequence will be defined as the sum of the component weights, that is

$$w_N(r) = \sum_{i=1}^{N} w_1(r_i) \tag{VIII–14}$$

The basic idea behind shell mapping is that the M^N possible ring blocks are arranged in an ordered list with lower weight blocks appearing closer to the beginning of the list. There may be many blocks with the same weight and these are called a shell. We will see how to lexicographically order the blocks in a shell. The block at the beginning of the list will be labelled or indexed by 0 and the one at the end by $M^N – 1$. The 2^K blocks closest to the beginning of the list which are also the 2^K lowest weight blocks are used as the shell sequences. Encoding is performed by using the binary K-tuples of shaping bits as the indexes of the list elements. Decoding is performed, basically, by observing that given a ring block, its index is the number of blocks below it on the list. We will see how to achieve reasonable table storage requirements by a "divide and conquer" approach where the N dimensional problem is divided into two N/2 dimensional problems, etc. The problem, then, is to efficiently assign shaping K-tuples to ring blocks and vice versa.

To perform shell mapping, the number of vectors with a given weight must be known. Let $M_p(j)$ be the number of p-tuples of ring indices $r = \{r_1, \ldots, r_p\}$ with total weight $j = w_1(r_1) + \ldots + w_1(r_p)$. The weight generating function is defined to be

$$G_p(z) = \sum_k M_p(k) z^k \tag{VIII–15}$$

For example, with the Motorola weight function $w_1(i)=i$, the number of 1-tuples of weight j is $M_1(j) = 1$ for $j = 0, \ldots, M–1$ and is 0 otherwise. The corresponding generating function is

$$G_1\,(z) = \sum_{k=0}^{M-1} z^k = \frac{1-z^M}{1-z} \qquad\qquad \text{(VIII–16)}$$

The number of (2p)-tuples of weight j can be computed from the number of p-tuples of each weight in the following way. Each 2p-tuple can be considered to be the concatenation of a pair of p-tuples. The weight of the 2p-tuple is the sum of the weights of the two p-tuples. If the first p-tuple has weight k, then the second p-tuple must have weight j–k to make the total weight equal to j. Thus the number of 2p-tuples of weight j must be

$$M_{2p}\,(j) = \sum_{k=0}^{j} M_p(k)\,M_p(j{-}k) \qquad\qquad \text{(VIII–17)}$$

This sum is just a convolution so the corresponding generating function is

$$G_{2p}\,(z) = G_p^2\,(z) \qquad\qquad \text{(VIII–18)}$$

If the shaping block length N is a power of 2, $M_N(j)$ can be found by successively applying (VIII–17) for 2p = 2, 4, ... , N since the initial sequence $M_1(j)$ is easily determined. The shell mapping encoding and decoding algorithms assume that the intermediate results $M_k(j)$ have been stored in memory for k = 1, 2, 4, ... , 2^n, ... , N/2 and all relevant j.

Another sequence that will be used in shell mapping is the number of ring blocks with weight less than or equal to j. This will be designated by $C_N(j)$ and can be computed from $M_N(j)$ by

$$C_N\,(j) = \sum_{k=0}^{j} M_N\,(k) \qquad\qquad \text{(VIII–19)}$$

These values should also be stored in a table.

The number of ring blocks required is 2^K. Thus, the maximum value, J, required for j is the smallest j such that $C_N(j) \geq 2^K$. Then 2^K blocks can be selected from the $C_N(J)$ blocks. J is called the SVQ threshold.

C. The Decoding Algorithm

The shell mapping technique can be best understood by first looking at the decoding algorithm, which is the method for mapping a received block of N rings back into the original input block of K shaping bits. In the receiver, the received signal is demodulated and Viterbi decoded to give a maximum likelihood estimate of the transmitted sequence of constellation points. Then blocks of N estimated 2D constellation points are quantized into blocks of ring indexes.

Given a received block r of N ring indexes, the decoding function $D_N(r)$ computes the index of r in the list. The decoding function can also be expressed as

$$D_N(r) = \{\text{number of blocks below } r \text{ on the list}\} \tag{VIII–20}$$

Binary shaping K-tuples can be assigned to ring blocks in many ways. The method we will examine is based on a splitting algorithm that successively divides blocks into pairs of half the length until blocks of length 1 are reached. Therefore, we will assume that the original block length is a power of 2, that is, $N = 2^q$.

At each step, the i-tuples of rings will be ordered according to rules which will be given shortly. The decoding, ordering, or indexing function on i-tuples will be called $D_i(\cdot)$.

The ordering of 1-tuples is easy. According to (VIII–13), the ring weights must form a nondecreasing sequence. Therefore, 1-tuples will be listed in numerical order. That is, the one-dimensional decoding function is

$$D_1(r) = r \text{ for } r = 0, ..., M–1 \tag{VIII–21}$$

This is the starting point for building higher dimensional decoding functions.

As a matter of notation, let an i-tuple of ring indexes be

$$r^{[i]} = \left[r_1, ..., r_i \right] \tag{VIII–22}$$

The first and second halves of $r^{[i]}$ will be designated by

$$r_1^{[i]} = [r_1, ..., r_{i/2}] \text{ and } r_2^{[i]} = [r_{i/2+1}, ..., r_i] \tag{VIII–23}$$

We will now see how to define a decoding function for i-tuples in terms of one for i/2-tuples. Assume $D_{i/2}(\cdot)$ is known. First, order i-tuples of rings according to the following rules for the three possible cases:

Case 1: Different Weight i-tuples

An i-tuple $u^{[i]}$ is listed below $v^{[i]}$ if $w_i\left(u^{[i]}\right) < w_i\left(v^{[i]}\right)$ where $w_i(\cdot)$ is the i-dimensional weight function.

Case 2: Equal Weight i-tuples, but First Halves have Different i/2-dimensional Indexes

When $w_i\left(u^{[i]}\right) = w_i\left(v^{[i]}\right)$, then $u^{[i]}$ is listed below $v^{[i]}$ if

$$D_{i/2} \ (u_1^{[i]}) < D_{i/2} \ (v_1^{[i]})$$

Case 3: Equal Weight i-tuples. Indexes of 1st Halves are the same

When $w_i\left(u^{[i]}\right) = w_i\left(v^{[i]}\right)$ and $u_1^{[i]} = v_1^{[i]}$, then list $u^{[i]}$ below $v^{[i]}$ if $D_{i/2} \ (u_2^{[i]}) < D_{i/2} \ (v_2^{[i]})$.

The i-dimensional decoding function can also be expressed in terms of the following function:

$$N_i\left(v^{[i]}\right) = \{\text{number of i-tuples } u^{[i]} \text{ with } w_i(u^{[i]}) = w_i(v^{[i]}) \tag{VIII–24}$$

$$\text{and } u^{[i]} \text{ below } v^{[i]}\}$$

This quantity will be called the offset into the shell. Then, the i-dimensional decoding function can be expressed as

$$
\begin{aligned}
D_i\left(v^{[i]}\right) &= \{\text{number of i-tuples below } v^{[i]}\} \\
&= \{\text{number of i-tuples } u^{[i]} \text{ with } w_i(u^{[i]}) < w_i(v^{[i]})\} \\
&\quad + \{\text{number of i-tuples } u^{[i]} \text{ with } w_i(u^{[i]}) = w_i(v^{[i]}) \\
&\quad \text{and } u^{[i]} \text{ below } v^{[i]}\} \\
&= C_i\left[w_i\left(v^{[i]}\right) - 1\right] + N_i\left(v^{[i]}\right)
\end{aligned}
\tag{VIII–25}
$$

The quantities $D_i(\cdot)$ for $i < N$ do not have to be computed. Also, $C_i(\cdot)$ is needed only for $i=N$. Finally, we will see in the next paragraph how to recursively compute $N_i(\cdot)$ for $i = 2, 4, \ldots, N$ using $N_{i/2}(\cdot)$ and $M_{i/2}(\cdot)$. Then $D_N(\cdot)$ can be computed by (VIII–25) for $i=N$.

The vectors counted in $N_i(v^{[i]})$ can be partitioned into the following three types:

Type 1: 1st Halves Differ in Weight

Consider the set

$$\left\{ u^{[i]} \;\middle|\; w_i\,(u^{[i]}) \;=\; w_i(v^{[i]}) \;\cap\; w_{i/2}\,(u_1^{[i]}) \;<\; w_{i/2}\,(v_1^{[i]}) \right\} \tag{VIII–26}$$

The number of elements in this set is

$$a_1 \;=\; \sum_{k=0}^{w_{i/2}\,(v_1^{[i]})\,-1} M_{i/2}\,(k)\; M_{i/2}\,[w_i\,(v^{[i]})-k] \tag{VIII–27}$$

Type 2: 1st Halves Differ but Have the Same Weight

Consider the set

$$\left\{ u^{[i]} \;\middle|\; w_i\,(u^{[i]}) \;=\; w_i\,(v^{[i]}) \;\cap\; w_{i/2}\,(u_1^{[i]}) \;=\; w_{i/2}\,(v_1^{[i]}) \;\cap\; D_{i/2}\,(u_1^{[i]}) \;<\; D_{i/2}\,(v_1^{[i]}) \right\} \tag{VIII–28}$$

According to (VIII–24), the number of choices for $u_1^{[i]}$ is $N_{i/2}(v_1^{[i]})$. The total weight must be $w_i(v^{[i]})$, so the number of choices for $u_2^{[i]}$ is $M_{i/2}[w_i(v^{[i]}) - w_{i/2}(v_1^{[i]})]$. Thus the number of type 2 vectors is

$$a_2 \;=\; N_{i/2}\,(v_1^{[i]})\; M_{i/2}\,[w_i\,(v^{[i]})-w_{i/2}\,(v_1^{[i]})] \tag{VIII–29}$$

Type 3: 1st Halves Identical

The number of vectors in the set

$$\left\{ u^{[i]} \;\middle|\; w_i\,(u^{[i]}) \;=\; w_i(v^{[i]}) \;\cap\; u_1^{[i]} \;=\; v_1^{[i]} \;\cap\; D_{i/2}\left(u_2^{[i]}\right) \;<\; D_{1/2}\left(v_2^{[i]}\right) \right\} \tag{VIII–30}$$

is

$$a_3 \;=\; N_{i/2}\left(v_2^{[i]}\right) \tag{VIII–31}$$

Notice, also, that the weight of $v_2^{[i]}$ is

$$w_{i/2}\left(v_2^{[i]}\right) \;=\; w_i\left(v^{[i]}\right) \;-\; w_{i/2}\left(v_1^{[i]}\right) \tag{VIII–32}$$

Adding the numbers for the three cases gives

$$N_i\left(v^{[i]}\right) = \sum_{k=0}^{w_{i/2}\left(v_1^{[i]}\right)-1} M_{i/2}\left(k\right) M_{i/2}\left[w_i\left(v^{[i]}\right) - k\right]$$

$$+ N_{i/2}\left(v_1^{[i]}\right) M_{i/2}\left[W_{i/2}\left(v_2^{[i]}\right)\right] + N_{i/2}\left(v_2^{[i]}\right)$$

(VIII–33)

The decoding operation is performed iteratively. First, $r = v^{[N]}$ is divided into N/2 pairs of ring indexes and $N_2(\cdot)$ is computed for each of the pairs using (VIII–33). Adjacent pairs are then combined into N/4 4-tuples and $N_4(\cdot)$ is computed for each. The doubling procedure is repeated until $N_N(r)$ is computed. Then the index $D_N(r)$ is computed by (VIII–25) with i=N.

Example VIII–1.   N = 2 and Motorola Weight Function

Let the one-dimensional weight function be

$$w_1\left(r\right) = r \text{ for } r = 0, \dots, M - 1$$

(VIII–34)

and designate 2-tuples by $v^{[2]} = [r1, \ r2]$. The number of 1-tuples of each weight is

$$M_1\left(k\right) = \begin{cases} 1 & \text{for } k = 0, \dots, M - 1 \\ 0 & \text{elsewhere} \end{cases}$$

(VIII–35)

For this example,

$$v_1^{\ [2]} = \left[r_1\right] \text{ and } v_2^{\ [2]} = \left[r_2\right]$$

Then, according to (VIII–33), the shell offset is

$$N_2\left(v^{[2]}\right) = \sum_{k=0}^{w_1(r_1) - 1} M_1\left(k\right) M_1\left[w_2\left(v^{[2]}\right) - k\right]$$

$$+ N_1\left(r_1\right) M_1\left[w_2\left(v^{[2]}\right) - w_1\left(r_1\right)\right] + N_1\left(r_2\right)$$

(VIII–36)

Since there is a single 1-tuple of each weight, $N_1(r) = 0$, so the last line of (VIII–36) is zero and

$$N_2\left(v^{[2]}\right) = \sum_{k=0}^{r_1-1} M_1(k)\, M_1(r_1 + r_2 - k) = \sum_{k=0}^{r_1-1} M_1(r_1 + r_2 - k) \qquad \text{(VIII–37)}$$

From Figure VIII–2 it can be seen that this sum is

$$N_2\left(\left[r_1,\ r_2\right]\right) = \begin{cases} r_1 & \text{for } 0 \le r_1 + r_2 \le M - 1 \\ M - 1 - r_2 & \text{for } M \le r_1 + r_2 \le r_1 + M - 2 \end{cases} \qquad \text{(VIII–38)}$$

Example of Shell Sequence Ordering for N = 4 and M = 3

$D_1(r)$	r		
0	0	$M_1(0) = 1$	
1	1	$M_1(1) = 1$	
2	2	$M_1(2) = 1$	

$D_2(r)$	r		
0	00	$M_2(0) = 1$	
1	01	$M_2(1) = 2$	
2	10		
3	02	$M_2(2) = 3$	
4	11		
5	20		
6	12	$M_2(3) = 2$	
7	21		
8	22	$M_2(4) = 1$	

$D_4(r)$	r		
0	0000	$M_4(0) = 1$	
1	0001	$M_4(1) = 4$	$C_4(0) = 1$
2	0010		
3	0100		
4	1000		

5	0002	$M_4(2) = 10$	$C_4(1) = 5$
6	0011		
7	0020		
8	0101		
9	0110		
10	1001		
11	1010		
12	0200		
13	1100		
14	2000		
15	0012	$M_4(3) = 16$	$C_4(2) = 15$
16	0021		
17	0102		
18	0111		
19	0120		
20	1002		
21	1011		
22	1020		
23	0201		
24	0210		
25	1101		
26	1110		
27	2001		
28	2010		
29	1200		
30	2100		
31	0022	$M_4(4) = 19$	$C_4(3) = 31$
32	0112		
33	0121		
34	1012		
35	1021		
36	0202		
37	0211		
38	0220		
39	1102		
40	1111		

41	1120		
42	2002		
43	2011		
44	2020		
45	1201		
46	1210		
47	2101		
48	2110		
49	2200		
50	0122	$M_4(5) = 16$	$C_4(4) = 50$
51	1022		
52	0212		
53	0221		
54	1112		
55	1121		
56	2012		
57	2021		
58	1202		
59	1211		
60	1220		
61	2102		
62	2111		
63	2120		
64	2201		
65	2210		
66	0222	$M_4(6) = 10$	$C_4(5) = 66$
67	1122		
68	2022		
69	1212		
70	1221		
71	2112		
72	2121		
73	2202		
74	2211		
75	2220		
76	1222	$M_4(7) = 4$	$C_4(6) = 76$

77	2122		
78	2212		
79	2221		
80	2222	$M_4(8) = 1$	$C_4(7) = 80$

$$C_4(8) = 81$$

$M_1 (r_1 + r_2 - k)$

1

$r_1 + r_2 - (M - 1)$ 0 $r_1 - 1$ $r_1 + r_2$

k

(a) $0 \leq r_1 + r_2 \leq M - 1$

$M_1 (r_1 + r_2 - k)$

1

0     $r_1 + r_2 - (M - 1)$ $r_1 - 1$ $r_1 + r_2$

k
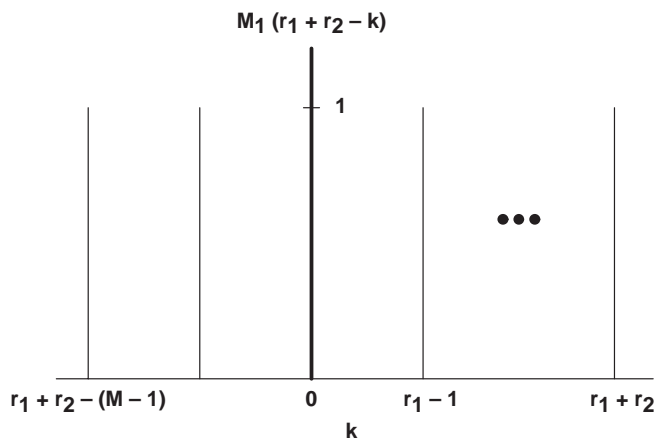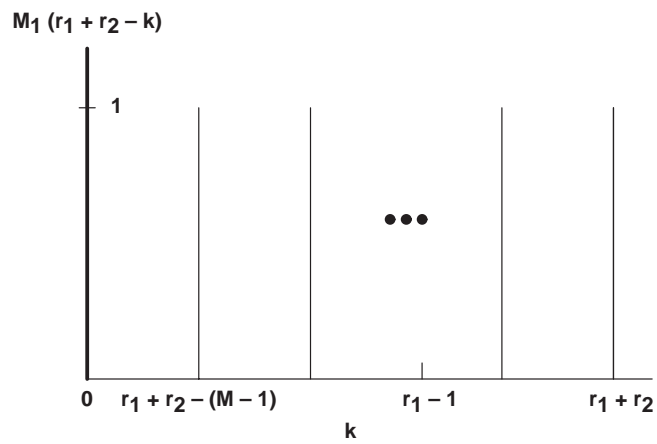
(b) $M \leq r_1 + r_2 \leq r_1 + M - 2$

Figure VIII–2.   Plot of Function in Summation

D. The Encoding Algorithm

The encoding algorithm maps binary K-tuples of shaping bits into N-tuples of ring indexes. The mapping is based on the ordering of ring blocks described in the previous section. Let the shaping bit K-tuple be the binary representation for the index $D_N(v^{[N]})$. The problem is to find $v^{[N]}$. According to (VIII–25)

$$D_N\left(v^{[N]}\right) = C_N\left[w_N\left(v^{[N]}\right) - 1\right] + N_N\left(v^{[N]}\right) \qquad \text{(VIII–39)}$$

Remember that $N_N(v^{[N]})$ is the number of blocks with the same weight as $v^{[N]}$ that are below it on the list. Also, $M_N(w_N(v^{[N]}))$ is the number of blocks with the weight of $v^{[N]}$. Therefore,

$$0 \leq N_N\left(v^{[N]}\right) < M_N\left[w_N\left(v^{[N]}\right)\right] \tag{VIII–40}$$

Also,

$$C_N\left[w_N\left(v^{[N]}\right)\right] = C_N\left[w_N\left(v^{[N]}\right) - 1\right) + M_N\left[w_N\left(v^{[N]}\right)\right] \tag{VIII–41}$$

Thus,

$$C_N\left[w_N\left(v^{[N]}\right) - 1\right] \leq D_N\left(v^{[N]}\right) < C_N\left[w_N\left(v^{[N]}\right)\right] \tag{VIII–42}$$

This shows that the index D_N will always fall in an interval bracketed by a pair of successive C_N's.

Encoding Step 1: Compute the Weight of $v^{[N]}$

Based on (VIII–42), the weight of $v^{[N]}$ is

$$w_N\left(v^{[N]}\right) = \max_n\left\{n \mid C_N\left(n - 1\right) \leq D_N\left(v^{[N]}\right)\right\} \tag{VIII–43}$$

Encoding Step 2: Compute the Offset

Once the weight is known, we can compute the offset as

$$N_N\left(v^{[N]}\right) = D_N\left(v^{[N]}\right) - C_N\left[w_N\left(v^{[N]}\right) - 1\right] \tag{VIII–44}$$

According to (VIII–33),

$$N_N\left(v^{[N]}\right) = \left\{ \sum_{k=0}^{w_{N/2}\left(v_1^{[N]}\right) - 1} M_{N/2}\left(k\right) M_{N/2}\left[w_N\left(v^{[N]}\right) - k\right]\right\} \tag{VIII–45}$$

$$+ \left\{N_{N/2}\left(v_1^{[N]}\right) M_{N/2}\left[W_{N/2}\left(v^{(N)}\right)\right]\right\}$$

$$+ \left\{N_{N/2}\left(v_2^{[N]}\right)\right\}$$

Of the weight $w_N(v^{[N]})$ N-tuples, the number with the same weight as $v^{[N]}$ in the first half, that is, $w_{N/2}(u_1^{[N]}) = w_{N/2}(v_1^{[i]})$, is

$$M_{N/2}\left[w_{N/2}\left(v_1^{[N]}\right)\right] M_{N/2}\left[w_{N/2}\left(v_2^{[N]}\right)\right] \tag{VIII–46}$$

where $w_{N/2}(v_2^{[N]}) = w_N(v^{[N]}) - w_{N/2}(v_1^{[N]})$. Thus, the sum of the second and third terms in the curly braces in (VIII–45), which is the number of N-tuples below $v^{[N]}$ with the same total weight as $v^{[N]}$ and the same weight in the first half as $v_1^{[M]}$, must satisfy,

$$0 \le N_{N/2}\left(v_1^{[M]}\right) M_{N/2}\left[w_{N/2}\left(v_2^{[M]}\right)\right] + N_{N/2}\left(v_2^{[n]}\right) < M_{N/2}\left[w_{N/2}\left(v_1^{[M]}\right)\right] M_{N/2}\left[w_N\left(v_2^{[M]}\right)\right] \tag{VIII–47}$$

Rearranging this last inequality gives

$$\left\{N_{N/2}\left(v_1^{[M]}\right) M_{N/2}\left[w_{N/2}\left(v_2^{[M]}\right)\right] + N_{N/2}\left(v_2^{[M]}\right)\right\} - M_{N/2}\left[w_{N/2}\left(v_1^{[M]}\right)\right] M_{N/2}\left[w_N\left(v_2^{[M]}\right)\right] \tag{VIII–48}$$

$$= \left\{N_N\left(v^{[M]}\right) - \sum_{k=0}^{w_{N/2}\left(v_1^{[M]}\right)-1} M_{N/2}(k)\, M_{N/2}\left[w_N\left(v^{[M]}\right) - k\right]\right\}$$

$$-M_{N/2}\left[w_{N/2}\left(v_1^{[M]}\right)\right] M_{N/2}\left[w_{N/2}\left(v_2^{[M]}\right)\right] < 0$$

Encoding Step 3: Finding the Weights of $v_1^{[N]}$ and $v_2^{[N]}$

From the inequality (VIII–48), we see that the weight of the first half of $v^{[N]}$ must be

$$W_{N/2}\left(v_1^{[M]}\right) = \max_{n}\left\{n \mid \sum_{k=0}^{n-1} M_{N/2}[k]\, M_{N/2}\left[w_{N/2}\left(v^{[M]}\right) - k\right] \le N_N\left(v^{[M]}\right)\right\} \tag{VIII–49}$$

The weight of the second half can then be computed as

$$w_{N/2}\left(v_2^{[M]}\right) = w_N\left(v^{[M]}\right) - w_{N/2}\left(v_1^{[M]}\right) \tag{VIII–50}$$

Encoding Step 4: Compute the Partial Offset

Now the following partial offset d can be computed:

$$d \ = \ N_{N/2} \left(v_1^{[M]} \right) M_{N/2} \left[w_{N/2} \left(v_2^{[M]} \right) \right] + N_{N/2} \left(v_2^{[M]} \right)$$

$$= \ N_N \left(v^{[M]} \right) - \sum_{k=0}^{w_{N/2}\left(v_1^{[M]} \right) - 1} M_{N/2} \left(k \right) M_{N/2} \left[w_N \left(v^{[M]} \right) - k \right] \tag{VIII–51}$$

Encoding Step 5: Compute the Offsets of the 1st and 2nd Halves

Remember that $N_{N/2} \left(v_2^{[M]} \right)$ is the number of weight $w_{N/2} \left(v_2^{[M]} \right)$ N/2-tuples below $v_2^{[M]}$. The total number of N/2-tuples with weight $w_{N/2} \left(v_2^{[M]} \right)$ is $M_{N/2} \left[w_{N/2} \left(v_2^{[M]} \right) \right]$, so

$$0 \le N_{N/2} \left(v_2^{[M]} \right) < M_{N/2} \left[w_{N/2} \left(v_2^{[M]} \right) \right] \tag{VIII–52}$$

Using the same reasoning, we see that the following inequality must also be true:

$$0 \le N_{N/2} \left(v_1^{[M]} \right) < M_{N/2} \left[w_{N/2} \left(v_1^{[M]} \right) \right] \tag{VIII–53}$$

Using the Euclidean division algorithm, $N_{N/2} \left(v_2^{[M]} \right)$ and $N_{N/2} \left(v_1^{[M]} \right)$ can be found by dividing d by $M_{N/2} \left[w_{N/2} \left(v_2^{[M]} \right) \right]$. $N_{N/2} \left(v_1^{[M]} \right)$ is the remainder and $N_{N/2} \left(v_2^{[M]} \right)$ is the quotient.

Thus, to complete step 5, compute the following:

$$N_{N/2} \left(v_1^{[M]} \right) \ = \ \text{int} \left\{ d / M_{N/2} \left[w_{N/2} \left(v_2^{[M]} \right) \right] \right\} \tag{VIII–54}$$

and

$$N_{N/2} \left(v_2^{[M]} \right) \ = \ d - M_{N/2} \left[w_{N/2} \left(v_2^{[M]} \right) \right] N_{N/2} \left(v_1^{[M]} \right) \tag{VIII–55}$$

Encoding Step 6: Iterate the Procedure

Steps 3, 4, and 5 can now be applied to the two N/2-tuples to find the weights and offsets of four N/4-tuples, and the procedure can be repeated until 1-tuples are reached. Then the ring index for a 1-tuple will be obvious from its weight $w_1(\cdot)$ and offset $N_1(\cdot)$.

Example 1. (Continued)

For the Motorola ring weight assignment, it is not necessary to decompose 2-tuples into 1-tuples since the results have been analytically computed and can be found from (VIII–38). Let $v = [r_1, r_2]$ so $w_2(v) = r_1 + r_2$. Then from the first part of Example 1, it follows that

$$r_1 = \begin{cases} N_2\ (v) & \text{for} \quad w_2\ (v) \le M - 1 \\ W_2\ (v) + N_2\ (v) - (M - 1) & \text{for} \quad M - 1 < W_2\ (v) \end{cases} \qquad \text{(VIII–56)}$$

$$r_2 = w_2\ (v) - r_1$$

Encoding Example for N = 4 and M = 3

Suppose $D_4\left(r^{[4]}\right) = 13$

Step 1

From the ordered list of 4-tuples, we see that

$$C_4\ (1) = 5 \le 13 < C_4\ (2) = 15$$

Thus, according to Step 1, $w_4\left(r^{[4]}\right) = 2$.

Step 2

The offset is $D_4\left(r^{[4]}\right) - C_4\ (1) = 13 - 5 = 8$

Step 3   Find Weights of 1st and 2nd Halves

$$M_2\ (0)\ M_2\ (2) + M_2\ (1)\ M_2\ (1) = 1 \times 3 + 2 \times 2 = 7 \le 8$$

$$< M_2\ (0)\ M_2\ (2) + M_2\ (1)\ M_2\ (1) + M_2\ (2)\ M_2\ (0) = 10$$

Thus

$$w_2\left(r_1^{[4]}\right) = 2 \text{ and } w_2\left(r_1^{[4]}\right) = w_4\left(r^{[4]}\right) - w_2\left(r_1^{[4]}\right) = 2 - 2 = 0$$

Step 4

The partial offset is $d = 8 - 7 = 1$

Step 5

Find Offsets of 1st and 2nd Halves

$$M_2\left(w_2\left(r_2^{[4]}\right)\right) = M_2(0) = 1$$

$$N_2\left(r_1^{[4]}\right) = \text{int}\left\{d \,/\, M_2\left(w_2\left(r_2^{[4]}\right)\right)\right. = 1$$

$$N_2\left(r_2^{[4]}\right) = d - M_2\left(w_2\left(r_2^{[4]}\right)\right) N_2\left(r_1^{[4]}\right) = 1 - 1 \times 1 = 0$$

Using (VIII–56) we find that

$$r_1 = N_2\left(r_1^{[4]}\right) = 1, \quad r_2 = w_2\left(r_1^{[4]}\right) - r_1 = 2 - 1 = 1$$

$$r_3 = N_2\left(r_2^{[4]}\right) = 0, \quad r_4 = w_2\left(r_2^{[4]}\right) - r_3 = 0 - 0 = 0$$

So the encoded ring block is $r^{[4]} = [\,1100\,]$

Appendix VIII-A

Justification for using the Motorola weight function is given in this appendix. Suppose that the rings are formed from M concentric circles as shown in Figure VIII–3.
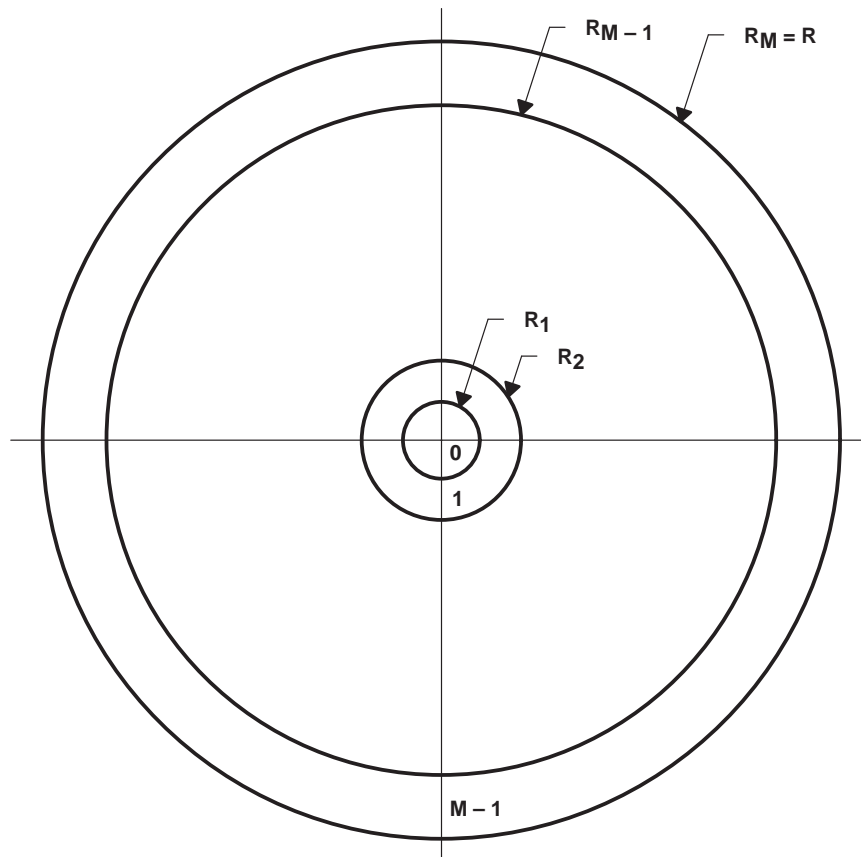
Figure VIII–3.   Concentric Shaping Rings

The M concentric circles form shaping rings labelled 0, 1, ... , M–1. The radii R1, ... , R_M are selected so the rings all have the same area. Thus, the area of ring 0 must be

$$A_0 = \pi R_1^2 = \pi R^2 / M \tag{VIII–57}$$

so

$$R_1^2 = R^2 / M \tag{VIII–58}$$

The area enclosed by the outer circle of ring k is

$$A_k = \pi R_{K+1}^2 = (k+1) A_0 = (k+1) \pi R_1^2 \qquad \text{(VIII–59)}$$

so

$$R_{K+1}^2 = (k+1) R_1^2 = (k+1) R^2 / M \qquad \text{(VIII–60)}$$

The average power of ring k is

$$P_k = \int_{R_k}^{R_{k+1}} r^2 \frac{dA}{A_k} = \int_{R_k}^{R_{k+1}} r^2 \frac{2\pi r\ dr}{\pi \left(R_{k+1}^2 - R_k^2 \right)}$$

(VIII–61)

$$= \frac{R_{k+1}^4 - R_k^4}{2 \left(R_{k+1}^2 - R_k^2 \right)} = \frac{R_{k+1}^2 + R_k^2}{2}$$

On using (VIII–60) this reduces to

$$P_k = \frac{R^2}{M}\ (k + 0.5)$$

(VIII–62)

The average power in a ring block $r = [r_1, r_2, ..., r_N]$ is

$$P(r) = \sum_{k=1}^{N} P_{r_k} = \frac{R^2}{M} \left(0.5N + \sum_{k=1}^{N} r_k \right)$$

(VIII–63)

Thus, any ring sequence with the same sum has the same average power. This justifies using the sum of the ring values as a weight function.

Appendix B.

Selected excerpts from Dr. Steven A. Tretter's "Fundamentals of Trellis Shaping and Precoding" on the subject of obtaining the inputs to the convolutional encoder. Used with permission.

A.30   A Method for Determining the Binary Subset Label From the Coordinates of a 2D Point

The combined precoding and trellis coding scheme for V.34 will be presented in Chapter X. Finding the input to the trellis encoder involves finding the subset binary labels for 2D constellation points. A simple method for finding these labels will now be presented.

Let (x,y) be a 2D constellation point. First translate this point to a lattice point by forming

$$(x_0, y_0) \; = \; (x, y) - (0.5, 0.5) \tag{IX-9}$$

According to (IX–8)

$$(x_0, y_0) \in 2RZ^2 + \left[J_2 \oplus (J_0 \cdot \bar{J}_1) \right] (2, 0) + J_1(1, 1) + J_0(0, 1) \tag{IX-10}$$

The sum of the components of any point in $2RZ^2$ is some even number $2n$. Thus,
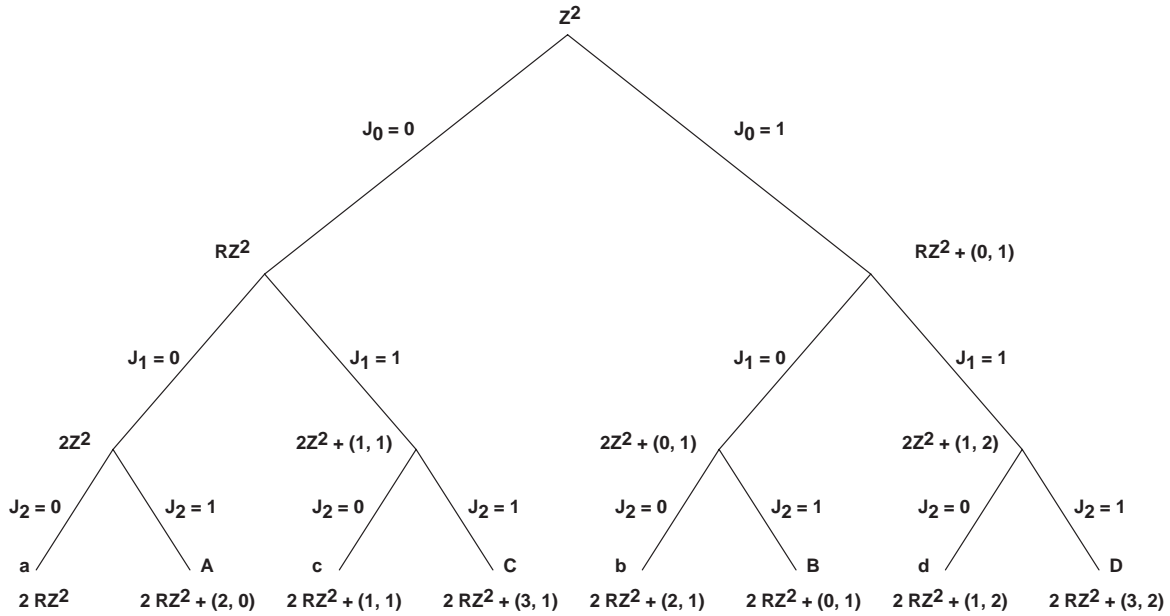
$$x_0 + y_0 \; = \; 2n + 2\left[J_2 \oplus (J_0 \cdot \bar{J}_1) \right] + 2J_1 + J_0 \tag{IX-11}$$

This sum is even if $J_0 = 0$ and is odd if $J_0 = 1$. Let the function mod(*a,b*) be defined to be

$$\text{mod(a, b) = remainder when a is divided by b} \tag{IX-12}$$

Then the rule for finding J_0 is

$$J_0 \; = \; \text{mod}(x_0 + y_0, \; 2) \tag{IX-13}$$

Z^2

$J_0 = 0$ $J_0 = 1$

RZ^2 $RZ^2 + (0, 1)$

$J_1 = 0$ $J_1 = 1$ $J_1 = 0$ $J_1 = 1$

$2Z^2$ $2Z^2 + (1, 1)$ $2Z^2 + (0, 1)$ $2Z^2 + (1, 2)$

$J_2 = 0$ $J_2 = 1$ $J_2 = 0$ $J_2 = 1$ $J_2 = 0$ $J_2 = 1$ $J_2 = 0$ $J_2 = 1$

a   A   c   C   b   B   d   D

$2RZ^2$ $2RZ^2 + (2, 0)$ $2RZ^2 + (1, 1)$ $2RZ^2 + (3, 1)$ $2RZ^2 + (2, 1)$ $2RZ^2 + (0, 1)$ $2RZ^2 + (1, 2)$ $2RZ^2 + (3, 2)$

Figure IX–3.   The 2D Partition Tree

The x component of any point in $2RZ^2$ must be some even integer $2n_1$. Therefore, it follows from (IX–10) that

$$x_0 = 2n_1 + 2\left[J_2 \oplus \left(J_0 \cdot \bar{J}_1 \right) \right] + J_1 \tag{IX-14}$$

which is even or odd depending on whether J_1 is even or odd. Therefore,

$$J_1 = \mathrm{mod}\left(x_0,\ 2 \right) = \text{lsb of } x_0 \tag{IX-15}$$

Once J_0 and J_1 have been determined, their effects can be subtracted out to form

$$\left(x_1,\ y_1 \right) = \frac{\left(x_0,\ y_0 \right) - J_1 (1, 1) - J_0 (0, 1)}{2} \tag{IX-16}$$

$$\in \frac{2RZ^2 + \left[J_2 \oplus \left(J_0 \cdot \bar{J}_1 \right) \right] (2, 0)}{2} = RZ^2 + \left[J_2 \oplus \left(J_0 \cdot \bar{J}_1 \right) \right] (1, 0)$$

The sum of the coordinates of any point in RZ^2 must be some even number $2n_3$. Therefore,

$$x_1 + y_1 = 2n_3 + \left[J_2 \oplus \left(J_0 \cdot \bar{J}_1 \right) \right] \tag{IX-17}$$

so

$$\left[J_2 \oplus \left(J_0 \cdot \overline{J}_1\right)\right] \ = \ \mathrm{mod}\!\left(x_1 + y_1, \ 2\right) \qquad\qquad\qquad (\mathtt{IX-18})$$

and

$$J_2 \ = \ \mathrm{mod}\!\left(x_1 + y_1, \ 2\right) \oplus \left(J_0 \cdot \overline{J}_1\right) \qquad\qquad\qquad (\mathtt{IX-19})$$