SOFTWARE IMPLEMENTATION OF VITERBI
MAXIMUM-LIKELIHOOD DECODING

A Thesis Presented to The
Faculty of the College of Engineering and Technology
of Ohio University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by

Caonabo Almonte

September 1981

OHIO UNIVERSITY
LIBRARY
ACKNOWLEDGEMENTS

I would like to take this opportunity to thank all those that have made this research work possible. Specially, to Dr. Robert W. Lilley for the long hours that we spent together working in the simulator. Also, I would like to thank Mr. Rick Turner for his help and support. To the Wright-Patterson Air Force Base for supporting my research work here at Ohio University I owe a great deal. Last but not least, to Dr. Joseph Essman, for his patience and understanding in finishing this work.

Special thanks go to my friends, Dr. Raymond Luebbers, Mr. Lawrence Thorn, and Mr. Paul Lee Ewing for their constant support.

I dedicate this thesis to my father and mother for their unending love, understanding, and motivation.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Acknowledgements</td>
<td>iii</td>
</tr>
<tr>
<td></td>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>I.</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II.</td>
<td>CONVOLUTIONAL ENCODING</td>
<td>10</td>
</tr>
<tr>
<td>2.0.</td>
<td>Introduction</td>
<td>10</td>
</tr>
<tr>
<td>2.1.</td>
<td>Coding Trees</td>
<td>13</td>
</tr>
<tr>
<td>2.2.</td>
<td>Software Design of a Convolutional Encoder</td>
<td>20</td>
</tr>
<tr>
<td>III.</td>
<td>VITERBI DECODING</td>
<td>24</td>
</tr>
<tr>
<td>3.0.</td>
<td>Introduction</td>
<td>24</td>
</tr>
<tr>
<td>3.1.</td>
<td>General Principles of Viterbi Decoding</td>
<td>24</td>
</tr>
<tr>
<td>3.2.</td>
<td>Example of Viterbi Decoding</td>
<td>27</td>
</tr>
<tr>
<td>3.3.</td>
<td>Computer Algorithm</td>
<td>33</td>
</tr>
<tr>
<td>3.4.</td>
<td>On the Optimality of the Viterbi Algorithm</td>
<td>38</td>
</tr>
<tr>
<td>3.5.</td>
<td>Software Design of a Viterbi Decoder</td>
<td>39</td>
</tr>
<tr>
<td>3.6.</td>
<td>Conclusions</td>
<td>45</td>
</tr>
<tr>
<td>3.7.</td>
<td>Trellis Tree Generation</td>
<td>46</td>
</tr>
<tr>
<td>IV.</td>
<td>CODE SELECTION</td>
<td>51</td>
</tr>
<tr>
<td>4.0.</td>
<td>Distance Properties of Convolutional Codes</td>
<td>51</td>
</tr>
<tr>
<td>4.1.</td>
<td>Minimum Distance Computation and Error Correction</td>
<td>51</td>
</tr>
<tr>
<td>4.2.</td>
<td>Systematic and Nonsystematic Convolutional Codes</td>
<td>55</td>
</tr>
<tr>
<td>4.3.</td>
<td>Code Selection</td>
<td>58</td>
</tr>
<tr>
<td>4.4.</td>
<td>Path Memory Truncation</td>
<td>61</td>
</tr>
<tr>
<td>4.5.</td>
<td>Known Starting State</td>
<td>62</td>
</tr>
<tr>
<td>V.</td>
<td>THEORETICAL AND EXPERIMENTAL RESULTS</td>
<td>64</td>
</tr>
<tr>
<td>5.1.</td>
<td>Bitchange Model of a Channel</td>
<td>64</td>
</tr>
<tr>
<td>5.2.</td>
<td>Experimental Results</td>
<td>73</td>
</tr>
<tr>
<td>VI.</td>
<td>CONCLUSIONS AND RECOMMENDATIONS</td>
<td>88</td>
</tr>
<tr>
<td>VII.</td>
<td>REFERENCES</td>
<td>90</td>
</tr>
<tr>
<td>VIII.</td>
<td>APPENDIX (Fortran Program Listings)</td>
<td>92</td>
</tr>
<tr>
<td>Figure 1.1.</td>
<td>Simulator Block Diagram.</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>Figure 2.1.</td>
<td>Rate b/v, Constraint Length K Convolutional Encoder.</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2.2.</td>
<td>Rate 1/2, K=4 Convolutional Encoder.</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2.3.</td>
<td>Tree Diagram—K=4, R=1/2 Convolutional Encoder.</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2.4.</td>
<td>Trellis Tree - K=4, R=1/2 Encoder.</td>
<td>14</td>
</tr>
<tr>
<td>Figure 2.5.</td>
<td>State Diagram K=4, R=1/2 Convolutional Encoder.</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2.6.</td>
<td>Trellis Tree Showing Encoded Message Path.</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2.7.</td>
<td>Bit Number Arrangement.</td>
<td>19</td>
</tr>
<tr>
<td>Figure 2.8.</td>
<td>Shift Register Set Up.</td>
<td>21</td>
</tr>
<tr>
<td>Figure 2.9.</td>
<td>R=1/2, K=4 Convolutional Encoder Flow Chart.</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.1.</td>
<td>Binary Symmetric Channel.</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3.2.</td>
<td>Trellis Tree for K=4, R=1/2 Viterbi Decoder.</td>
<td>25</td>
</tr>
<tr>
<td>Figure 3.3.</td>
<td>Trellis Tree Showing Encoded Message Path.</td>
<td>28</td>
</tr>
<tr>
<td>Figure 3.4.</td>
<td>Viterbi Decoded Trellis Tree Showing Maximum-Likelihood Path.</td>
<td>29</td>
</tr>
<tr>
<td>Figure 3.5.</td>
<td>Sample Computer Output for Example of Figure 3.4.</td>
<td>30</td>
</tr>
<tr>
<td>Figure 3.5a.</td>
<td>Example of Bubble Sort.</td>
<td>34</td>
</tr>
<tr>
<td>Figure 3.6.</td>
<td>Trellis Chunk for K=4, R=1/2 Convolutional Encoder - Viterbi Decoder.</td>
<td>35</td>
</tr>
<tr>
<td>Figure 3.7.</td>
<td>Two Paths on L-Step Trellis.</td>
<td>36</td>
</tr>
<tr>
<td>Figure 3.8.</td>
<td>Flow Chart for Bubble Sort.</td>
<td>39</td>
</tr>
<tr>
<td>Figure 3.9.</td>
<td>Viterbi Decoding Flow Chart.</td>
<td>43</td>
</tr>
<tr>
<td>Figure 3.10.</td>
<td>Trellis 'Chunk' for K=4, R=1/2 Convolutional Encoder - Viterbi Decoder.</td>
<td>47</td>
</tr>
<tr>
<td>Figure 4.1.</td>
<td>Trellis Tree Showing Three Distinct Paths for Calculating Minimum 'Free' Distance.</td>
<td>50</td>
</tr>
</tbody>
</table>
Figure 4.2. Trellis Tree for K=4 R=1/2 Viterbi Decoder Showing Path Numbers.

Figure 4.3. Comparison of Systematic and Nonsystematic R=1/2 Code Distances.

Figure 4.4. Coder that Exhibits Catastrophic Error Propagation.

Figure 4.5. Partial Trellis Showing Equivalent Paths.

Figure 5.1. BSC Channel.

Figure 5.2. Flow Chart Bitchan Module.

Figure 5.3. Comparisons Between IBM (ERFC) Routine and Locally Generated Function (CERFNC).

Figure 5.4. Theoretical FSK Curve.

Figure 5.5. Theoretical PSK Curve.

Figure 5.6. Theoretical DPSK Curve.

Figure 5.7. Viterbi Theoretical Upper Bound.

Figure 5.8. Simulator Configuration for Obtaining Experimental Results.

Figure 5.8a. Sample "SMD" File to Generate Experimental Results.

Figure 5.9. K=3 R=1/2 Viterbi Decoder Performance Curve (PSK Channel).

Figure 5.10. K=4 R=2/3 Viterbi Decoder Performance Curve (PSK Channel).

Figure 5.11. K=4 R=1/2 Viterbi Decoder Performance Curve (PSK Channel).

Figure 5.12. K=5 R=1/2 Viterbi Decoder Performance Curve (PSK Channel).

Figure 5.13. K=6 R=1/2 Viterbi Decoder Performance Curve (PSK Channel).

Figure 5.14. K=3 R=1/2 Viterbi Decoder Performance Curve (FSK Channel).
Figure 5.15. $K=4$ $r=2/3$ Viterbi Decoder Performance Curve (FSK Channel).

Figure 5.16. $K=4$ $r=1/2$ Viterbi Decoder Performance Curve (FSK Channel).

Figure 5.17. $K=5$ $r=1/2$ Viterbi Decoder Performance Curve (FSK Channel).

Figure 5.18. $K=6$ $r=1/2$ Viterbi Decoder Performance Curve (FSK Channel).
| Table 3.1 | Table of Modulo-2 Addition. | 26 |
| Table 3.2 | Trace Back Table. | 44 |
| Table 3.3 | Table of Transition Values. | 48 |
I. INTRODUCTION

Unlike typical engineering developments, coding theory had as its beginning a very curious phenomenon. First came the ultimate theorems, and the applications that explored the theory came later. In 1948, C.E. Shannon first proposed the basic theorems. The problem with these is that they are existence theorems. That is, he showed that within the many ways of coding, it was possible to code data so that one could attain arbitrarily low error rates, so long as the rate of transmission did not exceed the channel capacity. However, the theorems provided the applications people with no clues as to how this theoretical limit could be reached.

There was also another basic problem. The channels that were being dealt with at the time, such as telephone lines, cable, etc., did not resemble the channel assumed in theory. This channel assumed statistical independence in the noise from symbol to symbol. In most of the channels being used at the time the noise disturbances occurred in bursts extending across many consecutive bits. Many other anomalies do arise in practice from other sources of electrical noise adding to the problem of accurately modeling these channels.

There have been two major developments that have resulted in the increased interest in coding theory. First of all, coding schemes such as block coding and convolutional coding have been developed. Their main advantage is that sufficiently simple decoding algorithms have been developed for them. The latter include algebraic structures for block types and sequential decoding for convolutional codes. An outgrowth, or
perhaps, a slight improvement on this last algorithm is the algorithm first presented by A.J. Viterbi in 1967 [1]. The second development is the obvious communications need that arose due to deep-space probes and orbiting satellites. It was found that the space and satellite channels are quite close to the statistical model first used to develop the original theorems of information theory. Furthermore these channels are statistically regular, of nonbursty type noise, and consequently, exactly what the original information theory theorems addressed. The space channel and satellite channels lend themselves to accurate predictions, detailed computer simulations, and, of course, are ideal for a project such as the WPAFB/OU Communications Simulator.

The starting point for this thesis work was a library search for pertinent materials. From it selected articles were collected that have contributed greatly to the overall development of this thesis, and in fact constitute the very foundation upon which all the work was done. A very broad introduction to coding theory and its beginnings was provided by G.D. Forney [8] in an article published in the IEEE Spectrum, "Coding Theory and its Application in Space Communication". Here Forney explains the beginnings of coding theory and its practical applications in the age of digital space communications. Fano [4] in his "A Heuristic Discussion of Probabilistic Decoding" [4], provides a good explanation of sequential decoding. This is important because Viterbi decoding is a slight improvement on sequential decoding. The principles explained in Fano's article can serve as a source of intuitive knowledge which helps to understand the overall simplification of computation that constitutes the Viterbi algorithm.
Viterbi's [1] first article, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm", provides the theoretical background upon which the algorithm is based. It provides theoretical bounds to probability of error from both above and below. In his second article [2] on the subject, "Convolutional Codes and their Performance in Communication Systems", Viterbi provides a tutorial approach to the basic properties of convolutional codes and provides examples of maximum likelihood decoders. These were used to develop the first two sets of working modules for the simulator. These two articles are the backbone of all the work reported in this thesis.

J. A. Heller and I. M. Jacobs [13] in their article, "Viterbi Decoding for Satellite and Space Communication", present examples of efficient systems on power limited and space channels. They also show the effect of hard vs. soft quantization of the received signal as well as an actually implemented system of constraint length 7. Finally, they compare Viterbi and sequential decoding, both algorithms for decoding convolutional codes. Other hardware implementations were found in an article by G.C. Clark,Jr., and R.C. Davis [7]. Its title implies its content, "Two Recent Applications of Error-Correction Coding to Communications Systems Design." Both of these articles were helpful in providing the author with a measure of realism, i.e., a feel for actual hardware implementations. This was invaluable in later decisions on implementations on the simulator itself.

There are three articles, however, that contributed the most to the actual implementation of the Viterbi algorithm in the simulator.
Listed in order of increasing importance, first, "The Viterbi Algorithm" by G.D. Forney, Jr. [11], which provided the author with a basic structure of the algorithm in detail. Second in importance is, "Error Protection Manual for AFCS," [12] Vol. II, by P.J. Trafton, N.F. McAllister, B.P. Tunstall, J.C. Elliot, N. Brienza, H.I. Paul, and P.W. Fox, published 30 November, 1972. The examples presented in the latter included challenging explanations and helpful hints as to theory and implementation, but in a more straight theoretical treatises. The third one is, "Error Control Coding Handbook (Final Report)," by J.P. Odenwalder, [10] published 15 July 1976. It is by far the most straightforward approach to convolutional codes, Viterbi decoding, and their performance. This last report is, in the opinion of the author, the most authoritative single article on the subject of the Viterbi algorithm, and at the same time, is relatively easy to understand. It also provided a guide to choosing 'good' convolutional codes, which was invaluable in this project. All of this latter work, of course, was based on what Viterbi proposed in 1967 in his first article on the subject.

But besides its theoretical lucidity in describing the space and satellite channels, one may ask, what is the practical use of coding? The answer is that coding is a cost-effective way to improve communication efficiency. When we deal with communications, such as those we are trying to model, between a far away object and a ground station, certain unique problems arise. The most important of these, practically speaking, is the relatively high cost of power at the space station in terms of size and cost to put that payload in space. For this reason we would, ideally, want to minimize the spacecraft's weight.
The efficiency of a communication system is measured by the energy/bit to noise ratio--(Eb/No)--that is required to achieve a certain probability of error. This can be expressed in terms of modulating signal power by:

\[
\frac{Eb}{No} = \frac{P}{No R}
\]  

(1)

where \(R\) = rate of information in bits/sec  
\(P\) = transmitter power in watts  
\(Eb\) = energy/bit in watts/Hz/bit  
\(No\) = single-sided noise spectral density in watts/Hz

Rearranging terms to get \(R\),

\[
R = \frac{P/No}{Eb/No}
\]  

(2)

Thus, by reducing \(Eb/No\), we can increase the rate of transmission, or the amount of power \(P\) can be reduced to accommodate a desired rate \(R\), with a given probability of error.

For the space channel, it is assumed that it is power limited and not bandwidth limited. More and more, however, the trend in space communications is towards higher \(P/No\) with the same bandwidth requirements. This is one of the reasons for choosing code rates of 2 or less in the simulator routines. Also, the latter seem to provide excellent results with medium decoder complexity. The advantage of these code rates is explained by Heller et al. [13]. Coding, in general, reduces the required
Eb/No for a particular bit error probability (see eq. (2)). Thus, it increases communication efficiency at the expense of bandwidth and computation time.

The WPAFB/OU Communications Simulator is an attempt to model diverse communication techniques by the use of FORTRAN routines combined with operating and programming procedures. Its main advantage is its flexibility in accommodating different communication system configurations. The simulator is written in such a way as to provide timing, data flow and user commands. The user may monitor the I/O of any block that he chooses, as the simulator is running (on the terminal), and he may obtain a copy of the results if he so desires. He may also plot certain simulator parameters. Figure 1.1 shows the overall system architecture. This is the execution order for the configuration at run time. I/O follows the lines shown in this diagram as a general rule. However, the user may alter this at any time.

All the research and implementation performed in this thesis pertains to block 6, channel encoder block, block 9, bit-changing module, and block 23, channel decoder. These may be used to encode any digital source originating to the left of block 6 in the diagram (see Figure 1.1). The channel encoder block contains the convolutional encoding capability. The user may choose between different constraint length codes at configuration time. The bitchanging module was designed to simulate the effects of varying (Eb/No) values on the overall probability of error performance of these modules. The channel decoder capability consists of the Viterbi Maximum-likelihood decoders that match in rate
Figure 1.1. Simulator Block Diagram.
and size the before-mentioned encoders. It must be emphasized that there are two alternate routes that model a channel. The simplest is the bit-changing routine of block 9. This is where theoretical P(e) curves for FSK, PSK, and DPSK modulation techniques are simulated. The other route is through the 'regular' channel, i.e., through those blocks that use modulator, noise generator, demodulator, etc. The latter is more realistic since we are then using actual modulator, demodulator models, etc., but both methods are comparable in generating channel models that resemble those encountered in a real-time communication system.

The intent of these schemes, and of this thesis consequently, is to generate encoding-decoding procedures that ensure good P(e) results, and, therefore, accurately simulate actual convolutional encoding and Viterbi decoding techniques. By using these techniques, it is the intent of the author to be able to encode a string of bits (generally 1-24 bits in size), and after passing them through the channel, to derive those same bits at the other end for further processing if so desired.

The most outstanding achievement of this research work is that now any arbitrary convolutional encoding, Viterbi decoding scheme can be implemented in software, with a minimum of programming time delay, by using a few basic principles of convolutional coding-Viterbi decoding theory. By following the examples expounded in this thesis, it should be quite easy to expand the simulator capability in designing and implementing new and more complex versions of these techniques. It may also be possible to use these techniques in hardware implementations although that was not the aim at the outset. Simulation results for all modules
have been compiled to show this technique's superior bit error rate performance over uncoded PSK, and FSK.
II. CONVOLUTIONAL ENCODING

2.0 Introduction

In general, a rate = b/v, K constraint length, convolutional encoder is a b-input, v-output device which has K binary shift register stages and linear logic, usually modulo-2 addition (see Figure 2.1). The outputs v depend on the K variables of which b are the current inputs. The constraint length K is defined as the total number of binary register stages in the convolutional encoder. A convolutional encoder is so called because the encoded symbols do not resemble the original data.

All the work done for the Wright-Patterson project consisted of rates 1/2 and 2/3. From the literature it was realized that if such rates were used, we would not only get good codes (i.e., codes with good distance properties, see Chapter IV), but we would get the best codes discovered so far [13], [10]. In order to describe the basic convolutional coding concepts, one particular example, rate=1/2, K=4 convolutional encoder will be described (Figure 2.2). This coder is somewhat more complex than the one that appears frequently in the literature (rate=1/2, K=3). However, the basic software design procedure is identical, and a progression from a simple design to more complex ones can then be shown. The block diagram for this coder is shown in Figure 2.2, and the equations describing its behavior are:

\[ v_1 = s_1 \oplus s_2 \oplus s_3 \oplus s_4 \]

\[ v_2 = s_1 \oplus s_3 \oplus s_4 \]
Constraint Length = $K = K_1 + K_2 + \cdots + K_b$

Figure 2.1. Rate $b/v$, Constraint Length $K$ Convolutional Encoder.
Figure 2.2. Rate 1/2, K = 4, Convolutional Encoder.
2.1 Coding Trees

Traditionally, the behavior of a convolutional code is exhibited by a tree diagram. Figure 2.3 shows a drawing of the diagram for this sample encoder (K=4, rate=1/2).

The way to interpret this diagram is that if the first input is a 0, then the encoder moves up on the upper branch. If a 1 is input, then the encoder moves on the lower branch. The symbols shown on each branch are those corresponding to each input. In the same manner, if the second bit is a 0, the encoder moves on the upper branch, and if it is a 1 on the lower branch, and so on for each step. The thirty two possible outputs for the first five inputs are shown in the diagram.

There are a few very interesting conclusions that can be drawn from the tree diagram. From it we can see that after four branches, the tree structure becomes repetitive. In general, this will occur for any encoder after K branches (where K=constraint length). The reason for this is quite apparent after some thought. As the fifth data bit enters the coder, the first data bit drops off the left side of the shift register. It can also be seen from the tree diagram that the nodes labeled 'a', both have the same branch symbols emanating from it. Therefore, they can be joined together. It is also important to notice that both nodes labeled 'a' have branch symbols which are complements of each other.

Having redrawn the tree diagram in the manner prescribed above we get a structure called the trellis diagram (Figure 2.4), so called
Figure 2.3. Tree Diagram—K=4, R=1/2 Convolutional Encoder.
Figure 2.4. Trellis Tree - $K = 4$, $R = 1/2$ Encoder.
because a 'trellis is a tree-like structure with remerging branches' [2]. The most often used convention for the latter is that a 'zero' input to the coder is represented by solid lines, whereas a 'one' input is shown by dashed lines. This convention may vary, however, from reference to reference.

The trellis tree itself is also highly repetitive in nature and can be further reduced to a state diagram representation of the coder (see Figure 2.5). The so called states are labeled according to the nodes of the trellis diagram. The states correspond essentially to the last three input bits to the coder. Thus, these bits are used to denote the states of the state diagram. The one advantage that is directly apparent from studying both these representations is that the trellis tree can keep track of time. The state diagram is a more concise representation, but for our purposes the trellis representation is better (see Figure 2.4).

It must also be remembered that the tree diagram is used to determine all the different combinations of inputs that the coder goes through and the output symbols generated. It is actually a visual description of all the possible outputs of the encoder when affected by data symbols. In this way it is a visual tabulation of all those events. But, as seen from this example, the tree itself becomes repetitive, and some of the calculations are duplicated in its generation. For more complicated designs, a more straightforward method of generating the trellis tree has been devised based on the finite state properties of the convolutional encoder. This saves both time and the possibility of error from mindless
Figure 2.5. State Diagram $K=4$ $R=1/2$ Convolutional Encoder.
repetition of calculations that are not really necessary (see Chapter III, Table 3.3 for details).

From either the state diagram or the trellis tree, certain properties are apparent.

1). There are always 2 paths leaving and entering each state. This is because the next data bit can only be a 1 or a 0.

2). The symbols of each path leaving (or entering) each state are always complements of each other. This gives good distance properties.

3). It is very important to note that as the next bit is shifted in, the 'old' MSB is dropped off the shift register and no longer affects the outcome of the encoder.

Let us now take a sample input sequence, for example, 10110100...
Starting from state 000 in the trellis diagram, the encoder travels along the dashed line into state 001 and generates the code symbols 11 (see Figure 2.6). From state 001, the encoder travels to state 010 thru the solid line with symbols 10 generated. Then via the dashed line to state 101 with output symbols 00. Next, from state 101 to state 011 via the dashed line with output symbols 10. And then, via the solid line from state 011 to state 110, generating the output symbols 01. Next, via the dashed line from state 110 to state 101 generating the symbols 11. Then, from the state 101 to state 010 via the solid line generating the symbols 01. Lastly, from state 010 to state 100 the encoder travels the solid line generating the output symbols 11. (Note: it should be remembered that solid lines denote a 0 input whereas a dashed line denotes a 1
Figure 2.6. Trellis Tree Showing Encoded Message Path.
input). Demultiplexing all these symbols we get the encoded sequence 1110001001110111. Notice that this sequence bears no resemblance to the original data sequence. The decoder must accept this output sequence, with some bits changed perhaps, and must derive from it the original data sequence. To do this the decoder must trace all possible paths thru the trellis and come up with the right path as its most likely one.

2.2 Software Design of a Convolutional Encoder

The core of the software design of the convolutional encoder is:

1). I/O section

2). Shift register section

3). Modulo-2 arithmetic section.

The I/O section of the convolutional encoder takes advantage of two subroutines available in the simulator library, PBIT/GBIT. These routines are used to 'Put' and 'Get' bits from array storage. The general form is:

CALL PBIT(IARRAY,IBNO,IBIT)
IARRAY=THE BIT STORAGE ARRAY NAME
IBNO=BIT NUMBER IN THE ARRAY
IBIT=THE BIT VALUE (1 OR 0)

In the same manner,
CALL GBIT(IARRAY,IBNO,IBIT)

Subroutines PBIT/GBIT pack and retrieve individual bits, up to 24 in number per hardware word. The order of the bits is 1 thru 24 from
right to left as shown in Figure 2.7. Calling PBIT places a bit into the bit slot number provided by IBNO. In all the work done for this thesis the bits are retrieved and packed sequentially starting with bit 1. PBIT/GBIT routines adjust bit 25 in IARRAY(2) automatically. The main advantage of these routines is that they allow data flow efficiency by passing 24 or more bits per run time, instead of one bit per run time. This amounts to a considerable savings in computer time [18].

Figure 2.7. Bit Number Arrangement.

It must be noted that IARRAY must be dimensioned, and is cleared at the beginning of the program to avoid erroneous results.

This encoder module can take up to 24 data bits and thus it codes up to 48 bits. The number of these may change each run time, and the software adjusts this value automatically for all the modules down stream.

The shift register section is accomplished by using one-dimension arrays. Also, there must be a way to shift the bits left one place. For example, for the shift register of the coder being discussed, we must shift all bits left one place and then input the new data bit.
Because of the apparent pattern, one can generalize the above using a DO loop as shown in the program listing in the appendix.

For the modulo-2 arithmetic section, it was found that by using the 'MOD' function available in FORTRAN, the software work would be cut down significantly. An alternate route was to use logical IF statements to accomplish the equivalent EXOR operations. This method did not prove to be as efficient though.

The logical set up for the CHEN14 module is provided in the next page via a flowchart (Figure 2.9). The latter pertains to the 'Simulation Run' part of this module. The documentation, common block, etc. are standard for all simulator software design [18].
Figure 2.9. $R = 1/2$, $K = 4$ Convolutional Encoder Flow Chart.
III. VITERBI DECODING

3.0 Introduction

"The Viterbi Algorithm is a solution to the problem of maximum a posteriori estimation of the state sequence of a finite-state discrete time Markov process observed in memoryless noise" [11]. To put it more simply, the Viterbi algorithm resolves the problem of finding the highest probability that a particular path thru the trellis was taken. The algorithm takes advantage of the remerging structure of convolutional codes. The highest conditional probability means the shortest length route thru the map (trellis). Most of the work done in the WPAFB/OU Simulator was done with hard quantization; i.e., bits entered into the algorithm were already considered individual 1's and 0's instead of preprocessing (into levels) thru soft quantization. This was done to take advantage of the particular I/O requirements of the simulator (packing 24 bits/IBM word).

3.1 General Principles of Viterbi Decoding

In order to discuss Viterbi decoding of convolutional codes, it is very important to clarify the corresponding nomenclature. First of all, one must talk about the "Binary Symmetric Channel." In this type of channel, a transmitted symbol will be changed from 0-->1 or 1--->0 with a probability 'p', and these channel errors are introduced independently from symbol to symbol. This is important to know because later on we will see that this is the way that the BITCHANGE module is constructed, (see Chapter V).
Thus, if we input into the channel a certain number of encoded symbols $x_1, x_2, x_3, \ldots, x_j$ and on the receiver end we get $y_1, y_2, y_3, \ldots, y_j$ which have been corrupted by noise, then the decoder which derives the closest correct sequence is one which minimizes the "overall error probability." That is, the decoder that finds the sequence that is closest in the Hamming sense is the maximum-likelihood decoder. The Hamming sense refers to that sequence which is different from the received sequence in the minimum number of symbols. By this it is not meant that the algorithm should examine the entire sequence at once, which in some applications could be quite large, but that it should examine those paths that are present in the trellis diagram. And then it should choose that sequence which differs from the received sequence in the least number of symbols. This is equivalent to saying that the algorithm should choose the path thru the trellis which has the lowest value of cumulative distance.

As stated earlier the code trellis tree has one advantage over the state diagram representation. That is, it can keep track of discrete
time. At each step thru the trellis the algorithm must choose between the $2^{b}$ paths (in general) entering each state. The algorithm must also remember the value of that minimum distance which we will call "metric" (by convention). In conclusion, the Viterbi algorithm, at each step thru the trellis, must remember two things:

1). Which path out of the two (in this case) entered that state.
2). The value of cumulative state metric at that point.

The only problem arises when we have a tie into the particular state. In that case, one may flip a coin, as is done for block codes, since subsequent states will be affected equally by either choice. In an ideal situation, one would like to have this occur the minimum number of times. However, at times ties may be inevitable. This is a crucial point to remember when we talk about the choice of codes selected for this project. Essentially, that choice was made so that this event would not happen often (see Chapter IV).

The Hamming distance, which is the essence of the branch calculations, is most easily calculated by using modulo-2 addition as in Chapter II for the encoder section. The rationale behind this is obvious once one examines a table of modulo-2 addition. In this table "A" represents the symbol coming from the channel.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A $\oplus$ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.1. Table of Modulo-2 Addition.

(Note: Notice that when the symbols are different, in the second and third cases, the Hamming distance=1).
"B" represents the symbol in a hypothetical branch of the trellis tree. Since there are only 2 symbols being transmitted per data bit (rate=1/2), then there are only 4 different distances possible. Thus, the following calculations must be performed each time 2 new symbols are read:

\[
\begin{align*}
Y(1) + 0 & \quad Y(1) + 1 \\
Y(2) + 0 & \quad Y(2) + 1
\end{align*}
\]

By combining these calculations each branch metric can be calculated.

3.2. Example of Viterbi Decoding

To illustrate Viterbi decoding, a specific example will be explained and generalization of the technique will be presented following the example. A very valid question concerning the Viterbi Decoding algorithm is what must it do computationally speaking? In order to show this more concretely, the decoding mechanism will be explained in sufficient detail here by using the example of Chapter II. For convenience, Figures 2.5 and 2.6 are re-drawn here as Figures 3.2 and 3.3 respectively. Figure 3.2 is the complete trellis diagram (rules for generating the trellis diagram will be given later in Table 3.3) with all the paths shown. It is the intent here also to show that the path taken by the decoder is the same as the one shown in Figure 3.3, and, thus, show the decoding capability of the Viterbi algorithm.

First of all, let us analyze the forward calculations in Figure 3.4. Since we start from state 000 at discrete time \(k=0\), then there are only two paths that can be taken (see Chapter IV for explanation of known starting state). Path 1 generates a Hamming distance of 2 (Hamming
Figure 3.2. Trellis Tree for K = 4, R = 1/2 Viterbi Decoder.
Figure 3.3. Trellis Tree Showing Encoded Message Path.
Figure 3.4. Viterbi Decoded Trellis Tree Showing Maximum-Likelihood Path.
difference between transmitted values 11 and hypothetical branch values, 00). Thus at state 000, k=1 time the total state metric is equal to the starting state value + the Hamming distance in that branch (0+2)=2. For path 2 a similar calculation is made, except that in this instance the Hamming distance is 0 (0+0=0). Those two values are recorded in the diagram at k=1. For k=2, the received symbols are 10. For path 1, the Hamming distance is 1. Thus, the new state metric for state 000 at k=2 is equal to 3 (2+1). The same calculations are done for paths 2, 3, and 4. Notice that these are the only paths possible at this point. For k=3 discrete time, the same type of calculations are repeated, except for the fact that there are now 8 different states and 8 different paths. Only when we reach k=4 discrete time is there a significant difference in the behavior of the decoder. Let's examine state 000. It has 2 possible paths entering it. The first path is path 1. The total metric along this path is equal to the state metric at k=3, which happens to be 3 in this case, added to the Hamming distance between the received symbols and the hypothetical symbols in this branch (received=10, hypothetical=00, Hamming distance=1). Therefore, the total distance along this path is 4. The other possible path is path 9. It comes from state 100, and the value of state metric at k=3 is equal to 2. The Hamming distance along this branch is equal to 1 (received=10, hypothetical=11). Therefore, the total distance along this path is 3. According to the theory, the decoder must choose the path "which is closest in the Hamming sense" [2]; i.e., the path with the lowest value of cumulative metric. Therefore, the decoder chooses path 9, and must record the new state metric value at k=4, state 000 to be 3. The decoder must also record the path number for the "backward" calculations. This same procedure is repeated for all 8
states at this point in discrete time. Thus, we end up with 8 paths and 8 values of state metrics. For the next step in discrete time, k=5, the same procedure is repeated again with the same results, 8 paths out of 16 possible, and 8 new values of state (cumulative) metrics. This procedure is repeated until the received symbols are exhausted. At this point, the trellis tree must be retraced backwards in order to decode the most likely path.

Again, if one looks at the sample decoded trellis tree in Figure 3.4, the last column of state metrics has all the values at that point in time. The one with the smallest value of 0, is the lowest cumulative distance of all the paths, and, consequently, it is the most likely path. It ends at state 100, k=9, and the decoder could only have arrived there by the path shown. From Figure 3.2, we know that the transition that made the decoder enter that state had to be a "0". Stepping backwards thru the trellis this path leads us to state 010. The only state leading to that state comes from state 101. From Figure 3.2 we know that the only possible transition into that state had to be a "0". The only path leading to state 101 comes from state 110. From Figure 3.2, we know that the transition had to be a "1". This is repeated all the way back to state 001 at k=1 (It is obvious from Figure 3.4 that there is only one path leading all the way back to the beginning of the trellis tree).

Figure 3.2 can be used to decipher the bits originally transmitted. The decoded message is 10110100, which is the one originally encoded in Chapter II. In fact, the path traced in Figure 3.4 is exactly the same that appears in Figure 3.3 for the encoder. Essentially, thus, the decoding capability of the Viterbi algorithm has been shown.
3.3. **Computer Algorithm**

The question remains, how does the software accomplish all of this just described? Figure 3.5 shows a sample output for the rate=1/2, k=4 Viterbi decoder. In it are shown the "RMETRI" and "PATHNO" arrays. It must be remembered that this constitutes a complete history of all the calculations done by the decoder, and can be used for debugging purposes. Figure 3.5 is also the same example as the one in Chapter II. It is the intent here to show how the software works, and that it does indeed follow the general Viterbi decoding principles. For the first three steps (in general, for constraint length K-1 steps), the algorithm is restricted to picking paths 1 and 2 for k=1; 1,2,3,4 for k=2, and so on. The reasoning behind this is obvious from Figure 3.2 (the complete code trellis). At discrete time k=4 is where the comparisons start. As stated earlier in this example, first the total metric along each of the two possible paths is calculated, their value is then compared, and then a path and its total value are chosen. For instance, the paths leading to state 001 are path 2 and path 10 (see trellis tree "chunk", Figure 3.6 for the appropriate path number; more about the meaning of this 'chunk' later in this example). Along path 2 the total distance is equal to the state metric value at state 000, which is 3, added to the Hamming distance between the received symbols 10 and the hypothetical branch symbols, 1. This total distance is 4. Along path 10 the total distance is equal to the state metric value at state 100, which is 2, added to the Hamming distance along this branch, which is also 1 (total metric=3). Out of the two hypothetical branches, the decoder chooses path 10 since its total distance is 3. This now becomes the new state metric value at state 001, depth k=4. Notice from Figure 3.5 that the RMETRI array
JK = 5

**This is Rmetric array:**

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>3</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**This is Pathno array:**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>9</th>
<th>1</th>
<th>9</th>
<th>9</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>12</td>
<td>4</td>
<td>12</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>14</td>
<td>6</td>
<td>14</td>
<td>14</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>15</td>
<td>7</td>
<td>15</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>

**Decoded message:**

1 0 1 1 0 1 0 0

Figure 3.5. Sample Computer Output for Example of Figure 3.4.
contains the value 3 for the element RMETRI(2,4), and the path number recorded in PATHNO(2,4) is 10. This is exactly as it should be. The same calculation is repeated for each and every state up to state 111. It was mentioned in Chapter II that tree diagrams describing the behavior of a convolutional code are repetitive in nature. As can be seen from Figure 3.2, the trellis tree itself and its calculations are also highly repetitive. For this reason a total description of the code trellis can be given by simply taking one section of the total trellis tree, as is shown in Figure 3.6. The trellis tree "chunk" is the foundation of all Viterbi decoding designs for the Wright-Patterson project.

The calculations just mentioned constitute the "forward" calculations. Now let's illustrate how the software works to trace back thru the trellis tree. The last metric column contains a very important piece of information. And that is the smallest value of cumulative metric at that point in time. A "Bubble Sort" section was designed to decide on the smallest value in this column and to set the flag "JK" (see Figure 3.5a). This flag is the parameter needed to initialize the "trace back" section of the program. Actually, there are two very important pieces of information that determine the decoded transition. First the value of JK. This value points to the particular state being analyzed. Depending on that state, the decoder knows, for sure, the value of transition (transmitted bit) that got it to that point in the trellis. If one observes closely the trellis 'chunk' of Figure 3.6, it is apparent that both paths into, for example, state 001 denote a "1" transition. The same is true, for instance, for state 010, except that in this case the transitions were both a "0". From Figure 3.5 we know that the first value of JK provided by the "Bubble sort" is 5. The decoder
\begin{itemize}
\item \(\times\) Compare 1 vs. 2 - Fail test
\item \(\checkmark\) Compare 2 vs. 3 - Pass test
\item \(\checkmark\) Compare 2 vs. 4 - Pass test
\item \(\times\) Compare 2 vs. 5 - Fail test
\item \(\checkmark\) Compare 5 vs. 6 - Pass test
\item \(\checkmark\) Compare 5 vs. 7 - Pass test
\item \(\checkmark\) Compare 5 vs. 8 - Pass test
\end{itemize}

\(\therefore 5\) is smallest value metric \(\Rightarrow JK = 5\)

Figure 3.5a. Example of Bubble Sort.
Figure 3.6. Trellis Chunk for K=4, R=1/2 Convolutional Encoder–Viterbi Decoder.
immediately knows that the transition was a "0". The decoder then checks the element PATHNO(5,9) for its value. Since this value is a 5, then the decoder knows that the next value of JK should be a 3 (see the trace back Table 3.2 of Figure 3.6 for details). Since the pointer JK is now at state 3, the transition had to be a "0". Next the decoder checks PATHNO(3,8). Since its value is 11, then the decoder knows that JK had to be reset to 6. For JK=6, the corresponding transition had to have been a "1". And the next value checked is PATHNO(6,7). This is done until we get to the beginning of the trellis tree. As can be seen from Figure 3.3 the decoded message is 10110100, which is also the message originally sent. To sum up, the decoder software is tracing back through all the transitions backwards as one would do by hand, except that the software does this with numerical values. The decoded bits are then packed and the subroutine returns control to the simulator exec. It must be noted that this example is "noise-free", and thus no symbol errors were introduced. The same analysis applies when errors are introduced into the message. Experimental results (Chapter V) show the performance to be quite good even under those circumstances.

3.4. On the Optimality of the Viterbi Algorithm

The Viterbi algorithm (VA) is useful because it is optimum in the Hamming distance sense. The "VA" selects the most probable path given a particular received sequence. For any algorithm to be optimum, it must choose the path with the lowest cumulative Hamming distance out of the possible 2**L paths. Let us assume that we have 2 paths in a L step trellis tree. And in this tree we have 2 paths ACD and BCD that meet at the ith node. The incremental metrics from points C-->D on both paths are
Figure 3.7. Two Paths in L-step Trellis.

equivalent and thus have no point in the argument. However, at the ith node, if the algorithm were to choose AC over BC, then the incremental metrics at that point (i) for path AC must be lower than those of BC. The "VA" is constructed so that the lowest value of cumulative metric at each point is chosen, i.e., it takes the lowest value of incremental metric at each node of discrete time. Therefore, it could not discard the AC portion of path ACD. This is true for any value of i, that is, for each and every node along the trellis. At the conclusion of the forward calculations, the algorithm chooses the lowest cumulative metric path, and thus, it chooses the highest probability path. Therefore, we can conclude that the "VA" is optimum in this sense.

3.5. Software Design of a Viterbi Decoder

The following are the major subdivisions of a Viterbi decoder:

1). I/O section.

2). Hamming distance calculation section.

3). Trellis tree description of the decoder, i.e., metrics out of each state.

3a). (K-1) bits require only some states to be filled—have a section take care of that also.
4). Comparison of metrics, then choose smallest incremental value metric "into" each state. Record the path taken.

5). Short length message section.

6). "Bubble sort" section to find smallest value state metric in last column of the trellis map.

7). Trace-back section to work backwards thru the trellis map.

As in the encoder design, the I/O section takes advantage of PBIT/GBIT routines first mentioned in Chapter II. Also, the decoder accommodates a variable word length capability that can change each run time.

The Hamming distance calculation section works basically on the principles outlined at the beginning of this chapter, i.e., by using modulo-2 addition. This was found to be the simplest and most straightforward way to perform this calculation.

The next section is crucial to the successful design of a Viterbi decoder. This is the trellis tree description section of the decoder. A copy of the "chunk" of trellis used for calculations in the example of Chapter II is shown in Figure 3.6. What this means is that we take the trellis tree diagram that has been obtained from direct tabulation, and we incorporate its number of states, its number of branches and their Hamming distance value into the decoder. This is the section that varies from module to module. The reason is obvious once analyzed; each encoder has a different trellis tree description, so each decoder, if it is to decipher that code, must have the same trellis tree description as the
encoder. This is part of the "forward" calculations of the decoder. That is, first we create the entire trellis map forward and then we retrace thru the map backwards to decode the original message.

It must also be remembered that all decoder modules start at known state 0. Progressively the decoder branches out to all states after (K-l) bits (2 at a time, 2**b in general--see Figure 3.2). Since this approach has been taken, the software must reflect this feature. Later it was found that this method yields better, more accurate results when up to (K-l) bits were transmitted/run time.

Next, since there are two branches into each state, one of them must be discarded (the largest) and the smallest kept. To do this systematically arrays were introduced into the branch metric calculations. This allows the user a considerable savings in time and computation. Also, this section provides for the case that the metrics into each state are equal. When that occurs, a coin is tossed (by calling random number generator, 'ARAND'), as is done for code block words. This section also records the particular path taken. This is for future reference when the decoder makes the final decision on the most likely message sent.

Short length messages are taken care of in the following manner. As shown in the diagram on Figure 3.2, only a limited number of states is reached before the constraint length K (in this case 3 bits). In those instances, if the decoder knows that only those bits were transmitted, it adjusts accordingly and only checks that limited number of states. The remaining states not yet reached are "don't care" states and should not enter into the calculation. This method has proven quite satisfactory in actual simulator runs.
The "Bubble Sort" section is quite useful because it is highly portable from module to module. It compares, for instance, state 1 with every other state until a lower value of state metric is found. At that time it changes to this new state number and continues to compare the remaining states. At the end of the calculation, it has effectively checked every state metric against every other state metric, and it has set the flag "JK" with the number of the state where the smallest value state metric exists. Figure 3.8 shows a flowchart for this section. The trace back section then uses the information from the bubble sort for its own purposes. Higher degrees of decoder complexity caused by larger constraint lengths do not affect the "bubble sort" section greatly. By changing the parameter "NCOMPA" (see Appendix for program listing) only, we can accommodate larger and larger size decoders. This was thought to be quite necessary because the number of computations would greatly increase from module to module unless this feature was implemented (see Figure 3.5a for an example).

The trace back section was also an improvement in terms of portability. Its main feature relies on the fact that if we are on an even numbered state, then the transition must have been a 1. The opposite is true for odd numbered states. Also, after careful tabulation of 'next' state values going back thru the trellis showed a very distinctive pattern that is incorporated into the software. This simplifies the design of simpler Viterbi decoders, but more importantly, it facilitates the design of higher size decoders with a minimum time dedicated to this section. Again it is seen that a careful tabulation of results yielded a pattern that could be incorporated into the software and decrease the number of steps required as well as software designer time. This
FROM MAIN PROGRAM

INITIALIZE
i=1, j=2
(Compare 1st and 2nd Ele)

RMETRI(i, KNB) <
RMETRI(j, KNB?)

YES

Set JK
j=j+1
(Look at next one)

NO

DONE?

NO

i=j, j=j+1
Choose latter value
Look at next one
down Set JK

YES

Continue

Figure 3.8. Flow Chart Bubble Sort.
page contains a complete table of trace back values for the K=4, r=1/2 decoder. This table describes the sort of calculations that the decoder must go thru at each step of the trellis tree while tracing back thru the map. For example, if JK=2 at the kth step of the trellis, then the decoder knows immediately that the transmitted bit had to be a "1". Also, it knows to check the "PATHNO" array at the appropriate location for its value. If this value were 2, for instance, the decoder would reset the JK value to 1, and the same calculation would be repeated again (see Table 3.2). This is done until the decoder reaches the beginning of the trellis tree (see example at the beginning of this chapter for details).

TRACE BACK TABULATION FOR K=4 R=1/2 VITERBI DECODER.

For JK=1     ITRANS=0
   IF(PATHNO(1,J)=1----)SET NEXT JK=1
   IF(PATHNO(1,J)=9----)SET NEXT JK=5

For JK=2     ITRANS=1
   IF(PATHNO(2,J)=2----)SET NEXT JK=1
   IF(PATHNO(2,J)=10---)SET NEXT JK=5

For JK=3     ITRANS=0
   IF(PATHNO(3,J)=3---)SET NEXT JK=2
   IF(PATHNO(3,J)=11---)SET NEXT JK=6

For JK=4     ITRANS=1
   IF(PATHNO(4,J)=4---)SET NEXT JK=2
   IF(PATHNO(4,J)=12---)SET NEXT JK=6

For JK=5     ITRANS=0
   IF(PATHNO(5,J)=5---)SET NEXT JK=3
   IF(PATHNO(5,J)=13---)SET NEXT JK=7
FOR JK=6    ITRANS=1
    IF(PATHNO(6,J)=6---)SET NEXT JK=3
    IF(PATHNO(6,J)=14---)SET NEXT JK=7

FOR JK=7    ITRANS=0
    IF(PATHNO(7,J)=7---)SET NEXT JK=4
    IF(PATHNO(7,J)=15---)SET NEXT JK=8

FOR JK=8    ITRANS=1
    IF(PATHNO(8,J)=8---)SET NEXT JK=4
    IF(PATHNO(8,J)=16---)SET NEXT JK=8

Table 3.2. Trace Back Table.

PATTERNS FOR THIS TRELLIS TREE:

1). IF JK is odd----->itrans=0
   If JK is even----->itrans=1

2). Rules for comparison:
   IF(PATHNO(JK,J).EQ.JK)---next jk=present jk
   IF(PATHNO(JK,J).EQ.JK+8)---next jk=(present jk+4)

3). Difference in value between next JK's is 4.

4). Next step backwards thru the trellis is:
   a). 1/2 of present value+ .5---if JK is odd.
   b). 1/2 of present value--------if JK is even.

3.6. Conclusions

To conclude, in as far as it was possible, all sections were made quite portable. That is, with minor changes they could be lifted in
their entirety and re-used in more complex designs. However, it must be remembered that each decoder contains a trellis description that is quite different in size from the previous simpler decoder. This description cannot be generalized and has to be tailored to each distinct coder-decoder set. Clearly, the trellis section is the one that entails the highest amount of design time now. The others can be reutilized with minor changes to the software itself (such as the size of counters, the size of arrays and the like). This fact also simplifies the debugging procedures greatly, since this is the only section that changes significantly and the others are known to work correctly from the previous modules, then any problems arising in the decoder must be rooted here.

Figure 3.9 at the end of the Chapter shows a complete flowchart for the Viterbi decoding algorithm. This is provided to give a general overview of the logical procedure that the decoder must follow. It can be concluded then that the software design has been narrowed down to a simple tabulation of the particular trellis tree for a particular convolutional code (this is presented in the next section of this Chapter). All other computational sections can be obtained from the existing software and re-used for a new design with minor changes to the software.

3.7. Trellis Tree Generation

The following are some brief notes to the Viterbi designer on the generation of the trellis code, and-or state diagram representation for any convolutional encoder. In order to do this, the following steps must be taken:

1). Pick a coder with "good" distance properties and no "catastrophic error propagation" (see Chapter IV) [10].
Figure 3.9. Viterbi Decoding Flow Chart.
2). From the constraint length "K" the number of states can be calculated, $2^{(K-1)}$. From the number of bits, b, input at a time, the number of paths into each state can be calculated, $2^{*b}$.

3). Tabulate the transitions from state to state and their value. This should be done systematically by taking each state and inputting a 0, and then a 1. This procedure provides the value of the transitions and the value of the next state. After these tabulations are finished, the trellis tree, which is more useful than the state diagram in this case, can be drawn directly. The trellis tree is the foundation for the design of the Viterbi decoder at hand.

4). Utilize previous Viterbi decoder modules as models for the present design. The changes are mainly in the area which calculates the Hamming distance "out of" each state. Also, note that the size of the operating arrays may be too small and must be adjusted also.

To solidify these principles, the derivation of the trellis tree code for $k=4$, rate=1/2 Viterbi decoder is detailed here.

Step 1. Using reference [10], a rate 1/2 $k=4$ constraint length was picked (this is the same one as shown in Figure 2.2).

Step 2. The number of states=$2^{*(4-1)}=8$. The number of paths into each state=$2^{*1}=2$. 
Step 3. Tabulation of results:

<table>
<thead>
<tr>
<th>STATE</th>
<th>INPUT</th>
<th>'SR' STATUS</th>
<th>PRES.----&gt;NEXT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0000</td>
<td>000---&gt;000</td>
<td>00</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>0001</td>
<td>000---&gt;001</td>
<td>10</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>0010</td>
<td>001---&gt;010</td>
<td>10</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>0011</td>
<td>001---&gt;011</td>
<td>01</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>0100</td>
<td>010---&gt;100</td>
<td>11</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>0101</td>
<td>010---&gt;101</td>
<td>00</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>0110</td>
<td>011---&gt;110</td>
<td>01</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>0111</td>
<td>011---&gt;111</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1000</td>
<td>100---&gt;000</td>
<td>11</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1001</td>
<td>100---&gt;001</td>
<td>00</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>1010</td>
<td>101---&gt;010</td>
<td>01</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>1011</td>
<td>101---&gt;011</td>
<td>10</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>1100</td>
<td>110---&gt;100</td>
<td>00</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>1101</td>
<td>110---&gt;101</td>
<td>11</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>1110</td>
<td>111---&gt;110</td>
<td>10</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1111</td>
<td>111---&gt;111</td>
<td>01</td>
</tr>
</tbody>
</table>

Table 3.3. Table of Transition Values.

The results of this tabulation are again re-drawn here for convenience (Figure 3.10).

Step 4. This step must be done by the would-be designer. He may at this time also enter his own "flavor" to the software. This last step is intended as a guide, not as a rigid rule to be followed.
Figure 3.10. Trellis 'Chunk' for K=4, R=1/2 Convolutional Encoder - Viterbi Decoder.
IV. CODE SELECTION

4.0. Distance Properties of Convolutional Codes

By far the most important factor in choosing "good" convolutional codes is the state metric distance that is generated at the Viterbi decoder. This is so important because if this distance is relatively small, it will be quite difficult for the decoder, after a few steps of execution, to distinguish between the paths taken. This, of course, is critical, since if too many ties occur, then the decoder logic will simply be flipping coins to decide which path was taken into each state, and none of the inherent advantages of the Viterbi method will help to decode the original message. Fortunately a lot of research work has gone into finding codes with "good" distances [10].

4.1. Minimum Distance Calculation and Error Correction

It may be asked, "How do you compute the value of distance for an arbitrary code?" There is no loss of generality by computing the distance between the all zeros word and all other possible code words. This is the same distance obtained when any other specific code word is compared to all others. Let us take the example of Chapters II and III. Figure 2.5 is re-drawn here as Figure 4.1 to show the distance of each branch from the all zeros path. Figure 4.1 shows three paths of interest. In order to make the explanation of this calculation clearer, Figure 3.8 is re-drawn here as Figure 4.2 with some slight changes. Figure 4.2 has all the path numbers derived in an arbitrary fashion for ease of comparison between the paths. They represent all the possible paths starting at depth k=4. The first path of interest is the one
Figure 4.1. Trellis Tree Showing Three Distinct Paths for Calculating Minimum 'Free' Distance.
Figure 4.2. Trellis Tree for $R=1/2$, $K=4$ Viterbi Decoder Showing Path Numbers.
derived from the input sequence 111000. Through the trellis this path starts at state 000 and moves through path 2 (dashed line). At \( k=2 \) it travels through path 4 (dashed line). At \( k=3 \) it travels through path 8 (dashed line). At \( k=4 \) it travels path 15 (solid line). At \( k=5 \) it travels path 13 (solid line). Finally at \( k=6 \) it travels through path 9 (solid line) and rejoins the all zeros path (Note: solid lines represent a "0" bit being transmitted whereas dashed lines represent a "1"). The total Hamming difference between this message and an all zeros message is 7 (at each step of the trellis each branch is given a Hamming difference between itself and the all zeros path). This total value is arrived at by adding the incremental values of metric at each step of the trellis (in this case \( 2+1+1+1+0+2=7 \)). The second path of interest is the one generated by the input data 100000. The same type of calculation is done and the total distance is 7 also \( (2+1+2+2) \). The third path of interest is the one generated by input data 110000. By following the same procedure again, it can be determined that the total distance through this path is 6. Other paths can be drawn similarly with varying values of total distance, but this one has the minimum total distance. It can be thus concluded that the minimum distance for this code trellis is 6.

This assertion is corroborated by both Viterbi [2], and Odenwalder [10]. This minimum "free" distance also gives a measure of how many errors can be corrected. This means that 2 errors can occur, and still be corrected. If 3 errors occur, there is a 50-50 chance that they will be corrected since that will put the original sequence at a distance 3 from all other sequences.

It is also important to note that there are certain critical spots, so to speak, where the errors introduced may not be able to be corrected. The reason for this is obvious once one observes the nature
of the Viterbi computations. When errors are introduced early in the message they do not affect the decoding performance as much as when errors are introduced near the end of the message. The reason is that when errors are introduced at the beginning, the decoder has a chance to recuperate so to speak. That is, the decoder paths remerge every \((K-1)\) bits. Therefore, any ties can be broken as the message becomes more "solidified", and the decoder can discern between the paths. When the errors occur near the end of the message, this process is halted prematurely and this "self-synchronizing" property cannot take effect. It can be concluded then that the longer the message, the more chance we will have of obtaining the maximum-likelihood message that was originally sent. There is one other factor that influences the choice of 'good' codes. And that is the concept of catastrophic error propagation. But before we talk about that, we must explain the difference between systematic and nonsystematic convolutional codes.

4.2. Systematic and Nonsystematic Convolutional Codes

Viterbi [2] gives a rather complete definition of systematic codes. He says, "The term systematic convolutional code refers to a code on each of whose branches one of the code symbols is just the data bit generating that branch." In other words, a systematic convolutional code is one that has at least one of its commutator taps coming directly from the first stage of the shift register. That is, the next input bit is one of the encoded output bits. The importance of systematic codes stems from the fact that in block codes, nonsystematic codes can be transformed into systematic codes that have a comparable performance level. The same is not true in convolutional codes. The reason for this
is that, as mentioned previously, the performance for convolutional codes depends largely on the minimum free distance between codewords. When we essentially eliminate an adder from the encoder, we tend to reduce that minimum free distance. Viterbi [2] shows comparisons between equal constraint length K systematic and nonsystematic codes. For convenience this is reproduced here (Figure 4.3). For larger values of K, the differences are even larger. As a matter of fact, this difference is so great for larger values of K, that systematic codes have the performance equivalent to nonsystematic codes of half the constraint length, while at the same time, these decoders require the decoder computing time and complexity of larger size K. This has been the main reason for not choosing any systematic convolutional codes to be implemented in the WPAFB/OU Communications Simulator. It should be emphasized that for instance, for the K=4, rate=1/2 coder used in Chapters II and III, the minimum free distance for the systematic code is 4 while for the nonsystematic (which we are using) it is 6 (see Figure 4.3).

There is another important reason for treating systematic convolutional codes. And that is they do not suffer from catastrophic error propagation. The latter is defined as "The event that a finite number of channel symbol errors causes an infinite number of data bit errors to be decoded" [2], [10]. But, one may ask, how can it be predicted that a certain code will have catastrophic error propagation? From the state diagram, catastrophic errors will occur if and only if any closed path on the diagram has a zero weight [2], [10]. Consequently, if the decoder were to choose this path with the zero weight, it could conceivably go through that path many times and generate a large number of bit errors. Another way to predict this condition for binary trees is if each adder
Maximum, Minimum Free Distance

<table>
<thead>
<tr>
<th>K</th>
<th>Systematic</th>
<th>Nonsystematic</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 4.3. Comparison of Systematic and Nonsystematic R=1/2 Code Distance.
has an even number of connections. This causes the all ones state to have zero weight and the above scenario occurs. To clarify this point further let us take an example. Figure 4.4 shows a coder that displays the catastrophic error propagation. Also included are its state diagram and a sample trellis tree with an encoded message. Notice that if the original message were all ones, the decoder would perform perfectly. However, if the message were all zeros, the decoder would decipher the message as all ones. This is the exact meaning of the self-loop with zero weight. Actually, there are two self-loops with zero weight that can be confused. This is the way that the decoder must work, and as can be seen from the trellis diagram, it would be wrong 50 percent of the time and for as many bits as the self-loop was traversed (see Figure 4.5).

4.3. Code Selection

There are two different criteria for choosing the convolutional coders, and, thus, the Viterbi decoders being used in the WPAFB/OU Communications Simulator. First and foremost is whether or not the particular code has good distance properties. By this it is meant that out of the codes available we choose one that has the maximum free distance for that size coder. The second criterion is that the particular code does not exhibit catastrophic error propagation. Reference [10] provides a set of optimum (in the sense explained above) short constraint length codes of rate 1/2 and 1/3. Almost all rate 1/2 were used in this project plus a rate 2/3 k=4 coder, which was used to test its performance against those of rate 1/2. The 2/3 code selected was the example appearing in Viterbi [2] and Odenwalder [10]. This last example clarified certain
Figure 4.4. Coder that Exhibits Catastrophic Error Propagation.
Figure 4.5. Partial Trellis Showing Equivalent Paths.
fundamental principles of Viterbi decoding, so in that way it was doubly helpful.

It must be emphasized at this point that each coder selected for the WPAFB/OU Simulator generated a slightly different trellis tree. And thus, it was decided early on in the project that a modular approach to the design of the Viterbi decoding capability for the simulator should be taken. Each module was set up as a different size convolutional encoder with its corresponding Viterbi decoder. And, of course, this simplified the debugging procedures and will help future endeavors in this area.

4.4. Path Memory Truncation

The subject of path memory truncation is a much more involved one when we are dealing with hardware implementations than it is in our case. There are two reasons for this. First of all, we can reserve relatively large amounts of storage by using dimensional arrays. Second, we know before-hand the largest number of bits that can be sent through the channel to the decoder per simulation run time. For most convolutional encoding modules this number is 24 bits, except for CHEN12 (rate=2/3 K=4 module) which has the capacity to send a total of 32 bits.

If these simplifications were not possible, however, there are a few alternatives that could be used. One of these would be to force the encoder, every so often, to an all zeros state. This would act as a flag to the decoder to output its decoded message. Another alternative is to output the oldest bit on a particular sequence after four of five times the constraint length K has been processed [13]. This last scheme ensures that only a few additional errors would be introduced and they
would not affect the overall bit error probability significantly. This scheme also reinforces the assertion made earlier that the longer the message, the more state metrics, etc., calculated, and the more chance that the correct message will be decoded, and consequently, the lower the overall bit error probability.

4.5. Known Starting State

All Viterbi decoding modules assume a known starting state of zero. This may, at first, seem undesirable, but since all state metrics are reset to 0 when re-entering the Viterbi modules, simulation results have shown this to be a good idea. In a worst case scenario, the first 3 or 4 constraint lengths worth of calculations would be somewhat unreliable because the decoder did not know accurately the starting state. However, after that the self-synchronizing property of the Viterbi decoder would take over [13]. In other words, it would be the same thing as the decoder making a finite number of errors and then remerging into the right path. In this way, steady state operation is assured after a reasonable amount of decoding (discrete) time.

There is another added advantage to taking this approach. By starting at state 0 every simulation run we narrow down the number of paths into each state up to the constraint length K (see Figure 3.4). At that point, all states should have been reached and the comparisons into each state should begin. This, of course, is reflected in the software logic of all the Viterbi decoder modules. The interesting point is that when 1, 2, or, in general, (K-1) bits are transmitted only, the probability of decoding the correct message is enhanced by using this method. In fact, when an unknown starting state was allowed, the message
decoded was wrong more than 50 percent of the time. This method of known starting state also follows the examples appearing in the literature [13]. This factor must be kept in mind when reviewing all software of this class.
5.1. *Bitchange Model of a Channel*

The BITCHAN module is a convenient way to model a channel theoretically without having to use modulator-demodulator blocks, etc. As stated in Chapter III, it is based on the concept of the BINARY SYMMETRIC CHANNEL. The BSC is a channel that changes a 0 to a 1 or a 1 to a 0 according to a transition probability "p" (see Figure 5.1).

![Figure 5.1. BSC Channel.](image-url)

The question remains, "How do you set this probability p?" There are two parameters that must be set at "Set-up" time of the simulation run. They are the type of P(e) curve desired and the value of Signal-to-noise ratio in db. For instance, if we choose the P(e) curve for a PSK type signal, and we set the SNR to 3 dB, the program then calculates the theoretical value of error probability for the particular curve (see Figure 5.5). This value is then used to generate a threshold value that determines whether a bitchange will occur for the bit in question. So, in essence, we are setting the probability "p" by the type of curve chosen and the value of SNR. Then the program simply takes each bit
individually and applies the threshold criterion. That is, depending on whether the random value called from 'ARAND' is greater than this threshold, then the bit is "flipped". If the value of random number is smaller, then the bit remains the same (see Figure 5.2). This operation is done to all bits coming through this pseudo-channel. We have thus introduced noise independently from symbol to symbol into the channel because we have gone from an input symbol that was, for instance, a 0 and changed it into a 1 according to the above mentioned criteria. This is exactly what a regular channel would do causing the receiver to decide on a different symbol value than the one originally sent. It stands to reason then that the lower the value of SNR used, the higher the \( P(e) \) value, and consequently, the lower the threshold value. When this condition occurs we will have more bits being changed, and, thus, a more "noisy" channel. This, of course, agrees with actual practice.

There is an additional feature that enhances the portability of this module, and that is the "ERFC(X)" function generation. This function is not available as part of the standard FORTRAN package for the DEC-10 machine at Wright-Patterson. Therefore, it was decided to include this as part of the FORTRAN source made available for the project. To do this, an original IBM assembly language routine was re-written in FORTRAN. Results comparing the routine available in the 370 and the one generated for the project are provided in Figure 5.3. When compared the values agreed to the sixth place and above. Therefore, it is felt that this accurately models this function which is necessary for the \( P(e) \) curve calculations. Figures 5.4, 5.5, 5.6, 5.7 are all the curves available in the BITCHAN module; i.e., PSK, FSK, DPSK, and Viterbi curves. The latter represents the theoretical bound provided in [2], [14]. This
Figure 5.2. Flow Chart Bitchan Module.
<table>
<thead>
<tr>
<th>$X$</th>
<th>CERFNC</th>
<th>ERFC</th>
<th>DIFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3.90000</td>
<td>0.20000000F</td>
<td>0.20000000F</td>
<td>0.0</td>
</tr>
<tr>
<td>-3.80000</td>
<td>0.19999990E</td>
<td>0.19999990E</td>
<td>0.0</td>
</tr>
<tr>
<td>-3.70000</td>
<td>0.19999990E</td>
<td>0.19999990E</td>
<td>0.0</td>
</tr>
<tr>
<td>-3.60000</td>
<td>0.19999990E</td>
<td>0.19999990E</td>
<td>0.0</td>
</tr>
<tr>
<td>-3.50000</td>
<td>0.19999990E</td>
<td>0.19999990E</td>
<td>0.0</td>
</tr>
<tr>
<td>-3.40000</td>
<td>0.19999990E</td>
<td>0.19999990E</td>
<td>0.0</td>
</tr>
<tr>
<td>-3.30000</td>
<td>0.19999990E</td>
<td>0.19999990E</td>
<td>0.0</td>
</tr>
<tr>
<td>-3.20000</td>
<td>0.19999990E</td>
<td>0.19999990E</td>
<td>0.0</td>
</tr>
<tr>
<td>-3.10000</td>
<td>0.19999990E</td>
<td>0.19999990E</td>
<td>0.0</td>
</tr>
<tr>
<td>-3.00000</td>
<td>0.19999990E</td>
<td>0.19999990E</td>
<td>0.0</td>
</tr>
<tr>
<td>-2.90000</td>
<td>0.19999980E</td>
<td>0.19999980E</td>
<td>0.0</td>
</tr>
<tr>
<td>-2.80000</td>
<td>0.19999247E</td>
<td>0.19999247E</td>
<td>0.0</td>
</tr>
<tr>
<td>-2.70000</td>
<td>0.19976355E</td>
<td>0.19976355E</td>
<td>0.0</td>
</tr>
<tr>
<td>-2.69999</td>
<td>0.19999528E</td>
<td>0.19999528E</td>
<td>0.0</td>
</tr>
<tr>
<td>-2.59999</td>
<td>0.19993114E</td>
<td>0.19993114E</td>
<td>0.0</td>
</tr>
<tr>
<td>-2.49999</td>
<td>0.19988565E</td>
<td>0.19988565E</td>
<td>0.0</td>
</tr>
<tr>
<td>-2.39999</td>
<td>0.19981365E</td>
<td>0.19981365E</td>
<td>0.0</td>
</tr>
<tr>
<td>-2.29999</td>
<td>0.19970198E</td>
<td>0.19970198E</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.99999</td>
<td>0.19953213E</td>
<td>0.19953213E</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.89999</td>
<td>0.19927902E</td>
<td>0.19927902E</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.79999</td>
<td>0.19890900E</td>
<td>0.19890900E</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.69999</td>
<td>0.19837894E</td>
<td>0.19837894E</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.59999</td>
<td>0.19763470E</td>
<td>0.19763470E</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.49999</td>
<td>0.19661036E</td>
<td>0.19661036E</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.39999</td>
<td>0.19522829E</td>
<td>0.19522829E</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.29999</td>
<td>0.19340057E</td>
<td>0.19340057E</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.19999</td>
<td>0.19103107E</td>
<td>0.19103107E</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.09999</td>
<td>0.18802013F</td>
<td>0.18802013F</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.99999</td>
<td>0.18426962E</td>
<td>0.18426962E</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.89999</td>
<td>0.17969027E</td>
<td>0.17969027E</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.79999</td>
<td>0.17420940E</td>
<td>0.17420940E</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.69999</td>
<td>0.16777935E</td>
<td>0.16777935E</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.59999</td>
<td>0.16038475E</td>
<td>0.16038475E</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.49999</td>
<td>0.15204897E</td>
<td>0.15204897E</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.39999</td>
<td>0.14283810E</td>
<td>0.14283819E</td>
<td>0.95367432E-06</td>
</tr>
<tr>
<td>-0.29999</td>
<td>0.13286152F</td>
<td>0.13286152F</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.19999</td>
<td>0.12226906E</td>
<td>0.12226906E</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.09999</td>
<td>0.11124506E</td>
<td>0.11124506E</td>
<td>0.0</td>
</tr>
<tr>
<td>0.00000</td>
<td>0.99998802F</td>
<td>0.99998802F</td>
<td>0.0</td>
</tr>
<tr>
<td>0.10000</td>
<td>0.88752526E</td>
<td>0.88752520E</td>
<td>-0.59604645E-07</td>
</tr>
<tr>
<td>0.20000</td>
<td>0.77728599F</td>
<td>0.77728593F</td>
<td>-0.59604645E-07</td>
</tr>
<tr>
<td>0.30000</td>
<td>0.67136240E</td>
<td>0.67136234E</td>
<td>-0.59604645E-07</td>
</tr>
<tr>
<td>0.40000</td>
<td>0.57159758E</td>
<td>0.57159752F</td>
<td>-0.59604645E-07</td>
</tr>
<tr>
<td>0.50000</td>
<td>0.47949094E</td>
<td>0.47949088E</td>
<td>-0.59604645E-07</td>
</tr>
<tr>
<td>0.60000</td>
<td>0.39613569E</td>
<td>0.39613563F</td>
<td>-0.59604645E-07</td>
</tr>
<tr>
<td>0.70000</td>
<td>0.32219156E</td>
<td>0.32219154E</td>
<td>-0.59604645E-07</td>
</tr>
<tr>
<td>0.80000</td>
<td>0.25789273F</td>
<td>0.25789279F</td>
<td>0.59604645E-07</td>
</tr>
<tr>
<td>0.90000</td>
<td>0.20308650E</td>
<td>0.20308661F</td>
<td>0.11920929E-06</td>
</tr>
</tbody>
</table>

Figure 5.3. Comparisons Between IBM (ERFC) Routine and Locally Generated Function (CERFNC).
<table>
<thead>
<tr>
<th>row</th>
<th>value</th>
<th>value</th>
<th>value</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.40001</td>
<td>0.11979145E-00</td>
<td>0.11979187E-00</td>
<td>0.41723251E-06</td>
</tr>
<tr>
<td>1</td>
<td>1.70001</td>
<td>0.89683414E-01</td>
<td>0.89683712E-01</td>
<td>0.29002322E-06</td>
</tr>
<tr>
<td>1</td>
<td>1.30001</td>
<td>0.65990150E-01</td>
<td>0.65990388E-01</td>
<td>0.23841858E-06</td>
</tr>
<tr>
<td>1</td>
<td>1.40001</td>
<td>0.47713555E-01</td>
<td>0.47713738E-01</td>
<td>0.18253922E-06</td>
</tr>
<tr>
<td>1</td>
<td>1.50001</td>
<td>0.33893932E-01</td>
<td>0.33894058E-01</td>
<td>0.12665987E-06</td>
</tr>
<tr>
<td>1</td>
<td>1.60001</td>
<td>0.23650989E-01</td>
<td>0.23651082E-01</td>
<td>0.93132257E-07</td>
</tr>
<tr>
<td>1</td>
<td>1.70001</td>
<td>0.16209129E-01</td>
<td>0.16209196E-01</td>
<td>0.67055225E-07</td>
</tr>
<tr>
<td>1</td>
<td>1.80001</td>
<td>0.10909233E-01</td>
<td>0.10909285E-01</td>
<td>0.52154064E-07</td>
</tr>
<tr>
<td>1</td>
<td>1.90001</td>
<td>0.72040523E-02</td>
<td>0.72094426E-02</td>
<td>0.37252903E-07</td>
</tr>
<tr>
<td>2</td>
<td>2.00000</td>
<td>0.46776347E-02</td>
<td>0.46776570E-02</td>
<td>0.22351742E-07</td>
</tr>
<tr>
<td>2</td>
<td>2.10000</td>
<td>0.29794215E-02</td>
<td>0.29794227E-02</td>
<td>0.11641532E-08</td>
</tr>
<tr>
<td>2</td>
<td>2.20000</td>
<td>0.18628226E-02</td>
<td>0.18628221E-02</td>
<td>-0.46566129E-09</td>
</tr>
<tr>
<td>2</td>
<td>2.30000</td>
<td>0.11431642E-02</td>
<td>0.11431640E-02</td>
<td>-0.23283064E-09</td>
</tr>
<tr>
<td>2</td>
<td>2.40000</td>
<td>0.68850839E-03</td>
<td>0.68850862E-03</td>
<td>-0.23283064E-09</td>
</tr>
<tr>
<td>2</td>
<td>2.50000</td>
<td>0.40694978E-03</td>
<td>0.40694978E-03</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>2.60000</td>
<td>0.23603403E-03</td>
<td>0.23603406E-03</td>
<td>0.29103830E-10</td>
</tr>
<tr>
<td>2</td>
<td>2.70000</td>
<td>0.13433286E-03</td>
<td>0.13433295E-03</td>
<td>0.87311491E-10</td>
</tr>
<tr>
<td>2</td>
<td>2.80000</td>
<td>0.75013537E-04</td>
<td>0.75013522E-04</td>
<td>-0.14551915E-10</td>
</tr>
<tr>
<td>2</td>
<td>2.90000</td>
<td>0.41098203E-04</td>
<td>0.41098188E-04</td>
<td>-0.14551915E-10</td>
</tr>
<tr>
<td>3</td>
<td>3.00000</td>
<td>0.22090753E-04</td>
<td>0.22090753E-04</td>
<td>0.63664629E-11</td>
</tr>
<tr>
<td>3</td>
<td>3.10000</td>
<td>0.11648838E-04</td>
<td>0.11648845E-04</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>3.20000</td>
<td>0.60258872E-05</td>
<td>0.60258872E-05</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>3.30000</td>
<td>0.30577876E-05</td>
<td>0.30577876E-05</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>3.40000</td>
<td>0.15220385E-05</td>
<td>0.15220385E-05</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>3.50000</td>
<td>0.74312396E-06</td>
<td>0.74312396E-06</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>3.60000</td>
<td>0.35587738E-06</td>
<td>0.35587738E-06</td>
<td>-0.17053026E-12</td>
</tr>
<tr>
<td>3</td>
<td>3.70000</td>
<td>0.16715859E-06</td>
<td>0.16715859E-06</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>3.80000</td>
<td>0.77007826E-07</td>
<td>0.77007830E-07</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>3.90000</td>
<td>0.34794219E-07</td>
<td>0.34794226E-07</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>4.00000</td>
<td>0.15418227E-07</td>
<td>0.15418223E-07</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>4.10000</td>
<td>0.67005672E-08</td>
<td>0.67005637E-08</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>4.20000</td>
<td>0.28557192E-08</td>
<td>0.28557188E-08</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>4.30000</td>
<td>0.11935770E-08</td>
<td>0.11935768E-08</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>4.40000</td>
<td>0.48921622E-09</td>
<td>0.48921600E-09</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>4.50000</td>
<td>0.19663740E-09</td>
<td>0.19663743E-09</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>4.60000</td>
<td>0.77504142E-10</td>
<td>0.77504100E-10</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>4.70000</td>
<td>0.29955954E-10</td>
<td>0.29955954E-10</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>4.80000</td>
<td>0.11353630E-10</td>
<td>0.11353629E-10</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>4.90000</td>
<td>0.42194850E-11</td>
<td>0.42194841E-11</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>5.00000</td>
<td>0.15376745E-11</td>
<td>0.15376745E-11</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>5.10000</td>
<td>0.54961683E-12</td>
<td>0.54961673E-12</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>5.20000</td>
<td>0.19252178E-12</td>
<td>0.19252173E-12</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>5.30000</td>
<td>0.66142296E-13</td>
<td>0.66142296E-13</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>5.40000</td>
<td>0.22280819E-13</td>
<td>0.22280829E-13</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>5.50000</td>
<td>0.73592120E-14</td>
<td>0.73592120E-14</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>5.60000</td>
<td>0.23833140E-14</td>
<td>0.23833131E-14</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>5.70000</td>
<td>0.75678074E-15</td>
<td>0.75678032E-15</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>5.80000</td>
<td>0.23560984E-15</td>
<td>0.23560984E-15</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>5.90000</td>
<td>0.71920932E-16</td>
<td>0.71920893E-16</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>6.00000</td>
<td>0.21524709E-16</td>
<td>0.21524709E-16</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Figure 5.3. Continued.
Figure 5.4: Theoretical FSK Curve.
Figure 5.5. Theoretical PSK Curve.
<table>
<thead>
<tr>
<th>Value</th>
<th>2.42E00</th>
<th>2.07E00</th>
<th>1.38E00</th>
<th>1.04E00</th>
<th>6.90E-01</th>
<th>3.45E-01</th>
<th>5.23E-05</th>
</tr>
</thead>
<tbody>
<tr>
<td>E/N (Viterbi)</td>
<td>7.48E00</td>
<td>7.68E-01</td>
<td>1.54E00</td>
<td>2.30E00</td>
<td>3.07E00</td>
<td>3.84E00</td>
<td>4.61E00</td>
</tr>
</tbody>
</table>

Figure 5.7: Viterbi Theoretical Upper Bound.
is a rather loose bound specially around 1 to 2 dB range. In fact, a lot of literature does not include any bounds below 2 dB. The equations upon which these theoretical curves are based are the following:

For FSK---\( P(E) = 0.5 \times \exp(-E/2N) \)

For PSK---\( P(E) = 0.5 \times \text{erfc}(\sqrt{E/N}) \)

For DPSK---\( P(E) = 0.5 \times \exp(-E/N) \)

For Viterbi upper bound---\( P(e) < (2\sqrt{p(1-p)})^{5/2} \)

\[ \frac{1-4\sqrt{p(1-p)}}{1-4\sqrt{p(1-p)}} \]

where \( p \) = error probability for PSK signal.

A copy of the BITCHAN listing is provided in the Appendix for convenience. It must also be remembered that all I/O to this module follows the same procedures as explained in Chapters II and III [18].

5.2. Experimental Results

Figure 5.8 shows the simulator configuration used to run all the performance experiments. Here the DSOURCE module is DSOR10 which generates random binary bits. Figure 5.8a is a sample 'SMD' file to run such a simulation. It must be remembered that all data in these experiments is "hard quantized" data in order to take advantage of the simulator I/O structure. In general, hard quantization does not provide as good results as those in Odenwalder [10], and Heller, et al [13]. This was a necessary tradeoff in order to utilize the simulator routines most efficiently (see Chapter II for details).

Figures 5.9, 5.10, 5.11, 5.12, and 5.13 show the results for the

- \( k=3 \) rate=1/2
- \( k=4 \) rate=2/3
- \( k=4 \) rate=1/2
- \( k=5 \) rate=1/2
- \( k=6 \) rate=1/2
Figure 5.8. Simulator Configuration for Obtaining Experimental Results.
Figure 5.8a. Sample "SMD" File to Generate Experimental Results.
Figure 5.9. $K=3, R=1/2$ Viterbi Decoder Performance Curve (PSK Channel).
Figure 5.10. $K=4$, $R=2/3$ Viterbi Decoder Performance (PSK channel).
Figure 5.11. K=4, R=1/2 Viterbi Decoder Performance Curve (PSK Channel).
Figure 5.12. $K=5, r=1/2$ Viterbi Decoder Performance Curve (PSK Channel).
Figure 5.13. $K=6, r=1/2$ Viterbi Decoder Performance Curve (PSK Channel).
codes respectively. This was done by using PSK theoretical curve of Figure 5.5. These results are somewhat surprising though. All the rate 1/2 modules exhibit performance results which are quite similar. As a matter of fact, except for Figure 5.13 (k=6 rate=1/2), all other rate 1/2 code curves are nearly identical. These are superior to the rate 2/3 k=4 code though. The only advantage of the latter is that it can process more total bits (32) than the rate 1/2 codes. At higher signal-to-noise ratios the rate 2/3 k=4 begins to show some gradual improvement, however.

This brings up another important point; due to large amounts of computer time required to generate these data points (on the average 241 virtual CPU seconds for a 20,000 bit simulation to generate one point), results were obtained to a maximum of 5 dB. If a signal-to-noise ratio was set such that a probability of error was near 10**(-5) (approximately 7-8 dB), at least 500,000 bits (on the average) would have to be run. This was thought to be unreasonable and too costly.

An FSK system, as expected from theory, is worse than a PSK system in terms of bit error performance, specially around the region of interest (0 --->5 dB). Figures 5.14, 5.15, 5.16, 5.17, and 5.18 show the results of simulation. The latter were restricted to a maximum of 20,000 bits due to the low expected probability of error. Unlike the PSK case, however, convolutional coding and Viterbi decoding do not provide a measure of communication improvement in the range studied. The trend in these latter curves is, as the signal-to-noise ratio increases, a better bit error rate results, but in a rather slow manner. The cross-over point is close to 5 dB. There is one interesting observation that these results highlight; and that is for this type of coding decoding techniques, an FSK system is undesirable (the literature corroborates this
Figure 5.14. $K=3, R=1/2$ Viterbi Decoder Performance Curve (FSK Channel).
Figure 5.15. $K=4$, $r=2/3$ Viterbi Decoder Performance Curve. (FSK Channel).
Figure 5.16. K=4, $r=\sqrt{2}$ Viterbi Decoder Performance Curve (FSK Channel).
Figure 5.17. $K=5$ $r=1/2$ Viterbi Decoder Performance Curve (FSK Channel).
Figure 5.18. $K=6, r=1/2$ Viterbi Decoder Performance Curve (FSK Channel).
fact; [2], [10], [13] use PSK). Once again, however, the improvement from coder to coder is not significant in this range. Thus, it can be concluded that for the signal-to-noise ratios studied, all the encoder-decoder modules exhibit nearly equivalent curves except for the k=4, rate=2/3 set (Figure 5.15) The latter's performance is quite poor even when compared to theoretical FSK.
VI. CONCLUSIONS AND RECOMMENDATIONS

The conclusions that can be drawn from my work are the following. First of all, the design and implementation of convolutional encoders and Viterbi decoders (in software) has been reduced to a step-by-step method. That is, by selecting a certain size and rate coder, the design can be implemented in software in a relatively short designer time. Second of all, it has been shown that the design process is narrowed down to the generation of the appropriate trellis diagram. By using the existing designs and by following the general flow charts given in Chapters II and III, a new design becomes almost trivial. However, it must be remembered that the complexity of the decoder rises exponentially for larger and larger size coders. For this reason all separate section designs were made as compact and portable as possible. In this way, re-designs of entire sections would be avoided. This type of software implementation could, conceivably, be used to create real-time systems assuming, of course, that the system was fast enough. By studying the basic principles of convolutional coding and Viterbi decoding, it may be possible to implement these techniques in hardware also. Most important, however, is the fact that at the time of this writing, the WPAFB/OU Communications Simulator has the capability to use five different convolutional encoding-Viterbi decoding schemes (a K=3, r=1/2, a K=4, r=2/3, a K=4, r=1/2, a K=5, r=1/2, and a K=6, r=1/2). These can be used whenever error correction is desired for any simulation.

Performance results were given which substantiate the overall theory that coding improves communication efficiency (bit error rate performance) at the expense of enlarged bandwidth and greater computation.
time. In order to improve these results even more, it would be advantageous to incorporate soft-quantization into the decoding process. Analog values could be read directly from the channel and pre-processed before the actual Viterbi decoding process would start. Odenwalder [10] and Heller [13] have shown significant improvement in results by using soft quantization. It is suggested that such a scheme be implemented in the WPAFB/OU communications simulator while keeping the I/O requirements of the simulator in a manageable and practical fashion.
VII. REFERENCES


VIII. APPENDIX

SUBROUTINE CCENCD
C*****************************************************************************
C
C SUBROUTINE CHEN10 IS A CONVOLUTIONAL ENCODER
C TO BE USED ALONG WITH THE VITERBI MAXIMUM LIKELIHOOD DECODER.
C
C SUBROUTINE INPUT: DI(1,NBEXA)=BINARY WORD UP TO 24 BITS.
DQ(1,NBEXA)=NO. OF BITS IN DI(1,NBEXA)
C
C SUBROUTINE OUTPUT: DI(1,IBEX)=ENCODED WORD IN PBIT FORMAT.
DI(2,IBEX)=2ND HALF OF ENCODED WORD.
DQ(1,IBEX)=NO. OF BITS OF DI(1,IBEX)
DQ(2,IBEX)=NO. OF BITS OF DI(2,IBEX)
C
VARIABLES ENTERED: NONE
C
SUBROUTINES CALLED: GBIT,PBIT.
C
*****************************************************************************
C
COMMON BLOCK
C*****************************************************************************
C
COMMON/IO/IRTM,IWTM,LGDR,MODE,LSMML,LUWF,LTEMP,ITITLE(20),
* ILPT,IDS1,IDS2,IDS3,IDS4,IPL1,IPL2,IPL3,IPL4,IRTPO
COMMON/MON/IMB(12,S),IDB(12,S),RIVL(12),INB(12),IOP(12),IBLS(80),
* IBLN(80,2),IBMAX,MCMD
COMMON/EXEC/GDR(20),IBEX,TINC,CTIME,NSPV,SINC,TIME(3,31),DI(3,31),
* DQ(3,31),IBEXM,NBEXA,NBEXB,NBEXC
EQUIVALENCE (GDR(1),TSTART),(GDR(2),TSTOP),(GDR(3),ISEED)
DIMENSION IARRAY(2),KARRAY(2),IBIT(2)
INTEGER DATA(3)
C SET INITIAL WORD LENGTH OUTPUT
C
NBZ=DQ(1,NBEXA)+DQ(2,NBEXA)
NBZ=NBZ*2
DQ(1,IBEX)=24
DQ(2,IBEX)=0
IF (NBZ.GT.24)DQ(2,IBEX)=NBZ-24
IF(NBZ.LT.24)DQ(1,IBEX)=NBZ
WRITE(IWTM,245)NBZ
245 FORMAT(' CHEN10: INITIAL OUTPUT WORD LENGTH IS ',I2,' BITS')
RETURN
C
C SIMULATION RUN
C
ENTRY RCENCD
NB1=DQ(1,NBEXA)
NB2=DQ(2,NBEXA)
IF(NBIT1.LT.24)NB1=0
NBIT=NB1+NB2

C
C ARRAY INITIALIZATION

DO 2 I=1,3
  DATA(I)=0
DO 6 I=1,2
  IBIT(I)=0
  IARRAY(1)=DI(1,NBEXA)
  IARRAY(2)=DI(2,NBEXA)
DO 5 KK=1,2
  KARRAY(KK)=0

C CONVOLUTIONAL ENCODER

K=1
II=1
DO 4 I=1,NBIT

C SHIFT CONTENTS LEFT ONE PLACE...

J=3
DO 3 JI=1,2
  DATA(J)=DATA(J-1)
  J=J-1

C GET NEW BIT ......

CALL GBIT(IARRAY,II,DATA(1))

C DO EXOR OPERATION......

IDATA=DATA(1)+DATA(3)
  IBIT(2)=MOD(IDATA,2)
  IBIT(1)=MOD(IDATA+DATA(2),2)

C PACK OUTPUT

CALL PBIT(KARRAY,K,IBIT(1))
CALL PBIT(KARRAY,K+1,IBIT(2))

C INCREMENT BIT NO.....

II=II+1
K=K+2

C VARIABLE WORD LENGTH CAPABILITY....

NBIT2X=2*NBIT
IF(NBIT2X.GT.24)GO TO 25
DQ(1,IBEX)=NBIT2X
DQ(2,IBEX)=0.
GO TO 8

25 DQ(1,IBEX)=24.
    DQ(2,IBEX)=NBIT2X-24.
C
C   EXIT
C
8   DI(1,IBEX)=KARRAY(1)
     DI(2,IBEX)=KARRAY(2)
     RETURN
     END
SUBROUTINE CCDECD

CDECI0 IS AN UPDATED VERSION OF VITERBI MAXIMUM-LIKELIHOOD
DECODER FOR BSC CHANNEL.

SUBROUTINE INPUT: DI(1,NBEXA)=1ST HALF ENCODED WORD IN PBIT
FORMAT.
DI(2,NBEXA)=2ND HALF OF ENCODED WORD; IFF
NO. OF BITS > 24.
DQ(1,NBEXA)=NO. OF BITS IN DI(1,NBEXA)
DQ(2,NBEXA)=NO. OF BITS IN DI(2,NBEXA).
DQ(3,NBEXA)=WORD TRANSITION MARKER

SUBROUTINE OUTPUT: DI(1,IBEX)=DECODED WORD.
DQ(1,IBEX)=NO. OF BITS IN DI(1,IBEX).
DQ(3,IBEX)=WORD TRANSITION MARKER

VARIABLES ENTERED: NONE

SUBROUTINES CALLED: GBIT, ARAND, PB1T.

C******************************************************************************
C
C DIMENSION TMETRI(8), HAMDIS(4), ITRANS(25)
INTEGER PATHNO(4,25), Y(2), RMETRI(4,25)
DIMENSION IARRAY(2), KARRAY(2)
COMMON/IO/ IRTM, IWTM, LGDR, MODE, LSML, LSMD, LUWF, LTEMP, ITITLE(20),
* ILPT, IDSK1, IDSK2, IDSK3, IDSK4, IPL1, IPL2, IPL3, IPL4, IRTPO
COMMON/MON/ IMB(12,S), IDB(12,S), RIVL(12), INB(12), IOP(12), IBLS(80),
* IBLN(80,2), IBMAX, MCMMD
COMMON/EXEC/GDR(20), IBEX, TINC, CTIME, NSPV, SINC, TIME(3,31), DI(3,31),
* DQ(3,31), IBEXM, NBEXA, NBEXB, NBEXC
EQUIVALENCE (GDR(1), TSTART), (GDR(2), TSTOP), (GDR(3), ISEED)
RETURN

C 2ND ENTRY POINT

ENTRY RCDECD
DQ(3,IBEX)=DQ(3,NBEXA)
IF(DQ(3,NBEXA).NE.1.0)RETURN
NB1=DQ(1,NBEXA)
NB2=DQ(2,NBEXA)
NB=(NB1+NB2)/2
KNB=NB+1
DQ(1,IBEX)=NB
DQ(2,IBEX)=0.

C GET INPUT ARRAYS...
IARRAY(1)=DI(1,NBEXA)
IARRAY(2)=DI(2,NBEXA)
DO31J=1,KNB
DO31I=1,4
RMETRI(I,J)=0.
C PATHNO(I,J)=0
C ZERO OUT OUTPUT ARRAY.
DO 555I=1,2
555 KARRAY(I)=0
DO 556 I=1,KNB
556 ITRANS(I)=0
C SET UP MAIN LOOP
II=1
DO 333 K=1,NB
C READ CHANNEL OUTPUT...
CALL GBIT(IARRAY,II,Y(1))
CALL GBIT(IARRAY,II+1,Y(2))
C CALCULATE ALL POSSIBLE HAMMING DISTANCES...
CHECK1=MOD(0+Y(1),2)
CHECK2=MOD(0+Y(2),2)
CHECK3=MOD(1+Y(1),2)
CHECK4=MOD(1+Y(2),2)
HAMDIS(1)=CHECK1+CHECK2
HAMDIS(2)=CHECK3+CHECK4
HAMDIS(3)=CHECK3+CHECK2
HAMDIS(4)=CHECK1+CHECK4
C COMPUTE ALL METRICS OUT OF EACH STATE....
C METRICS OUT OF STATE 0...
TMETRI(1)=RMETRI(1,K)+HAMDIS(1)
TMETRI(2)=RMETRI(1,K)+HAMDIS(2)
C METRICS OUT OF STATE 1....
TMETRI(3)=RMETRI(2,K)+HAMDIS(3)
TMETRI(4)=RMETRI(2,K)+HAMDIS(4)
C METRICS OUT OF STATE 2.....
TMETRI(5)=RMETRI(3,K)+HAMDIS(2)
TMETRI(6)=RMETRI(3,K)+HAMDIS(1)
C METRICS OUT OF STATE 3.....
TMETRI(7)=RMETRI(4,K)+HAMDIS(4)
TMETRI(8)=RMETRI(4,K)+HAMDIS(3)
C INTRODUCE LOGIC TO TAKE CARE OF FIRST TWO STEPS THRU TRELLIS.
C GO TO (100,200),K
C COMPARISONS OF METRICS; DECIDE WHICH ONE IS SMALLEST.
DO31I=1,4
IF(TMETRI(I).GT.TMETRI(I+4))GO TO 2
IF(TMETRI(I).LT.TMETRI(I+4))GO TO 1
C NEITHER WORKS, TOSS A COIN, CHOOSE ONE....
CALL ARAND(XC)
IF(XC.GT.0.5)GO TO 1
GO TO 2
1 RMETRI(I,K+1)=TMETRI(I)
PATHNO(I,K+1)=I
GO TO 3
RMETRI(I,K+1)=TMETRI(I+4)
PATHNO(I,K+1)=I+4

CONTINUE
GO TO 333

K=1 CASE...

DO 9 I=1,2
RMETRI(I,K+1)=TMETRI(I)
PATHNO(I,K+1)=I
GO TO 333

K=2 CASE

DO 10 I=1,4
RMETRI(I,K+1)=TMETRI(I)
PATHNO(I,K+1)=I
10 II=II+2

SPECIAL CONDITIONS MUST BE SET UP IFF 1 BIT IS BEING TRANSMITTED/RUN TIME....
GO TO (150),NB

ALGORITHM TO FIND SMALLEST OF 'NCOMPA'
RMETRICS IN LAST COLUMN....
SET UP INDICES...

NCOMPA=4
I=1
J=2
K=1

HEART OF ALGORITHM...

IF(RMETRI(I,KNB).LT.RMETRI(J,KNB))GO TO 500
GO TO 600

KEEP SAME METRIC, COMPARE NEXT ONE DOWN...

JK=I
J=J+1
K=K+1

IF(K.LT.NCOMPA)GO TO 400
GO TO 800

CHANGE INDEX OF COMPARISON TO NEXT ONE;
COMPARE NEXT ONE DOWN....

JK=J
I=J
J=J+1
K=K+1
GO TO 525
800 CONTINUE
C
C SPECIAL CONDITIONS SET UP HERE...
C
GO TO 899
150 NCOMPA=2
GO TO 399
C DECRYPT PATH NOS TO PROVIDE PREVIOUS STATE AND PRESENT
C TRANSITION. THIS MUST BE DONE RECURSIVELY UNTIL WE GET TO
C THE BEGINNING OF THE TRELLIS TREE.
899 DO 900 K=1,NB
J=KNB+1-K
C
C IS FLAG 'JK' ODD OR EVEN??...
C
XJK=JK
A1=XJK/2.
I1=A1
A2=A1-I1
IF(A2.EQ.0.)GO TO 667
IF(A2.EQ.0.5)GO TO 668
WRITE(IWTM,669)
669 FORMAT(5X,'CDEC10: PROBLEM EXISTS IN TRACE BACK SECTION')
GO TO 1399
C
C EVEN CASE...
C
667 ITRANS(J)=1
C
C SET NEW FLAG, KEEP OLD 'JK'...
C
JK=A1
GO TO 670
C
C ODD CASE...
C
668 ITRANS(J)=0
C
C MAKE IT EVEN...ROUND UP!!.
A1=A1+0.5
C
C SET NEW FLAG..
JK=A1
670 IF(PATHNO(JK,J).EQ.JK)GO TO 671
IF(PATHNO(JK,J).EQ.JK+4)GO TO 672
WRITE(IWTM,673)
673 FORMAT(5X,'CDEC10: ERROR EXISTS IN PATHNO ARRAY.')
GO TO 1399
671 JK=JK
GO TO 900
672 JK=JK+2
900 CONTINUE
C
C OUTPUT INFORMATION IN PROPER ARRAY..
1399 DO 666 K=1,NB
IDEC=ITRANS(K+1)
I=K
666 CALL PBIT(KARRAY,I,IDE
DI(1,IBEX)=KARRAY(1)
DI(2,IBEX)=KARRAY(2)
RETURN
END
SUBROUTINE CCENC

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

SUBROUTINE CHEN12 IS K=4, RATE 2/3 CONVOLUTIONAL ENCODER. IT SHOULD BE USED WITH SAME LENGTH AND RATE VITERBI DECODER.

SUBROUTINE INPUT: DI(1,NBEXA)=BINARY WORD UP TO 24 BITS.
DI(2,NBEXA)=2ND 24 BITS (IF NECESSARY).
DQ(1,NBEXA)=NO. OF BITS IN DI(1,NBEXA)
DQ(2,NBEXA)=NO. OF BITS IN DI(2,NBEXA)

SUBROUTINE OUTPUT: DI(1,IBEX)=ENCODED WORD IN PBIT FORMAT.
DI(2,IBEX)=2ND HALF OF ENCODED WORD;
ONLY REQUIRED IF DI(1,IBEX) > 24 BITS.
DQ(1,IBEX)=NO. OF BITS OF DI(1,IBEX)
DQ(2,IBEX)=NO. OF BITS OF DI(2,IBEX)

VARIABLES ENTERED: NONE

SUBROUTINES CALLED: GBIT, PBIT.

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

COMMON BLOCK

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

COMMON/IO/IRTM, IWTM, LGDR, MODE, LSML, LSMD, LUWF, LTEMP, ITITLE(20),
* ILPT, IDSK1, IDSK2, IDSK3, IDSK4, IPL1, IPL2, IPL3, IPL4, IRTPO
COMMON/MON/IMB(12,5), IDB(12,5), RIVL(12), INB(12), IOP(12), IBLS(80),
* IBLN(80,2), IBMAX, MCMD
COMMON/EXEC/GDR(20), IBEX, TINC, CTIME, NSPV, SINC, TIME(3,31), DI(3,31),
* DQ(3,31), IBEXM, NBEXA, NBEXB, NBEXC
EQUIVALENCE (GDR(1), TSTART), (GDR(2), TSTOP), (GDR(3), ISEED)
COMMON/FLAG/IFLAG
DIMENSION IDATA(4), IBIT(3), IARRAY(2), KARRAY(2)

SET INITIAL WORD LENGTH

NBZ=DQ(1,NBEXA)+DQ(2,NBEXA)
NBZ=NBZ*3
NBZ=NBZ/2
DQ(1,IBEX)=24
DQ(2,IBEX)=0
IF (NBZ.GT.24) DQ(2,IBEX)=NBZ-24
IF(NBZ.LT.24) DQ(1,IBEX)=NBZ
WRITE(IWTM,245)NBZ
245 FORMAT(' CHEN12: INITIAL OUTPUT WORD LENGTH IS ',I2,' BITS')
RETURN

SIMULATION RUN
ENTRY RCENCD

ADJUSTABLE WORD LENGTH FEATURE FINDS
NO. OF BITS TO BE PROCESSED...

X1=DQ(1,NBEXA)
X2=DQ(2,NBEXA)
X3=(X1+X2)/2.
I1=X3
A1=X3-I1
IFLAG=0
IFL2=0
IF(A1.EQ.0.)GO TO 224
IF(A1.EQ.0.5)GO TO 223
WRITE(IWTM,225)

225 FORMAT(3X,'CHEN12: PROBLEM IN VARIABLE WORD LENGTH FEATURE')
GO TO 31

SET FLAG!!!...

223 IFLAG=1
I1=I1+1

224 NNB=I1

CLEAR OUT SHIFT REG. & OUTPUT BUFFERS..

DO 1 I=1,4
1 IDATA(I)=0

DO 2 I=1,3
2 IBIT(I)=0

INPUT ARRAY SETUP
IARRAY(1)=DI(1,NBEXA)
IARRAY(2)=DI(2,NBEXA)

ZERO OUT OUTPUT 'PBIT' ARRAY...

DO 3 KK=1,2
3 KARRAY(KK)=0

CONVOLUTIONAL ENCODER..

INITIALIZE ALL COUNTERS FOR INPUT AND OUTPUT ARRAYS AND
MAIN ENCODING LOOP...

K=1
II=1
KI=1

SHIFT TWO BITS RIGHT WITHIN REGISTER....

DO 4 I=1,2
4 IDATA(I+2)=IDATA(I)

GET TWO NEW BITS
CALL GBIT(IARRAY,II,IDATA(2))
CALL GBIT(IARRAY,II+1,IDATA(1))
IF(IFL2.EQ.1)IDATA(1)=0

NOW DO EXOR OPERATIONS...

IBIT(3)=MOD(IDATA(1)+IDATA(3),2)
INTER=MOD(IDATA(1)+IDATA(2),2)
IBIT(2)=MOD(INTER+IDATA(4),2)
IBIT(1)=MOD(IBIT(3)+IDATA(4),2)

PACK OUTPUT BITS

CALL PBIT(KARRAY,K,IBIT(1))
CALL PBIT(KARRAY,K+1,IBIT(2))
CALL PBIT(KARRAY,K+2,IBIT(3))

IS SECOND FLAG SET??
IF SO THEN I MUST GET OUT LOOP...

IF(IFL2.EQ.1)GO TO 22

INCREMENT BIT NO FOR INPUT ARRAY...
II=II+2

INCREMENT OUTPUT BIT NO.....
K=K+3

ARE WE DONE YET??

KI=KI+1
IF(KI.LT.NNB)GO TO 20
IF(IFLAG.EQ.0.AND.KI.EQ.NNB)GO TO 20
IF(IFLAG.EQ.1)GO TO 21
GO TO 22

SET 2ND FLAG, AND UPDATE
VALUE OF 'NNB' FOR FUTURE
REFERENCE...

21 IFL2=1
GO TO 20

22 NO3X=NNB*3
IF(NO3X.GT.24)GO TO 25
DQ(1,IBEX)=NO3X
DQ(2,IBEX)=0.
GO TO 30

25 DQ(1,IBEX)=24.
DQ(2,IBEX)=NO3X-24.

30 DI(1,IBEX)=KARRAY(1)
DI(2,IBEX)=KARRAY(2)

RETURN
END
SUBROUTINE CCDECD
C***********************************************************************
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C
  C 2ND ENTRY POINT
  C
ENTRY RCDECD
DQ(3,IBEX)=DQ(3,NBEXA)
IF(DQ(3,NBEXA).NE.1.0)RETURN
C
DATA INITIALIZATION...

NB1=DQ(1,NBEXA)
NB2=DQ(2,NBEXA)
NB=(NB1+NB2)/3
C SET UP A VARIABLE TO DETERMINE THE
C SIZE OF OUTPUT WORD...
NO2X=2*NB
IF(IFLAG.EQ.1)NO2X=NO2X-1
C IF IFLAG=0, NO. MUST BE EVEN, LEAVE EVERYTHING
C THE SAME....
IF(NO2X.GT.24)GO TO 223
DQ(1,IBEX)=NO2X
DQ(2,IBEX)=0.
GO TO 224
223 DQ(1,IBEX)=24.
DQ(2,IBEX)=NO2X-24.
224 KNB=NB+1
C SET ALL PROCESSING ARRAYS---->0
DO 2 J=1,KNB
DO 2 I=1,4
RMETRI(I,J)=0
2 PATHNO(I,J)=0
DO 9 J=1,33
9 ITRANS(J)=0
DO 555 1=1,2
555 KARRAY(I)=0
C C SET UP MAIN LOOP
C C GET INPUT ARRAYS..
C IARRAY(1)=DI(1,NBEXA)
IARRAY(2)=DI(2,NBEXA)
II=1
DO 333 K=1,NB
C GET RECEIVED SYMBOLS FROM CHANNEL...
CALL GBIT(IARRAY,II,Y(1))
CALL GBIT(IARRAY,II+1,Y(2))
CALL GBIT(IARRAY,II+2,Y(3))
C C CALCULATE ALL POSSIBLE HAMMING DISTANCES....
C DO 4 I=1,3
4 CHECK(I)=MOD(I+Y(I),2)
JI=1
DO 5 I=4,6
CHECK(I)=MOD(I+Y(JI),2)
5 JI=JI+1
HAMDIS(1)=CHECK(1)+CHECK(2)+CHECK(3)
HAMDIS(2)=CHECK(4)+CHECK(5)+CHECK(6)
HAMDIS(3)=CHECK(1)+CHECK(5)+CHECK(3)
HAMDIS(4)=CHECK(4)+CHECK(2)+CHECK(6)
HAMDIS(5)=CHECK(4)+CHECK(5)+CHECK(3)
HAMDIS(6)=CHECK(1)+CHECK(2)+CHECK(6)
HAMDIS(7)=CHECK(4)+CHECK(2)+CHECK(3)
HAMDIS(8)=CHECK(1)+CHECK(5)+CHECK(6)
C C COMPUTE ALL METRICS OUT OF EACH STATE....
C STATE 0....

DO 6 I=1,4
TMETRI(I)=RMETRI(1,K)+HAMDIS(I)

C STATE 1....

J=4
DO 65 I=5,8
TMETRI(I)=RMETRI(2,K)+HAMDIS(J)
J=J-1

C STATE 2...

DO 7 I=9,12
J=I-4
TMETRI(I)=RMETRI(3,K)+HAMDIS(J)

C STATE 3....

J=9
DO 8 I=13,16
J=J-1
TMETRI(I)=RMETRI(4,K)+HAMDIS(J)

C TAKE CARE OF FIRST STEP THRU TRELLIS...

C IF(K.EQ.1)GO TO 100

C NOW COMPARE METRICS, DECIDE WHICH ONE IS SMALLEST...

DO 150 I=1,4
IF(TMETRI(I).LT.TMETRI(I+4))GO TO 10
IF(TMETRI(I+4).LT.TMETRI(I))GO TO 15
CALL ARAND(A)
IF(A.LT.0.5)GO TO 10
GO TO 15
10 IF(TMETRI(I).LT.TMETRI(I+8))GO TO 20
IF(TMETRI(I+8).LT.TMETRI(I))GO TO 25
CALL ARAND(B)
IF(B.LT.0.5)GO TO 20
GO TO 25
20 IF(TMETRI(I).LT.TMETRI(I+12))GO TO 30
IF(TMETRI(I+12).LT.TMETRI(I))GO TO 35
CALL ARAND(C)
IF(C.LT.0.5)GO TO 30
GO TO 35
30 RMETRI(I,K+1)=TMETRI(I)
PATHNO(I,K+1)=I
GO TO 150
15 IF(TMETRI(I+4).LT.TMETRI(I+8))GO TO 16
IF(TMETRI(I+8).LT.TMETRI(I+4))GO TO 25
CALL ARAND(D)
IF(D.LT.0.5)GO TO 16
GO TO 25
16 IF(TMTRI(I+4).LT.TMTRI(I+12))GO TO 18
IF(TMTRI(I+12).LT.TMTRI(I+4))GO TO 35
CALL ARAND(E)
IF(E.LT.0.5)GO TO 18
GO TO 35
18 RMTRI(I,K+1)=TMTRI(I+4)
PATHNO(I,K+1)=I+4
GO TO 150
25 IF(TMTRI(I+8).LT.TMTRI(I+12))GO TO 26
IF(TMTRI(I+12).LT.TMTRI(I+8))GO TO 35
CALL ARAND(F)
IF(F.LT.0.5)GO TO 26
35 RMTRI(I,K+1)=TMTRI(I+12)
PATHNO(I,K+1)=I+12
GO TO 150
26 RMTRI(I,K+1)=TMTRI(I+8)
PATHNO(I,K+1)=I+8
150 CONTINUE
GO TO 333
C
C K=1 CASE IS TAKEN CARE OF HERE!
C
100 DO 155 J=1,4
RMTRI(J,K+1)=TMTRI(J)
155 PATHNO(J,K+1)=J
C END OF MAIN RECEIVING LOOP...
333 II=II+3
C
ALGORITHM TO FIND SMALLEST OF 4
C RMETRICES IN LAST COLUMN...
C
C SET UP INDICES...
C
I=1
J=2
K=1
C
HEART OF ALGORITHM...
C
400 IF(RMTRI(I,KNB).LT.RMTRI(J,KNB))GO TO 500
GO TO 600
C
KEEP THE SAME METRIC, COMPARE NEXT ONE DOWN...
C
500 JK=1
J=J+1
K=K+1
525 IF(K.LT.4)GO TO 400
GO TO 800
C
CHANGE INDEX OF COMPARISON TO NEXT ONE;
C COMPARE NEXT ONE DOWN...
CONTINUE
C NOW, LET'S RETRACE THE TRELLIS PATH NUMBERS AND
C IN SO DOING FIND OUT THE PREVIOUS STATE AND THE
C TRANSITION THAT BROUGHT US HERE. THIS MUST BE
C DONE RECURSIVELY UNTIL WE GET TO THE BEGIN-
C NING OF THE TRELLIS TREE.....
C
DO 900 K=1,NB
J=KNB+1-K
JJ=2*J-1

C USE COMPUTED 'GO TO' STATEMENT...
C
GO TO (999,1499,1999,2498),JK
C IF NONE OF THESE CRITERIA IS MET, PRINT ERROR
C MESSAGE AND GET OUT OF TULSA!..
C
WRITE(IWTM,1100)
1100 FORMAT(10X,'CDEC12: ERROR EXISTS IN JK DEFINITION')
GO TO 1400
999 ITRANS(JJ)=0
ITRANS(JJ-1)=0
GO TO 2499
1499 ITRANS(JJ)=1
ITRANS(JJ-1)=0
GO TO 2499
1999 ITRANS(JJ)=0
ITRANS(JJ-1)=1
GO TO 2499
2498 ITRANS(JJ)=1
ITRANS(JJ-1)=1

C NOW, CHECK THE PATH NO. AND FIND THE PREVIOUS STATE,
C AND CONSEQUENTLY THE ORIGINAL SYMBOLS...
C
2499 IF(PATHNO(JK,J).EQ.JK)GO TO 1000
IF(PATHNO(JK,J).EQ.JK+4)GO TO 1500
IF(PATHNO(JK,J).EQ.JK+8)GO TO 2000
IF(PATHNO(JK,J).EQ.JK+12)GO TO 2500
C IF NONE OF THESE CRITERIA IS MET, PRINT ERROR MESSAGE...
WRITE(IWTM,1101)
1101 FORMAT(10X,'CDEC12: ERROR EXISTS IN PATHNO ARRAY')
GO TO 1400
1000 JK=1
GO TO 900
1500 JK=2
GO TO 900
2000 JK=3
GO TO 900
2500 JK=4
900 CONTINUE
1400 KB=2*NB+1
   KKB=KB-1
C PACK THOSE BABIES UP!!!!...
C
C
   DO 666 K=1,KKB
   IDEC=ITRANS(K+1)
   I=K
666 CALL PBIT(KARRAY,I,IDEC)
   DI(1,IBEX)=KARRAY(1)
   DI(2,IBEX)=KARRAY(2)
RETURN
END
SUBROUTINE CCENCD
C***********************************************************************
C
C SUBROUTINE CHEN14 IS K=4, RATE 1/2 CONVOLUTIONAL ENCODER. IT
C SHOULD BE USED WITH SAME SIZE AND RATE VITERBI DECODER FOR
C SATISFACTORY OPERATION.
C
C SUBROUTINE INPUT: DI(1,NBEXA)=BINARY WORD UP TO 24 BITS.
C DQ(1,NBEXA)=NO. OF BITS IN DI(1,NBEXA).
C
C SUBROUTINE OUTPUT: DI(1,IBEX)=ENCODED WORD IN PBIT FORMAT.
C DI(2,IBEX)=2ND HALF OF ENCODED WORD.
C DQ(1,IBEX)=NO. OF BITS IN DI(1,IBEX).
C DQ(2,IBEX)=NO. OF BITS IN DI(2,IBEX).
C
C VARIABLES ENTERED: NONE
C
C SUBROUTINES CALLED: GBIT, PBIT.
C
C***********************************************************************
C
C SET INITIAL WORD LENGTH
C
NBZ=DQ(1,NBEXA)+DQ(2,NBEXA)
NBZ=NBZ*2
DQ(1,IBEX)=24
DQ(2,IBEX)=0
IF (NBZ.GT.24)DQ(2,IBEX)=NBZ-24
IF(NBZ.LT.24)DQ(1,IBEX)=NBZ
WRITE(IWTM,245S)NBZ
245 FORMAT(' CHEN14: INITIAL OUTPUT WORD LENGTH IS ','I2',' BITS')
RETURN

C SIMULATION RUN...

ENTRY RCENCD
NB1=DQ(1,NBEXA)
NB2=DQ(2,NBEXA)
IF(NB1.LT.24)NB2=0
NB=NB1+NB2
CLEAR ALL WORKING REGISTERS...

DO 1 I=1,4
1  IDATA(I)=0
DO 2 I=1,2
2  IARRAY(I)=0
     KARRAY(I)=0
     IBIT(I)=0
     IARRAY(1)=DI(1,NBEXA)
     IARRAY(2)=DI(2,NBEXA)

CONVOLUTIONAL ENCODER...

II=1
K=1
DO 30 KK=1,NB

SHIFT ALL BITS LEFT ONE PLACE...

J=4
DO 5 I=1,3
5  J=J-1

GET NEW BIT...

CALL GBIT(IARRAY,II,IDATA(1))

NOW, DO EXOR OPERATIONS

INT1=MOD(IDATA(4)+IDATA(3),2)
IBIT(2)=MOD(INT1+IDATA(1),2)
IBIT(1)=MOD(IBIT(2)+IDATA(2),2)

OUTPUT ENCODED DATA...

PACK THOSE BABIES UP!!!

CALL PBIT(KARRAY,K,IBIT(1))
CALL PBIT(KARRAY,K+1,IBIT(2))

UPDATE RUNNING INDICES...

II=II+1
K=K+2
NO2X=NB*2
IF(NO2X.GT.24)GO TO 25
DQ(1,IBEX)=NO2X
DQ(2,IBEX)=0.
GO TO 26
25 DQ(1,IBEX)=24.
    DQ(2,IBEX)=NO2X-24.
26 DI(1,IBEX)=KARRAY(1)
    DI(2,IBEX)=KARRAY(2)
RETURN
END
SUBROUTINE CCDECD

C***************************************************************
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C**************************************************************

DIMENSION TMETRI(16),HAMDIS(4),ITRANS(25),CHECK(4)
INTEGER PATHNO(8,25),RMETRI(8,25),Y(2)
DIMENSION IARRAY(2),KARRAY(2)
COMMON/IO/IRTM,ITWM,LGDR,MODE,LSMML,LSMMD,LUWF,LTEMP,ITITLE(20),
* ILPT,IDSK1,IDSK2,IDSK3,IDSK4,IPL1,IPL2,IPL3,IPL4,IRTPO
COMMON/MON/IMB(12,5),IDB(12,5),RIVL(12),INB(12),IOP(12),IBLS(80),
* IBLN(80,2),IBMAX,MCMD
COMMON/EXEC/GDR(20),IBEX,TINC,CTIME,NSPV,SINC,TIME(3,31),DI(3,31),
* DQ(3,31),IBEXM,NBEXA,NBEXB,NBEXC
EQUIVALENCE (GDR(1),TSTART),(GDR(2),TSTOP),(GDR(3),ISEED)
RETURN
ENTRY RCDECD
DQ(3,IBEX)=DQ(3,NBEXA)
IF(DQ(3,NBEXA).NE.1.0)RETURN
NB1=DQ(1,NBEXA)
NB2=DQ(2,NBEXA)
NB=(NB1+NB2)/2
DQ(1,IBEX)=NB
DQ(2,IBEX)=0.
KNB=NB+1

C SET ALL PROCESSING ARRAYS---->0.

DO 1 I=1,2
IARRAY(I)=0
1 KARRAY(I)=0
DO 2 J = 1, KNB  
DO 2 I = 1, 8  
RMETRI(I, J) = 0  
PATHNO(I, J) = 0  

C GET INPUT ARRAYS...  
C  
IARRAY(1) = DI(1, NBEXA)  
IARRAY(2) = DI(2, NBEXA)  
C  
METRIC VALUE & PATHNO CALCULATIONS FOLLOW....  

C SET UP MAIN LOOP  
C  
II = 1  
DO 333 K = 1, NB  
C  
INPUT RECEIVED SYMBOLS...  
CALL GBIT(IARRAY, II, Y(1))  
CALL GBIT(IARRAY, II+1, Y(2))  
C  
CALCULATE ALL HAMMING DISTANCES...  
C  
DO 4 I = 1, 2  
CHECK(I) = MOD(O + Y(I), 2)  
JI = I  
DO 5 I = 3, 4  
CHECK(I) = MOD(1 + Y(JI), 2)  
JI = JI+1  
HAMDIS(1) = CHECK(1) + CHECK(2)  
HAMDIS(2) = CHECK(3) + CHECK(4)  
HAMDIS(3) = CHECK(3) + CHECK(2)  
HAMDIS(4) = CHECK(1) + CHECK(4)  
C  
COMPUTE ALL METRICS OUT OF EACH STATE....  
C  
OUT OF STATE 0...  
TMETRI(1) = RMETRI(1, K) + HAMDIS(1)  
TMETRI(2) = RMETRI(1, K) + HAMDIS(2)  
C OUT OF STATE 1....  
TMETRI(3) = RMETRI(2, K) + HAMDIS(3)  
TMETRI(4) = RMETRI(2, K) + HAMDIS(4)  
C OUT OF STATE 2...  
TMETRI(5) = RMETRI(3, K) + HAMDIS(2)  
TMETRI(6) = RMETRI(3, K) + HAMDIS(1)  
C OUT OF STATE 3...  
TMETRI(7) = RMETRI(4, K) + HAMDIS(4)  
TMETRI(8) = RMETRI(4, K) + HAMDIS(3)  
C OUT OF STATE 4...  
TMETRI(9) = RMETRI(5, K) + HAMDIS(2)  
TMETRI(10) = RMETRI(5, K) + HAMDIS(1)  
C OUT OF STATE 5...  
TMETRI(11) = RMETRI(6, K) + HAMDIS(4)  
TMETRI(12) = RMETRI(6, K) + HAMDIS(3)  
C OUT OF STATE 6
SPECIAL CONDITIONS MUST BE SET UP IFF 1 OR 2 BITS BEING TRANSMITTED/RUN TIME

REMEMBER THAT RMETRI & PATHNO ARRAYS ARE NOT YET FULLY FILLED IN THESE SITUATIONS

\[ \text{TMETRI}(13) = \text{RMETRI}(7,K) + \text{HAMDIS}(1) \]
\[ \text{TMETRI}(14) = \text{RMETRI}(7,K) + \text{HAMDIS}(2) \]

OUT OF STATE 7...

\[ \text{TMETRI}(15) = \text{RMETRI}(8,K) + \text{HAMDIS}(3) \]
\[ \text{TMETRI}(16) = \text{RMETRI}(8,K) + \text{HAMDIS}(4) \]

TAKE CARE OF FIRST 3 STEPS THRU TRELLIS...

GO TO (100,200,300),K
DO 6 I=1,8
IF(TMTRI(I).LT.TMTRI(I+8))GO TO 7
IF(TMTRI(I+8).LT.TMTRI(I))GO TO 8
CALL ARAND(XC)
IF(XC.LT.0.5)GO TO 7
GO TO 8
7 RMETRI(I,K+1) = TMETRI(I)
PATHNO(I,K+1) = I
GO TO 6
8 RMETRI(I,K+1) = TMETRI(I+8)
PATHNO(I,K+1) = I+8
6 CONTINUE
GO TO 333
C
C K=1 CASE--FILL ONLY FIRST 2 ROWS...
100 L=2
GO TO 302
C
K=2 CASE--FILL 4 ROWS...
200 L=4
GO TO 302
C
K=3 CASE--FILL 8 ROWS--NO COMPARISONS YET..
300 L=8
302 DO 303 IJ=1,L
RMETRI(IJ,K+1) = TMETRI(IJ)
303 PATHNO(IJ,K+1) = IJ
333 II=II+2
C
SPECIAL CONDITIONS MUST BE SET UP IFF 1 OR 2 BITS BEING TRANSMITTED/RUN TIME...
REMEMBER THAT RMETRI & PATHNO ARRAYS ARE NOT YET FULLY FILLED IN THESE SITUATIONS...
GO TO (150,250),NB
C
ALGORITHM TO FIND SMALLEST OF 8 RMETRICS IN LAST COLUMN...
SET UP INDICES...
NCOMPA=8
399 I=1
J=2
K=1
HEART OF ALGORITHM...

400 IF(RMETRI(I,KNB).LT.RMETRI(J,KNB))GO TO 500
GO TO 600

KEEP SAME METRIC, COMPARE NEXT ONE DOWN...

500 JK=I
J=J+1
K=K+1
525 IF(K.LT.NCOMPA)GO TO 400
GO TO 800

CHANGE INDEX OF COMPARISON TO NEXT ONE;
COMPARE NEXT ONE DOWN....

600 JK=J
I=J
J=J+1
K=K+1
GO TO 525
800 CONTINUE
GO TO 899

SPECIAL CONDITIONS ARE SET UP HERE

150 NCOMPA=2
GO TO 399
250 NCOMPA=4
GO TO 399

NOW RETRACE TRELLIS PATH NOS. & FIND TRANSITIONS...
THIS MUST BE DONE STARTING ON LAST COLUMN AND WORKING BACKWARDS UNTIL WE GET TO THE BEGINNING OF THE TRELLIS TREE....

899 DO 666 I=1,NB
J=KNB+1-I

IS FLAG 'JK' ODD OR EVEN??...

XJK=JK
A1=XJK/2.
I1=A1
A2=A1-I1
IF(A2.EQ.0.)GO TO 667
IF(A2.EQ.0.5)GO TO 668
WRITE(IWTM,669)
669 FORMAT(5X,'CDEC14: PROBLEM EXISTS IN TRACE BACK SECTION')
GO TO 9000

EVEN CASE...

667 ITRANS(J)=1
C SET NEW FLAG, KEEP 'JK' INTACT...

JJK=A1
GO TO 670

C ODD CASE...

668 ITRANS(J)=0
C MAKE IT EVEN...ROUND UP...
A1=A1+0.5
C SET NEW FLAG...
JJK=A1
670 IF(PATHNO(JK,J).EQ.JK)GO TO 671
IF(PATHNO(JK,J).EQ.JK+8)GO TO 672
WRITE(IWTM,673)
FORMAT(5X,'CDEC14: ERROR EXISTS IN PATHNO ARRAY')
GO TO 9000
671 JK=JJK
GO TO 666
672 JK=JJK+4
666 CONTINUE

C PACK BITS AND PREPARE TO GET OUT...

9000 DO 9090 K=1,NB
IDEC=ITRANS(K+1)
I=K
9090 CALL PBIT(KARRAY,I,IDEC)
DI(1,IBEX)=KARRAY(1)
DI(2,IBEX)=KARRAY(2)
RETURN
END
SUBROUTINE CCENCED
C***********************************************************************
C
C SUBROUTINE CHEN16 IS K=5, RATE=1/2 CONVOLUTIONAL ENCODER.
C IT SHOULD BE USED WITH SAME SIZE AND RATE VITERBI DECODER.
C
C SUBROUTINE INPUT: DI(1,NBEXA)=BINARY WORD UP TO 24 BITS.
C DQ(1,NBEXA)=NO. OF BITS IN DI(1,NBEXA).
C
C SUBROUTINE OUTPUT: DI(1,IBEX)=1ST HALF ENCODED WORD IN
C PBIT FORMAT.
C DI(2,IBEX)=2ND HALF ENCODED WORD.
C DQ(1,IBEX)=NO. OF BITS IN DI(1,IBEX).
C DQ(2,IBEX)=NO. OF BITS IN DI(2,IBEX).
C
C VARIABLES ENTERED: NONE.
C
C SUBROUTINES CALLED: GBIT, PBIT.
C
C***********************************************************************
C
COMMON BLOCK
C***********************************************************************
COMMON/IO/IRTM,IWTM,LGDR,MODE,LSMML,LSMMD,LUWF,LTEMP,ITITLE(20),
  * ILPT,IDSK1,IDSK2,IDSK3,IDSK4,IPL1,IPL2,IPL3,IPL4,IRTPO
COMMON/MON/IMB(12,S),IDB(12,5),RIVL(12),INB(12),IOP(12),IBLS(80),
  * IBLN(80,2),IBMAX,MCMD
COMMON/EXEC/GDR(20),IBEX,TINC,CTIME,NSPV,SINC,TIME(3,31),DQ(3,31),
  * DQ(3,31),IBEXM,NBEXA,NBEXB,NBEXC
EQUIVALENCE (GDR(l),TSTART),(GDR(2),TSTOP),(GDR(3),ISEED)
DIMENSION IDATA(S),IBIT(2),IARRAY(Z),KARRAY(Z)
C
C SET INITIAL WORD LENGTH
C
NBZ=DQ(1,NBEXA)+DQ(2,NBEXA)
NBZ=NBZ*2
DQ(1,IBEX)=24
DQ(2,IBEX)=0
IF (NBZ.GT.24)DQ(2,IBEX)=NBZ-24
IF(NBZ.LT.24)DQ(1,IBEX)=NBZ
WRITE(IWTM,245)NBZ
245 FORMAT(' CHEN16: INITIAL OUTPUT WORD LENGTH IS ',I2,' BITS')
RETURN
C
C SIMULATION RUN...
C
ENTRY RCENCED
NB1=DQ(1,NBEXA)
NB2=DQ(2,NBEXA)
IF(NB1.LT.24)NB2=0
NB=NB1+NB2
C CLEAR ALL WORKING REGISTERS...

DO 1 I=1,5
   IDATA(I)=0
DO 2 I=1,2
   IBIT(I)=0
   IARRAY(I)=0
   KARRAY(I)=0

2 GET INPUT ARRAYS...

IARRAY(1)=DI(1,NBEXA)
IARRAY(2)=DI(2,NBEXA)

C CONVOLUTIONAL ENCODER...

II=1
K=1
DO 30 KK=1,NB

C SHIFT ALL BITS LEFT ONE PLACE...

J=5
DO 5 I=1,4
   IDATA(J)=IDATA(J-1)
   J=J-1

5 GET NEW BIT...

CALL GBIT(IARRAY,II,IDATA(1))

C NOW, DO EXOR OPERATIONS

INT1=MOD(IDATA(1)+IDATA(5),2)
INT2=MOD(IDATA(4)+IDATA(3),2)
IBIT(1)=MOD(INT1+INT2,2)
IBIT(2)=MOD(INT1+IDATA(2),2)

C OUTPUT ENCODED DATA...

CALL PBIT(KARRAY,K,IBIT(1))
CALL PBIT(KARRAY,K+1,IBIT(2))

C UPDATE INDECES...

II=II+1
K=K+2
NO2X=NB*2
IF(NO2X.GT.24) GO TO 25
DQ(1,IBEX)=NO2X
DQ(2,IBEX)=0.
GO TO 26

25 DQ(1,IBEX)=24.
DQ(2,IBEX)=NO2X-24.
DI(1,IBEX)=KARRAY(1)
DI(2,IBEX)=KARRAY(2)
RETURN
END
SUBROUTINE CDECD

C***********************************************************************
C
C CDEC16 IS R=1/2, K=5 VITERBI MAX- LIKELIHOOD DECODER FOR BSC.
C SUBROUTINE INPUT: DI(1,NBEXA)=1ST HALF ENCODED WORD IN PBIT
C FORMAT
C DI(2,NBEXA)=2ND HALF ENCODED WORD.
C DQ(1,NBEXA)=NO. OF BITS IN DI(1,NBEXA).
C DQ(2,NBEXA)=NO. OF BITS IN DI(2,NBEXA).
C DQ(3,NBEXA)=WORD TRANSITION MARKER
C
C SUBROUTINE OUTPUT: DI(1,IBEX)=DECODED WORD.
C DQ(1,IBEX)=NO. OF BITS IN DI(1,IBEX).
C DQ(3,IBEX)=WORD TRANSITION MARKER
C
VARIABLES ENTERED: NONE.

SUBROUTINES CALLED: GBIT, ARAND, PBIT.

C***********************************************************************
COMMON BLOCK

DIMENSION TMETRI(32), HAMDIS(4), CHECK(4), ITRANS(25)
DIMENSION IARRAY(2), KARRAY(2)
INTEGER PATHNO(16,25), RMETRI(16,25), Y(2)
COMMON/IO/ IRTM, IWTM, LGDR, MODE, LSML, LSMD, LUWF, LTEMP, ITITLE(20),
* ILPT, IDSK1, IDSK2, IDSK3, IDSK4, IPL1, IPL2, IPL3, IPL4, IRTPO
COMMON/MON/ IMB(12,5), IDB(12,5), RIVL(12), INB(12), IOP(12), IBLS(80),
* IBLN(80,2), IMAX, MCMD
COMMON/EXEC/GDR(20), IBEX, TINC, CTIME, NSPV, SINC, TIME(3,31), DI(3,31),
* DQ(3,31), IBXM, NBEXA, NBEXB, NBEXC
EQUIVALENCE (GDR(1), TSTART), (GDR(2), TSTOP), (GDR(3), ISEED)
RETURN

SIMULATION RUN

ENTRY RCDECD
DQ(3,IBEX)=DQ(3,NBEXA)
IF(DQ(3,NBEXA).NE.1.0)RETURN
NB1=DQ(1,NBEXA)
NB2=DQ(2,NBEXA)
NB=(NB1+NB2)/2
DQ(1,IBEX)=NB
DQ(2,IBEX)=0.
KNB=NB+1

SET ALL PROCESSING ARRAYS—>0.
DO 1 I=1,2
IARRAY(I)=0
KARRAY(I)=0
DO 2 J=1,KNB
DO 2 I=1,16
RMETRI(I,J)=0
PATHNO(I,J)=0

GET INPUT ARRAYS...

IARRAY(1)=D1(1,NBEXA)
IARRAY(2)=D1(2,NBEXA)

METRIC VALUE & PATHNO CALCULATIONS FOLLOW....

II=1
DO 333 K=1,NB

INPUT RECEIVED SYMBOLS...

CALL GB1T(IARRAY,II,Y(1))
CALL GBIT(IARRAY,II+1,Y(2))

CALCULATE ALL HAMMING DISTANCES...

DO 4 I=1,2
4 CHECK(I)=MOD(0+Y(I),2)
JI=1
DO 5 I=3,4
5 CHECK(I)=MOD(1+Y(JI),2)
JI=JI+1
HAMDIS(1)=CHECK(1)+CHECK(2)
HAMDIS(2)=CHECK(3)+CHECK(4)
HAMDIS(3)=CHECK(3)+CHECK(2)
HAMDIS(4)=CHECK(1)+CHECK(4)

COMPUTE ALL METRICS OUT OF EACH STATE....

OUT OF STATE 0...
TMETRI(1)=RMETRI(1,K)+HAMDIS(1)
TMETRI(2)=RMETRI(1,K)+HAMDIS(2)

OUT OF STATE 1....
TMETRI(3)=RMETRI(2,K)+HAMDIS(4)
TMETRI(4)=RMETRI(2,K)+HAMDIS(3)

OUT OF STATE 2...
TMETRI(5)=RMETRI(3,K)+HAMDIS(3)
TMETRI(6)=RMETRI(3,K)+HAMDIS(4)

OUT OF STATE 3...
TMETRI(7)=RMETRI(4,K)+HAMDIS(2)
TMETRI(8)=RMETRI(4,K)+HAMDIS(1)

OUT OF STATE 4...
TMETRI(9)=RMETRI(5,K)+HAMDIS(3)
TMETRI(10)=RMETRI(5,K)+HAMDIS(4)

OUT OF STATE 5...
TMETRI(11)=RMETRI(6,K)+HAMDIS(2)
TMETRI(12)=RMETRI(6,K)+HAMDIS(1)

OUT OF STATE 6...
TMETRI(13)=RMETRI(7,K)+HAMDIS(1)
TMETRI(14)=RMETRI(7,K)+HAMDIS(2)

OUT OF STATE 7...
TMETRI(15)=RMETRI(8,K)+HAMDIS(4)
TMETRI(16)=RMETRI(8,K)+HAMDIS(3)

OUT OF STATE 8...
TMETRI(17)=RMETRI(9,K)+HAMDIS(2)
TMETRI(18)=RMETRI(9,K)+HAMDIS(1)

OUT OF STATE 9...
TMETRI(19)=RMETRI(10,K)+HAMDIS(3)
TMETRI(20)=RMETRI(10,K)+HAMDIS(4)

OUT OF STATE 10...
TMETRI(21)=RMETRI(11,K)+HAMDIS(4)
TMETRI(22)=RMETRI(11,K)+HAMDIS(3)

OUT OF STATE 11...
TMETRI(23)=RMETRI(12,K)+HAMDIS(1)
TMETRI(24)=RMETRI(12,K)+HAMDIS(2)

OUT OF STATE 12...
TMETRI(25)=RMETRI(13,K)+HAMDIS(4)
TMETRI(26)=RMETRI(13,K)+HAMDIS(3)

OUT OF STATE 13...
TMETRI(27)=RMETRI(14,K)+HAMDIS(1)
TMETRI(28)=RMETRI(14,K)+HAMDIS(2)

OUT OF STATE 14...
TMETRI(29)=RMETRI(15,K)+HAMDIS(2)
TMETRI(30)=RMETRI(15,K)+HAMDIS(1)

OUT OF STATE 15...
TMETRI(31)=RMETRI(16,K)+HAMDIS(3)
TMETRI(32)=RMETRI(16,K)+HAMDIS(4)

TAKE CARE OF FIRST 4 STEPS THRU TRELLIS...

GO TO (100,200,300,404),K
DO 6 I=1,16
IF(TMETRI(I).LT.TMETRI(I+16))GO TO 7
IF(TMETRI(I+16).LT.TMETRI(I))GO TO 8
CALL ARAND(XC)
IF(XC.LT.0.5)GO TO 7
GO TO 8
7 RMETRI(I,K+1)=TMETRI(I)
PATHNO(I,K+1)=I
GO TO 6
8 RMETRI(I,K+1)=TMETRI(I+16)
PATHNO(I,K+1)=I+16
6 CONTINUE
GO TO 333
C
C K=1 CASE
100 L=2
GO TO 402
C
C K=2 CASE...
200 L=4
GO TO 402
C
C K=3 CASE
300 L=8
GO TO 402
C
C K=4 CASE...
404 L=16
402 DO 401 KI=1,L
RMETRI(KI,K+1)=TMETRI(KI)
401 PATHNO(KI,K+1)=KI
333 II=II+2
C
C IF ONLY 1,2,3 BITS TRANSMITTED DO THE
C FOLLOWING...
C
GO TO (150,250,350),NB
C
C ALGORITHM TO FIND SMALLEST OF 'NCOMPA'
C RMETRICS IN LAST COLUMN...
C
C SET UP INDICES...
C
NCOMPA=16
399 I=1
J=2
K=1
C
C HEART OF ALGORITHM...
C
400 IF(RMETRI(I,KNB).LT.RMETRI(J,KNB))GO TO 500
GO TO 600
C
C KEEP SAME METRIC, COMPARE NEXT ONE DOWN...
C
500 JK=I
J=J+1
K=K+1
525 IF(K.LT.NCOMPA)GO TO 400
GO TO 800

C CHANGE INDEX OF COMPARISON TO NEXT ONE;
C COMPARE NEXT ONE DOWN....

C
600 JK=J
   I=J
   J=J+1
   K=K+1
   GO TO 525
GO TO 525
800 CONTINUE
   GO TO 665
150 NCOMPA=2
   GO TO 399
250 NCOMPA=4
   GO TO 399
350 NCOMPA=8
   GO TO 399
C
C NOW RETRACE TRELLIS PATH NOS. & FIND TRANSITIONS...
C THIS MUST BE DONE STARTING ON LAST COLUMN AND WORKING
C BACKWARDS UNTIL WE GET TO THE BEGINNING OF THE
C TRELLIS TREE.....
665 DO 666 I=1,NB
   J=KNB+1-I
C
C IS 'JK' ODD OR EVEN??
C
   XJK=JK
   A1=XJK/2.
   I1=A1
   A2=A1-I1
   IF(A2.EQ.0.)GO TO 667
   IF(A2.EQ.0.5)GO TO 668
   WRITE(IWTM,669)
   GO TO 9000
669 FORMAT(5X,'**DECC16: PROBE EXISTS IN TRACE BACK SECTION')
C
C EVEN CASE...
C
667 ITRANS(J)=1
   JJK=A1
   GO TO 670
C
C ODD CASE...
C
668 ITRANS(J)=0
C ROUND UP...
   A1=A1+0.5
   JJK=A1
C
C THIS SECTION DETERMINES THE VALUE OF
C THE NEXT 'JK' SO THAT WE CAN TAKE
C ONE MORE STEP BACKWARD THRU THE
C TRELLIS..

C

670 IF(PATHNO(JK,J).EQ.JK)GO TO 671
IF(PATHNO(JK,J).EQ.JK+16)GO TO 672
WRITE(IWTM,673)
673 FORMAT(5X,'CDEC16: ERROR EXISTS IN PATHNO ARRAY??!!')
GO TO 9000
671 JK=JJK
GO TO 666
672 JK=JJK+8
666 CONTINUE

C NEXT, OUTPUT METRIC & PATHNO ARRAYS FOR INSPECTION....

C PACK BITS AND PREPARE TO GET OUT

C

9000 DO 9090 K=1,KNB
IDE=ITRANS(K+1)
I=K
9090 CALL PBIT(KARRAY,I,IDE)
DI(1,IBEX)=KARRAY(1)
DI(2,IBEX)=KARRAY(2)
RETURN
END
SUBROUTINE CCENCD

C***********************************************************************
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C***********************************************************************
C
C
C
C
C
C

COMMON BLOCK

C***********************************************************************
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C
C***********************************************************************
C
C
C
C
C
C

C** COMMON BLOCK **

COMMON/IO/IRTM,IWTM,LGDR,MODE,LSML,LSMMD,LUWF,LTEMP,ITITLE(20),
* ILPT,IDSk1,IDSk2,IDSk3,IDSk4,IPL1,IPL2,IPL3,IPL4,IRTPO
COMMON/MON/IMB(12,5),IDB(12,5),RIVL(12),INB(12),IOP(12),IBLS(80),
* IBLN(80,2),IMAX,MCMD
COMMON/EXEC/GDR(20),IBEX,TINC,CTIME,NSPV,SINC,TIME(3,31),DI(3,31),
* DQ(3,31),IBEXM,NBEXA,NBEXB,NBEXC
EQUIVALENCE (GDR(1),TSTART),(GDR(2),TSTOP),(GDR(3),ISEED)
DIMENSION IDATA(6),IBIT(2),IARRAY(2),KARRAY(2)

C

SET INITIAL WORD LENGTH OUTPUT

NBZ=DQ(1,NBEXA)+DQ(2,NBEXA)
NBZ=NBZ*2
DQ(1,IBEX)=24.
DQ(2,IBEX)=0.
IF(NBZ.GT.24)DQ(2,IBEX)=NBZ-24
IF(NBZ.LT.24)DQ(1,IBEX)=NBZ
WRITE(IWTM,245)NBZ
245 FORMAT(' CHEN18: INITIAL OUTPUT WORD LENGTH IS ',12,' BITS')
RETURN

C

SIMULATION RUN...

ENTRY RCENCD
NB1=DQ(1,NBEXA)
NB2=DQ(2,NBEXA)
IF(NB1.LT.24)NB2=0
NB=NB1+NB2
CLEAR ALL WORKING REGISTERS...

DO 1 I=1,6
   IDATA(I)=0
DO 2 I=1,2
   IBIT(I)=0
   IARRAY(I)=0
   KARRAY(I)=0

GET INPUT ARRAYS...

IARRAY(1)=DI(1,NBEXA)
IARRAY(2)=DI(2,NBEXA)

CONVOLUTIONAL ENCODER...

II=1
K=1
DO 30 KK=1,NB

SHIFT ALL BITS LEFT ONE PLACE...

J=6
DO 5 I=1,5
   IDATA(J)=IDATA(J-1)
   J=J-1

GET NEW BIT...

CALL GBIT(IARRAY,II,IDATA(1))

NOW, DO EXOR OPERATIONS

INT1=MOD(IDATA(1)+IDATA(4),2)
INT2=MOD(INT1+IDATA(6),2)
INT3=MOD(IDATA(3)+IDATA(5),2)
IBIT(1)=MOD(INT2+INT3,2)
INT4=MOD(IDATA(2)+IDATA(6),2)
IBIT(2)=MOD(INT1+INT4,2)

OUTPUT ENCODED DATA...

CALL PBIT(KARRAY,K,IBIT(1))
CALL PBIT(KARRAY,K+1,IBIT(2))

UPDATE INDECES...

II=II+1
K=K+2
NO2X=NB*2
IF(NO2X.GT.24)GO TO 25
DQ(1,IBEX)=NO2X
DQ(2,IBEX)=0.
GO TO 26
25 DQ(1, IBEX)=24.
    DQ(2, IBEX)=NO2X-24.
26 DI(1, IBEX)=KARRAY(1)
    DI(2, IBEX)=KARRAY(2)
RETURN
END
SUBROUTINE CCDECD

CDEC18 IS R=1/2, K=6 VITERBI MAX-LIKELIHOOD DECODER FOR BSC.

SUBROUTINE INPUT: DI(1,NBEXA)=1ST HALF ENCODED WORD IN PBIT FORMAT.
DI(2,NBEXA)=2ND HALF ENCODED WORD.
DQ(1,NBEXA)=NO. OF BITS IN DI(1,NBEXA).
DQ(2,NBEXA)=NO. OF BITS IN DI(2,NBEXA).
DQ(3,NBEXA)=WORD TRANSITION MARKER

SUBROUTINE OUTPUT: DI(1,IBEX)=DECODED WORD.
DQ(1,IBEX)=NO. OF BITS IN DI(1,IBEX).
DQ(3,IBEX)=WORD TRANSITION MARKER

VARIABLES ENTERED: NONE.

SUBROUTINES CALLED: GBIT, ARAND, PBIT.

COMMON BLOCK

DIMENSION TMETRI(64), HAMDIS(4), CHECK(4), ITRANS(25)
DIMENSION LARRAY(2), KARRAY(2)
INTEGER PATHNO(32,25), RMETRI(32,25), Y(2)
COMMON/IO/IRTM, IWTM, LGDR, MODE, LSMML, LSMMD, LUWF, LTEMP, ITITLE(20),
   * ILPT, IDS1, IDS2, IDS3, ID5, ID6, ID7, ID8, ID9, ID10,
   * ITRANS
COMMON/MON/IMB(12,S), IDB(12,S), RIVL(lZ), INB(lZ), IOP(12), IBLS(80),
   * IBLN(80,2), IBMAX, MCMD
COMMON/EXEC/GDR(20), IBEX, TINC, CTIME, NSPV, IINC, TIME(3,31), DI(3,31),
   * DQ(3,31), IBEX, NBEXA, NBEXB, NBEXC
EQUIVALENCE (GDR(1), TSTART), (GDR(2), TSTOP), (GDR(3), ISEED)
RETURN

SIMULATION RUN

ENTRY RCDECD
DQ(3,IBEX)=DQ(3,NBEXA)
IF(DQ(3,NBEXA).NE.1.0)RETURN
NB1=DQ(1,NBEXA)
NB2=DQ(2,NBEXA)
NB=(NB1+NB2)/2
DQ(1,IBEX)=NB
DQ(2,IBEX)=0.
KNB=NB+1

SET ALL PROCESSING ARRAYS---0.
DO 1 I=1,2
IARRAY(I)=0
1
KARRAY(I)=0
DO 2 J=1,KNB
DO 2 I=1,32
RMETRI(I,J)=0
PATHNO(I,J)=0
2
C
GET INPUT ARRAYS...
C
IARRAY(1)=DI(1,NBEXA)
IARRAY(2)=DI(2,NBEXA)
C
METRIC VALUE & PATHNO CALCULATIONS FOLLOW....
C
II=1
DO 333 K=1,NB
C
INPUT RECEIVED SYMBOLS...
C
CALL GBIT(IARRAY,II,Y(1))
CALL GBIT(IARRAY,II+1,Y(2))
C
CALCULATE ALL HAMMING DISTANCES...
C
DO 4 I=1,2
4
CHECK(I)=MOD(0+Y(I),2)
JI=1
DO 5 I=3,4
CHECK(I)=MOD(1+Y(JI),2)
5
JI=JI+1
HAMDIS(1)=CHECK(1)+CHECK(2)
HAMDIS(2)=CHECK(3)+CHECK(4)
HAMDIS(3)=CHECK(3)+CHECK(2)
HAMDIS(4)=CHECK(1)+CHECK(4)
C
COMPUTE ALL METRICS OUT OF EACH STATE....
C
C OUT OF STATE 0...
TMETRI(1)=RMETRI(1,K)+HAMDIS(1)
TMETRI(2)=RMETRI(1,K)+HAMDIS(2)
C OUT OF STATE 1....
TMETRI(3)=RMETRI(2,K)+HAMDIS(4)
TMETRI(4)=RMETRI(2,K)+HAMDIS(3)
C OUT OF STATE 2...
TMETRI(5)=RMETRI(3,K)+HAMDIS(3)
TMETRI(6)=RMETRI(3,K)+HAMDIS(4)
C OUT OF STATE 3...
TMETRI(7)=RMETRI(4,K)+HAMDIS(2)
TMETRI(8)=RMETRI(4,K)+HAMDIS(1)
C
C OUT OF STATE 4...
TMETRI(9)=RMETRI(5,K)+HAMDIS(2)
TMETRI(10)=RMETRI(5,K)+HAMDIS(1)
OUT OF STATE 5...
TMETRI(11)=RMETRI(6,K)+HAMDIS(3)
TMETRI(12)=RMETRI(6,K)+HAMDIS(4)

OUT OF STATE 6...
TMETRI(13)=RMETRI(7,K)+HAMDIS(4)
TMETRI(14)=RMETRI(7,K)+HAMDIS(3)

OUT OF STATE 7...
TMETRI(15)=RMETRI(8,K)+HAMDIS(1)
TMETRI(16)=RMETRI(8,K)+HAMDIS(2)

OUT OF STATE 8...
TMETRI(17)=RMETRI(9,K)+HAMDIS(3)
TMETRI(18)=RMETRI(9,K)+HAMDIS(4)

OUT OF STATE 9...
TMETRI(19)=RMETRI(10,K)+HAMDIS(2)
TMETRI(20)=RMETRI(10,K)+HAMDIS(1)

OUT OF STATE 10...
TMETRI(21)=RMETRI(11,K)+HAMDIS(1)
TMETRI(22)=RMETRI(11,K)+HAMDIS(2)

OUT OF STATE 11...
TMETRI(23)=RMETRI(12,K)+HAMDIS(4)
TMETRI(24)=RMETRI(12,K)+HAMDIS(3)

OUT OF STATE 12...
TMETRI(25)=RMETRI(13,K)+HAMDIS(4)
TMETRI(26)=RMETRI(13,K)+HAMDIS(3)

OUT OF STATE 13...
TMETRI(27)=RMETRI(14,K)+HAMDIS(1)
TMETRI(28)=RMETRI(14,K)+HAMDIS(2)

OUT OF STATE 14...
TMETRI(29)=RMETRI(15,K)+HAMDIS(2)
TMETRI(30)=RMETRI(15,K)+HAMDIS(1)

OUT OF STATE 15...
TMETRI(31)=RMETRI(16,K)+HAMDIS(3)
TMETRI(32)=RMETRI(16,K)+HAMDIS(4)

OUT OF STATE 16...
TMETRI(33)=RMETRI(17,K)+HAMDIS(2)
TMETRI(34)=RMETRI(17,K)+HAMDIS(1)

OUT OF STATE 17...
TMETRI(35)=RMETRI(18,K)+HAMDIS(3)
TMETRI(36)=RMETRI(18,K)+HAMDIS(4)

OUT OF STATE 18...
TMETRI(37) = RMETRI(19,K) + HAMDIS(4)
TMETRI(38) = RMETRI(19,K) + HAMDIS(3)

C OUT OF STATE 19...
TMETRI(39) = RMETRI(20,K) + HAMDIS(1)
TMETRI(40) = RMETRI(20,K) + HAMDIS(2)

C OUT OF STATE 20...
TMETRI(41) = RMETRI(21,K) + HAMDIS(1)
TMETRI(42) = RMETRI(21,K) + HAMDIS(2)

C OUT OF STATE 21...
TMETRI(43) = RMETRI(22,K) + HAMDIS(4)
TMETRI(44) = RMETRI(22,K) + HAMDIS(3)

C OUT OF STATE 22...
TMETRI(45) = RMETRI(23,K) + HAMDIS(3)
TMETRI(46) = RMETRI(23,K) + HAMDIS(4)

C OUT OF STATE 23...
TMETRI(47) = RMETRI(24,K) + HAMDIS(2)
TMETRI(48) = RMETRI(24,K) + HAMDIS(1)

C OUT OF STATE 24...
TMETRI(49) = RMETRI(25,K) + HAMDIS(4)
TMETRI(50) = RMETRI(25,K) + HAMDIS(3)

C OUT OF STATE 25....
TMETRI(51) = RMETRI(26,K) + HAMDIS(1)
TMETRI(52) = RMETRI(26,K) + HAMDIS(2)

C OUT OF STATE 26...
TMETRI(53) = RMETRI(27,K) + HAMDIS(2)
TMETRI(54) = RMETRI(27,K) + HAMDIS(1)

C OUT OF STATE 27...
TMETRI(55) = RMETRI(28,K) + HAMDIS(3)
TMETRI(56) = RMETRI(28,K) + HAMDIS(4)

C OUT OF STATE 28...
TMETRI(57) = RMETRI(29,K) + HAMDIS(3)
TMETRI(58) = RMETRI(29,K) + HAMDIS(4)

C OUT OF STATE 29...
TMETRI(59) = RMETRI(30,K) + HAMDIS(2)
TMETRI(60) = RMETRI(30,K) + HAMDIS(1)

C OUT OF STATE 30...
TMETRI(61) = RMETRI(31,K) + HAMDIS(1)
TMETRI(62) = RMETRI(31,K) + HAMDIS(2)

C OUT OF STATE 31...
TMETRI(63) = RMETRI(32,K) + HAMDIS(4)
TMETRI(64) = RMETRI(32,K) + HAMDIS(3)
C TAKE CARE OF FIRST 5 STEPS THRU TRELLIS...

GO TO (100,200,300,404,504),K
DO 6 I=1,32
   IF(TMRET(I).LT(TMRET(I+32)) GO TO 7
   IF(TMRET(I+32).LT(TMRET(I)) GO TO 8
   CALL ARAND(XC)
   IF(XC.LT.0.5) GO TO 7
   GO TO 8
7    RMRET(I,K+1)=TMRET(I)
     PATHNO(I,K+1)=I
     GO TO 6
8    RMRET(I,K+1)=TMRET(I+32)
     PATHNO(I,K+1)=I+32
6    CONTINUE
    GO TO 333

C K=1 CASE
100 L=2
    GO TO 502

C K=2 CASE...
200 L=4
    GO TO 502

C K=3 CASE
300 L=8
    GO TO 502

C K=4 CASE...
404 L=16
    GO TO 502

C K=5 CASE
504 L=32
502 DO 501 KI=1,L
   RMRET(KI,K+1)=TMRET(KI)
501 PATHNO(KI,K+1)=KI
333 II=II+2

C IF ONLY 1,2,3, OR 4 BITS TRANSMITTED DO THE FOLLOWING...
C GO TO (150,250,350,450),NB
C ALGORITHM TO FIND SMALLEST OF 'NCOMPA'
C RMETRICS IN LAST COLUMN...
C SET UP INDICES...
C NCOMPA=32
399 I=1
   J=2
K=1

HEART OF ALGORITHM...

400 IF(RMETRI(I,KNB).LT.RMETRI(J,KNB))GO TO 500
GO TO 600

KEEP SAME METRIC, COMPARE NEXT ONE DOWN...

500 JK=I
J=J+1
K=K+1
525 IF(K.LT.NCOMPA)GO TO 400
GO TO 800

CHANGE INDEX OF COMPARISON TO NEXT ONE;
COMPARE NEXT ONE DOWN....

600 JK=J
I=J
J=J+1
K=K+1
GO TO 525
800 CONTINUE
GO TO 665

NCOMPA=2
GO TO 399
NCOMPA=4
GO TO 399
NCOMPA=8
GO TO 399
NCOMPA=16
GO TO 399

NOW RETRACE TRELLIS PATH NOS. & FIND TRANSITIONS...
THIS MUST BE DONE STARTING ON LAST COLUMN AND WORKING
BACKWARDS UNTIL WE GET TO THE BEGINNING OF THE
TRELLIS TREE.....
665 DO 666 I=1,NB
J=KNB+I-1

IS 'JK' ODD OR EVEN??

XJK=JK
A1=XJK/2.
I1=A1
A2=A1-I1
IF(A2.EQ.0.)GO TO 667
IF(A2.EQ.0.5)GO TO 668
WRITE(IWTM,669)
669 FORMAT(5X,'CDEC18: PROB. EXISTS IN TRACE BACK SECTION')
GO TO 9000

EVEN CASE....
C
667  ITRANS(J)=1
     JJK=Al
     GO TO 670

C
C  ODD CASE...
C
668  ITRANS(J)=0
C  ROUND UP...
     Al=Al+0.5
     JJK=Al

C
C  THIS SECTION DETERMINES THE VALUE OF
C  THE NEXT 'JK' SO THAT WE CAN TAKE
C  ONE MORE STEP BACKWARD THRU THE
C  TRELLIS...
C
670  IF(PATHNO(JK,J).EQ.JK)GO TO 671
     IF(PATHNO(JK,J).EQ.JK+32)GO TO 672
     WRITE(IWTM,673)
673  FORMAT(5X,'CDEC18: ERROR EXISTS IN PATHNO ARRAY??!!')
     GO TO 9000
671  JK=JK
     GO TO 666
672  JK=JJK+16
666  CONTINUE
C
C  PACK BITS AND PREPARE TO GET OUT
C
C
9000  DO 9090 K=1,NB
     IDEC=ITRANS(K+1)
     I=K
9090  CALL PBIT(KARRAY,I,IDEC)
     DI(1,IBEX)=KARRAY(1)
     DI(2,IBEX)=KARRAY(2)
     RETURN
END
SUBROUTINE CBITCH
C*********************************************************************************
C SUBROUTINE BITC10 SIMULATES THE ADDITION OF AWGN TO A
C CONVOLUTIONALLY ENCODED SET OF BITS COMING FROM THE
C CENCODE BLOCK.
C
C SUBROUTINE INPUT: DI(1,NBEXA)=FIRST 24 ENCODED BITS.
C DI(2,NBEXA)=2ND 24 BITS.
C DQ(1,NBEXA)=NO. OF BITS IN DI(1,NBEXA)
C DQ(2,NBEXA)=NO. OF BITS IN DI(2,NBEXA)
C
C SUBROUTINE OUTPUT: DI(1,IBEX)=BITCHANGED WORD IN PBIT FORMAT.
C DI(2,IBEX)=2ND HALF OF BITCHANGED WORD.
C DQ(1,IBEX)=NO. OF BITS IN DI(1,IBEX)
C DQ(2,IBEX)=NO. OF BITS IN DI(2,IBEX)
C DQ(3,IBEX)=WORD TRANSITION MARKER (WHICH IS
C ALWAYS 1.0 FOR BIT CHANNEL SIMULA-
C TIONS)
C
C VARIABLES ENTERED: SIGNAL-TO-NOISE RATIO,SNR
C TYPE OF CURVE BEING MODELED,ICUR
C
C SUBROUTINES CALLED: GBIT,ARAND,PBIT.
C
C*********************************************************************************
C COMMON BLOCK
C*********************************************************************************
DIMENSION LARRAY(2),KARRAY(2)
COMMON/IO/IRTM,IWTM,LGDR,MODE,LSMML,LSMMD,LUWF,LTEMP,ITITLE(20),
ILPT,IDsK1,IDsK2,IDsK3,IDsK4,IP1,IP2,IP3,IP4,IRTPO
COMMON/MON/IMB(12,S),IDB(12,S),RIVL(12),INB(12),IOP(12),IBLS(80),
* IBLN(80,2),IBMAX,MCMD
COMMON/EXEC/GDR(20),IBEX,TINC,CTIME,NSPV,SINC,TIME(3,31),DI(3,31),
* DQ(3,31),IBEXM,NBEXA,NBEXB,NBEXC
EQUIVALENCE (GDR(1),TSTART),(GDR(2),TSTOP),(GDR(3),ISEED)

DATA CUE MESSAGE

111 CALL TERMWR('BITC10: ENTER SNR(IN DB) ',28,1,SNR)
IF(SNR.LT.0.)GO TO 112
IF(SNR.GT.S.)GO TO 112
CALL TERMD
GO TO 222
112 WRITE(IWTM,113)
113 FORMAT(2X,'SNR VALUE SHOULD BE BET. 0-->5 DB')
GO TO 111
222 CALL TERMWR('BITC10: ENTER CURVE TO BE MODELED: 1-FSK,2-PSK,3-DPSK
*,4-VITERBI ',64,2,ICUR)
CALL TERMD
DDB = SNR/10.
SNR1 = 10**(DDB)
GO TO (250, 260, 270, 280), ICUR
WRITE(IWTM, 290)
290 FORMAT(2X, 'USE ONE OF INDICATED VALUES...')
GO TO 222
C
C FOR 'FSK' SIGNAL...
C
C PE = 0.5*EXP(-E/2*N)
C
250 SNR2 = -0.5*SNR1
FSKPRO = 0.5*EXP(SNR2)
THRESH = 1 - FSKPRO
GO TO 667
C
C FOR 'PSK' SIGNAL...
C
C PE = 0.5*(1 - ERF(SQRT(E/N))
260 SNR2 = SQRT(SNR1)
PSKPRO = 0.5*ERFNC(SNR2)
THRESH = 1 - PSKPRO
GO TO 667
C
C FOR 'DPSK' SIGNAL...
C
C PE = 0.5*EXP(-E/N)
270 SNR2 = -SNR1
DPSKPR = 0.5*EXP(SNR2)
THRESH = 1 - DPSKPR
GO TO 667
C
C FOR 'VITERBI DECODING' SIGNAL...
C
C PE = ((2*SQRT(P*(1-P)))**5
-------------
1 - 4*SQRT(P*(1-P))
C
280 SNR2 = SQRT(SNR1)
PE = 0.5*ERFNC(SNR2)
PE1 = SQRT(PE*(1-PE))
PE2 = (2*PE1)**5
PE3 = 1 - 4*PE1
VITPRO = PE2/PE3
THRESH = 1 - VITPRO
667 WRITE(IWTM, 666) THRESH
666 FORMAT(5X, 'THRESH=', F10.7)
NSPV = 1
DQ(3, IBEX) = 1.0
RETURN
ENTRY RBITCH
C
C DATA INITIALIZATION
C
NBIT1=DQ(1,NBEXA)
NBIT2=DQ(2,NBEXA)
NBIT=NBIT1+NBIT2
DQ(1,IBEX)=NBIT1
DQ(2,IBEX)=NBIT2

C MAIN PROGRAM...
C
K=1
IARRAY(1)=DI(1,NBEXA)
IARRAY(2)=DI(2,NBEXA)
C ZERO OUT OUTPUT ARRAY...
DO 1 KK=1,2
1 KARRAY(KK)=0
C MAIN LOOP...
DO 2 I=1,NBIT
IQ=I
CALL GBIT(IARRAY,IQ,IBIT)
IF(IBIT.EQ.0)GO TO 3
NEWBIT=0
GO TO 4
3 NEWBIT=1
4 CALL ARAND(XC)
C DECISION: IS NOISE > THRESH??
C IF YES, CHANGE BIT.........
C IF NO, LEAVE IT ALONE AND PREPARE TO PACK....
IF(XC.GT.THRESH)GO TO 5
KBIT=IBIT
GO TO 6
5 KBIT=NEWBIT
6 CALL PBIT(KARRAY,K,KBIT)
2 K=K+1
DI(1,IBEX)=KARRAY(1)
DI(2,IBEX)=KARRAY(2)
RETURN
END

C THE FOLLOWING FUNCTION SUBPROGRAM GENERATES 'ERFC'
C WITHOUT USING CANNED IBM ROUTINE......
C
FUNCTION CERFNC(X)
DIMENSION B(9)
IF(X.LT.0)GO TO 1
FLAG=0.
GO TO 3
1 FLAG=1
X=ABS(X)
3 IF((X.GE.0.).AND.(X.LT.1.0))GO TO 5
IF((X.GE.1.0).AND.(X.LE.2.040452))GO TO 10
IF((X.GT.2.040452).AND.(X.LE.13.306))GO TO 20
IF(X.GT.13.306)GO TO 30
WRITE(IWTM,40)
40 FORMAT(5X,'BITClO:ERROR OCCURRED IN ERFC SUBPROGRAM...')
GO TO 100
FOR 0 < X < 1.0
ERF(X) = X * (A0 + A1 * X**2 + A2 * X**4 + ... + A5 * X**10)
ERFC(X) = 1 - ERF(X)

A0 = 0.12837912
A1 = -3.7612326
A2 = 1.1280180
A3 = -0.02671132
A4 = 0.00491756
A5 = -0.00056314
X1 = A1 * X * X
X2 = A2 * X**4
X3 = A3 * X**6
X4 = A4 * X**8
X5 = A5 * X**10
ERFNC = X * (A0 + X1 + X2 + X3 + X4 + X5)
ERFNC = ERFNC + X
CERFNC = 1 - ERFNC
IF(FLAG.EQ.0) GO TO 100
GO TO 98

FOR 1.0 < X < 2.0
ERFC(X) = B0 + B1 * Z + B2 * Z**2 + ... + B9 * Z**9,
WHERE Z = X - T0

T0 = 1.709472
Z = X - T0
B0 = 0.01562498
B(1) = -0.06071803
B(2) = 0.10379590
B(3) = -0.09805128
B(4) = -0.04920667
B(5) = 0.00423922
B(6) = -0.01067874
B(7) = -0.00633025
B(8) = -0.00043551
B(9) = 0.00150534

CERFNC = B(9) * Z + B(8)
J = 8
DO 11 I = 1, 7
    J = J - 1
11     CERFNC = CERFNC * Z + B(J)
CERFNC = CERFNC * Z + B0
IF(FLAG.EQ.0) GO TO 100
GO TO 98

FOR 2.040452 < X < 13.306
ERFC(X) = EXP(-X**2) * (C0 + C1 * X**4 + C2 * X**2 + C3) + C1 * X**4 + C2 * X**2 + C3
    X**6 + D1 * X**4 + D2 * X**2

C
C
20  CO=.56418951
   C1=-.28208106
   C2=-.96210474
   C3=-.14431247
   D1=4.9078886
   D2=4.1920066
   XSQ=EXP(-X*X)
   XMULT=XSQ/X
   X1=C1*X**4
   X2=C2*X*X
   X3=X**6
   X4=D1*X**4
   X5=D2*X*X
   FUNC=CO+(X1+X2+C3)/(X3+X4+X5)
   CERFNC=XMULT*FUNC
   IF(FLAG.EQ.0)GO TO 100
   GO TO 98
C
C FOR X>13.306
C ERFC(X)=0.
C
30  CERFNC=0.
    IF(FLAG.EQ.0)GO TO 100
98  CERFNC=2-CERFNC
    X=-X
100 RETURN
END