Further Developments in the Communication Link and Error ANalysis (CLEAN) Simulator

NASA GRANT NAG5-2006

Final Report
July 1, 1993 - June 30, 1994

Submitted to:

Mr. Warner Miller
Code 728.4
Instrument Electronic Systems Branch
Engineering Directorate
NASA/Goddard Space Flight Center
Greenbelt, MD 20771
301-286-8183

Submitted by:

William J. Ebel, Ph.D.
Frank M. Ingels, Ph.D.
Mississippi State University
Drawer EE
Mississippi State, MS 39762
601-325-3912

December 1995

Abstract

This report documents work performed for NASA Grant NAG5-2006 for the period July 1, 1993 through June 30, 1994. During this period, significant developments to the Communication Link and Error ANalysis (CLEAN) simulator were completed. Many of these were reported in the Semi-Annual report dated December 1993 which has been included in this report in Appendix A. Since December 1993, a number of additional modules have been added involving Unit-Memory Convolutional codes (UMC). These are:

1) Unit-Memory Convolutional Encoder module (UMCEncd)

2) Hard decision Unit-Memory Convolutional Decoder using the Viterbi decoding algorithm (VitUMC)

3) A number of utility modules designed to investigate the performance of UMC's such as

   a) UMC column distance function (UMCdc)

   b) UMC free distance function (UMCdfree)

   c) UMC row distance function (UMCdr)

   d) UMC Transformation (UMCTrans)

The study of UMC's was driven, in part, by the desire to investigate high-rate convolutional codes which are better suited as inner codes for a concatenated coding scheme. A number of high-rate UMC's were found which are good candidates for inner codes.

Besides the further developments of the simulation, a study was performed to construct a table of the best known Unit-Memory Convolutional codes. To date, a total of 100 new UMC's were found which are better than any previously known UMC's. These results have been submitted and accepted to the *IEEE Transactions on Information Theory* and will appear in print soon. A copy of the final paper is included in Appendix B for reference.

Finally, a preliminary study of the usefulness of the Periodic Convolutional Interleaver

Will Ebel
Assistant Professor

cc: Ms. Gloria R. Blanchard

I. Introduction

During the past year, CLEAN capabilities have grown substantially. Most of the new programs are briefly described in the semi-Annual report included in Appendix A. Among the developments is the integration of the RICE compression/decompression software into the simulation. Not included in that report are a number of modules involving encoding, decoding, and analysis of Unit-Memory convolutional codes.

Appendix A

Semi-Annual Report for the period July 1, 1993 through December 30, 1993

The Communication Link and Error ANalysis (CLEAN) Simulator

NASA GRANT NAG5-2006
July 1, 1993 - June 30, 1994

Semi-Annual Report
July 1, 1993 - December 30, 1993

Submitted to:

Mr. Warner Miller
Code 728.4
Instrument Electronic Systems Branch
Engineering Directorate
NASA/Goddard Space Flight Center
Greenbelt, MD 20771
301-286-8183

Submitted by:

William J. Ebel, Ph.D.
Frank M. Ingels, Ph.D.
Shane Crowe
Mississippi State University
Drawer EE
Mississippi State, MS 39762
601-325-3912

December 1993

Table of Contents

Abstract

This report documents work performed for NASA Grant NAG5-2006 for the period July 1, 1993 through December 30, 1993. During this period, significant developments to the Communication Link and Error ANalysis (CLEAN) simulator were completed and include:

1) Soft decision Viterbi decoding

2) Node synchronization for the Soft decision Viterbi decoder

3) Insertion/deletion error programs

4) Convolutional Encoder

5) Programs to investigate new convolutional codes

6) Pseudo-Noise sequence generator

7) Soft decision data generator

8) RICE compression/decompression (integration of RICE code generated by Pen-Shu Yeh at Goddard Space Flight Center)

9) Markov Chain channel modeling

10) % complete indicator when a program is executed

11) Header documentation

12) Help utility

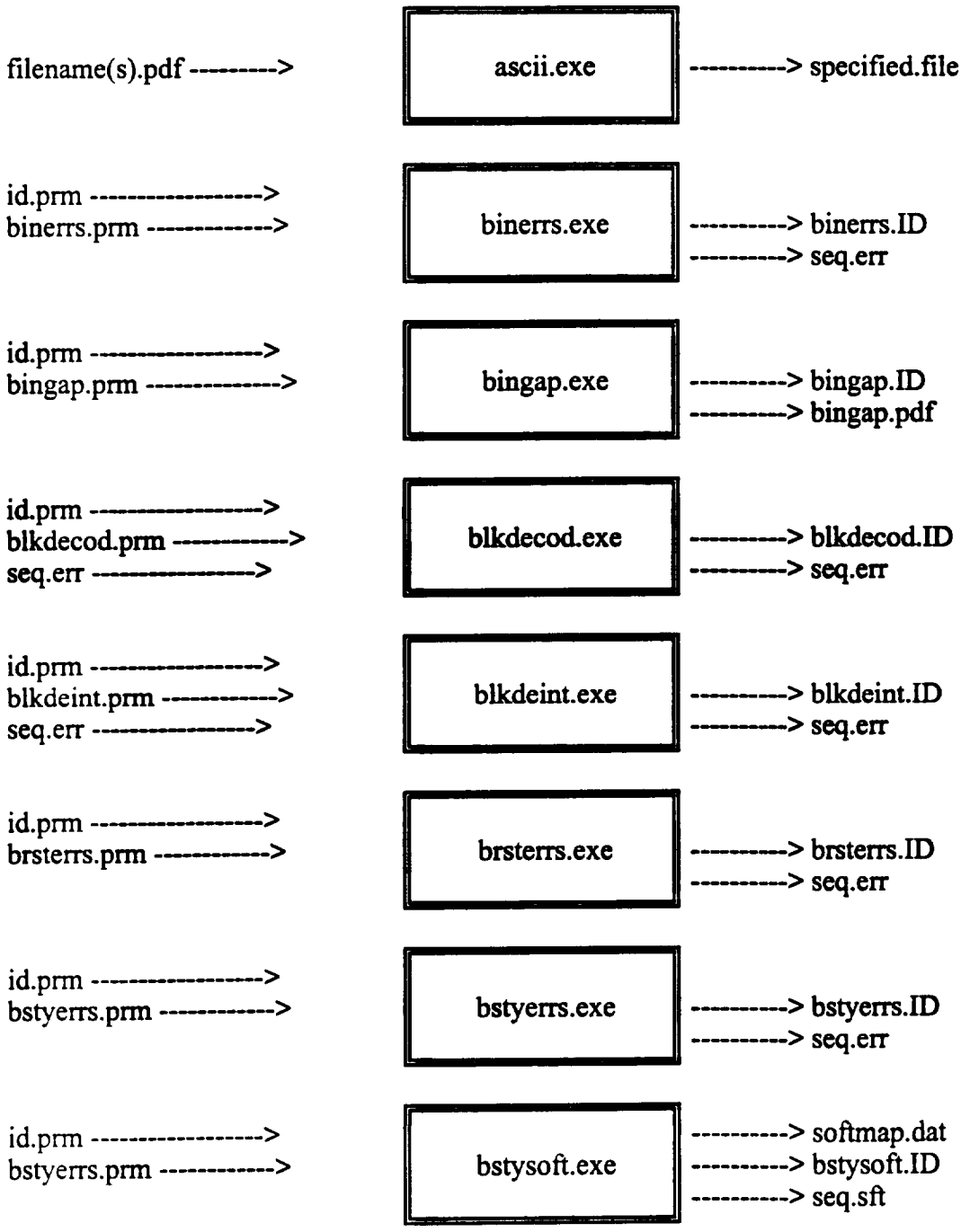
The CLEAN simulation tool is now capable of simulating a very wide variety of satellite communication links including the TDRSS downlink with RFI. The RICE compression/decompression schemes allow studies to be performed on error effects on RICE decompressed data. The Markov Chain modeling programs allow channels with memory to be simulated. Memory results from filtering, forward error correction encoding/decoding, differential encoding/decoding, channel RFI, non-linear transponders and from many other satellite system processes.

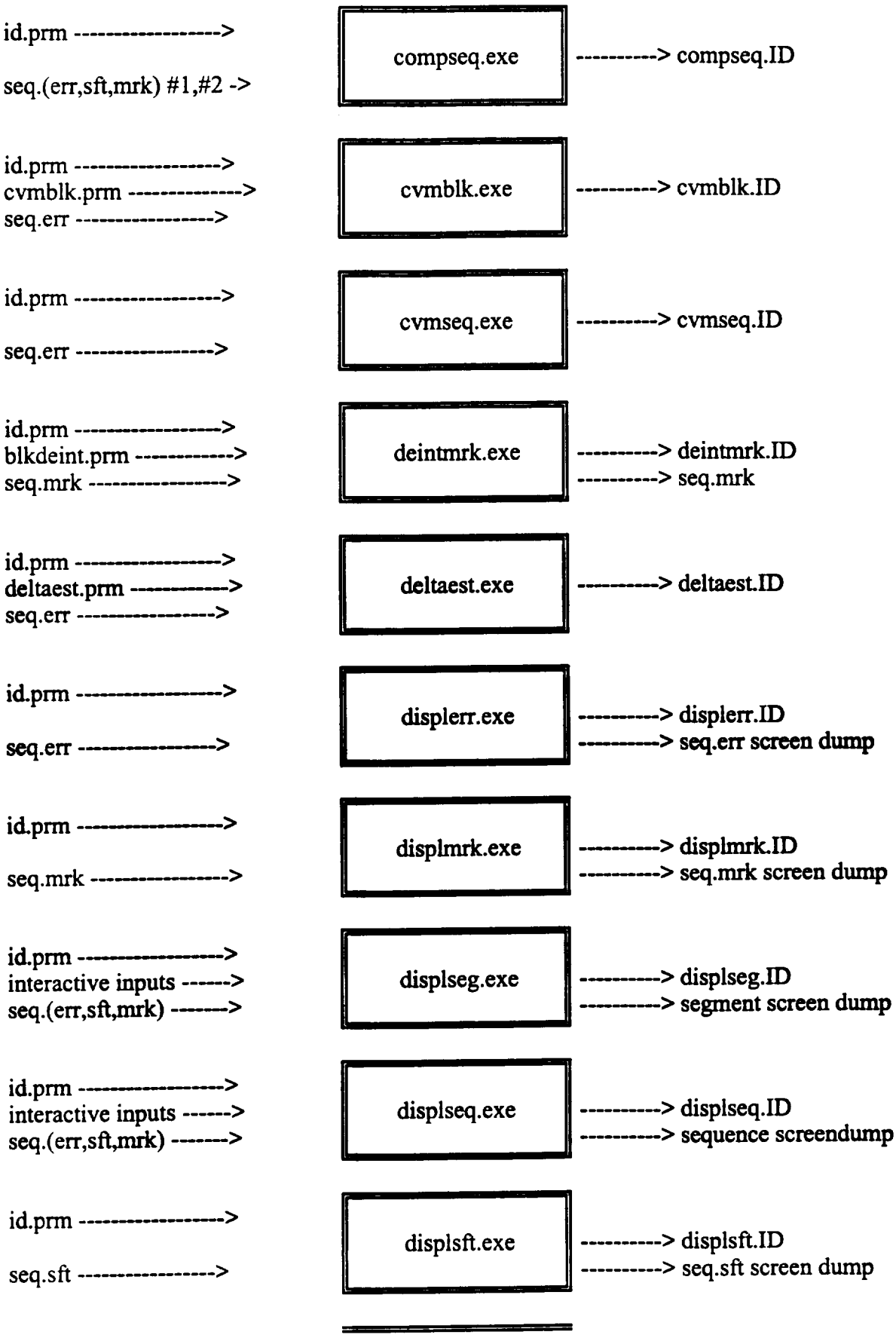
Besides the development of the simulation, a study was performed to determine whether the PCI provides a performance improvement for the TDRSS downlink. There exist RFI with several duty cycles for the TDRSS downlink. We conclude that the PCI does not improve performance for any of these interferers except possibly one which occurs for the TDRS East. Therefore, the usefulness of the PCI is a function of the time spent transmitting data to the WSGT through the TDRS East transponder.

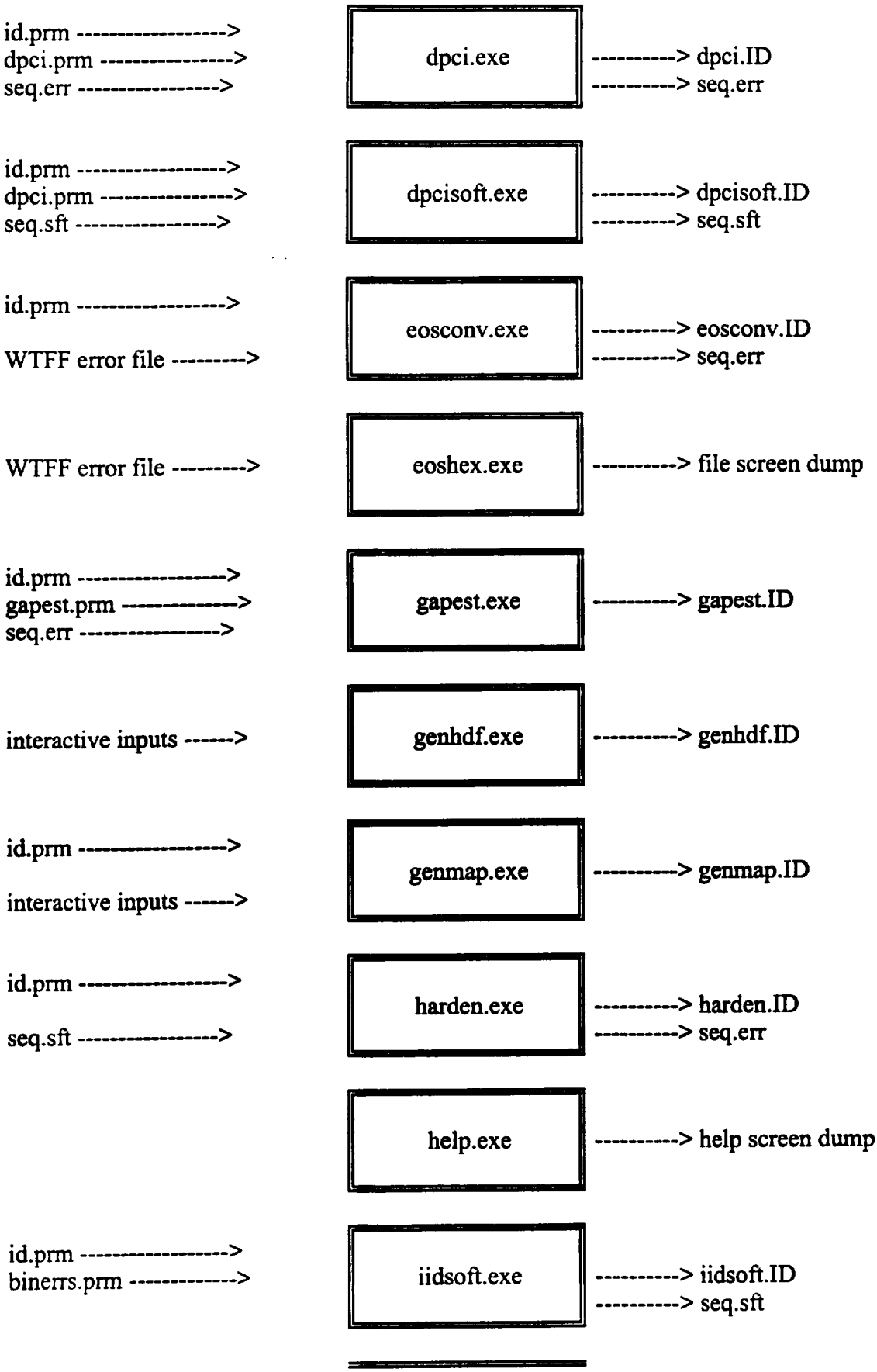
I. Introduction

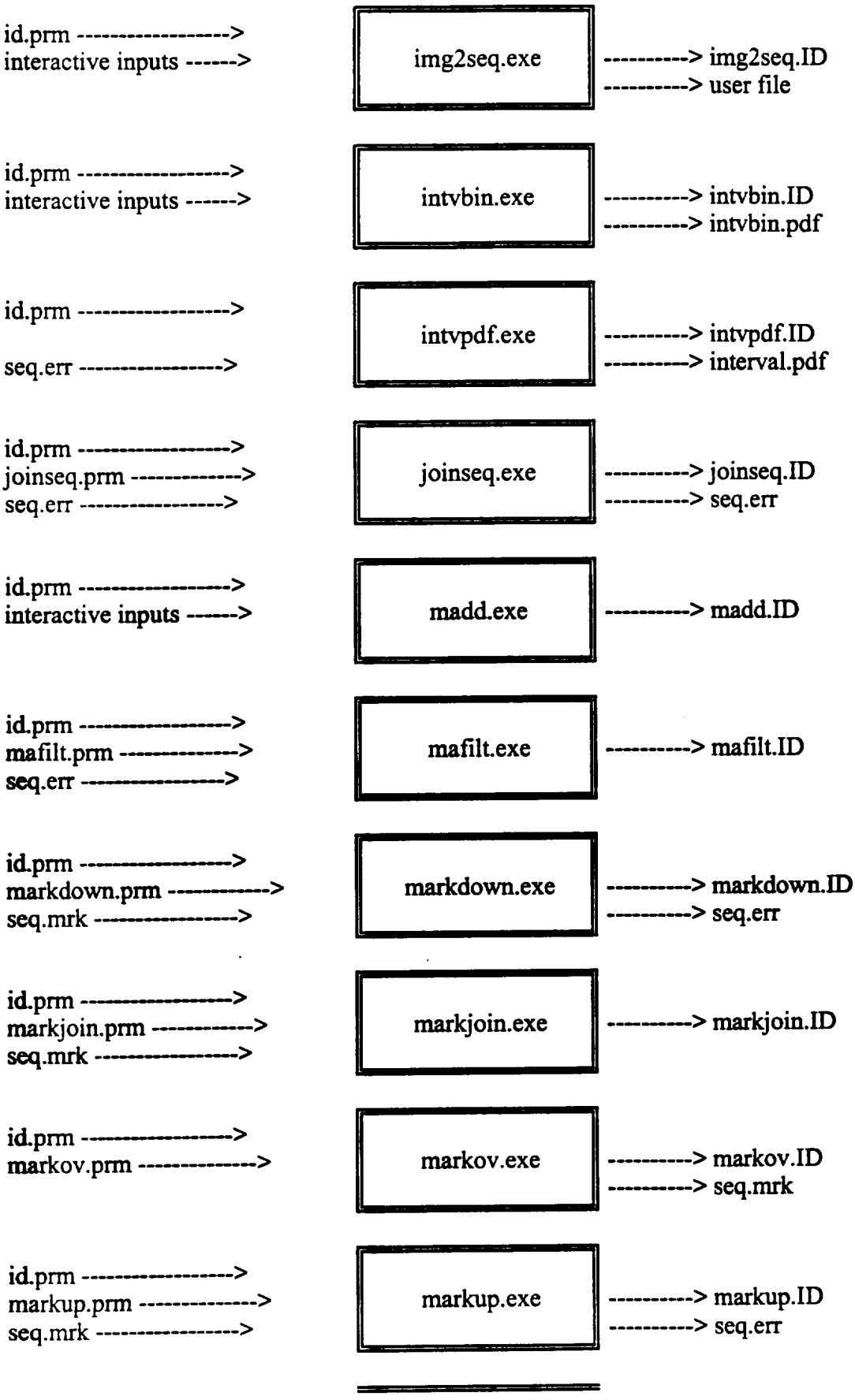
During the past 6 months, CLEAN capabilities have grown substantially. Most of the new programs are briefly described in Section II. Among the developments is the integration of the RICE compression/decompression software into the simulation. In the Appendix, the theory of RICE compression is described along with a description of CLEAN implementation. In Section III, some results on the question of whether the PCI is really necessary for the TDRSS downlink is discussed.

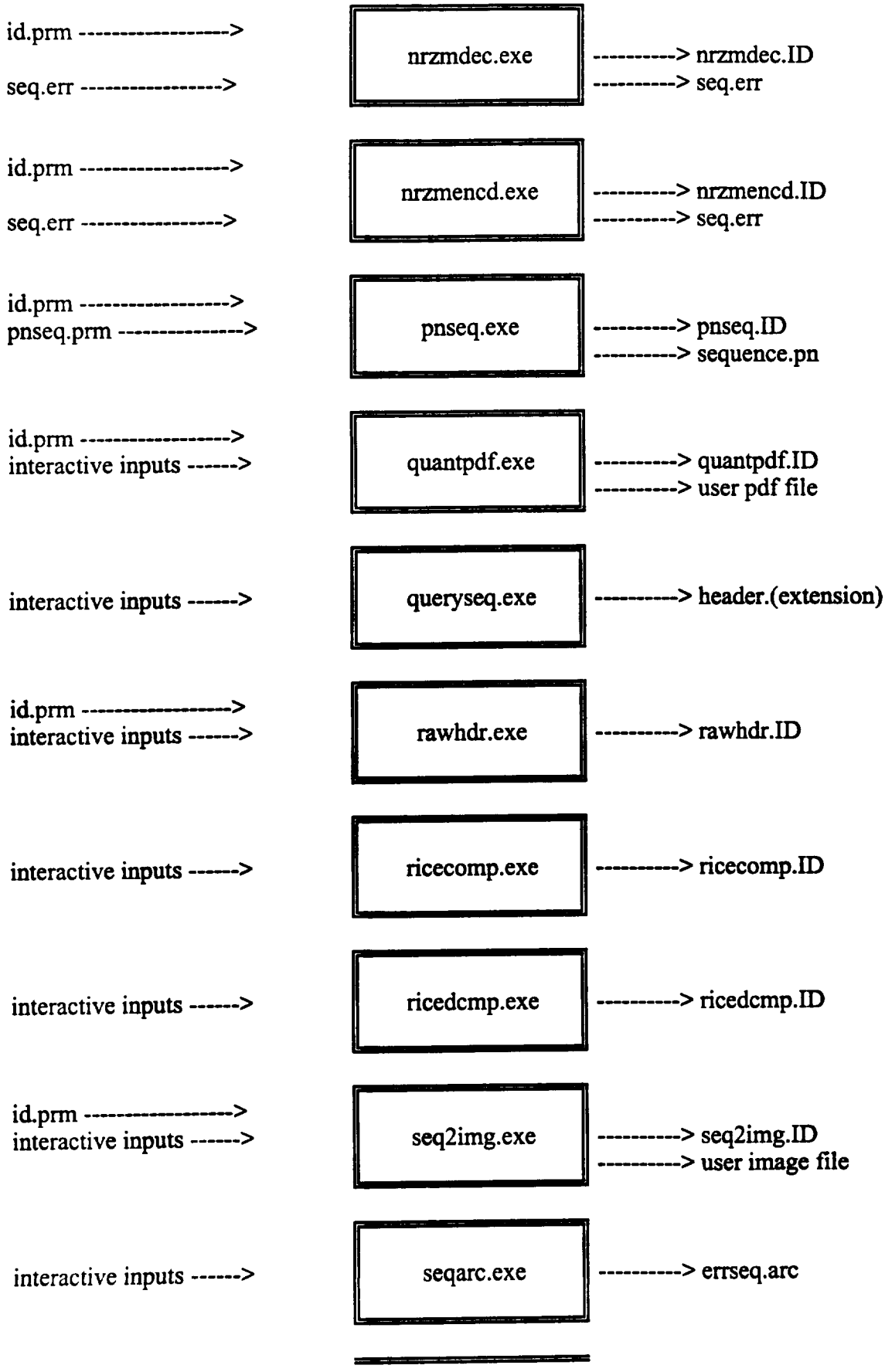
To help run the source code, the following list in given which provides a quick overview of the required input files and the output files which are associated with each program of CLEAN.

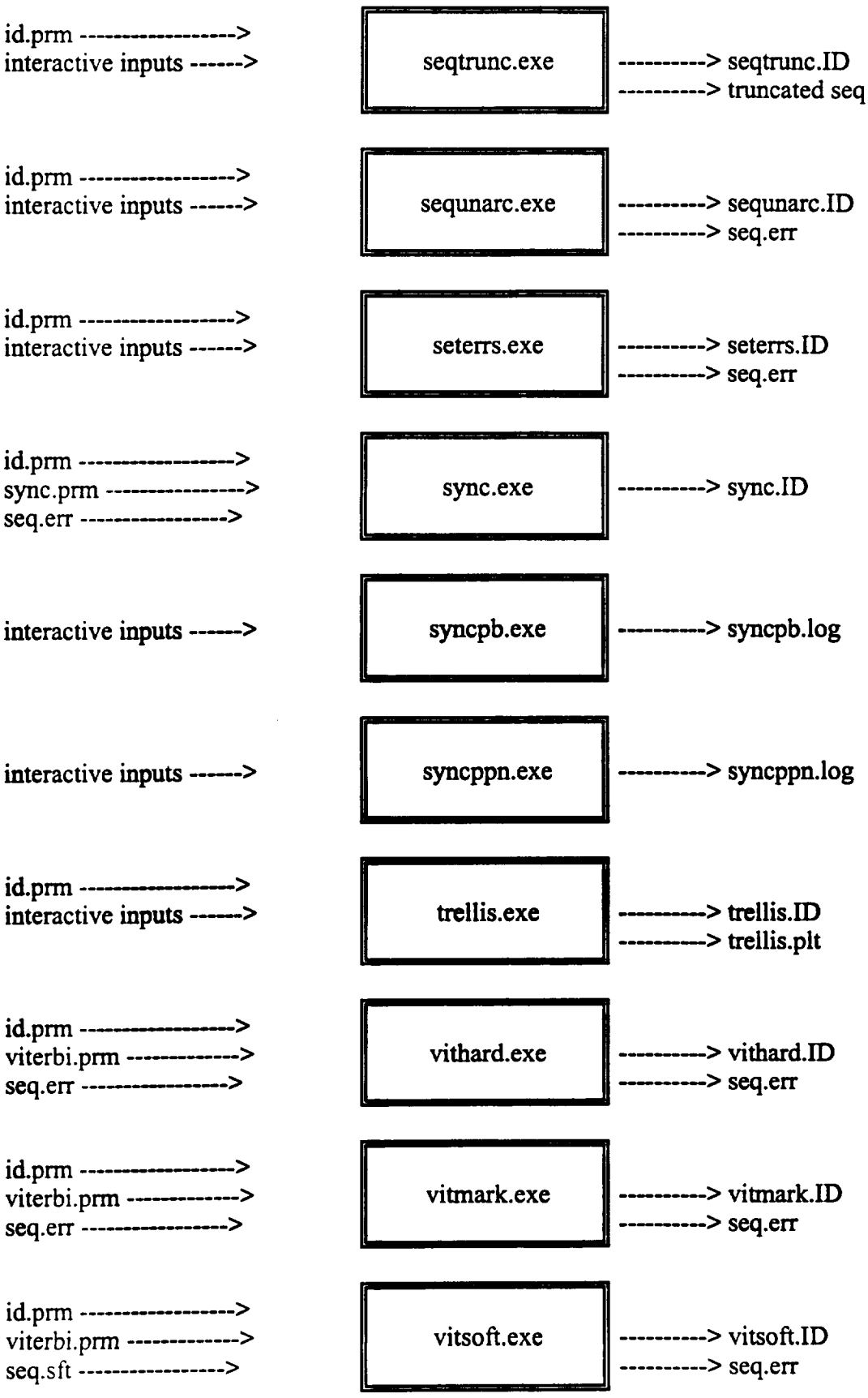



id.prm ----------->
seq.(err,sft,mrk) #1,#2 ->
compseq.exe
----------> compseq.ID
id.prm ----------->
cvmblk.prm ----------->
seq.err ----------->
cvmblk.exe
----------> cvmblk.ID
id.prm ----------->
seq.err ----------->
cvmseq.exe
----------> cvmseq.ID
id.prm ----------->
blkdeint.prm ----------->
seq.mrk ----------->
deintmrk.exe
----------> deintmrk.ID
----------> seq.mrk
id.prm ----------->
deltaest.prm ----------->
seq.err ----------->
deltaest.exe
----------> deltaest.ID
id.prm ----------->
seq.err ----------->
displerr.exe
----------> displerr.ID
----------> seq.err screen dump
id.prm ----------->
seq.mrk ----------->
displmrk.exe
----------> displmrk.ID
----------> seq.mrk screen dump
id.prm ----------->
interactive inputs ------>
seq.(err,sft,mrk) ------->
displseg.exe
----------> displseg.ID
----------> segment screen dump
id.prm ----------->
interactive inputs ------>
seq.(err,sft,mrk) ------->
displseq.exe
----------> displseq.ID
----------> sequence screendump
id.prm ----------->
seq.sft ----------->
displsft.exe
----------> displsft.ID
----------> seq.sft screen dump


id.prm ----------->
dpci.prm ----------->
seq.err ----------->
dpci.exe
----------> dpci.ID
----------> seq.err
id.prm ----------->
dpci.prm ----------->
seq.sft ----------->
dpcisoft.exe
----------> dpcisoft.ID
----------> seq.sft
id.prm ----------->
WTFF error file --------->
eosconv.exe
----------> eosconv.ID
----------> seq.err
WTFF error file --------->
eoshex.exe
----------> file screen dump
id.prm ----------->
gapest.prm ----------->
seq.err ----------->
gapest.exe
----------> gapest.ID
interactive inputs ------>
genhdf.exe
----------> genhdf.ID
id.prm ----------->
interactive inputs ------>
genmap.exe
----------> genmap.ID
id.prm ----------->
seq.sft ----------->
harden.exe
----------> harden.ID
----------> seq.err
help.exe
----------> help screen dump
id.prm ----------->
binerrs.prm ----------->
iidsoft.exe
----------> iidsoft.ID
----------> seq.sft


id.prm ----------->
interactive inputs ------>
img2seq.exe
----------> img2seq.ID
----------> user file
id.prm ----------->
interactive inputs ------>
intvbin.exe
----------> intvbin.ID
----------> intvbin.pdf
id.prm ----------->
seq.err ----------->
intvpdf.exe
----------> intvpdf.ID
----------> interval.pdf
id.prm ----------->
joinseq.prm ----------->
seq.err ----------->
joinseq.exe
----------> joinseq.ID
----------> seq.err
id.prm ----------->
interactive inputs ------>
madd.exe
----------> madd.ID
id.prm ----------->
mafilt.prm ----------->
seq.err ----------->
mafilt.exe
----------> mafilt.ID
id.prm ----------->
markdown.prm ----------->
seq.mrk ----------->
markdown.exe
----------> markdown.ID
----------> seq.err
id.prm ----------->
markjoin.prm ----------->
seq.mrk ----------->
markjoin.exe
----------> markjoin.ID
id.prm ----------->
markov.prm ----------->
markov.exe
----------> markov.ID
----------> seq.mrk
id.prm ----------->
markup.prm ----------->
seq.mrk ----------->
markup.exe
----------> markup.ID
----------> seq.err


id.prm ----------->
seq.err ----------->
nrzmdec.exe
----------> nrzmdec.ID
----------> seq.err
id.prm ----------->
seq.err ----------->
nrzmencd.exe
----------> nrzmencd.ID
----------> seq.err
id.prm ----------->
pnseq.prm ----------->
pnseq.exe
----------> pnseq.ID
----------> sequence.pn
id.prm ----------->
interactive inputs ------>
quantpdf.exe
----------> quantpdf.ID
----------> user pdf file
interactive inputs ------>
queryseq.exe
----------> header.(extension)
id.prm ----------->
interactive inputs ------>
rawhdr.exe
----------> rawhdr.ID
interactive inputs ------>
ricecomp.exe
----------> ricecomp.ID
interactive inputs ------>
ricedcmp.exe
----------> ricedcmp.ID
id.prm ----------->
interactive inputs ------>
seq2img.exe
----------> seq2img.ID
----------> user image file
interactive inputs ------>
seqarc.exe
----------> errseq.arc


id.prm ----------->
interactive inputs ------>
seqtrunc.exe
----------> seqtrunc.ID
----------> truncated seq
id.prm ----------->
interactive inputs ------>
sequnarc.exe
----------> sequnarc.ID
----------> seq.err
id.prm ----------->
interactive inputs ------>
seterrs.exe
----------> seterrs.ID
----------> seq.err
id.prm ----------->
sync.prm ----------->
seq.err ----------->
sync.exe
----------> sync.ID
interactive inputs ------>
syncpb.exe
----------> syncpb.log
interactive inputs ------>
syncppn.exe
----------> syncppn.log
id.prm ----------->
interactive inputs ------>
trellis.exe
----------> trellis.ID
----------> trellis.plt
id.prm ----------->
viterbi.prm ----------->
seq.err ----------->
vithard.exe
----------> vithard.ID
----------> seq.err
id.prm ----------->
viterbi.prm ----------->
seq.err ----------->
vitmark.exe
----------> vitmark.ID
----------> seq.err
id.prm ----------->
viterbi.prm ----------->
seq.sft ----------->
vitsoft.exe
----------> vitsoft.ID
----------> seq.err

II. Further Developments to Clean

This section briefly describes additional capabilities which have been added to CLEAN. The capabilities have been divided into two main sections. In Section A, additional error sequence manipulation programs, which represent system components, are briefly described and in Section B, programs written to evaluate theoretical formulas are briefly described.

A. Soft Decision Program Modules

To more accurately reflect the receiver, programs were written to simulate soft decision values which are output by the demodulator for the real TDRSS. These programs involve "soft" sequence generation programs as well as programs to mimic the receiver DPCI and Viterbi decoder on those soft values.

1. iidsoft

This program generates a "soft" error sequence with independent and identically distributed soft event occurrences. By definition, an ERROR sequence MUST refer to hard decision data at the demodulator output. In contrast, this program simulates 3-bit soft decision data which would be output by a soft decision demodulator, assuming that the signal transmitted corresponds to the transmission of a binary zero. The algorithm involves using the channel error probability, input by the user through the parameter file, to construct the conditional Normal densities for the random variable which would be input to a multilevel thresholder to determine the 3-bit soft decision output. For convenience, it is assumed that the received signals are identically +1,-1 for a binary 1,0 respectively and that the decision thresholds, used to construct the 3-bit soft decision data numbers, are located at equi-spaced distances around +1 and -1 inclusive. Then, the 3-bit binary number assigned to each level begins with 000 for the range below -1 and end with 111 for the range above +1. In summary, the thresholds are arbitrarily chosen as follows:

3-bit value	Binary rep.	Low Thresh.	High
7	111	infinity	3/2
6	110	3/2	1
5	101	2	1/2
4	100	1/2	0
3	011	0	-1/2
2	010	-1/2	-1
1	001	-1	-3/2
0	000	-3/2	- infinity

Note that the first binary value to be output is the least significant digit for the soft value. These threshold values were taken from Heller and Jacobs, "Viterbi Decoding for Satellite and Space Communication," IEEE Transactions Communication Technology, vol. COM19, no. 5, October 1971, pp. 835-848.

To determine the soft decision values for each signal output, the probability of occurrence for each level must be found and subsequently used to statistically determine the sequence output. The probability that the i(th) soft value occurs is stored in SoftProb(i) which can be found using the Q(x) function. As implemented below, the cumulative SoftProb is stored in SoftProb, that is, SoftProb(i) represents the probability that soft value i, or i-1, ..., or 0 occurs. This is done to optimize execution speed. It is possible to threshold the soft sequence with a threshold of 0 to perform hard decision demodulation.

This program inputs parameters from an ASCII data file with default name 'BinErrs.prm' and outputs the "soft" error sequence to data file with default name 'seq.sft'. In addition, various statistics are output to an ASCII data file with default name 'IIDSoft.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'BinErrs.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'seq.sft' file which contains a sequence (in packed format) with independent and identically distributed soft values. It does not matter whether the output file 'seq.sft' exists or not. If it exists, it is overwritten without a prompt to the user.

2. bstysoft

This program generates a "soft" error sequence with bursty errors. The method for generating the soft values is discussed in the previous section for the iidsoft program documentation. The application here is identical except that two SoftProb functions are required: one when a burst is occurring and one when no burst is occurring.

A discussion of the method by which the burst length and burst interval statistics are generated can be found in the documentation of program bstyerrs.for.

This program inputs parameters from an ASCII data file with default name 'BstyErrs.prm' and outputs the soft sequence to a data file with default name 'seq.sft'. In addition, various statistics are output to an ASCII data file with default name 'BstySoft.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'BstyErrs.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'seq.sft' file which contains a "soft" error sequence (in packed format). It does not matter whether the file 'seq.sft' exists or not. If it exists, it is overwritten without a prompt to the user.

Even though Poisson distributed bursts may overlap in theory, this program does not allow bursts to overlap. The user must take care to specify input parameters so that the probability of overlapping burst is negligible.

3. displsft

This program displays the soft sequence found in file 'seq.sft'. It is assumed that the 3-bit soft values stored in seq.sft are in the SSPS (Soft Sequence Packed Symbol) format.

4. harden

This program reads in 3-bit soft decision data and performs hard decision thresholding. This will effectively reduce the length of the sequence file by a factor of 3.

The program reads in the soft sequence by blocks and performs hard decision thresholding on each block and then writes the modified block back out to the 'seq.err' file. The program outputs several statistics to the user screen as well. Note that the sequence is read in from file 'seq.sft' (SeqType=2), with 3-bit soft data and is stored in file 'seq.err' (SeqType=1), with hard errors.

Executing the program causes the 'seq.sft' file to be read which contains a soft value sequence (in packed format). The 'seq.sft' file must exist prior to the execution of this program.

5. soften

This program maps binary data into soft values out of the soft decision demodulator. The method used to perform this mapping is to combine the data sequence with an already existing soft sequence. Consider a particular data bit and the corresponding soft value from the soft sequence. If the data bit is a zero, then the soft value which would occur at the demodulator output remains the same. However, if the data bit is a 1, then the soft value which would occur at the demodulator output is the bit complement of the corresponding soft value. The bit complement can be achieved by taking 8 and subtracting the base10 equivalent of the soft value. For a discussion of how soft values are generated at the demodulator output, see Section 1 above.

This program inputs the data from file 'seq.dat' and the soft sequence from file 'seq.sft' and stores the result in the 'seq.sft' file. In addition, various statistics are output to an ASCII data file with default name 'Soften.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

Executing the program causes the 'seq.sft' file to be modified. Before running this program, sequences 'seq.dat' and 'seq.sft' must exist.

6. dpcisoft

This program performs Periodic Convolutional Deinterleaving of the soft sequence found in file 'seq.sft'. It is assumed that the channel symbols corresponding to those values have already been interleaved using an (Ntaps,M) periodic convolution interleaver. The method used to implement the function of the periodic convolutional interleaver is a series of formulas as described below. These functions are applied to a portion of the 'seq.sft' array which is stored in a ring buffer.

The method used to implement the deinterleaver involves constructing a Tap offset array which gives the offset for the soft sequence index to deinterleave next, based upon the tap position of the deinterleaver commutator. The Cycle offset is then used to determine the offset for the current commutator cycle number which is also used to determine the soft sequence index to deinterleave.

Note that there is a problem deinterleaving the end of the 'seq.sft' file due to the sequential nature of the algorithm. The DPCI soft sequence file is truncated to eliminate the "don't cares".

This program inputs parameters from an ASCII data file with default name 'DPCI.prm' and outputs the soft sequence to data file with default name 'seq.sft'. In addition, various statistics are output to an ASCII data file with default name 'DPCISoft.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'DPCI.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'seq.sft' file which contains an soft sequence (in packed format) with deinterleaved values. The 'seq.sft' file must exist prior to the execution of this program.

7. vitsoft

This program performs soft decision Viterbi decoding assuming ANY data sequence is transmitted. The Viterbi decoding algorithm assumes that the trellis begins at the all zero state for the first received code symbol. The end of the decoding process does not terminate with flush bits. Instead, steady state Viterbi decoding is performed up to the end of the data seq.

This program inputs parameters from an ASCII data file with default name 'Viterbi.prm' and inputs the soft sequence from file 'seq.sft' and outputs the decoded data sequence to data file with default name 'seq.err'. In addition, various statistics are output to an ASCII data file with default name 'VitSoft.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'Viterbi.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'seq.err' file which contains an error sequence (in packed format) with the decoded error sequence. The 'seq.sft' file must exist prior to the execution of this program.

There are several assumptions associated with the implementation and output of this program.

1) The path with the maximum probability metric at the i(th) Trellis stage is used to find the decoded bit for the output

2) It is assumed that the convolutional encoder is either rate 1/2 or rate 1/3. It is straight forward to extrapolate this program to accommodate a rate 1/n encoder. It should also be possible to modify this program to accommodate a rate m/n encoder.

The Viterbi algorithm, as implemented here, updates the Trellis by iterating through each of the states at the next stage. The probability metric for each path entering a given state are computed and the survivor is kept while the other sequence is discarded. In case of a tie, a coin is flipped (via a Uniform RV in [0,1]) to determine the survivor. The survivor is identified by updating the MLStateTrace array. This array contains the state of the previous Trellis stage which connects to the given state being processed. For example, suppose that we are now processing the next stage in the Trellis, we first consider state 1 at the next stage. After investigating the probability metric for the two possible paths entering state 1, we find that the survivor path came from state 3 of the previous Trellis stage. Therefore, MLStateTrace(i,1) = 3 where i is the stage index.

To prevent overwriting the Metric array, two Metric arrays are alternately processed for each Trellis stage. This is why the algorithm performs two Trellis stage updates for each main loop. In the first Trellis stage update, the metrics are found in array MetricA and the new metrics are stored in MetricB. In the second Trellis stage update, the metrics are found in array MetricB and the new metrics are stored in MetricA.

The Trellis is defined via three arrays; PathCodeSym, PathLink, and PathBit. Since this program only accommodates rate 1/2 or 1/3 encoders, only two paths enter each state at a given trellis stage. Therefore, if there are N trellis states, then there are only 2*N possible paths between two trellis stages. These are sequentially numbered from 1 to 2*N where path number 1 and 2 enter state 1, path 3 and 4 enter state 2, etc. Array PathLink(i) gives the state number from which path i originates. Also, PathCodeSym(i) gives the code symbol associated with path i, and PathBit(i) gives the bit associated with path i. Taken together, these three arrays completely define the steady state trellis.

8. vit3sync

This program operates like the vitsoft program. However, this program mimics exactly what happens in the real LV7017C hardware which is documented in an interoffice Memorandum written by James Wang and Wei-Chung Peng of LinCom with subject, "Simulation and Validation of Viterbi Decoder", TM-8719-05-09 and TM-8707-06, 01 March 1989. The vitsoft modifications performed to construct this program are as follows.

1) The metrics which are accumulated are arbitrarily chosen as described in an interoffice Memorandum mentioned above. This program mimics exactly what occurs in the real LV7017C hardware.

2) The metrics are monitored to determine whether node synchronization is lost. If node synchronization is lost, then the alternate bit pairings of the received data is chosen in an attempt to resync. The metrics are monitored again to determine whether synchronization has been established.

This program inputs parameters from an ASCII data file with default name 'Viterbi.prm' and inputs the soft sequence from file 'seq.sft' and outputs the decoded data sequence to data file with default name 'seq.err'. In addition, various statistics are output to an ASCII data file with default name 'Vit3Sync.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'Viterbi.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'seq.err' file which contains an error sequence (in packed format) with the decoded error sequence. The 'seq.sft' file must exist prior to the execution of this program.

B. Markov Chain Program Modules

Most processes which are used to manipulate and communicate binary data from a source to an end user can be modelled accurately by a Markov Chain. This includes differential coding, error correction coding, filtering, non-linearities, and more. In short, it should be possible to model the TDRSS downlink using a Markov Chain with an appropriate number of states. It is only necessary to determine the number of states and the transition probabilities. Estimating the

transition probabilities can be accomplished using the Baum-Welch algorithm [4]. Although the Baum-Welch algorithm has not been implemented in the simulation, programs which involve Markov Chains have been incorporated into the simulation to meet this goal. These are described below.

1. markov

This program generates a sample state sequence which is representative of a Markov Chain with known transition probability matrix. Each state is assigned a number from 0 to N-1 where N is the number of states.

This program inputs parameters from an ASCII data file with default name 'Markov.prm' and outputs a state sequence with default name 'seq.mrk'. In addition, various statistics are output to an ASCII data file with default name 'Markov.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'Markov.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'seq.mrk' file which contains the state sequence It does not matter whether the output file 'seq.mrk' exists or not. If it exists, it is overwritten without a prompt to the user.

2. markup

This program reads in a state sequence and performs hard decision thresholding for the upper bound case.

The program reads in the state sequence by blocks and performs hard decision thresholding on each block and then writes the modified block out to file 'seq.err'. The program outputs several statistics to the user screen as well. Note that the sequence is read in from file 'seq.mrk' (SeqType=5), with Markov Chain states and is stored in file 'seq.err' (SeqType=1), with hard errors.

Executing the program causes the 'seq.mrk' file to be read which contains a state sequence. The 'seq.mrk' file must exist prior to the execution of this program.

3. markdown

This program reads in a state sequence and performs hard decision thresholding for the lower bound case.

The program reads in the state sequence by blocks and performs hard decision thresholding on each block and then writes the modified block out to file 'seq.err'. The program outputs several statistics to the user screen as well. Note that the sequence is read in from file 'seq.mrk' (SeqType=5), with Markov Chain states and is stored in file 'seq.err' (SeqType=1), with hard errors.

Executing the program causes the 'seq.mrk' file to be read which contains a state sequence. The 'seq.mrk' file must exist prior to the execution of this program.

4. markjoin

This program generates the joint event probabilities for joints events associated with received codewords in the state seq. It is assumed that each state in the received sequence corresponds to a code symbol. The algorithm involves partitioning the state sequence into n-state blocks, where n is the code blocklength, called the received codeword state. The number of each state which occurs within a received codeword state constitutes a single sample point for the joint state event. The number of each joint event is accumulated and the total for each is divided by the number of received codeword states to determine the empirical probability. The only problem with this procedure is defining an efficient method for identifying each joint event. The method used in this program is to define an array, Joint(i), in which all joint events would be stored in a unique location. If the Markov Chain has S states, then there are [(n+S+1) choose (n]) number of ways that a specific number of each state occurs in the received codeword state. If a received codeword state has n1, n2, ..., nS number of occurrences of states s1, s2, ..., sS, respectively, then the Joint array location which contains this joint event is computed on the fly as given in subroutine StateIndex.

This program inputs parameters from an ASCII data file with default name 'MarkJoin.prm' and inputs the state sequence from data file with default name 'seq.mrk' and outputs the joint probabilities to ASCII data file with default name 'MarkJoint.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'MarkJoin.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. The 'seq.mrk' file must exist prior to the execution of this program.

5. displmrk

This program displays the sequence found in the file 'seq.mrk'. This file must be of type SeqType = 5 (state).

6. deintmrk

This program performs block deinterleaving of the state sequence found in file 'seq.mrk'. It is assumed that the channel symbols corresponding to those states have already been interleaved using an (C,R,m) block interleaver. The deinterleaver groups every m state seq values together and deinterleaves them as a group. The method used to implement the function of the block interleaver is to read in a block of the state seq and to use a series of formulas to perform the block deinterleaving. These formulas are described in the blkdeint program [5]

This program inputs parameters from an ASCII data file with default name 'DeintMrk.prm' and outputs the state sequence to data file with default name 'seq.mrk'. In addition, various statistics are output to an ASCII data file with default name 'DeintMrk.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'DeintMrk.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'. Executing the program generates the 'seq.mrk' file which contains an state sequence with deinterleaved states. The 'seq.mrk' file must exist prior to the execution of this program.

C. RICE Program Modules

In an effort to investigate the interaction between RICE decompression and errors which may result from decoding failure, several programs were written to perform RICE compression/decompression and convert image sequences to/from the sequence format required by CLEAN. These are described here.

1. ricecomp

This is the same code received from Pen-Shu Yeh at Goddard Space Flight Center with slight modifications to work with CLEAN. The code reads in an image in JPL format and compresses it into a format defined by Pen-Shu Yeh.

2. ricedcmp

This is the same code received from Pen-Shu Yeh at Goddard Space Flight Center with slight modifications to work with CLEAN. The code reads in an image in JPL format and compresses it into a format defined by Pen-Shu Yeh.

3. img2seq

This program converts the Jet Propulsion Laboratory's image file format (ASCII) to the CLEAN code data file format (packed). Both, the .img and .seq, filenames are specified by the user on the command line. This program works for RICE-compressed or uncompressed files.

First the program reads the image header and writes it to the sequence file's header. The program determines whether or not the file is compressed by reading character*2 ch1 in the image header. Then the appropriate conversion routine is selected and executed.

4. seq2img

This program converts a sequence file to an image file in the Jet Propulsion Laboratory's format. The sequence file must contain the proper image header data in the sequence header so that the image file will be constructed correctly.

If ch1 character in the image header is 'C1' the image will be written in the compressed image format. If ch1 is 'U0' the image file will be written in the non-compressed format. If ch1 is neither of these, the program will end.

Portions of this code are adapted from JPL's source code.

D. Miscellaneous Program Modules

Several additional programs were developed to accommodate convolutional encoding, cycle slips in the demodulator which can cause insertion errors and deletion errors, as well as other programs described below.

1. convencd

This program performs convolutional encoding on a binary data sequence. The data is read in from file with default name 'seq.dat' and the output is stored in a file with default name 'codeseq.dat'. The encoder structure information is found in parameter file 'Viterbi.prm' (see vithard for a description of these parameters).

Executing the program causes the file 'codeseq.dat' to be created or modified. Before running this program, sequence 'seq.dat' must exist.

There are no assumptions associated with the implementation or output of this program.

2. delete

This program simply deletes user specified soft values from a soft sequence. This process mimics bit deletions in the channel due to receiver PLL cycle slips. This program only works with soft decision sequences.

This program inputs the soft values to be deleted from data file with default name 'delete.dat' and applies those deletions to sequence found in file 'seq.sft'. In addition, various statistics are output to an ASCII data file with default name 'Delete.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

Executing the program causes the 'seq.sft' file to be modified. Before running this program, data file 'delete.dat' and sequence 'seq.sft' must exist.

3. insert

This program simply inserts user specified soft values into a soft sequence. This process mimics bit insertions in the channel due to receiver PLL cycle slips. This program only works with soft decision sequences.

This program inputs the soft values to be inserted into the data file with default name 'insert.dat' and inserts those into the sequence found in file 'seq.sft'. In addition, various statistics are output to an ASCII data file with default name 'Insert.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

Executing the program causes the 'seq.sft' file to be modified. Before running this program, data file 'insert.dat' and sequence 'seq.sft' must exist.

4. help

This program simply puts help type information to the user screen concerning the usage of the multiple executable programs which make up the CLEAN simulator. The information shown on the user screen is as follows:

```
*****************************************************************************
***                                                                       ***
***                  Communication Link and Error ANalysis                ***
***                              (CLEAN)                                  ***
***                                                                       ***
***                  A communication link simulation tool                 ***
***                                                                       ***
***                           Developed for:                              ***
***                    NASA Goddard Space Flight Center                   ***
***                                                                       ***
***                           Developed by:                               ***
***                      Mississippi State University                     ***
***                         William J. Ebel, Ph.D.                        ***
***                              Drawer EE                                ***
***                       Mississippi State, MS 39762                     ***
***                             601-325-3912                              ***
***                                                                       ***
*****************************************************************************

* Simulation description:
    This simulation tool consists of a collection of separate executable
 programs which perform various operations found in the TDRSS downlink
 receiver.  The simulation is based upon sequences which are expected to occur
 at the receiver threshold device output (hard or soft decision).  Complex
 systems can be simulated by executing the appropriate programs, corresponding
 to the operations found at the receiver, in the proper order.

------------------------- SIMULATION EXECUTABLES -------------------------
                                                |
--------=> EVENT GENERATORS <=--------          | ---------=> NRZM UTILITIES <=---------
BinErrs:  Binomial error generator              | NRZMDec:  NRZM decoder
BrstErrs: Burst error generator                 | NRZMEncd: NRZM encoder
BstyErrs: Bursty error generator                | ---=> RICE COMPRESSION PROGRAMS <=----
SetErrs:  User set error seq                    | RICEComp: RICE compression (Pen-Shu)
BstySoft: Bursty Soft generator                 | RICEDcmp: RICE decompression (Pen-Shu)
IIDSoft:  Indep. Ident. Distr. Soft             | Img2Seq:  Image to seq.err conv.
Markov:   Markov Chain State generator          | Seq2Img:  seq.err to Image conv.
-----=> MARKOV CHAIN PROGRAMS <=------          | ----------=> STATISTICS <=----------
MarkDown: Conv. to lower bound errors           | DeltaEst: Delta burst stat. est.
MarkUp:   Conv. to upper bound errors           | GAPEst:   GAP method burst stat. est.
MarkJoin: Estimate joint event prob             | BinGAP:   Binomial theor. GAP distr.
---------=>  INTERLEAVERS <=----------          | IntvPDF:  Empirical interval distr.
BlkDeint: Block deinterleaver                   | IntvBin:  Interval distr. for bin errs
DeintMrk: Block Deint for M.C. States           | CVMBlk:   CVM bin test by block
DPCI:     Error seq PCI Deinterleaver           | CVMSeq:   Error seq CVM bin test
DPCISoft: Soft seq PCI Deinterleaver            | ------------=> UTILITIES <=-----------
---=> ERROR CORRECTING DECODERS <=----          | Ascii:    convert a PDF file to ascii
BlkDecod: Block, Reed-Solomon decoder           | CompSeq:  Compare sequences
VitHard:  Viterbi hard decision decode          | DisplErr: Displ seq.err to screen
VitMark:  Viterbi decode w/ Markov est          | DisplSeq: Display sequence segment
VitSoft:  Viterbi soft decision decode          | DisplSeq: Display sequence to screen
----=> SYNCHRONIZATION PROGRAMS <=----          | DisplSft: Displ seq.sft to screen
Sync:    Seq.err Sync stat. gen.                | DisplMrk: Displ seq.mrk to screen
SyncPb:  Theoretical sync stat. gen.            | EOSconv:  EOS data conversion
SyncPPN: Theoretical sync stat. gen.            | EOShex:   EOS data display in HEX
----------=> MISCELLANEOUS <=---------          | Harden:   Hard threshold soft values
GenHDF:  Gen. Hamming Distance Fnc.             | JoinSeq:  Join two sequences
GenMap:  Soft value mapping gen.                | MAdd:     Exclusive OR two error seq
PNseq:   Pseudo-Noise sequence gen.             | MAFilt:   Moving Average filter of seq
RawHdr:  Show raw header (for debug)            | QuantPDF: Quantize PDF
Trellis: Trellis generator                      | QuerySeq: Query sequence header
                                                | SeqArc:   EOS sequence archiver
                                                | SeqTrunc: Seq length truncator
                                                | SeqUnarc: Sequence unarchiver
```

5. madd

This program modulo adds two binary data sequences. Each file name is specified by the user throught the keyboard. Both sequence files should be in packed format. The results of the modulo addition are stored in second file in packed format. If the two files are different lengths, the extra length is truncated. The program also outputs the error sequence error density based on the assumption that a '1' corresponds to an error.

This program was written with the intention to modulo add the channel input to the channel output to yield the channel error sequence. The error sequence is stored in file with name SeqFileName2.

6. pnseq

This program generates a psuedo-noise (PN) sequence. The implementation used here is that of Figure 8-6, pg. 380 of "Digital Communications and Spread Spectrum Systems" by Ziemer and Peterson.

The input parameters (data sequence length, generator polynomial order, and random number generator seed) are specified in a file called 'pnseq.prm'. Only orders of 7, 10 17, 20, 25, or 28 are allowed. Any orders other that these will cease program execution. The maximum length sequence for each generator polynomial order is listed in 'pnseq.prm'.

The shift register in the PN sequence generator is initialized with random binary values.

The data sequence is stored in packed form in 'seq.dat'

7. trellis

This program generates and displays for the user the convolutional encoder trellis diagram along with useful parameters. The program also generates a plot file which contains line segments which will physically form the shape of the trellis.

This program was derived from the Trellis generation subroutine constructed for the Viterbi decoding program.

The Trellis is defined via three arrays; PathCodeSym, PathLink, and PathBit. Since this program only accommodates rate 1/2 or 1/3 encoders, only two paths enter each state at a given trellis stage. Therefore, if there are N trellis states, then there are only 2*N possible paths between two trellis stages. These are sequentially numbered from 1 to 2*N where path number 1 and 2 enter state 1, path 3 and 4 enter state 2, etc. Array PathLink(i) gives the state number from which path i originates. Also, PathCodeSym(i) gives the code symbol associated with path i, and PathBit(i) gives the bit associated with path i. Taken together, these three arrays completely define the steady state trellis.

This program inputs parameters from an ASCII data file with default name 'Viterbi.prm' and outputs the trellis structure along with useful parameters to an ASCII data file with default name 'Trellis.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'Viterbi.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'.

8. genhdf

This program performs convolutional encoding of a binary sequence for a single generator function. For now, the program will iterate through all possible generator function for a given constraint length, and generate the "Hamming weight sequence" function, which is the Hamming distance for all possible input sequences, for each generator function.

The method used to construct the unique input sequences is described next. Valid input sequences are all those possible which do not have a string of K consecutive zeros in them where K is the constraint length of the code. These sequences can be generated as follows:

1) Construct the following prefix code:

   $C = \{ 1, 01, 001, 0001, ..., 0^{K-2}1 \}$

   where 0^{K-2} denotes K-2 consecutive zeros. This is a prefix code because no vector in the set can be constructed from a group of other vectors

2) Construct the first sequence as the first prefix code vector 1.

3) Construct all subsequent sequences as combinations of the prefix code vectors as follows:

   a) Number the prefix code vectors as follows:

Number	Prefix Code
0	1
1	01
2	001
...	...
i	0^i1
...	...
K-2	$0^{K-2}1$

   Note that there are K-1 prefix code numbers.

   b) Now let an integer counter, j, iterate from 0 on up

   c) Consider the j(th) integer counter value. Suppose it has a base K-1 representation

   $j = j0 * (K-1)^0 + j1 * (K-1)^1 + j2 * (K-1)^2 + ...$

   where each coefficient is a number in the range 0,1,...,K-2. Next construct the base K-1 number by concatenating the coefficients together:

   j base 10 = [... j2 j1 j0] base (K-1)

   Now construct the j(th) Generator Hamming Distance function sequence by starting the sequence with a 1 and by concatenating the prefix code sequence for the base K-1 coefficients in the order from least significant to most significant. That is, the input sequence is constructed by

   sequence = ... (j2 PC) (j1 PC) (j0 PC) 1

where PC stands for Prefix Code.

The only problem with this formulation is that it excludes input sequences of the form 1^i for integer i greater than 2. However, only input sequences up to a given length (MaxSeqLength) are constructed. Therefore, the input sequences of the form 1^i for i=1,...,MaxSeqLength are constructed first and placed at the beginning of the sequence. This completes the description of how the encoder input sequences are constructed.

This program was derived from the Trellis generation program constructed for the Viterbi decoding program.

The Trellis is defined via three arrays; PathCodeSym, PathLink, and PathBit. Since this program only accommodates rate 1/2 or 1/3 encoders, only two paths enter each state at a given trellis stage. Therefore, if there are N trellis states, then there are only 2*N possible paths between two trellis stages. These are sequentially numbered from 1 to 2*N where path number 1 and 2 enter state 1, path 3 and 4 enter state 2, etc. Array PathLink(i) gives the state number from which path i originates. Also, PathCodeSym(i) gives the code symbol associated with path i, and PathBit(i) gives the bit associated with path i. Taken together, these three arrays completely define the steady state trellis.

This program inputs parameters from an ASCII data file with default name 'Viterbi.prm' and outputs the trellis structure along with useful parameters to an ASCII data file with default name 'GenHDF.ID', where ID is a three letter identifier for the current run which is input from file 'ID.prm'.

The program is run by editing the parameter file 'Viterbi.prm' and selecting the appropriate parameters and by choosing a program ID by editing file 'ID.prm'.

III. Periodic Convolutional Interleaver

Recently, the issue as to whether the PCI is necessary in TDRSS has surfaced. Two documents (presentation slides) have addressed this issue, one by Warner Miller [1] and one by Ted Kaplan and Ted Berman [2] which give conflicting results. Below, the main points and results of the documents are outlined and it is shown that the results are not comparable due to the fact that the channel models used are fundamentally different.

In document [1], OMV test results are presented to illustrate why the PCI is not necessary for the TRMM communications link. Bursts of a fixed length were input into the Viterbi decoder, one at a time, both with and without the PCI present to determine the effect (number of erred bits output by the Viterbi decoder). The main points of the document are as follows.

1) Viterbi output bursts are not extended. This is not entirely true. If a burst (in terms of code symbols) of length B is input to the Viterbi decoder, then generally a burst (in BITS) of length B+M is output where M is a number less than the memory length of the decoder (32 for the LV7017). However, for the length of bursts considered for the OMV tests (>50), the slight increase in burst length is not noticeable.

2) A (255,223,16) Reed-Solomon code can correct 16 code symbols or (at least) 628 consecutive bit errors. This is correct.

3) The OMV tests show that without the PCI, almost all the error bursts output by the Viterbi decoder can be corrected. When the PCI is present, synchronization loss causes error bursts at the Viterbi decoder output of >1000 bits which cannot be corrected by the RS decoder.

4) The OMV test results conflict with the CLASS analysis performed by Ted Kaplan.

In document [2], CLASS (it is assumed) is used to generate performance results which show that the PCI is necessary. The noise environment is modeled by Poisson occurring RFI pulses which affect 15 code symbols (30 binary symbols) at the Viterbi decoder input. The duty cycle of the RFI is taken to be .018 and thermal noise and False Loss of Viterbi Decoder Synchronization (FLDS) are ignored. The main points of the document, for the no PCI case, are as follows.

1) Without the PCI, the Viterbi decoder can't correct code symbol bursts of length 15. In fact, it is stated that the bursts at the Viterbi decoder output are longer than those at the input. See item (1) above. It is my belief that in principle, Document [1] agrees with this assessment.

2) With the PCI, the errors at the Viterbi decoder output are not present unless PCI synchronization is lost. In essence, the error probability at the Viterbi decoder output with the PCI is much less than the Viterbi decoder output without the PCI. It is my belief that in principle, Document [1] agrees with this assessment.

3) Therefore, it is concluded "that there should be an even larger difference after RS decoding (see Figure 1)". Figure 1 of Document [1] shows that the error probability at the RS decoder output is much worse without the PCI. This Figure is the source of the conflict between the OMV test results and CLASS results.

There are several important differences between the analyses which make comparison impossible. These are outlined here.

1) CLASS does not incorporate synchronization into the decoder performance analysis. This is obviously a critical issue which must be considered. Long burst lengths will occur at the Viterbi decoder output when PCI synchronization (and less so, Viterbi node synchronization) is lost.

2) The OMV test results only consider bursts of long length but don't consider the Poisson occurrence time of bursts in the real channel. Previous studies have shown that the S-band downlink is characterized by noise bursts which occur with Poisson statistics [3]. This is important because the duty cycle (taken to be 0.018 by Ted Kaplan in [1]) will result in more than one burst per interleaved Reed-Solomon code block. A duty cycle of 0.018 with bursts of length 30 will cause an average error free guardband between bursts of 30/0.018=1667 binary symbols. Therefore, one RS interleaved block which contains 10,200 binary symbols will result in approximately 10,200/1667=6 noise bursts. Each noise burst causes roughly 30 binary symbol errors, equivalent to roughly 30/8=4 RS code symbol errors. Therefore, 24 RS code symbols (24*8=192 binary symbols) will be in error on average due to the RFI. At first, it appears that these will be corrected with no trouble, however, because the occurrence times are Poisson for the RFI pulses, it is possible for some RS interleaved blocks to contain many more code symbol errors. It is unclear whether performance will be sufficient, in any case, the Poisson occurrences of the RFI bursts cannot be ignored. It is my belief that the RS decoder will have no trouble correcting the bursts which typically occur within one RS interleaved block. Note that the RS decoder does not allow error propagation due to the block nature of the decoder.

   The TDRS East environment is another matter, however. This environment is characterized by an interferer with a duty cycle of 11% or so. It is unclear whether the system, with or without the PCI, can handle this interferer.

The Communication Link and Error ANalysis (CLEAN) simulator developed by me at MSU can help resolve the problem. Poisson occurring bursts can be generated to simulate the RFI in the real link and a soft Viterbi decoding program, which emulates node synchronization exactly like the LV7017C hardware, can be applied. This work is currently in progress along with the RICE compression work.

Preliminary results suggest that the PCI is not necessary for the TDRSS West environment.

Appendix

The RICE Compression Algorithm: Theory and CLEAN Implementation

1.0 ABSTRACT

In communication systems such as satellite data links, it is necessary to keep the bandwidth small due to limited channel and/or transmitter complexity. One way to alleviate the problem is to use digital data compression algorithms which reduce the number of bits required to represent a given amount of information. The RICE compression algorithm is frequently used in data links transmitting digital images from satellites to earth [1-5].

This paper summarizes RICE compression theory and simulation for a noiseless environment. The RICE simulation presented is an application specific to the Voyager II spacecraft, and is integrated into CLEAN, an existing software package. In conclusion, questions are presented for research relating to noisy simulations.

2.0 INTRODUCTION

The goal of all data compression schemes is to take source data and perform a reversible mapping which averages fewer output bits per symbol than the source. In general, the source data is first divided into words (symbols) of equal length and ordered in terms of decreasing symbol probability. Then, the most probable words are assigned codewords which are short relative to the corresponding source symbols. Similarly, the least probable words are assigned codewords which are long in length relative to the source symbols. Ideally, the average codeword length will approach the source entropy (entropy is the minimum number of bits/symbol required to represent the source by using any code).

Many compression schemes, such as the Shannon-Fanno code, perform this mapping by table look-up. An example of a Shannon-Fanno code [6] is shown in Table 1. The source

symbols in Table 1 are 3 bits long and the average codeword length is

$$\overline{L} = \sum_{i=1}^{n} L(X_i)p_{X_i} \qquad (1)$$

or 2.75 bits.

Source Symbols	Probability	Codeword	Codeword Length
X_0	.2500	00	2
X_1	.2500	01	2
X_2	.1250	100	3
X_3	.1250	101	3
X_4	.0625	1100	4
X_5	.0625	1101	4
X_6	.0625	1110	4
X_7	.0625	1111	4

Table 1. Example Shannon-Fanno Code.

The constructs of the Shannon-Fanno code are not important. The point being made here is that the Shannon-Fanno code mapping, as well as many other code mappings, is based on *a priori* table look-up. In reality, the source symbol statistics vary, so the symbol probability ordering in Table 1 can change and data expansion can occur. Therefore "table look-up" codes fall short when the "least probable" symbols occur too often. Thus these types of compression algorithms only work for a certain entropy range. Figure 1 illustrates performance for a typical "table look-up" code with different source entropies. Note this particular code performs best for source entropies from 2.5-4.5 bits/symbol.

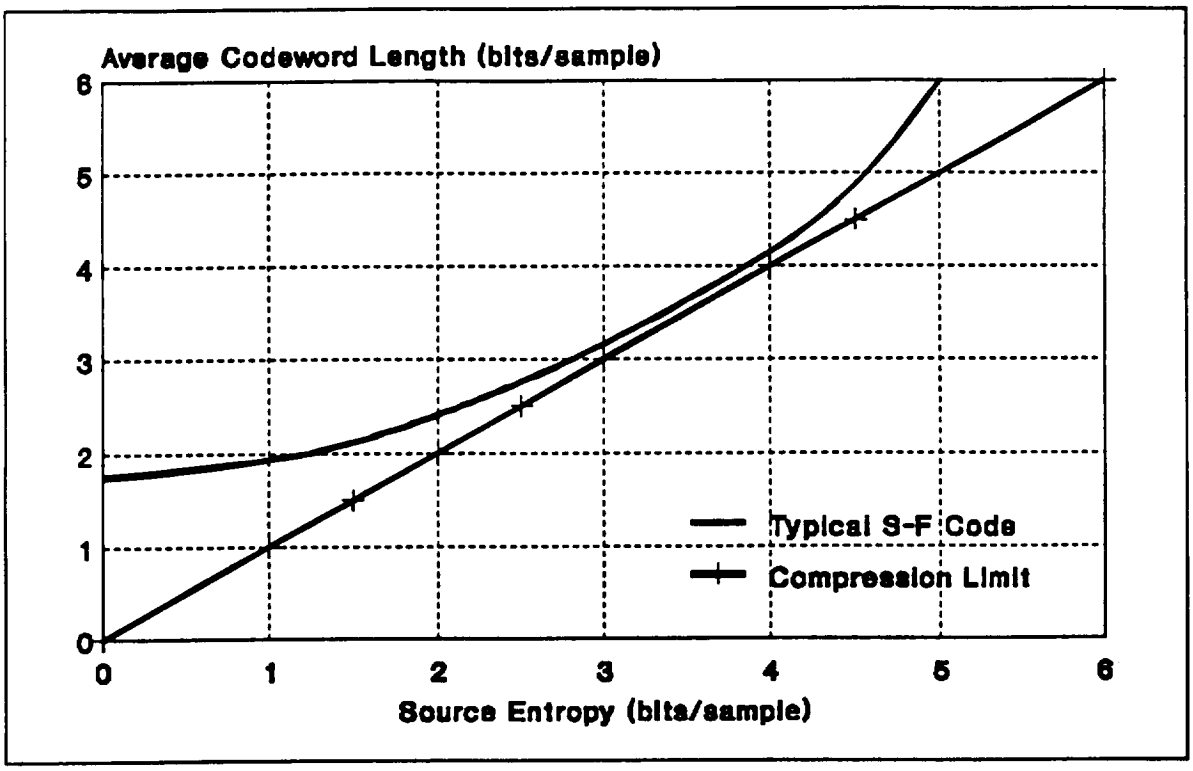


Figure 1. Average Performance for a Typical Shannon-Fanno Code.

The RICE compression algorithm is an adaptive code that employs ideas similar to the Shannon-Fanno code. RICE contains several different compression routines that each perform well under a different entropy range [4,5]. Basically, the RICE algorithm reads a block of source symbols, determines which compression routine is best suited for this block of data, encodes the symbols, and transmits the symbols along with a few ID bits which identify the compression routine used on this particular block. Therefore, the RICE compression algorithm can make adjustments for varying source symbol statistics.

This paper discusses the general RICE compression theory, a RICE application, and a

computer simulation. Also, questions dealing with RICE decoding in a noisy environment are presented.

3.0 THE RICE COMPRESSION ALGORITHM

Let the sequence of any symbols, x_1, x_2, ..., x_{q-1} be denoted as $X=\{x_i\}$. Then the entropy, H(X), is defined as

$$H(X) = -\sum_i p_i \log_2 p_i \quad bits/sample \qquad (2)$$

where p_i is the probability that x_i occurs. The entropy of a data source is the theoretical limit for how many (actually, how few) bits/symbol are required to represent it. Practically all data sources have time-varying entropies. The biggest advantage of RICE compression is that it can employ many types of compression algorithms, which collectively perform well over a wide range of entropies. The average performance plot for each RICE compression option looks like Figure 1, except each option is good over a different entropy range. The term, "RICE compression", does not imply the number or type of algorithms within it. This paper will only cover a few of them.

3.1 PREPROCESSING

No matter which code option is used, RICE's first task is to order the symbol probabilities for each block. This is accomplished by reversible preprocessing which usually removes correlation from the symbols and orders them using *a priori* knowledge. From now on, it is

assumed that the following condition is true for each block of samples:

$$p_0 \geq p_1 \geq p_2 \cdots \geq p_{q-1} \tag{3}$$

where q is the number of symbols output from the source. Reversible preprocessing is summarized in Figure 2. The actual preprocessing method used will depend on the application.

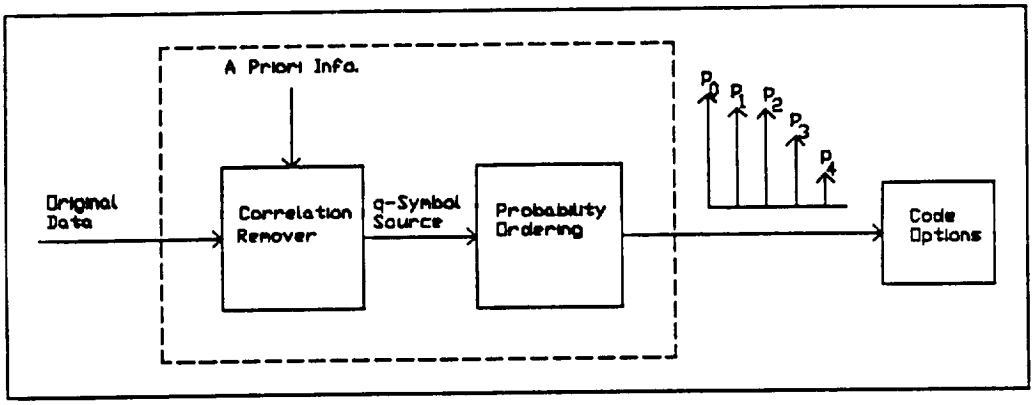


Figure 2. Reversible Preprocessing for RICE Compression.

Once the condition in (3) is true or well approximated, RICE can choose which compression option is best for the current block. Let the compression options be denoted as Ψ_i, where i is an identifier. The Ψ_i identifiers used in this paper are identical to those in [5].

One code option is an obvious, trivial case. If the source symbols happen to be completely random, there is no need to encode them. An attempt to code them would most likely result in data expansion. Therefore, the simplest compression option is

$$\psi_3[X] = X \tag{4}$$

3.2 Ψ_1: FUNDAMENTAL SEQUENCE

The simplest, non-trivial compression option is the fundamental sequence. The

fundamental sequence codeword operator is defined by

$$fs[i] = 000...0001 \tag{5}$$

where i is the magnitude of an input symbol and the output is i zeros followed by a "1". Obviously, the length of a fundamental sequence codeword is

$$l_i = L(fs[i]) = i+1 \quad bits \tag{6}$$

Encoding J symbols as fundamental sequence codewords is denoted by

$$\Psi_1[X] = FS[X] = fs[x_1] * fs[x_2] * \cdots * fs[x_J] \tag{7}$$

where * implies concatenation and Ψ_1 is called the fundamental sequence of X. The length of a fundamental sequence is

$$F = L(FS[X]) = \sum_{j=1}^{J} L(fs[x_j]) = J + \sum_{j=1}^{J} x_j \tag{8}$$

No matter how many bits each symbol contains, Ψ_1 would be powerful if lower magnitude symbols occurred most. This would be the case for highly-correlated data because the symbols output from the preprocessor (de-correlator) would be low in magnitude. Image data is a good example of this situation, since pixels on the same scan line are highly correlated [5]. The performance plot for the fundamental sequence is contained in Figure 4. Note that FS[X] performs well over H(X) of 1.5 to 3.0 bits/sample.

3.3 Ψ_2 AND Ψ_0: CFS[X] AND $\widetilde{CFS}$[X]

Let Y be a J-symbol sequence. Given a positive integer e, define the extended sequence

of Y to be Y concatenated with enough zeros to form a sequence whose length is a multiple of e. The extended sequence of Y is written as

$$Y' = Ext^{e}[Y] = (y_1 y_2 \cdots y_e) * (y_{e+1} y_{e+2} \cdots y_{2e}) * \cdots * (y_{J-1} y_J 00 \cdots 0) \quad (9)$$

There are $\lceil J/e \rceil$ groups of e symbols in $Ext^e[Y]$, so there are $e\lceil J/e \rceil$ symbols total in $Ext^e[Y]$ where $\lceil J/e \rceil$ is the smallest integer greater than or equal to J/e.

As an example, let Y by the 29 bit sequence

$$Y = 11010011010100111011010010111 \quad (10)$$

Then the 3rd extension of Y is given as

$$Y' = Ext^{3}[Y] = (110)*(100)*(110)*(101)*(001)* \\ (110)*(110)*(100)*(101)*(110) \quad (11)$$

where one dummy zero was added to complete the $\lceil 29/3 \rceil = 10^{th}$ symbol of Y'.

Compression options Ψ_2 and Ψ_0 attempt to remove any redundancy that may remain in the fundamental sequence. Clearly, from (5), it may be likely zeros in the fundamental sequence are more likely than ones. Define the second compression option as

$$\psi_2[X] = CFS[X] = cfs[x_1] * cfs[x_2] * \cdots \quad (12)$$

where cfs means code the 3rd extension of X mapped according to Table 2 [5]. The performance plot for CFS[X] is contained in Figure 4. Note that CFS[X] performs best for H(X) of 3 to 4.5 bits/sample.

Input 3-tuple α	Output Codeword: cfs[α]
000	0
001	100
010	101
100	110
011	11100
101	11101
110	11110
111	11111

Table 2. 8-Word Code, cfs[α].

It is clear from Table 2 that when zeros are most likely in FS[X], compression will occur. It is possible that ones are more likely in FS[X]. Therefore, define the next compression option to be

$$\psi_0[X] = \tilde{CFS}[X] = \tilde{cfs}[x_1] * \tilde{cfs}[x_2] * \cdots \qquad (13)$$

where $\tilde{cfs}$ comes from Table 2 with the left column complemented. The performance plot for $\tilde{CFS}$[X] is contained in Figure 4. Note that $\tilde{CFS}$[X] performs best for H(X) of 0 to 1.5 bits/sample.

3.4 THE BASIC COMPRESSOR

All of the RICE compression options mentioned thus far collectively perform close to source entropies which range from 0 to 4.5 bits/sample. A block diagram for the four basic code

options is shown in Figure 3. Together, these code operators comprise the "basic compressor"

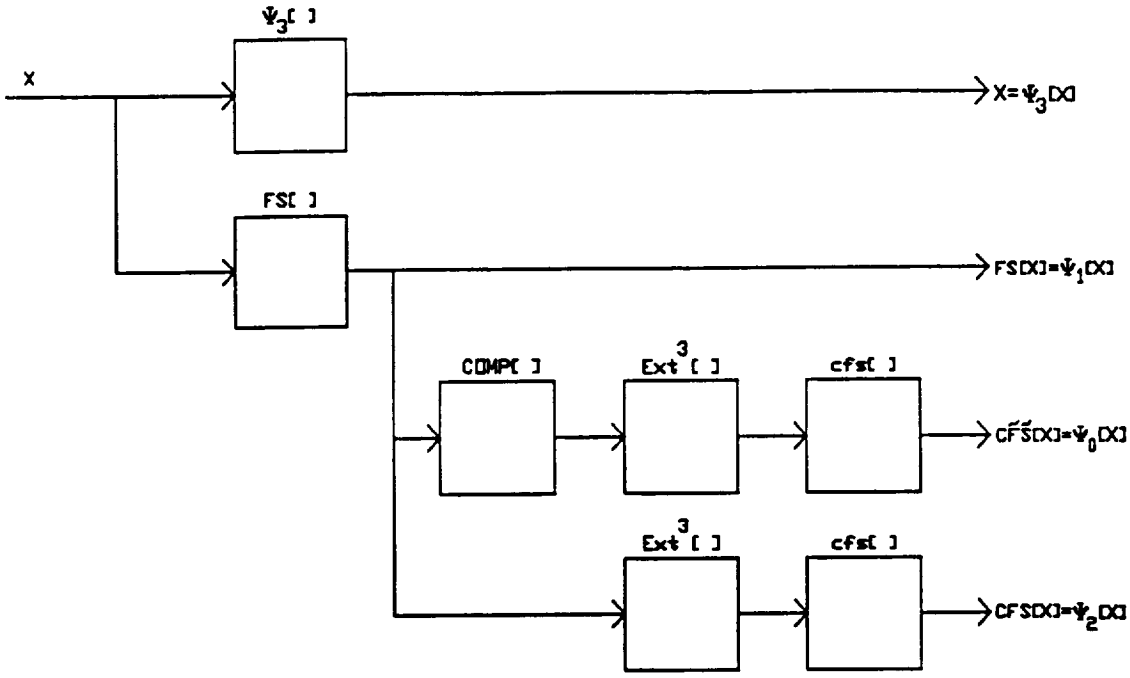


Figure 3. Four Basic Compressor Options.

which is denoted as

$$\Psi_4[X] = BC[X] = ID*\Psi_{ID}[X] \tag{14}$$

where ID is a concatenated 2-bit binary number representing 0, 1, 2, or 3, the compression option used for this sequence. Clearly, ID will be chosen such that

$$L(\Psi_{ID}[X]) = \min_j \{L(\Psi_j[X])\} \tag{15}$$

Rice suggests the ID decision rules outlined in Table [5]. The length of the basic compressor

would be

$$\frac{L(BC[X])}{J} = \frac{2}{J} + \frac{L(\Psi_{ID}[X])}{J} \quad bits/sample \qquad (16)$$

where J is the number of samples. The rightmost term in (16) is assumed to be the shortest of the code options.

Operator Decision	Condition for FS[X] Length
$\Psi_0[\ ]$	$F \leq 3 \lfloor J/2 \rfloor$
$\Psi_1[\ ]$	$3 \lfloor J/2 \rfloor < F \leq 3J$
$\Psi_2[\ ]$	$3J < F < 3(m-2J)$
$\Psi_3[\ ]$	$F \geq 3(m-2J)$

Table 3. Basic Compressor Decision Rules; F=FS length, J=no. samples, m=raw sequence length.

The overhead associated with the basic compressor is 2/J bits/sample, the length of the ID bits. It appears that the overhead could be minimized by keeping J large. However, a large block size would give the basic compressor fewer chances to choose the best code option and the rightmost term in (16) may not be optimum. Studies by Spencer and May [7] suggest that the best block size is 16 to 25 samples.

The performance plot for the basic compressor is shown in Figure 4. The trivial option, Ψ_3, has been left out of the plot. This option would be a horizontal line at q, the number of bits/sample output from the source.

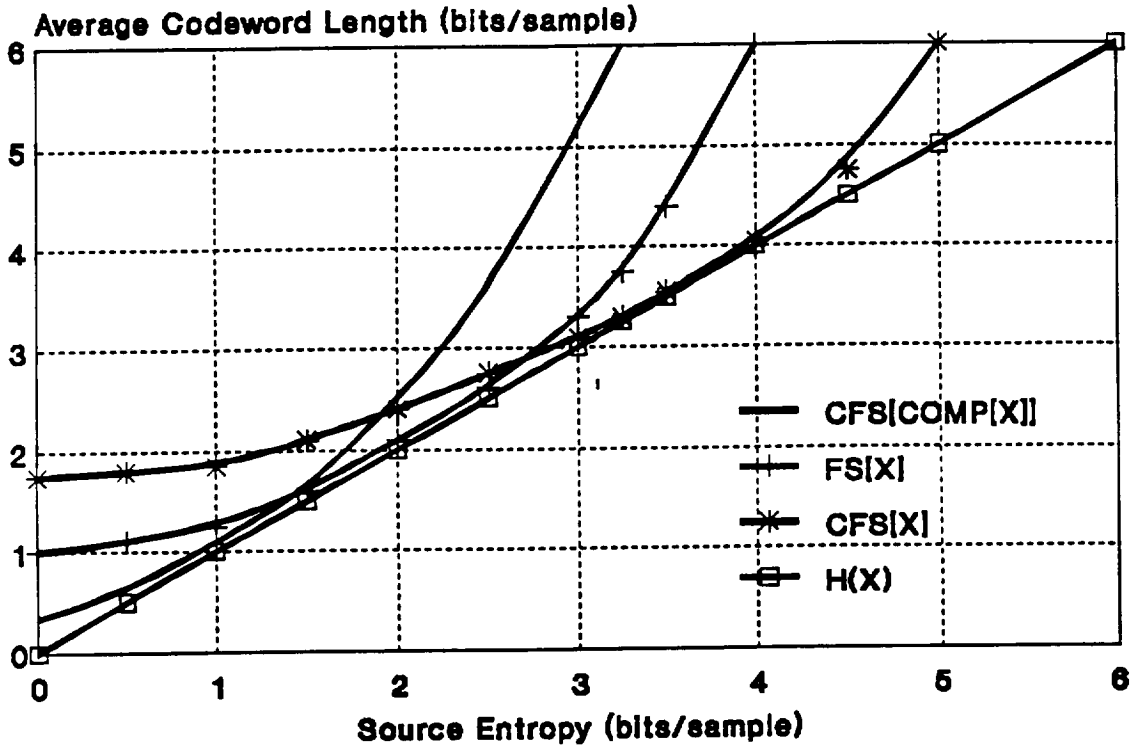


Figure 4: Basic Compressor Performance.

3.5 Ψ_5: BLOCK-BY-BLOCK BASIC COMPRESSOR

Let Y be an N sample sequence of samples partitioned into η smaller blocks such that

$$Y = Y_1 * Y_2 * \cdots * Y_\eta \tag{17}$$

and each Y_i is composed of J_i samples. Therefore

$$N = \sum_{i=1}^{\eta} J_i \tag{18}$$

The block-by-block Basic Compressor is the adaptive version of (14). That is, the block-

by-block Basic Compressor can change ID's in the middle of a sequence. Define the block-by-block compressor as

$$\psi_5[Y] = \psi_4[Y_1] * \psi_4[Y_2] * \cdots * \psi_4[Y_n] \qquad (19)$$

3.6 $\Psi_{i,k}$, Ψ_{11}: SPLIT-SAMPLE ENCODING

There are many other compression algorithms that could be incorporated into RICE compression. The only other algorithm covered in this paper is split-sample encoding. Split-sample encoding recognizes when (and how many) least significant bits (LSB's) in a source sample are random. When this is the case, these LSB's are output "as is" and the remaining most significant bits (MSB's) are compressed. When more LSB's are random, the source entropy is higher, and split-sample encoding performs better. Therefore, split-sample encoding works well for high entropies.

Let M_0^n be a sequence of N preprocessed samples of n bits/sample such that (3) is satisfied. Define the split-sample operator (not the encoder) as

$$SS_0^{n,k}[M_0^n] = \left\{L_k^0, M_0^{n,k}\right\} \qquad (20)$$

where L_k^0 is the N sample sequence consisting of k LSB's of each sample of M_0^n, and $M_0^{n,k}$ is the N sample sequence consisting of the n-k MSB's of each sample of M_0^n. The other subscript and superscript parameters will not be used, but are retained to stay consistent with [5]. A typical sample of this structure is illustrated in Figure 5.

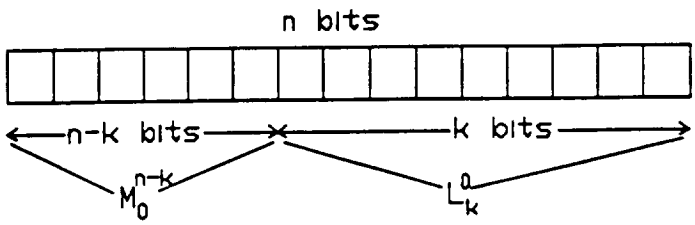


Figure 5. Typical Split-Sample Symbol.

Define the split-sample encoder as

$$\psi_{i,k}[M_0^n] = L_k^0 * \psi_i[M_0^{n-k}] \tag{21}$$

where i is the compression option used to encode the MSB's, n is the number of bits/sample in the original sequence, k is the number of LSB's, and n-k is the number of MSB's.

The decision criterion for i and k depends on the options available to the RICE compressor. Decision criterion for several options is given in [4,5]. In this paper, only the decision rules for i=1 will be examined.

Since i=1, the only decision to make is k. Obviously, k will be chosen such that

$$L\{\psi_{1,k}[M_0^n]\} = \min_k L\{\psi_{1,k}[M_0^n]\} \tag{22}$$

Clearly,

$$\begin{aligned} L\{\psi_{1,k}[M_0^n]\} &= L\{L_k^0\} + L\{\psi_1[M_0^{n-k}]\} \\ &= Nk + L\{\psi_1[M_0^{n-k}]\} \end{aligned} \tag{23}$$

Let the sequence, M_0^n, be represented in terms of its samples

$$M_0^n = m_1 * m_2 * \cdots * m_N \tag{24}$$

and let each sample be represented in terms of binary digits as

$$m_j = b_j^{n-1}2^{n-1}+b_j^{n-2}s^{n-2}+\cdots+b_j^0 = \sum_{l=0}^{n-1} b_j^l 2^l \qquad (25)$$

where b_j^{n-1} is the MSB and b_j^0 is the LSB. Substituting into (8), the length of the fundamental sequence of MSB's is

$$L\{\psi_1[M_0^{n-k}]\} = F_k = N+\sum_{j=1}^{N}\left(\sum_{l=k}^{n-1} b_j^l 2^{l-k}\right) \qquad (26)$$

Notice in (26) that the exponent on the two reflects the truncation of the k LSB's. Rice [5] shows that when (26) is modified and substituted into (23), the length of the split-sample sequence is

$$L\{\psi_{1,k}[M_0^n]\} = 2^{-k}F_0+\frac{N}{2}(1-2^{-k})+Nk \qquad (27)$$

where F_0 is (26) with k=0. Therefore, the RICE compressor must choose k such that (27) is minimized.

Finally, define Ψ_{11} as

$$\psi_{11}[M_0^n] = k'*\psi_{1,k}[M_0^n] \qquad (28)$$

where k' is the binary representation of k. Note the similarities between (27) and (14).

4.0 RICE SIMULATION

Any compression option could be used for i in (21), including the Basic Compressor. Rice [3] has shown that i=1 only provides good compression for the Voyager image entropy range. Therefore, this simulation only incorporates $\Psi_{1,k}$.

CLEAN, a communications simulation package developed by Mississippi State University,

is capable of incorporating RICE compression into many communication system configurations. The RICE portion of CLEAN has been adapted from existing code developed by the Jet Propulsion Laboratory (JPL).

The image file format input to the RICE simulator is described in Figure 6. Figure 6a

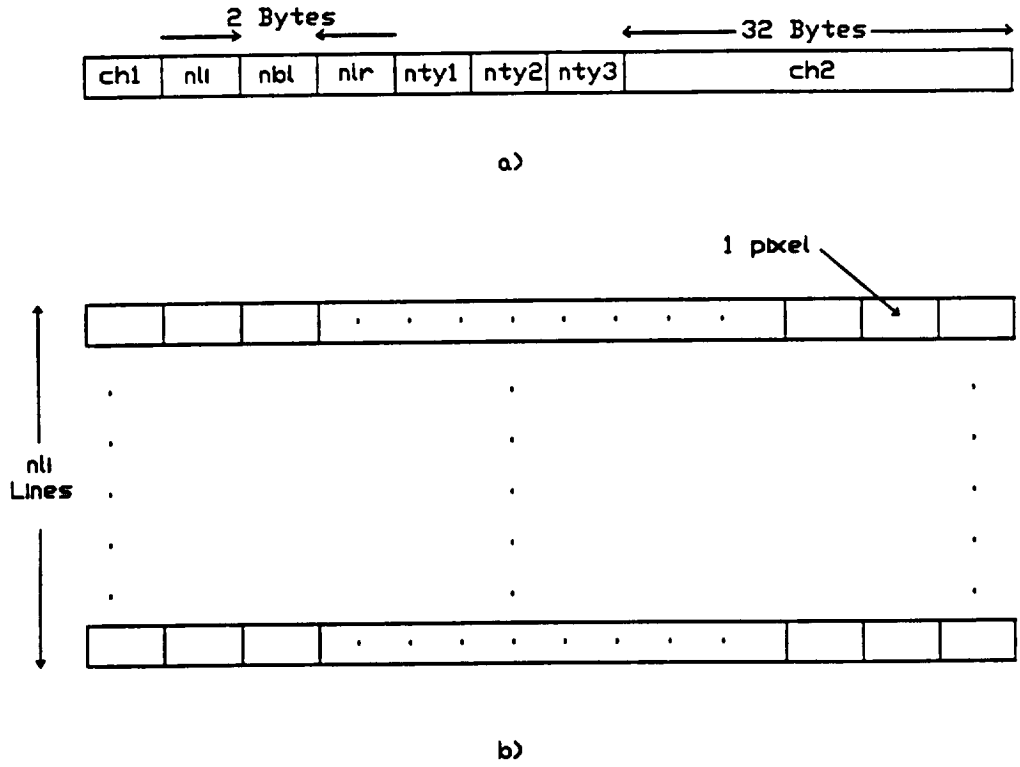


Figure 6. Image File Format.

shows that the first record in the file contains image header data. The header data is defined as

follows:

ch1: "U0" if image is uncompressed, "C1" if image is compressed

nli: number of lines in the image file

nbl: number of bytes per line

nbp: number of bits per pixel

nlr: number of label record

nty(3): type of image file - set to 0 0 0

ch2: user text - image title

Figure 6b describes the image portion of the image file. One record is equivalent to one scan line. Therefore, the image file format is similar to the manner in which pixels are laid over a monitor.

The RICE simulator processes one record at a time. Each record is broken into 16-pixel blocks. If a record is does not contain a multiple of 16 pixels, the last block is zero-filled. Therefore, for each scan line input to the RICE simulator, one reference pixel is output followed by 16 concatenations of (28) where n is the number of bits/pixel. In other words, the split-sample encoder has 16 opportunities per scan line to adjust to changing data statistics. A block diagram of the RICE simulator is shown in Figure 7.

Pixels on the same scan line of image data are highly correlated. For example, adjacent pixels are usually about the same color and intensity.

The purpose of the reversible preprocessing in Figure 7 is to alter the source symbols (pixels) such that (3) is well approximated. The probability ordering in (3) is achieved by using *a priori* information. In the case of a pixel, this *a priori* information is the previous pixel, or

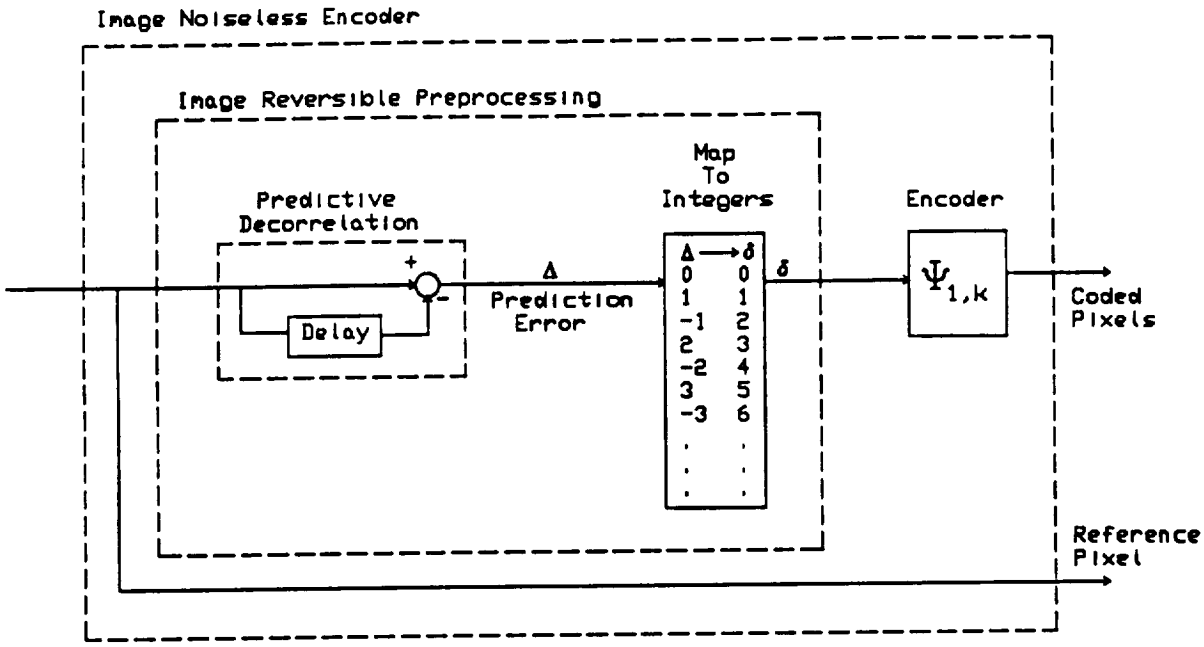


Figure 7. Block Diagram of the RICE Simulator.

reference pixel. The predictive decorrelator in Figure 7 subtracts the previous pixel from the current pixel, yielding a difference value, Δ. Since adjacent pixels are approximately equal, the most likely values for $|\Delta|$ are close to zero. Therefore, the mapping in Figure 6 outputs integers (δ's) whose probability ordering matches the condition in (3). This mapping is outlined in Table 3.

The δ values are well conditioned for split-sample encoding because they are mostly low in magnitude. Therefore, their MSB's will contain significant redundancy and their LSB's will be somewhat random.

Δ Condition	δ Assignment
$0 < \Delta \leq$ previous pixel	2Δ
$\Delta >$ previous pixel	pixel value
(previous pixel - maximum) $\leq \Delta \leq 0$	$2\lvert\Delta\rvert-1$
(previous pixel - maximum) $> \Delta > 0$	maximum - pixel value

Table 3. Δ→δ Mapping Rules.

Note that first pixel from each scan line, the reference pixel, is sent uncoded. At the decompressor, the reference pixel is used in conjunction with the δ values to reconstruct the scan line.

The compressed image file output from the RICE simulator is described in Figure 8. The header for the compressed file is the same as Figure 6a. Clearly, each record of compressed data will be variable in length. Therefore, the number of bytes for each compressed scan line is stored at the beginning of each record. The reference pixel will be used with the decoded series of δ

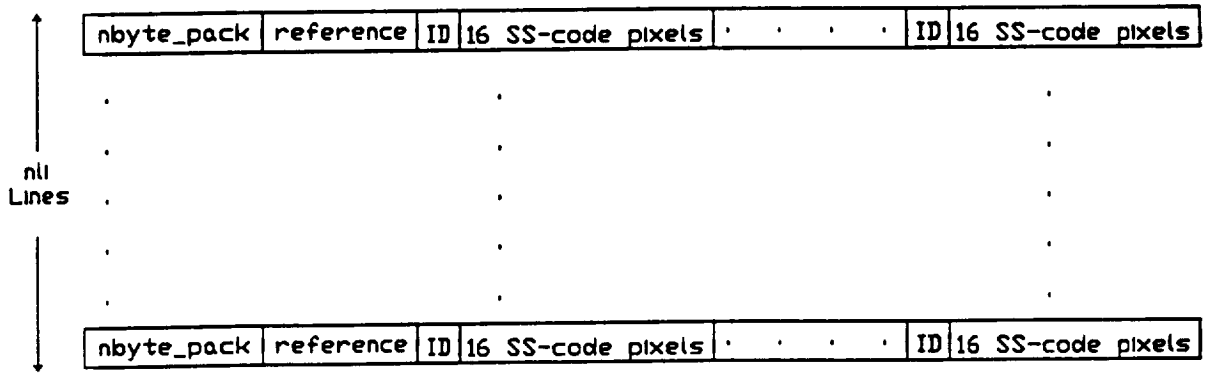


Figure 8. Compressed Image File Format.

values to reconstruct the original scan line.

The ID bits tell the decompressor how many LSB's (k) where split from the original pixel. Note that the ID bits say nothing about the length of the FS encoded MSB's. An example of a split-sample encoded pixel is shown in Figure 9.

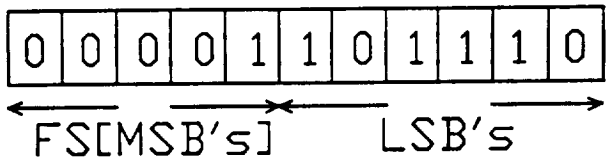


Figure 9. Typical Split-Sample Encoded Pixel.

As the RICE decompressor simulator reads the compressed image file from left to right, it must have some way of knowing where the encoded MSB's begin and end and where the LSB's begin and end. After the ID bits are read, the simulator will begin reading the FS encoded MSB's. As soon as the simulator reads a "1", it assumes that this is the end of the FS encoded MSB's and the next k bits are the LSB's for the current δ. This δ decompression is repeated 15 more times (remember, the δ's were encoded in 16-integer blocks), then the next ID is read and the process repeats until nbyte_pack bytes have been read in. Once all the δ values have been decompressed, they will be used with the reference pixel to reconstruct the original scan line. The decompressor repeats all of this until nli lines have been processed.

4.1 RICE/CLEAN INTEGRATION

Programs called img2seq and seq2img provide the interface between JPL's RICE compression code and MSU's CLEAN code. Img2seq converts the RICE image file format to

the CLEAN sequence file format. Conversely, seq2img converts the CLEAN sequence file format to the RICE image file format. Both programs work for compressed or uncompressed formats. The image header data from record 1 of the image file is stored on records 60-100 in the sequence file. The image data starts on record 101 of the sequence file.

The RICE/CLEAN integration exists to study the effects of a noisy channel on RICE decompression. Clearly, from Figure 9, errors in the LSB's will only result in pixel distortion. However, errors in the ID bits or FS encoded MSB's will may the decompressor to overlap the FS encoded MSB's with the LSB's. Consequently, synchronization of the FS encoded MSB's, LSB's, and ID bits would be lost. Block loss, or even line loss could occur. In other words, errors in the appropriate positions would cause "error propagation" in the decoded pixels.

5.0 CONCLUSION

RICE compression performs quite well over a broad entropy range. However, the effects of noise on the decompressor output are still relatively unknown. The following questions about RICE need to be answered. Do errors output from inner error-correction codes cause catastrophic errors output from the RICE decompressor? If so, and extra error-correction encoding is needed, what is the net coding gain Will error propagation occur? If so, how is it stopped in real systems? Does error propagation really matter? What error statistics are important: pixel distortion, block loss, line loss, etc.?

6.0 BIBLIOGRAPHY

[1] "Universal Source Encoder for Space - USES", MRC NASA Space Engineering Research Center Publication, pp. 1-27.

[2] Jack Venbrux and Norley Liu, "Lossless Image Compression Chip Set", Proceedings of Northcon, Seattle, WA, 1990, pp 145-150.

[3] Pen-Shu Yeh, Robert Rice, Wanrer Miller, "On the Optimality of Code Options for a Universal Noiseless Coder", JPL Publication, February 1991, pp. 1-44.

[4] Robert Rice, "Some Practical Universal Noiseless Coding Techniques", JPL Publication, March 1979, pp. 1-119.

[5] Robert Rice and Jun-Ji, "Some Practical Universal Noiseless Coding Techniques, Part II", JPL Publication, March 1983, pp. 1-56.

[6] R.E. Ziemmer and W.H. Tranter, Principles of Communications, 3rd Ed., Houghton Miffin Co., 1990, pp. 696-698.

[7] D. Spencer and C. May, "Data Compression for Earth Resource Satellites", Proceedings of the 1972 ITC Conference, October, 1972.

BIBLIOGRAPHY

1. W. Miller, "TRMM Performance in RFI without the PCI," NASA Goddard, Code 738.3, December 16, 1993.

2. T. Kaplan and T. Berman, "Performance of TRMM Communications in RFI With and Without the PCI," Stanford Telecom, Code 531.1, December 22, 1993.

3. T.M. McKenzie, H. Choi, and W.R. Braun, "Documentation of CLASS Computer Program for Bit Error Rate with RFI", LinCom, TR-0883-8214-2, August 1982.

4. W. Turin, Performance Analysis of Digital Transmission Systems. New York: Computer Science Press, 1990.

5. Ebel, W.J., and Ingels, F.M., "An Investigation of Error Characteristics and Coding Performance", MSU Department of Electrical and Computer Engineering, Technical Semi-Annual Report, December 30, 1993, NASA Grant NAG5-2006.

Appendix B

Preprint of a Technical paper accepted for publication in the

IEEE Transactions in Information Theory

To appear

A Directed Search Approach for Unit-Memory Convolutional Codes[1]

William J. Ebel, *Member, IEEE*[2]

Abstract - A set of heuristic algorithms to numerically search for good, binary Unit-Memory Convolutional Codes (UMC) are presented along with a large number of new codes for $2 \le k \le 8$ and code rate $1/4 \le R < 1$. Combinatorial optimization is used which involves *selecting* and then *pairwise matching* column vectors of the two (n,k) UMC tap weight matrices. The column selection problem is that of finding the best $(2n,k)$ binary, linear Block Code (BC). In this paper, the best BC generator matrix G is found by successively refining G using directed local exhaustive searches. In particular, the set of minimum weight codewords are used to find a subset of G to exhaustively search. The UMC search strategy (pairwise matching problem) uses a directed local exhaustive search similar to the BC directed search by using the concept of the terminated BC of the UMC. The heuristic algorithms developed in this paper are very robust and converge relatively quickly to the optimal or near optimal UMC. In addition, although it is generally possible to achieve the block code upper bound for free distance, we give a class of UMC's which cannot achieve this bound.

Index Terms - Binary block code, binary unit-memory convolutional code, extended row distance, combinatorial optimization.

I. INTRODUCTION

Many conventional communication systems employ rate $1/n$ convolutional codes. These codes have a high performance/complexity ratio which make them well suited for practical applications. These codes can be implemented with Viterbi decoders that have only $2S$ paths traversing a single trellis stage, where S is the number of decoder states. As an alternative, we consider the class of Unit-Memory Convolutional codes (UMC). A UMC has a fully connected

1 This work was supported in part by the NASA Goddard Space Flight Center under Contract NAG5-2006.

2 The author is with the Department of Electrical and Computer Engineering, Mississippi State University, Box 9571, Mississippi State, MS 39762.

decoder trellis which requires a higher implementation complexity than a rate $1/n$ code for the same number of decoder states. However, the best UMC generally has better distance properties. In this paper, we are concerned with finding the best (n,k) UMC's for $2 \leq k \leq 8$ and code rate $1/4 \leq R < 1$.

The Combinatorial Optimization (CO) search technique introduced by Said and Palazzo [1] involves *selecting* column vectors (columns selection problem) and then *pairwise matching* the column vectors (columns matching problem) to form the two tap weight matrices of an (n,k) binary Unit-Memory Code (UMC). The columns selection problem is essentially that of finding the best $(2n,k)$ binary, linear Block Code (BC). Said and Palazzo approached this using column vector substitutions and an objective function which adds a penalty for every codeword with Hamming weight less than the target d_{min}. The columns matching problem requires matching the columns of the best BC to achieve a UMC with good distance properties. They also approached this using an objective function based on the extended row distance and randomly swapped columns until the objective function was optimized. In this paper, the CO technique is also used, however the algorithms have been designed to direct the search to improve efficiency and result in more optimal codes.

Other UMC's have been found using different memory structures and different optimizing criteria. Justesen et. al. [6] found a set of UMC's with tap weight matrices composed of circulant submatrices using the free distance and the extended row distance as the optimizing criteria. Mooser [8] found many Periodically time-varying convolutional codes (PTVC), Lauer [4] and others [7] have found good Partial Unit-Memory Codes (PUMC). Multi-Memory Codes (MMC) have also been extensively studied [9,10,12]. It can be shown that the UMC is a superset of the PUMC, PTVC, and MMC [3,8].

In Section II to follow, the directed BC search algorithm is described, and in Section III, the directed UMC search algorithm is described. New Unit-Memory Convolutional codes are presented in Section IV and the conclusions follow in Section V.

II. DIRECTED BLOCK CODE SEARCH

A $(2n,k)$ block code (BC) is defined by the relation

$$y = xG$$

where G is the $k \times 2n$ generator matrix. The algorithm to be presented begins with an initial matrix G and then successively refines G until the best BC is found. Since the algorithm does not always operate on the best found generator matrix, let G denote the generator matrix being refined and let G^* denote the best found generator matrix. The set of codewords for generator matrix G is denoted C, the Hamming weight distribution is denoted W (W_i represents the number of codewords with weight i), and the minimum distance is d_{min} which is subsequently denoted δ. Similarly, C^*, W^*, and δ^* are associated with the generator matrix G^*.

We consider a BC optimal if δ is a maximum, the number of non-zero minimum weight codewords W_δ is a minimum, and if the weight distribution is optimal. By optimal weight distribution, we mean one which has maximum *partial second order moment* defined by

$$M_w = \sum_{i=\delta+1}^{\delta+w-1} (i-\delta)^2 W_i$$

where w is the weight window width. Following the convention established above, M_w^* denotes the partial second order moment of W^*. Since M_w is not a function of $W_{\delta+w+1}$ through W_{2n}, this definition leaves open the possibility for multiple "optimal" weight distributions. However, the best found $(2n,k)$ BC will subsequently be used to construct an (n,k) UMC. Since it is unclear what effect the weight distribution from $W_{\delta+w+1}$ through W_{2n} may have on the optimality of the best UMC, all BC's with the same δ, W_δ, and M_w are considered equally optimal.

The motivation for requiring maximum δ and minimum W_δ is due to the fact that the best (n,k) UMC constructed from a $(2n,k)$ BC will generally have a free distance of δ and a number of free distance paths equal to W_δ. This is discussed in Section III below. The motivation for defining the *partial* second order moment is twofold. First, we have observed that many

different weight distributions for a $(2n,k)$ BC have identical full second order moment $M_{2n-\delta}$, and therefore the full moment does not provide sufficient "sensitivity" during the refinement process. Second, by maximizing M_w, the refinement process is biased to select a weight distribution with the bulk of the weight grouped near $\delta+w$. Many weight distributions exhibit peaks for weights slightly greater than δ. For example, the best (12,5) BC has a weight distribution W = [1, 0, 0, 0, 1, 8, 12, 8, 1, 0, 0, 0, 1]. In this case, choosing w = 3 biases the refinement process to settle on a code with this distribution.

The main algorithm for successively refining G alternates between two algorithms called the *Row Iteration Algorithm* denoted A_r and the *Column Iteration Algorithm* denoted A_c. We use $A_c(G)$ to mean that algorithm A_c is performed with G as the initial BC and we denote the set of parameters to be optimized by $O(G)=\{\delta, W_\delta, M_w\}$. Furthermore, $O(G^*)>O(G)$ refers to the notion that G^* is better than G. That is, we say that G^* is better than G if the first criterion, in order of priority, which is different between G^* and G is bettered by G^*. With these definitions, the main algorithm is given by:

1) Randomly choose G. $G^* \leftarrow G$.

2) $G \leftarrow A_r(G)$. If $O(G)>O(G^*)$, then $G^* \leftarrow G$.

3) $G \leftarrow A_c(G, M_w^*)$. If $O(G)>O(G^*)$, then $G^* \leftarrow G$.

4) If not done, go to Step 2.

At Step 4, the algorithm terminates if no better BC has been found after N_A cycles through the main loop (Steps 2 and 3).

A. Row Iteration Algorithm

The row iteration algorithm requires that a set B of binary elements of G be selected and then exhaustively modified in order to refine G. As described above, a refinement takes place when $O(G)>O(G^*)$, where G^* represents the best found generator matrix in this algorithm. The set B is chosen by a voting method based upon those codewords which have weight near δ.

Let X be the set of input vectors defined by

$$X = \{x : \delta \le w(xG) \le \delta + w - 1\}$$

where $w(xG)$ denotes the Hamming weight of codeword xG. There are

$$\sum_{i=\delta}^{\delta+w-1} W_i$$

total vectors in X. Consider the input vector $x \varepsilon X$ which gives rise to the codeword xG. The element g_{ij} of G receives a vote if *both*, x has a 1 in the i^{th} component and xG has a zero in the j^{th} component. This vote simply means that complementing g_{ij} will change the weight of codeword xG. After all the votes have been cast for the vectors in X, the set B is chosen to be the N_B elements of G which received the most votes. Modifying this subset of G is more likely to alter the weight distribution (W_δ through $W_{\delta+w-1}$) than an arbitrary set. We conjecture that *an optimal BC is one with uniform (or nearly so) votes for the elements of G, and any transformation of G, in B.*

For example, we searched for the best (12,4) BC using $w = 2$ and $N_B = 8$. The maximum achievable minimum distance is 6. During the search, a suboptimal code with $d_{min} = 5$ and weight distribution W = [1, 0, 0, 0, 0, 5, 5, 2, 1, 1, 1, 0, 0] was found. The generator matrix G for this code had the following vote count for each corresponding element:

$$\begin{bmatrix} 4 & 3 & 2 & 2 & 4 & 3 & 3 & 2 & 3 & 3 & 3 & 3 \\ 2 & 2 & 1 & 2 & 2 & 1 & 1 & 1 & 2 & 2 & 2 & 3 \\ 2 & 2 & 1 & 0 & 2 & 3 & 3 & 1 & 2 & 2 & 2 & 1 \\ 2 & 1 & 1 & 1 & 2 & 1 & 1 & 0 & 2 & 0 & 2 & 1 \end{bmatrix}$$

At the completion of the algorithm, the optimal code with $d_{min} = 6$ and weight distribution W = [1, 0, 0, 0, 0, 0, 12, 0, 3, 0, 0, 0, 0] was achieved. The generator matrix vote count was found to be:

$$\begin{bmatrix} 4 & 3 & 0 & 3 & 3 & 3 & 3 & 3 & 4 & 3 & 4 & 3 \\ 3 & 4 & 3 & 3 & 3 & 4 & 0 & 3 & 3 & 4 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \end{bmatrix}$$

This is more uniform than the suboptimal one. We note that the elements with zero vote counts was a common occurrence in our search experience. Most codes had between one and four zero vote counts, even when the rest of the vote counts were somewhat uniform.

As a final note, before this algorithm was to be performed, a number of random transformations were applied to the input generator matrix G. A transformation preserves the weight distribution but generally gives rise to a different maximum vote count for the set of elements of G. The transformed generator matrix with the largest maximum vote count was chosen for the refinement procedure.

B. Column Iteration Algorithm

The column iteration algorithm involves exhaustively modifying each column, in order of priority, of G in order to refine G. The column priority is assigned by accumulating the vote counts of G for each column. If the dimension of the code is k, then there are $2^k - 1$ possible ways to modify a given column. Once a column has been exhaustively modified, the next column in order of priority (ties resolved randomly) is exhaustively modified. When all the columns have been modified, the algorithm starts over with the highest priority column again until N_C columns have been modified. As above, a refinement takes place when $O(G) > O(G^*)$, where G^* represents the best found generator matrix in this algorithm.

Since the solution space is not convex [1], it is possible for a local minimum to be reached. To allow the algorithm to "back out" of local minima, an additional criterion involving the partial moment is added. That is, we consider $O(G) > O(G^*)$ if the partial moment M_w is within a factor $0 < K_M \leq 1$ of the best BC partial moment M_w^*.

III. DIRECTED UNIT-MEMORY CODE SEARCH

An (n,k) Unit-Memory Convolutional Code (UMC) is described by the $k \times n$ binary tap weight matrices F_0 and F_1. These matrices relate the encoder input vector and output vector (called a code symbol) by

$$y_i = x_i F_0 + x_{i-1} F_1$$

where x_i is the i^{th}, k-dimensional binary input vector, y_i is the i^{th}, n-dimensional binary output vector, and where all operations are in GF(2). Let the μ-dimensional vector s_i denote the state of the encoder for the i^{th} input, where μ is called the *state complexity* [3,4]. If F_1 is not full rank, then $\mu < k$ and the UMC is called a Partial Unit-Memory Code (PUMC) [4]. We will assume that F_1 is full rank so that $\mu = k$ and $s_i = x_{i-1}$. Furthermore, we assume that $x_i = \mathbf{0}$ for $i < 0$ and $s_0 = \mathbf{0}$.

The tap weight matrices also define a BC. Observe that the input sequence $x_i = x_0, \mathbf{0}, \cdots$ gives the output sequence $y_i = x_0 F_0, x_0 F_1, \mathbf{0}, \cdots$ and can be written

$$[y_0 \quad y_1] = [x_0 F_0 \quad x_0 F_1] = x_0 [F_0 \quad F_1]$$

This also defines a BC and provides a relationship between the $(2n,k)$ BC $G = [F_0 \quad F_1]$ and the (n,k) UMC with tap weight matrices F_0 and F_1. For the BC G, let δ be the minimum distance, W_j be the weight distribution, and let d_{free} be the free distance of the UMC. Since the paths given by the codewords of G are valid paths on the UMC decoder trellis, we must have $d_{free} \leq \delta$. This is called the *block code upper bound* on free distance [3]. Brouwer and Verhoeff [5] have tabulated the tightest known bounds on δ for every binary BC with n and k ranging from 1 to 127.

There are a set of UMC's which cannot achieve the block code upper bound. First, it is easy to show that any $(2n,k)$ BC which contains the all ones codeword must give rise to a catastrophic (n,k) UMC. Now consider the first order $(2^{k-1},k)$ Reed-Muller code with generator matrix

$$G_k = \begin{bmatrix} 1 & \cdots 1 \\ & G'_{k-1} \end{bmatrix}$$

where G'_{k-1} is a $(k-1)\times(2^{k-1})$ matrix consisting of all possible 2^{k-1} column vectors. The weight distribution is

$$W_j = \begin{cases} 1, & j=0, j=n \\ 2^k-2, & j=n/2 \\ 0, & \text{else} \end{cases} \tag{1}$$

which achieves the d_{min} upper bound. However, no other $(2^{k-1},k)$ BC will achieve the d_{min} upper bound. Modifying any of the non-zero columns of G'_{k-1} must necessarily result in a decrease in d_{min}. Similarly, modifying the zero column of G'_{k-1} or the first row must also necessarily result in a decrease in d_{min}.

Theorem 1: Any $(2^{k-2},k)$ UMC cannot achieve the block code upper bound.

Proof: Since every $(2^{k-1},k)$ BC with maximum d_{min} has a weight distribution which includes the all ones codeword, the resulting UMC must be catastrophic.

Thommesen and Justesen have shown [2] that it is possible to extend the connection between a UMC and a BC. The BC with generator matrix

$$G_j^r = \begin{bmatrix} F_0 & F_1 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & F_0 & F_1 & \cdots & \mathbf{0} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \mathbf{0} & \cdots & \cdots & F_0 & F_1 \end{bmatrix}$$

is called the j^{th} terminated BC of the UMC, where j denotes the number of row block matrices, and the minimum distance is called the j^{th} row distance d_j^r. To minimize the decoded error-probability, it is necessary to maximize the row distances d_j^r for $j \geq 1$. Indeed, since the maximum value for d_1^r is δ (i.e. $G_1^r = [F_0 \quad F_1]$) and since d_j^r must be a non-increasing function of j, the maximum possible row distances are

$$d_j^r = \delta, \quad j \geq 1$$

which occur when the block code upper bound is achieved.

Consider a subset of the codewords generated by G_j^r. If I_j is the set of input sequences where $x_i \neq \mathbf{0}$ for $i \leq j$ and $x_i = \mathbf{0}$ for $i > j$, then the extended row distance is defined by

$$\hat{d}_j^r = \min_{x \in I_j} \{w(xG_j^r)\} .$$

This represents the minimum weight of those codewords which start at the all zero state and return for the first time at the $(j+1)^{th}$ trellis stage. Therefore, the BC defined by G_j^r on the set of input sequences I_j has minimum distance $\hat{d}_j^r$. The free distance is given by

$$d_{free} = \min_j (\hat{d}_j^r) .$$

Clearly, $\hat{d}_j^r \geq d_j^r$ which gives $d_{free} = \hat{d}_j^r \geq \delta$ for $j \geq 1$ when the block code upper bound is achieved.

Since $\hat{d}_j^r$ is the minimum distance for the BC defined by G_j^r on the set of input sequences I_j, the columns matching problem can be approached in a manner similar to the columns selection problem which was described in the previous section. In this paper, we optimize the second terminated BC defined by

$$G_2^r = \begin{bmatrix} F_0 & F_1 & \mathbf{0} \\ \mathbf{0} & F_0 & F_1 \end{bmatrix}$$

on the set of input sequences I_2 and with minimum distance $\hat{d}_2^r$. The directed local exhaustive search is in the form of column permutations and the chosen optimizing criteria is a superset of those used in the row and column iteration algorithms. Column permutations have no effect on the weight distribution of G_1^r.

The algorithm presented below begins with an initial matrix $G = [F_0 \quad F_1]$ and then successively refines G with column permutations until the best UMC is found. The subsequent development follows closely with that given in Section II above. Let G denote the UMC being refined and let G^* denote the best found UMC. The j^{th} terminated code is given by G_j^r and $(G_j^r)^*$

for G and G^*, respectively. The set of codewords for generator matrix G_j^r on the set of input vectors I_j is denoted C_j, the weight distribution is denoted W_j ($W_{j,i}$ represents the i^{th} weight value for the j^{th} terminated BC), and the minimum distance is d_j^r which is subsequently denoted δ_j. Similarly, C_j^*, W_j^*, and δ_j^* are associated with the matrix $(G_j^r)^*$.

In this paper, the UMC optimizing criteria, in order of priority, are chosen to be; (1) maximum d_{free}, (2) minimum number of free distance paths N_{free}, (3) maximum δ_2, (4) minimum W_{2,δ_2}, (5) maximum δ_3, (6) minimum W_{3,δ_3}, (7) maximum partial second order moment of W_2 which is denoted $M_{2,W}$, (8) maximum δ_β for some $\beta \gg 1$, (9) and minimum W_{β,δ_β}, where β is an arbitrary, preselected index. The number of free distance paths is

$$N_{free} = \sum_{\substack{j \\ \delta_j = d_{free}}} W_{j,d_{free}}.$$

Criteria (1) through (7) optimize G_1^r through G_3^r. We have observed that maximizing $M_{2,W}$ promotes the selection of a UMC with a rapidly growing distance profile. Finally, criteria (8) and (9) are included to promote the elimination of a catastrophic code. Again, we use $O(G^*) > O(G)$ to denote the notion that G^* is better than G using these prioritized criteria.

The Viterbi decoding algorithm is used to determine the extended row distances δ_j which gives δ_2, δ_3, δ_β, and d_{free}. As long as $\hat{d}_j^r = d_{free}$ occurs for $j \le \beta$, it is only necessary to generate $\hat{d}_j^r$ for $1 \le j \le \beta$. By counting the number of paths which achieve δ_j for each j gives N_{free}, W_{2,δ_2}, W_{3,δ_3}, and W_{β,δ_β}. The second order partial moment $M_{2,W}$ is found by constructing the weight distribution W_2 from G_2^r on the set of input sequences I_2.

The main algorithm for successively refining $G = [F_0 \;\; F_1]$ alternates between two algorithms called the *Column Permutation Algorithm* A_p and the *Column Swap Algorithm* A_s and is given by:

1) Input G (from the BC search algorithm in Section II).

2) $G \leftarrow A_p(G)$. If $O(G) > O(G^*)$, then $G^* \leftarrow G$.

3) $G \leftarrow A_s(G, M^*_{2,W})$. If $O(G) > O(G^*)$, then $G^* \leftarrow G$.

4) If not done, go to Step 2.

At Step 4, the algorithm terminates if no better UMC has been found after some number of cycles through the main loop (Steps 2 and 3). The tap weight matrices for the final UMC are $[F_0 \quad F_1] = G^*$.

A. Column Permutation Algorithm

The column permutation algorithm $A_p(G)$ requires that a set of columns be identified and then systematically permuted (local exhaustive search) to refine G. Let $x_{2,\delta}$ be the set of input vectors which result in minimum weight codewords for G_2^r. That is,

$$x_{2,\delta} = \{x : x \, \varepsilon \, I_2 \, , \, w(x G_2^r) = \delta_2\}$$

We define $c_{2,\delta}$ to be the set of minimum weight codewords in G_2^r and c_i for $1 \le i \le 3n$ to be the number of codewords in $c_{2,\delta}$ which contain a binary zero in the i^{th} column. Changes in those columns which correspond to large values of c_i are more likely to alter the weight distribution of G_2^r and therefore the columns to be selected should come from this set. To localize the permutation search, however, pairs of columns are selected so that if column $i \le n$ is chosen then column $i + n$ is also chosen. Operating on D_c column pairs can only change δ_2 by D_c.

To select a useful set of column pairs, c_i is folded to give

$$c'_i = c_i + c_{i+n} + c_{i+2n} \quad , \quad 0 \le i \le n \quad .$$

The function c'_i gives a measure of how likely the column pair $(i, i+n)$ will change the weight distribution of G_2^r. Therefore, the chosen set of D_c column pairs correspond to the largest c'_i values with ties resolved randomly. The Column Permutation Algorithm simply involves determining the set of column pairs via c'_i, exhaustively permuting those column pairs, and retaining the UMC with the best optimizing criteria $O(G)$.

B. Column Swap Algorithm

This algorithm is similar to the column permutation algorithm. However, the permutation method is an exhaustive procedure where each trial involves swapping two columns of G. For n columns, this requires $n(n-1)/2$ swaps for one complete permutation set. As with the column iteration algorithm, the solution space is not convex and therefore it is possible for a local minimum to be reached. To allow the algorithm to "back out" of local minima, an additional criterion involving the partial moment is added. That is, we consider $O(G) > O(G^*)$ if the partial moment $M_{2,w}$ is within a factor $0 < K_M \le 1$ of the best partial moment $M^*_{2,w}$.

IV. NEW UNIT-MEMORY CODES

The algorithms described in the previous sections were used to search for the best UMC's with $2 \le k \le 8$ and with code rate $1/4 \le R < 1$. In searching for the best $(2n,k)$ BC, the algorithms described in Section II generally used $w = 3$, $12 \le N_B \le 20$, and $K_M = 0.7$. In searching for the best (n,k) UMC, the algorithms in Section III used $w = 3$, $\beta = 10$, and $K_M = 0.7$.

Table I provides a list of parameters for the currently best known UMC's. The minimum weight and number of minimum weight codewords for the $(2n,k)$ BC with $G = [F_0 \quad F_1]$ is given along with the block code upper bound [5]. Also shown are the free distance, number of free distance paths, the extended row distance $\hat{d}_r$ for $1 \le r \le 5$, and the corresponding number of paths W_{r,δ_r}. The corresponding tap weight matrices are given in Table II, where each tap weight matrix column is represented by a base10 number as shown.

A total of 105 UMC's are presented in Table II, most of which are new except for the following. Said and Palazzo [1] previously found the (10,7), (13,7), (11,8), and (24,8) UMC's shown in the table, and Palazzo [12] found the (3,2), (4,2), (6,2), and (4,3) UMC's using a network flow approach. Also, Dettmar and Shavgulidze found the (7,3) UMC [7]. Concerning the (8,7) UMC, our algorithms could only find a $d_{free} = 4$ code. The one given in the table was brought to our attention by O. Ytrehus [11]. Finally, although all our codes are better than the Quasi-Cyclic UMC's found by Justesen et. al. [6] based upon our criteria, a few of the Quasi-Cyclic UMC's have faster rising extended row distance function.

The large number of discovered codes which achieve the block code upper bound attest to the robustness of the directed search algorithms. In most cases, the (n,k) UMC which was found from the best $(2n,k)$ BC not only achieved $d_{free} = d_{min}$ but also achieved N_{free} equal to the number of BC minimum weight codewords. The most noteworthy exceptions are the high rate codes which either do not attain the upper bound, or have a number of free distance paths which exceed the number of minimum weight codewords. The codes which do not attain the upper bound are the (7,6), (8,7), (9,7), (9,8), and (10,8) codes.

Concerning the (15,6) UMC, we could not find a $d_{\min} = 14$ (30,6) BC without the all ones codeword. We suspect that one does not exist. Therefore, we found the best $d_{free} = 13$ (15,6) UMC. Also shown in the table are the modified upper bounds for the set of $(2^{k-1},k)$ UMC's via Theorem 1 and the upper bound for the (3,2) code which can be shown to be 3 via exhaustive search. In some cases, the BC which achieves the maximum $d_{\min}$ was not found. These are the (46,7), (50,8), (52,8), (58,8), and (64,8) BC's. We present these UMC's as the best which have been found to date but note that more optimal ones almost surely exist.

As a final note, a 66MHz Pentium-based PC was used to search for the codes. The BC search required between 5 minutes and an hour to converge, and the UMC search required between 5 minutes to (approximately) 24 hours to converge. The exponential growth in required computer time inhibited the search for codes larger than those presented in this paper.

V. SUMMARY AND CONCLUSIONS

In this paper, algorithms have been described which use combinatorial optimization and directed local exhaustive searches to find the best known Unit-Memory Convolutional (UMC) codes. A total of 105 UMC's are presented most of which are better than previously known codes. In addition, a class of UMC's are given which cannot achieve the block code upper bound.

ACKNOWLEDGEMENTS

Several of the codes listed in the table had already been published and were brought to the author's attention by the reviewers. The author thanks the reviewers for this and for other helpful comments and suggestions.

REFERENCES

[1] A. Said, and R. Palazzo Jr., "Using Combinatorial Optimization to Design Good Unit-Memory Convolutional Codes", *IEEE Transactions on Information Theory,* Vol. 39, No. 3, May 1993, pp. 1100-1108.

[2] C. Thommesen, and J. Justesen, "Bounds on Distances and Error Exponents of Unit Memory Codes", *IEEE Transactions on Information Theory,* Vol. IT-29, No. 5, September 1983, pp. 637-649.

[3] L. Lee, "Short Unit-Memory Byte-Oriented Binary Convolutional Codes Having Maximal Free Distance", *IEEE Transactions on Information Theory,* Vol. 22, May 1976, pp. 349-359.

[4] G.S. Lauer, "Some Optimal Partial-Unit-Memory Codes", *IEEE Transactions on Information Theory,* Vol. 25, No. 2, March 1979, pp. 240-243.

[5] A.E. Brouwer and T. Verhoeff, "An Updated Table of Minimum-Distance Bounds for Binary Linear Codes", *IEEE Transactions on Information Theory,* Vol. IT-39, No. 2, March 1993, pp. 662-677.

[6] J. Justesen, E. Paaske, M. Ballan, "Quasi-Cyclic Unit Memory Convolutional Codes", *IEEE Transactions on Information Theory,* Vol. IT-36, No. 3, May 1990.

[7] U. Dettmar, and S.A. Shavgulidze, "New Optimal Partial Unit Memory Codes", *Electronics Letters,* Vol. 28, No. 18, August 27 1992, pp. 1748-1749.

[8] M. Mooser, "Some Periodic Convolutional Codes Better than any Fixed Code", *IEEE Transactions on Information Theory,* Vol. IT-29, No. 5, September 1983.

[9] P.J. Lee, "New Short Constraint Length, Rate 1/N Convolutional Codes Which Minimize the Required SNR for Given Desired Bit Error Rates", *IEEE Transactions on Communications,* Vol. COM-33, No. 2, February 1985, pp. 171-177.

[10] R. Johannesson, and E. Paaske, "Further Results on Binary Convolutional Codes with an Optimum Distance Profile", *IEEE Transactions on Information Theory,* Vol. IT-24, No. 2, March 1978, pp. 264-268.

[11] O. Ytrehus, private communication.

[12] R. Palazzo Jr., "A Network Flow Approach to Convolutional Codes", *IEEE Transactions on Communications,* Vol. 43, No. 2/3/4, February/March/April 1995, pp. 1429-1440.

William J. Ebel [M'87] was born in St. Louis MO on August 25, 1962. He received the BSEE, MSEE, and Ph.D. degrees in Electrical Engineering from the University of Missouri-Rolla in 1983, 1985, and 1991, respectively.

From 1985 to 1991, he was a senior research engineer with McDonnell Douglas Corporation, McAir division studying issues related to Infrared Search and Track Systems. Since 1991 he has been with the Department of Electrical and Computer Engineering, Mississippi State University, as an assistant professor. Currently, he is involved with NASA studying various coding and synchronization issues related to the Tracking and Data Relay Satellite System (TDRSS). His current research interests include satellite communications, error correcting codes, and system simulation. He is a member of Eta Kappa Nu, Tau Beta Pi, and ASEE.

This work was supported in part by the NASA Goddard Space Flight Center under Contract NAG5-2006.

The author is with the Department of Electrical and Computer Engineering, Mississippi State University, MS 39762 USA.

Table I. Comprehensive list of the best known Unit-Memory Convolutional (UMC) Codes for $2 \le k \le 8$ and code rate $1/4 \le R < 1$.

		$[H_0 \quad H_1]$ BC			UMC		d_r					W_{r,δ_r}				
n	k	UB	d_{min}	N_{dmin}	d_{free}	N_{free}	1	2	3	4	5	1	2	3	4	5
3	2	3 [1]	3 [2]	1	3	1	3	4	4	5	5	1	2	1	4	1
4	2	5	5 [2]	2	5	2	5	6	7	8	9	2	3	4	5	6
5	2	6	6	1	6	1	6	8	9	10	11	1	4	4	5	6
6	2	8	8 [2]	3	8	3	8	10	12	14	16	3	6	12	24	48
7	2	9	9	2	9	2	9	11	12	15	17	2	2	1	2	2
8	2	10	10	1	10	1	10	13	16	19	22	1	2	3	4	5
4	3	4	4 [2]	3	4	10	4	4	4	4	5	3	4	2	1	5
5	3	5	5	3	5	5	5	5	6	7	8	3	2	4	10	20
6	3	6	6	2	6	2	6	7	8	9	11	2	1	1	1	17
7	3	8	8 [3]	7	8	7	8	10	12	14	16	7	21	63	189	567
8	3	8	8	1	8	1	8	11	12	14	15	1	11	2	8	2
9	3	10	10	6	10	6	10	12	14	16	20	6	6	4	2	63
10	3	11	11	4	11	4	11	14	16	19	22	4	10	1	1	2
11	3	12	12	3	12	3	12	16	19	22	25	3	15	13	19	26
12	3	13	13	3	13	3	13	17	20	23	27	3	4	3	2	8
5	4	4	4	3	4	9	4	4	4	5	5	3	4	2	18	8
6	4	6	6	12	6	46	6	6	6	6	8	12	23	10	1	84
7	4	7	7	8	7	19	7	7	7	8	9	8	10	1	1	1
8	4	8	8	7	8	7	8	9	10	12	13	7	20	9	54	25
9	4	8	8	1	8	1	8	10	11	12	13	1	4	2	1	1
10	4	10	10	10	10	10	10	12	12	14	16	10	19	1	1	1
11	4	11	11	7	11	7	11	13	16	18	20	7	4	22	5	5
12	4	12	12	6	12	6	12	15	17	20	23	6	16	3	10	15
13	4	13	13	6	13	6	13	16	19	23	26	6	1	1	8	5
14	4	14	14	4	14	4	14	18	21	25	29	4	5	2	6	6
15	4	16	16	15	16	15	16	20	24	28	32	15	20	26	33	43
16	4	16	16	3	16	3	16	21	25	30	34	3	6	2	5	2
6	5	4	4	2	4	8	4	4	4	5	4	2	4	1	26	1
7	5	6	6	15	6	58	6	6	6	6	8	15	31	10	2	163
8	5	7 [4]	7	7	7	20	7	7	7	8	9	7	11	2	3	6
9	5	8	8	6	8	9	8	8	9	10	11	6	3	2	2	2
10	5	9	9	8	9	8	9	10	11	12	13	8	21	12	10	1
11	5	10	10	6	10	6	10	11	13	14	16	6	7	11	1	5
12	5	12	12	28	12	32	12	12	14	16	18	28	4	6	6	6
13	5	12	12	4	12	4	12	14	15	18	19	4	9	1	6	1
14	5	14	14	24	14	24	14	16	18	20	22	24	57	27	10	5
15	5	15	15	16	15	16	15	16	19	21	25	16	2	7	1	2
16	5	16	16	15	16	15	16	18	21	24	28	15	10	5	1	3
17	5	16	16	3	16	3	16	20	23	26	29	3	21	8	3	1
18	5	17	17	5	17	5	17	21	24	29	33	5	4	2	4	4
19	5	18	18	4	18	4	18	23	27	31	35	4	21	4	4	2
20	5	20	20	26	20	26	20	24	28	32	38	26	11	3	1	8
7	6	5	5	10	4 [5]	6	5	4	4	5	5	10	3	3	29	15
8	6	6	6	9	6	47	6	6	6	6	7	9	26	10	2	13
9	6	8	8	45	8	307	8	8	8	8	8	45	140	95	24	3
10	6	8	8	7	8	12	8	8	8	10	10	7	4	1	4	1
11	6	9	9	9	9	9	9	10	10	12	12	9	26	2	8	1
12	6	10	10	7	10	7	10	11	12	13	15	7	13	10	2	5
13	6	12	12	45	12	53	12	12	14	14	18	45	8	18	1	34

14	6	12	12	6	12	6	12	14	15	18	20	6	30	1	15	12
15	6	14	13 [6]	3	13	3	13	15	16	19	20	3	11	2	4	1
16	6	15 [4]	15	15	15	15	15	16	18	21	22	15	9	4	2	1
17	6	16	16	44	16	44	16	18	20	22	24	44	32	6	1	1
18	6	16	16	2	16	2	16	19	22	26	30	2	9	7	1	7
19	6	18	18	30	18	30	18	20	24	28	32	30	6	14	17	18
20	6	18	18	2	18	2	18	21	23	29	33	2	5	2	4	2
21	6	20	20	45	20	45	20	22	26	30	36	45	3	6	3	12
22	6	21	21	22	21	22	21	24	28	33	38	22	2	3	1	2
23	6	22	22	15	22	15	22	23	30	36	40	15	1	1	11	1
24	6	24	24	60	24	60	24	26	32	38	44	60	1	6	13	18
8	7	6	5 [9]	11	5	119	5	5	5	5	5	11	31	33	18	17
9	7	7	7	33	6 [3]	45	7	6	6	6	6	33	28	10	6	1
10	7	8	8 [7]	54	8	337	8	8	8	8	8	54	135	103	39	6
11	7	8	8	5	8	12	8	8	9	9	9	5	7	17	3	2
12	7	10	10	48	10	106	10	10	10	12	12	48	51	7	32	4
13	7	11	11 [7]	37	11	66	11	11	11	12	14	37	28	1	1	2
14	7	12	12	28	12	48	12	12	13	15	16	28	20	11	10	2
15	7	12	12	4	12	4	12	13	14	15	17	4	4	6	2	1
16	7	14	14	54	14	58	14	14	16	18	20	54	4	16	8	2
17	7	15	15	36	15	36	15	16	16	19	22	36	21	1	2	2
18	7	16	16	37	16	37	16	17	19	21	24	37	9	6	1	2
19	7	16	16	5	16	5	16	18	21	24	26	5	4	14	5	2
20	7	18	18	54	18	54	18	20	22	26	30	54	33	7	8	10
21	7	19	19	35	19	35	19	21	24	26	31	35	12	8	2	2
22	7	20	20	38	20	38	20	22	23	29	33	38	7	1	1	1
23	7	21	20 [8]	3	20	3	20	24	25	30	34	3	26	1	2	1
24	7	22	22	54	22	54	22	26	26	32	38	54	113	1	2	4
25	7	24	24	108	24	108	24	26	30	36	42	108	5	3	8	11
26	7	24	24	35	24	35	24	28	31	36	43	35	22	1	1	3
27	7	24	24	5	24	5	24	29	33	38	43	5	19	3	3	1
28	7	26	26	52	26	52	26	32	34	42	48	52	127	1	4	1
9	8	6	6	29	4 [5]	5	6	4	4	4	5	29	1	2	2	24
10	8	8	8	130	6 [5]	68	8	6	6	6	6	130	43	15	8	2
11	8	8	8 [7]	50	8	325	8	8	8	8	8	50	159	89	23	4
12	8	8	8	1	8	11	8	8	8	9	10	1	8	2	3	5
13	8	10	10	55	10	139	10	10	10	10	12	55	73	10	1	8
14	8	11	11	36	11	55	11	11	11	12	14	36	17	2	1	5
15	8	12	12	59	12	98	12	12	14	14	16	59	39	125	8	10
16	8	13	13	38	13	48	13	13	14	15	16	38	10	5	1	1
17	8	14	14	28	14	36	14	14	15	15	19	28	8	3	1	3
18	8	16	16	153	16	219	16	16	18	20	22	153	66	46	10	2
19	8	16	16	54	16	56	16	16	18	18	22	54	2	2	1	2
20	8	16-17	16	6	16	6	16	18	20	23	25	6	23	4	3	1
21	8	18	18	80	18	80	18	20	20	24	28	80	79	2	2	2
22	8	19	19	52	19	52	19	21	21	23	28	52	38	1	1	3
23	8	20	20	101	20	101	20	22	24	28	32	101	25	1	1	1
24	8	22	22 [7]	144	22	146	22	22	26	30	34	144	2	9	1	1
25	8	23	22 [8]	83	22	83	22	24	28	30	36	83	11	19	3	7
26	8	24	23 [8]	56	23	56	23	25	29	32	39	56	1	2	1	3
27	8	24	24	62	24	62	24	27	30	36	41	62	18	4	4	3
28	8	24-25	24	46	24	46	24	28	30	38	44	46	11	1	6	7
29	8	26	25 [8]	10	25	10	25	29	34	39	46	10	7	6	1	2
30	8	26-28	26	40	26	40	26	32	34	40	48	40	144	1	2	8
31	8	28	28	118	28	118	28	32	36	42	50	118	38	3	1	1
32	8	29-30	28 [8]	37	28	37	28	32	38	46	52	37	2	1	12	2

(1) Via exhaustive search, upper bound is reduced to 3.
(2) See Reference [12].
(3) See Reference [7].
(4) Block code upper bound reduced by one due to Theorem 1.
(5) $d_{free} < d_{min}$ of the $[H_0 \quad H_1]$ BC.
(6) Algorithms failed to find a $d_{min} = 14$, (30,6) BC without the all ones codeword.
(7) See Reference [1].
(8) Upper bound not achieved for the $[H_0 \quad H_1]$ BC [5].
(9) See Reference [11].

Table II. Tap weight matrices for the best known Unit-Memory Convolutional (UMC) Codes for $2 \le k \le 8$ and code rate $1/4 \le R < 1$.

n	k	H_0 / H_1
3	2	1 2 2 3 2 1
4	2	1 2 3 1 1 3 3 2
5	2	1 2 3 1 3 3 2 3 1 2
6	2	1 2 1 2 3 3 3 3 2 1 1 2
7	2	1 2 2 1 3 3 2 3 3 2 2 1 3 1
8	2	1 2 3 1 3 3 1 2 3 1 2 2 3 1 1 2
4	3	1 2 4 6 7 1 3 5
5	3	1 2 4 5 3 2 7 6 3 7
6	3	1 2 4 5 6 3 1 7 6 5 2 3
7	3	1 2 4 6 3 5 7 6 5 7 4 2 3 1
8	3	1 2 4 4 2 3 7 5 6 1 2 4 7 6 3 5
9	3	1 2 4 3 5 6 1 7 3 3 4 7 6 5 1 5 7 2
10	3	1 2 4 7 6 3 5 5 1 3 7 7 4 3 2 2 6 1 6 5
11	3	1 2 4 3 7 5 6 1 3 4 6 1 7 2 4 2 5 3 6 5 1 7
12	3	1 2 4 1 6 6 7 4 3 2 5 5 2 4 1 5 7 3 7 6 2 3 5 6
5	4	1 2 4 8 14 13 15 7 14 11
6	4	1 2 4 8 11 7 15 14 5 13 3 9
7	4	1 2 4 8 13 3 6 12 9 7 11 5 10 15
8	4	1 2 4 8 12 3 7 9 9 11 14 10 15 13 5 6
9	4	1 2 4 8 13 9 15 11 7 10 6 5 3 2 11 1 14 12
10	4	1 2 4 8 13 14 10 7 5 12 6 13 5 15 11 9 14 3 7 1
11	4	1 2 4 8 7 3 11 13 14 11 7 14 9 10 6 13 1 8 15 4 5 12
12	4	1 2 4 8 11 10 13 4 9 6 7 14 12 11 7 8 5 13 1 2 15 10 14 3
13	4	1 2 4 8 10 3 9 14 6 11 12 15 5 11 13 10 8 9 12 2 14 1 6 5 7 15
14	4	1 2 4 8 9 7 15 12 6 5 14 10 13 11 3 11 4 7 8 13 5 15 10 1 12 6 2 14
15	4	1 2 4 8 7 15 3 14 6 5 11 9 12 13 10 11 15 4 5 3 14 9 10 1 7 8 13 6 12 2
16	4	1 2 4 8 15 11 13 9 12 5 6 6 3 14 7 10 2 15 3 6 10 5 12 8 13 4 11 1 9 14 7 2
6	5	1 2 4 8 16 29 9 12 7 26 23 6
7	5	1 2 4 8 16 14 27 21 26 25 13 31 9 7
8	5	1 2 4 8 16 25 31 14 28 13 5 22 11 26 7 21
9	5	1 2 4 8 16 3 21 29 27 30 15 13 10 25 6 18 23 28
10	5	1 2 4 8 16 14 3 9 26 22 21 31 5 29 28 25 17 23 15 27
11	5	1 2 4 8 16 13 28 15 31 5 22 23 11 19 17 29 9 3 21 25 27 7
12	5	1 2 4 8 16 10 7 27 30 19 29 22 31 9 25 26 13 24 21 14 28 12 11 15
13	5	1 2 4 8 16 3 17 23 21 27 13 15 28 30 6 11 18 10 5 29 22 9 26 12 14 25
14	5	1 2 4 8 16 14 17 11 24 31 28 29 19 5 10 25 6 22 18 20 13 9 7 30 23 27 21 26
15	5	1 2 4 8 16 21 25 17 28 26 20 3 30 11 6 13 29 7 9 12 31 10 15 24 19 14 23 22 5 27
16	5	1 2 4 8 16 18 22 30 11 12 29 13 19 25 9 3 31 28 24 7 20 17 2 26 14 23 10 6 27 21 15 5
17	5	1 2 4 8 16 19 12 11 3 26 7 25 18 5 21 28 13 26 17 10 30 29 27 23 14 9 21 19 6 24 31 28 20 22
18	5	1 2 4 8 16 22 9 11 13 24 18 28 3 23 15 27 6 10 25 18 27 19 26 12 30 17 31 7 29 14 8 16 20 21 13 5
19	5	1 2 4 8 16 23 14 28 11 13 24 18 29 3 25 26 22 7 31 30 27 8 6 25 20 19 22 18 5 7 9 10 21 17 16 31 12 29
20	5	1 2 4 8 16 18 29 9 27 7 10 24 19 22 15 30 12 21 28 23 22 21 5 26 11 9 6 13 3 11 25 15 1 5 31 24 20 27 7 17
7	6	1 2 4 8 16 32 62 53 7 45 51 14 34 26
8	6	1 2 4 8 16 32 29 55 56 27 33 62 19 38 15 52
9	6	1 2 4 8 16 32 21 27 19 48 30 45 41 39 28 38 42 63
10	6	1 2 4 8 16 32 19 57 14 63 29 11 58 47 45 25 7 37 53 51
11	6	1 2 4 8 16 32 11 63 45 21 51 39 48 42 31 58 60 46 14 54 24 57
12	6	1 2 4 8 16 32 26 23 37 61 52 43 62 9 47 51 34 6 57 21 40 14 5 54
13	6	1 2 4 8 16 32 37 19 46 28 49 63 25 7 62 60 42 56 10 55 13 31 12 43 26 41
14	6	1 2 4 8 16 32 21 19 63 14 60 28 38 43 10 27 52 15 9 45 49 22 57 33 50 7 44 23
15	6	1 2 4 8 16 32 13 22 56 59 13 61 23 37 49 50 11 19 38 28 14 35 44 41 26 55 52 21 42 7

16	6	1 2 4 8 16 32 41 44 37 26 28 31 55 38 35 49 22 61 42 7 19 50 25 52 47 21 14 59 11 56 13 40
17	6	1 2 4 8 16 32 22 37 58 48 14 29 41 55 27 60 34 17 40 10 53 46 31 54 28 50 9 38 19 13 47 5 11 49
18	6	1 2 4 8 16 32 13 62 42 25 3 23 36 50 31 56 47 52 14 59 21 19 38 49 35 63 28 55 22 41 37 26 11 7 61 44
19	6	1 2 4 8 16 32 27 56 37 28 5 46 35 59 23 29 52 42 18 36 53 6 49 60 54 19 41 50 39 15 45 12 20 25 63 11 30 26
20	6	1 2 4 8 16 32 48 14 39 27 33 5 28 15 42 45 52 22 41 12 62 50 29 3 21 63 57 38 53 51 56 43 26 46 9 19 23 44 36 17
21	6	1 2 4 8 16 32 14 46 63 25 12 15 40 10 26 62 21 35 5 19 54 37 6 41 18 55 20 3 7 40 45 60 50 53 24 29 33 57 62 51 34 25
22	6	1 2 4 8 16 32 3 19 7 50 31 25 9 21 54 26 44 46 61 55 56 57 28 6 27 42 22 60 41 37 38 12 52 20 11 14 43 33 59 49 36 30 13 35
23	6	1 2 4 8 16 32 42 28 43 62 20 5 39 56 44 63 49 26 50 15 38 3 17 33 19 41 11 25 6 59 14 60 22 12 27 54 57 52 36 30 46 53 9 29 45 35
24	6	1 2 4 8 16 32 42 28 43 62 20 5 39 56 44 63 49 26 50 15 38 3 17 33 19 25 6 41 11 59 14 60 22 12 27 54 57 52 36 30 46 53 9 29 45 35 23 51
8	7	1 2 4 8 16 32 64 127 18 31 51 80 116 106 94 105
9	7	1 2 4 8 16 32 64 29 119 59 91 124 30 120 46 111 77 41
10	7	1 2 4 8 16 32 64 118 107 88 83 89 21 36 7 106 41 100 30 63
11	7	1 2 4 8 16 32 64 19 71 113 63 72 58 45 53 91 111 35 93 73 86 102
12	7	1 2 4 8 16 32 64 29 118 97 91 46 111 112 7 85 73 56 100 59 67 106 55 125
13	7	1 2 4 8 16 32 64 23 61 115 103 73 124 56 27 37 85 54 122 76 17 79 42 96 67 110
14	7	1 2 4 8 16 32 64 103 54 98 15 127 75 30 70 45 120 93 87 43 108 113 25 46 21 69 116 60
15	7	1 2 4 8 16 32 64 25 41 115 30 58 71 105 60 49 108 50 127 10 90 15 92 39 81 120 116 67 101 23
16	7	1 2 4 8 16 32 64 14 44 69 43 124 29 70 63 27 118 24 119 41 113 82 100 62 75 96 99 30 115 81 89 84
17	7	1 2 4 8 16 32 64 38 5 107 85 41 30 112 50 123 97 49 42 23 94 29 88 83 19 109 110 60 119 84 74 15 76 103
18	7	1 2 4 8 16 32 64 85 28 88 47 114 7 58 52 99 74 75 109 94 102 104 124 119 76 105 69 116 81 53 123 43 22 13 82 91
19	7	1 2 4 8 16 32 64 38 14 35 19 86 97 95 122 107 88 113 77 57 109 74 73 83 54 11 85 103 102 25 125 13 60 114 104 119 66 92
20	7	1 2 4 8 16 32 64 43 82 84 23 81 90 112 57 30 103 52 71 75 101 105 37 91 51 38 93 25 47 118 33 92 40 58 78 106 20 108 87 88
21	7	1 2 4 8 16 32 64 103 52 114 75 94 22 120 35 42 63 85 105 59 14 66 56 19 112 65 90 89 115 38 37 62 26 108 55 69 78 13 7 111 47 92
22	7	1 2 4 8 16 32 64 28 88 74 82 65 47 44 83 7 57 118 105 93 39 111 54 116 58 31 33 123 76 101 34 26 23 94 70 53 43 51 75 126 112 79 98 9
23	7	1 2 4 8 16 32 64 120 96 89 122 92 27 55 37 63 108 50 52 70 103 13 15 28 126 90 86 76 68 71 42 98 53 67 49 127 116 81 19 25 41 97 35 9 107 14
24	7	1 2 4 8 16 32 64 36 3 38 23 54 65 42 63 56 70 94 47 123 124 100 112 19 35 50 30 80 71 107 62 21 109 114 88 15 74 90 111 13 76 119 27 57 125 9 106 97
25	7	1 2 4 8 16 32 64 22 31 91 102 35 108 101 56 92 111 116 45 123 85 42 15 82 73 57 94 72 33 26 114 62 71 67 53 120 6 28 111 50 43 97 75 13 5 17 54 124 119 80
26	7	1 2 4 8 16 32 64 62 122 44 87 23 97 92 17 39 57 33 107 69 53 100 43 104 116 106 71 115 53 127 124 31 90 47 78 5 10 9 86 103 50 28 76 19 13 88 6 58 40 66 89 113
27	7	1 2 4 8 16 32 64 112 125 51 123 70 21 48 103 25 67 46 92 82 24 89 15 55 98 126 120 22 27 38 44 60 105 57 45 18 39 91 49 42 97 30 79 100 108 111 76 19 43 58 69 54 102 115
28	7	1 2 4 8 16 32 64 108 107 52 46 41 83 97 84 59 119 78 114 77 120 102 11 126 49 61 23 39 90 66 18 13 89 85 106 26 22 92 72 36 62 116 123 71 56 37 68 14 73 111 86 75 127 99 93 65
9	8	1 2 4 8 16 32 64 128 231 23 113 54 223 189 107 45 232 179
10	8	1 2 4 8 16 32 64 128 235 190 141 89 52 119 78 43 147 214 236 241
11	8	1 2 4 8 16 32 64 128 117 233 15 141 166 106 226 124 154 221 206 90 57 89
12	8	1 2 4 8 16 32 64 128 95 214 241 54 251 158 108 163 216 230 106 57 237 195 116 75
13	8	1 2 4 8 16 32 64 128 227 55 218 77 102 75 84 19 246 253 234 120 141 215 127 164 101 171
14	8	1 2 4 8 16 32 64 128 27 86 62 204 177 235 251 166 98 41 88 101 180 172 151 217 199 218 81 47
15	8	1 2 4 8 16 32 64 128 178 183 123 223 225 185 238 117 232 120 197 158 216 31 78 37 171 214 141 60 98 84
16	8	1 2 4 8 16 32 64 128 206 247 166 180 219 62 149 131 108 95 118 61 120 89 197 83 212 42 177 107 242 146 13 27
17	8	1 2 4 8 16 32 64 128 167 245 79 169 200 118 93 51 67 248 87 188 94 158 153 38 178 58 227 219 45 177 52 212 63 88
18	8	1 2 4 8 16 32 64 128 105 62 188 187 147 223 254 75 50 112 86 45 243 157 92 167 26 127 245 49 170 164 71 208 200 217 21 198
19	8	1 2 4 8 16 32 64 128 56 191 190 172 62 170 102 240 89 125 233 65 94 205 142 165 166 59 238 115 130 85 52 163 111 41 23 116 220 181
20	8	1 2 4 8 16 32 64 128 6 209 78 232 44 242 109 203 162 53 92 69 179 147 10 131 23 165 121 196 252 245 150 154 43 225 125 58 112 190 95 157
21	8	1 2 4 8 16 32 64 128 83 50 13 173 39 208 198 155 253 235 242 90 236 212 218 175 44 101 122 195 181 53 133 150 98 126 20 220 11 96 92 99 201 77
22	8	1 2 4 8 16 32 64 128 104 135 190 184 21 200 58 215 113 183 220 235 155 46 212 61 22 116 203 14 94 90 245 148 65 39 140 254 231 120 119 19 209 109 37 178
23	8	1 2 4 8 16 32 64 128 90 61 84 6 195 47 249 148 252 206 228 31 100 83 223 143 120 114 58 133 40 11 154 135 162 134 107 193 231 177 147 242 182 234 205 216 171 53
24	8	1 2 4 8 16 32 64 128 127 221 152 237 248 210 134 209 47 94 67 121 13 26 52 104 207 190 196 246 159 228 200 178 227 245 234 170 188 206 185 219 151 137 161 165 243 148 175 131
25	8	1 2 4 8 16 32 64 128 216 179 214 189 255 137 244 83 49 100 11 241 20 89 134 115 30 186 205 45 53 60 230 207 220 164 235 165 159 29 74 122 86 104 63 197 7 97 168 195 147 131
26	8	1 2 4 8 16 32 64 128 58 176 10 253 61 198 241 161 31 131 197 113 212 168 166 86 141 158 229 98 188 221 227 83 142 92 78 152 201 143 151 232 202 57 148 79 187 235 145 250 120 238 170 103
27	8	1 2 4 8 16 32 64 128 11 53 102 123 225 13 222 180 136 243 142 201 203 30 86 209 83 109 49 199 31 181 121 25 57 226 90 205 162 174 104 236 163 68 229 54 157 74 119 146 246 24 116 58 169 134
28	8	1 2 4 8 16 32 64 128 167 196 159 103 48 56 135 189 148 13 186 66 235 110 105 21 164 253 179 92 42 204 97 219 109 240 54 116 172 182 95 114 252 58 239 213 98 201 55 162 177 27 69 210 220 36 247 60
29	8	1 2 4 8 16 32 64 128 15 35 95 208 59 165 225 115 42 147 28 186 7 70 255 249 174 50 107 86 21 159 73 131 172 26 110 216 121 246 162 212 183 60 188 233 177 66 199 137 237 221 27 251 211 96 17 222 55 119
30	8	1 2 4 8 16 32 64 128 74 58 91 233 178 71 39 200 126 196 229 224 152 172 193 220 163 119 203 253 50 173 155 143 95 27 148 177 85 166 180 222 165 238 53 61 51 213 134 46 70 30 56 194 191 107 215 73 82 63 255 87
31	8	1 2 4 8 16 32 64 128 119 193 60 207 131 108 246 179 222 38 182 204 7 20 33 172 126 82 26 42 187 85 157 152 226 206 98 105 168 146 113 249 143 91 209 134 133 124 88 63 189 25 71 198 95 13 106 58 245 111 12 231 143 170
32	8	1 2 4 8 16 32 64 128 23 142 28 49 42 103 188 19 85 95 167 122 91 144 18 159 47 225 108 169 137 182 204 221 232 207 11 187 99 87 249 110 245 55 226 65 6 198 44 174 179 237 40 244 189 109 120 131 149 215 33 240 164 30 210 88

Appendix C

Technical Note dated March 17, 1994

Documenting Preliminary Results of the PCI Study

Technical Note

Date: March 17, 1994
To: Warner Miller, Victor Sank, GSFC
From: William J. Ebel, Mississippi State University
Subject: TRMM Performance *Without* the PCI
Cc: Frank Ingels

Abstract - The developments of the Communication Link and Error ANalysis (CLEAN) simulator have reached the point where accurate assessments of the performance of the TDRSS downlink *without* the PCI but *with* node synchronization in the Viterbi decoder can be performed. In this note, the applicable developments of CLEAN are described and preliminary results of the PCI study are presented.

I. Introduction

As you are well aware, the TDRSS downlink includes an optional PCI and a required rate 1/2 constraint length 7 convolutional code. The demodulator provides 8-level soft-decision data to the DePCI and subsequently to the Viterbi decoder. Tap synchronization of the DePCI or node synchronization of the Viterbi decoder must be established at the receiver depending upon whether the PCI is switched on. The question which has been posed and discussed [1,2] is whether the PCI is necessary for the TDRSS downlink as used by TRMM.

In Section II to follow, applicable developments of CLEAN are described and their relevance to the actual system is discussed. In Section III, preliminary studies of the performance of the TDRSS downlink for TRMM *without* the PCI is presented and conclusions follow in Section IV.

II. CLEAN Simulator Developments

Two recent CLEAN developments are relevant to the issue at hand; 1) program *vit3sync* which mimics the LV7017C Viterbi decoder with node synchronization, and 2) program *rfi* which mimics the effect at the soft-decision demodulator output of multiply occurring RFI sources in the TDRSS downlink. Besides these, program *blkdeint* which performs depth 5 block deinterleaving and *blkdecod* which performs Reed-Solomon (RS) decoding for the (255,223) RS code are used to estimate performance. These latter two programs are described in [3].

The soft-decision data output by the TDRSS downlink receiver at White Sands is input to the LV7017C hardware for PCI/node synchronization and Viterbi decoding. The specific details of the synchronization strategy and Viterbi decoding can be found in [4]. The algorithm to perform node synchronization, when the PCI is not switched on, and soft-decision Viterbi decoding in the LV7017C hardware has been ***exactly duplicated*** in the CLEAN program *vit3sync*. This includes the Viterbi decoder trellis metrics, the metric renormalization strategy, the node synchronization strategy using a SyncCounter, the path memory length, etc.

Both functional and statistical verifications of program *vit3sync* were performed. For the statistical verification, two critical performance factors for the LV7017C Viterbi decoder were considered; (1) the Viterbi decoder output Bit Error Rate (BER), and (2) the average time to detect loss of node synchronization. To consider item (1), Figure 3.1 shown on the next page was extracted from [4]. This figure shows a comparison of the LinCom Viterbi decoder simulation results with theoretical bounds assuming an AWGN channel. The cross hairs show corresponding simulation results for program *vit3sync* of the CLEAN simulator. The BER for *vit3sync* matches the LinCom simulation exceptionally well. To consider item (2), various soft values were deleted from a post threshold data stream. The average time to detect loss of node synchronization has been shown to be roughly 200-300 bits at the Viterbi decoder output [4,5] for $E_b/N_0 = 5$dB. For *vit3sync*, average time to detect loss of node synchronization was observed to be 200-350 bits at the Viterbi decoder output.

Program *rfi* was written to mimic the effect several RFI sources have on the soft values output by the 8-level receiver threshold device and also includes thermal noise. It is assumed that the occurrence time for noise bursts are Poisson in distribution and that the burst duration spans 15 threshold output soft values. The program requires the following inputs:

1) Hard-decision error probability due to thermal noise alone
2) Number of RFI sources

and for each RFI source, the following inputs are required:

1) Hard-decision error probability during a burst (inclusive of thermal noise)
2) Average interval between bursts (reciprocal of rate of burst occurrence)

3) Burst length (in terms of the number of threshold output soft values)

The philosophy for determining the soft values output by the 8-level threshold device is as follows. When no RFI occurs, the random variable at the threshold input is Gaussian with a mean and variance which are related to the signal power and noise power, respectively. The ratio of the signal power to the noise power is the signal-to-noise ratio. In fact, there is a Q-function relationship between the hard-decision error probability and the signal to noise ratio for the random variable at the threshold input. When an RFI noise burst occurs, it has the effect of increasing the noise power but does not affect the signal power, and therefore has the effect of increasing the hard-decision error probability. However, the probability density function of the random variable at the threshold input is still Gaussian. Therefore, to determine the probability density function of the random variable at the threshold input, it is only necessary to know the hard-decision error probability at the threshold output during each RFI source burst. In summary, to determine which soft value is output by the thresholder, it necessary to determine:

1) which RFI sources are currently causing a burst to occur
2) the total error probability for the current threshold output value
3) the variance of the Gaussian random variable to be thresholded

The method used to statistically generate the soft output value for a given hard-decision error probability is described in [6] for program *iidsoft*.

In effect, the program can accurately represent the statistical nature of the soft values at the 8-level threshold output due to thermal noise and multiple RFI sources. To justify the appropriateness of this model, recall that all the distortions which occur in the real TDRSS downlink including satellite non-linearities, hardware distortion, etc. are lumped into a single parameter called the *implementation loss* which is realized in terms of an offset to the signal-to-noise ratio at the receiver [7].

III. Preliminary PCI Study Results

Several studies were conducted to investigate the performance of the TDRSS downlink for TDRS West and TDRS East and for the SSA return link. These RFI environments may be found in [8] and are summarized below.

The first study conducted involved simulating the TDRS West environment as described by Ted Kaplan in [2] for TRMM. The TRMM data rate is 2Mbps (4MHz channel rate) and each RFI pulse is 3.5 usec long so that at most 15 threshold output soft values will be affected by a single burst. The signal-to-thermal noise ratio at the receiver was taken to be $E_b/N_0 = 4.5$dB so that the Viterbi decoder output error probability is roughly 10^{-5} (with no RFI). This corresponds to a thermal noise hard-decision post threshold error probability of about 0.0465. Of the 5

known RFI sources which exist for the SSA return link for TDRS West, the two were ignored and three were treated as a single source with infinite power consistent with [2]. This is summarized in Table I [8].

Given this channel model, the following simulation was conducted using CLEAN.

1) Generate PN code to simulate data (1.5×10^7 bits) using program *pnseq*
2) Convolutionally encode the PN sequence using the NASA rate 1/2 constraint length 7 convolutional code using program *convencd*
3) Generate the soft sequence which would occur at the threshold output for the encoded binary sequence using programs *bstysoft* and *soften*.
4) Viterbi decode the received soft sequence (***no*** PCI present) using *vit3sync* which includes node synchronization and mimics the LV7017C hardware decoder
5) Determine the error sequence at the Viterbi decoder output using program *madd*
6) Block deinterleave the error sequence (depth 5) using program *blkdeint*
7) Reed-Solomon decode the deinterleaved sequence using program *blkdecod*

The results are summarized in Table II and suggest the following. If the signal EIRP is chosen so that the Viterbi decoded BER (with no RFI) is 10^{-5} per spec ($E_b/N_0 = 4.5$dB) but RFI for TDRS West happens to be in the channel (clearly a worse case scenario), then the Reed-Solomon decoder can correct many of the errors and provide a total system BER in the neighborhood of 2×10^{-6}. It is difficult to draw tangible conclusions from these results because the TDRS West RFI environment has been simplified (as in [2]) and because the simulation only resulted in 1 RS decoding failure out of 7350 codewords which is not a very good statistical estimate. However, these results bring into question the results shown in Figure 1 of reference [2] which shows the RS decoder output BER to be about 2×10^{-4} for an $E_b/N_0 = 4.5$dB and no PCI. If this were the true RS decoder output BER, then there should have been on the order of 3000 binary errors at the RS decoder output.

To more accurately represent the TDRS East and TDRS West RFI environments, program *rfi* of the CLEAN simulator was used to simulate all the RFI sources for two different thermal noise EIRP values as shown in Tables III, IV, V, and VI. The signal EIRP was taken to be 31.6dBW which gives a hard-decision threshold output BER of 0.0283 when 29dBW thermal noise is present with no RFI and 3.47×10^{-4} when 24dBW thermal noise is present with no RFI. The burst length of every RFI burst was taken to be 15 soft-decision threshold output values consistent with [2].

These environments were used with the CLEAN simulation described above to investigate the performance of the NASA concatenated coding scheme including the rate 1/2 constraint length 7 convolutional inner code with the (255,223) RS outer code. The PCI was *not* switched on and program *vit3sync* was used to perform Viterbi decoding which includes node

synchronization. The results are shown in Tables VII, VIII, IX, and X. These results suggest that for the worst case scenario, the system will fail for the TDRS East environment. Although decoding failure did not occur for the TDRS West environment, no tangible conclusions can be drawn at this time due to insufficient statistics.

IV. Conclusions

In conclusion, the Communication Link and Error ANalysis (CLEAN) simulator is a useful tool for investigating the performance of the TDRSS downlink. The issue of whether the PCI is necessary for the TDRSS downlink has been investigated. Preliminary results suggest that the TDRS West RFI environment may not be a problem without the PCI, but that the worst case TDRS East environment may give rise to significant decoding failures out of the Reed-Solomon decoder.

Bibliography

1. W. Miller, "TRMM Performance in RFI without the PCI," NASA Goddard, Code 738.3, December 16, 1993.
2. T. Kaplan and T. Berman, "Performance of TRMM Communications in RFI With and Without the PCI," Stanford Telecom, Code 531.1, December 22, 1993.
3. Ebel, W.J., and Ingels, F.M., "An Investigation of Error Characteristics and Coding Performance", MSU Department of Electrical and Computer Engineering, Technical Semi-Annual Report, December 30, 1992, NASA Grant NAG5-2006.
4. Wang, James, and Peng, Wei-Chung, "Simulation and Validation of Viterbi Decoder", Interoffice Memorandum, LinCom Corporation, TM-8719-05-09 and TM-8707-06, March 1, 1989.
5. Operational and Maintenance Manual for General Purpose LV7017C Convolutional Encoder-Viterbi-Decoder and Interleaver-Deinterleaver, Hughes Network Systems, Hughes Aircraft Company, TM 200090, October 1989.
6. Ebel, W.J., Ingels, F.M., and Crowe, S., "The Communication Link and Error ANalysis (CLEAN) Simulator", MSU Department of Electrical and Computer Engineering, Technical Semi-Annual Report, December 30, 1993, NASA Grant NAG5-2006.
7. Wang, J., and Lai, D., "Simulation of Convolutional and Reed-Solomon Coded System", LinCom, TR-8719-13, April 30, 1990.
8. McKenzie, T.M., and Choi, H., "User's Guide to CLASS Computer Program for Synchronization and Doppler Tracking with RFI: Revision 2", LinCom, TR-8512-22, December 1985.

Table I. Model of the TDRS West RFI environment

EIRP (dBW)	Burst Length	Duty Cycle (%)	Comments
25	15	3.5	ignored
35	15	2.2	ignored
45	15	0.6	1.8% D.C.
55	15	1.1	single source
65	15	0.1	BER = 1/2

Table II. Performance results for the TDRS West RFI environment model

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	3×10^7	1.64×10^6 (2.68×10^5 in bursts)	0.0547
Viterbi Decoder Output	1.5×10^7	1.0×10^5 (no node sync loss)	6.68×10^{-3}
Reed-Solomon Decoder Output	1.5×10^7	34	2×10^{-6} (1 decoding failure out of 7350 codewords)

Table III. Model of the TDRS East RFI environment for thermal noise EIRP of 29dBW

EIRP (dBW)	Duty Cycle (%)	Total Noise w/ Thermal (dBW)	E_s/N_0	Raw Channel BER
Thermal	N/A	29	2.6	0.0283
20	10	29.5	2.1	0.0359
30	13	32.5	-.9	0.101
40	3	40.3	-8.7	0.301
50	2	50.0	-18.4	Single
60	1.8	60.0	-28.4	Source
70	0.2	70.0	-38.4	BER = 1/2

Table IV. Model of the TDRS East RFI environment for thermal noise EIRP of 24dBW

EIRP (dBW)	Duty Cycle (%)	Total Noise w/ Thermal (dBW)	E_s/N_0	Raw Channel BER
Thermal	N/A	24	7.6	3.47×10^{-4}
20	10	25.5	6.1	0.00216
30	13	31.0	0.6	0.0649
40	3	40.0	-8.5	0.297
50	2	50.0	-18.4	Single
60	1.8	60.0	-28.4	Source
70	0.2	70.0	-38.4	BER = 1/2

Table V. Model of the TDRS West RFI environment for thermal noise EIRP of 29dBW

EIRP (dBW)	Duty Cycle (%)	Total Noise w/ Thermal (dBW)	E_s/N_0	Raw Channel BER
Thermal	N/A	29	2.6	0.0283
25	3.5	30.4	1.1	0.0543
35	2.2	36.0	-4.4	0.197
45	0.6	45.1	-13.5	0.382
55	1.1	55.0	-23.4	Single
65	0.1	65.0	-33.4	BER = 1/2

Table VI. Model of the TDRS West RFI environment for thermal noise EIRP of 24dBW

EIRP (dBW)	Duty Cycle (%)	Total Noise w/ Thermal (dBW)	E_s/N_0	Raw Channel BER
Thermal	N/A	24	7.6	3.47×10^{-4}
25	3.5	27.5	4.1	0.0117
35	2.2	35.3	-3.7	0.178
45	0.6	45.0	-13.4	0.380
55	1.1	55.0	-23.4	Single
65	0.1	65.0	-33.4	BER = 1/2

Table VII. Performance results for the TDRS East RFI environment model with 29dBW thermal noise EIRP

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	2×10^6	1.31×10^5 (8.99×10^4 in bursts)	0.0657
Viterbi Decoder Output	1.0×10^6	2.6×10^4 (no node sync loss)	0.0260
Reed-Solomon Decoder Output	1.0×10^6	2.31×10^4	0.0232 (411 decoding failures out of 490 codewords)

Table VIII. Performance results for the TDRS East RFI environment model with 24dBW thermal noise EIRP

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	2×10^6	7.35×10^4 (7.30×10^4 in bursts)	0.0367
Viterbi Decoder Output	1.0×10^6	1.83×10^4 (no node sync loss)	0.0183
Reed-Solomon Decoder Output	1.0×10^6	9.65×10^3	0.00965 (213 decoding failures out of 490 codewords)

Table IX. Performance results for the TDRS West RFI environment model with 29dBW thermal noise EIRP

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	2×10^6	8.11×10^4 (2.84×10^4 in bursts)	0.0406
Viterbi Decoder Output	1.0×10^6	6.33×10^3 (no node sync loss)	0.00633
Reed-Solomon Decoder Output	1.0×10^6	0	0.0 (0 decoding failures out of 490 codewords)

Table X. Performance results for the TDRS West RFI environment model with 24dBW thermal noise EIRP

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	2×10^6	2.58×10^4 (2.51×10^4 in bursts)	0.0129
Viterbi Decoder Output	1.0×10^6	4.59×10^4 (no node sync loss)	0.00459
Reed-Solomon Decoder Output	1.0×10^6	0	0.0 (0 decoding failures out of 490 codewords)

Model of the new TDRS East RFI environment
for thermal noise EIRP of 29dBW (BW = 20MHz)

EIRP (dBW)	Duty Cycle High RFI	Duty Cycle Low RFI	Total Noise w/ Thermal (dBW)	E_s/N_0	Raw Channel BER
Thermal	N/A	N/A	29	2.6	0.0283
30	6.5	4.5	32.5	-0.9	0.101
40	2.5	1.5	40.3	-8.7	0.301
50	3.5	1.0	50	-18.4	1/2

Performance results for the new TDRS East High RFI environment model
with 29dBW thermal noise EIRP

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	1×10^6	1.11×10^5 (6.15×10^4 in bursts)	0.0557
Viterbi Decoder Output	5.0×10^5	9.33×10^3 (no node sync loss)	0.0186
Reed-Solomon Decoder Output	5.0×10^5	4.50×10^3	0.00900 (95 decoding failures out of 245 codewords)

Performance results for the new TDRS East Low RFI environment model
with 29dBW thermal noise EIRP

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	1×10^6	8.06×10^4 (2.77×10^4 in bursts)	0.0403
Viterbi Decoder Output	5.0×10^5	2.97×10^3 (no node sync loss)	0.00594
Reed-Solomon Decoder Output	5.0×10^5	0	0 (0 decoding failures out of 245 codewords)

Model of the new TDRS East RFI environment
for thermal noise EIRP of 23.5dBW (BW = 20MHz)

EIRP (dBW)	Duty Cycle High RFI	Duty Cycle Low RFI	Total Noise w/ Thermal (dBW)	E_s/N_0	Raw Channel BER
Thermal	N/A	N/A	23.5	8.1	1.64×10^{-4}
30	6.5	4.5	30.9	0.7	0.0627
40	2.5	1.5	40.1	-8.5	0.297
50	3.5	1.0	50	-18.4	1/2

Performance results for the new TDRS East High RFI environment model
with 23.5dBW thermal noise EIRP

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	1×10^6	5.69×10^4 (5.66×10^4 in bursts)	0.0284
Viterbi Decoder Output	5.0×10^5	6.99×10^3 (no node sync loss)	0.0140
Reed-Solomon Decoder Output	5.0×10^5	1.51×10^3	0.00302 (37 decoding failures out of 245 codewords)

Performance results for the new TDRS East Low RFI environment model
with 23.5dBW thermal noise EIRP

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	1×10^6	2.49×10^4 (2.45×10^4 in bursts)	0.0124
Viterbi Decoder Output	5.0×10^5	2.09×10^3 (no node sync loss)	0.00419
Reed-Solomon Decoder Output	5.0×10^5	0	0 (0 decoding failures out of 245 codewords)

Model of the new TDRS East RFI environment
for thermal noise EIRP of 22dBW (BW = 4MHz)

EIRP (dBW)	Duty Cycle High RFI	Duty Cycle Low RFI	Total Noise w/ Thermal (dBW)	E_s/N_0	Raw Channel BER
Thermal	N/A	N/A	22	9.6	9.76×10^{-5}
30	6.5	4.5	25.5	6.1	0.00216
40	2.5	1.5	33.3	-1.7	0.122
50	3.5	1.0	43	-11.4	0.351

Performance results for the new TDRS East High RFI environment model
with 22dBW thermal noise EIRP

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	1×10^6	3.06×10^5 (3.04×10^4 in bursts)	0.0153
Viterbi Decoder Output	5.0×10^5	3.77×10^3 (no node sync loss)	0.00753
Reed-Solomon Decoder Output	5.0×10^5	0	0 (0 decoding failures out of 245 codewords)

Performance results for the new TDRS East Low RFI environment model
with 22dBW thermal noise EIRP

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	1×10^6	1.10×10^4 (1.10×10^4 in bursts)	0.00548
Viterbi Decoder Output	5.0×10^5	918 (no node sync loss)	0.00184
Reed-Solomon Decoder Output	5.0×10^5	0	0 (0 decoding failures out of 245 codewords)

Model of the new TDRS East RFI environment
for thermal noise EIRP of 16.5dBW (BW = 4MHz)

EIRP (dBW)	Duty Cycle High RFI	Duty Cycle Low RFI	Total Noise w/ Thermal (dBW)	E_s/N_0	Raw Channel BER
Thermal	N/A	N/A	16.5	15.1	4.32×10^{-16}
30	6.5	4.5	23.9	7.7	3.01×10^{-4}
40	2.5	1.5	33.1	-1.5	0.117
50	3.5	1.0	43	-11.4	0.351

Performance results for the new TDRS East High RFI environment model
with 16.5dBW thermal noise EIRP

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	1×10^6	3.00×10^4 (3.00×10^4 in bursts)	0.0150
Viterbi Decoder Output	5.0×10^5	3.37×10^3 (no node sync loss)	0.00674
Reed-Solomon Decoder Output	5.0×10^5	0	0 (0 decoding failures out of 245 codewords)

Performance results for the new TDRS East Low RFI environment model
with 16.5dBW thermal noise EIRP

Simulation Location	Simulation Length	Total Observed Binary Errors	Observed BER
Hard-decision Threshold Output	1×10^6	1.07×10^4 (1.07×10^4 in bursts)	0.00533
Viterbi Decoder Output	5.0×10^5	786 (no node sync loss)	0.00157
Reed-Solomon Decoder Output	5.0×10^5	0	0 (0 decoding failures out of 245 codewords)