View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NASA Technical Reports Server

NASA-CR-195243

IN-61-CR

209788

47P

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

(NASA-CR-195243) THE COMMUNICATION
LINK AND ERROR ANALYSIS (CLEAN)
SIMULATOR Semiannual Report, 1 Jul.
1993 - 30 Jun. 1994 (Mississippi
State Univ.) 47 p

N94-27168

Unclas

G3/61 0209788

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.

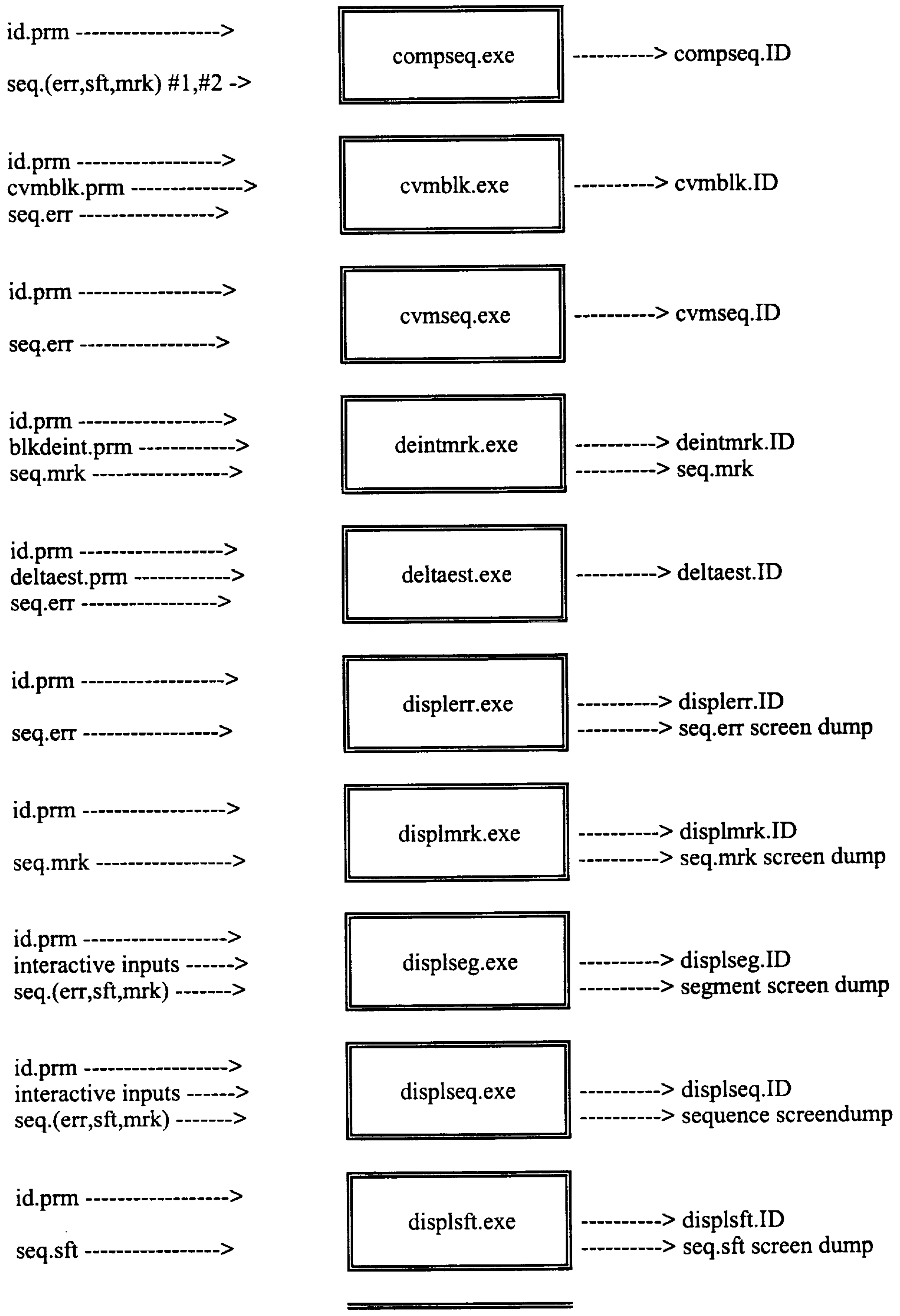
Besides the development of the simulation, a study was performed to determine whether the PCI provides a performance improvement for the TDRSS downlink. There exist RFI with several duty cycles for the TDRSS downlink. We conclude that the PCI does not improve performance for any of these interferers 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.

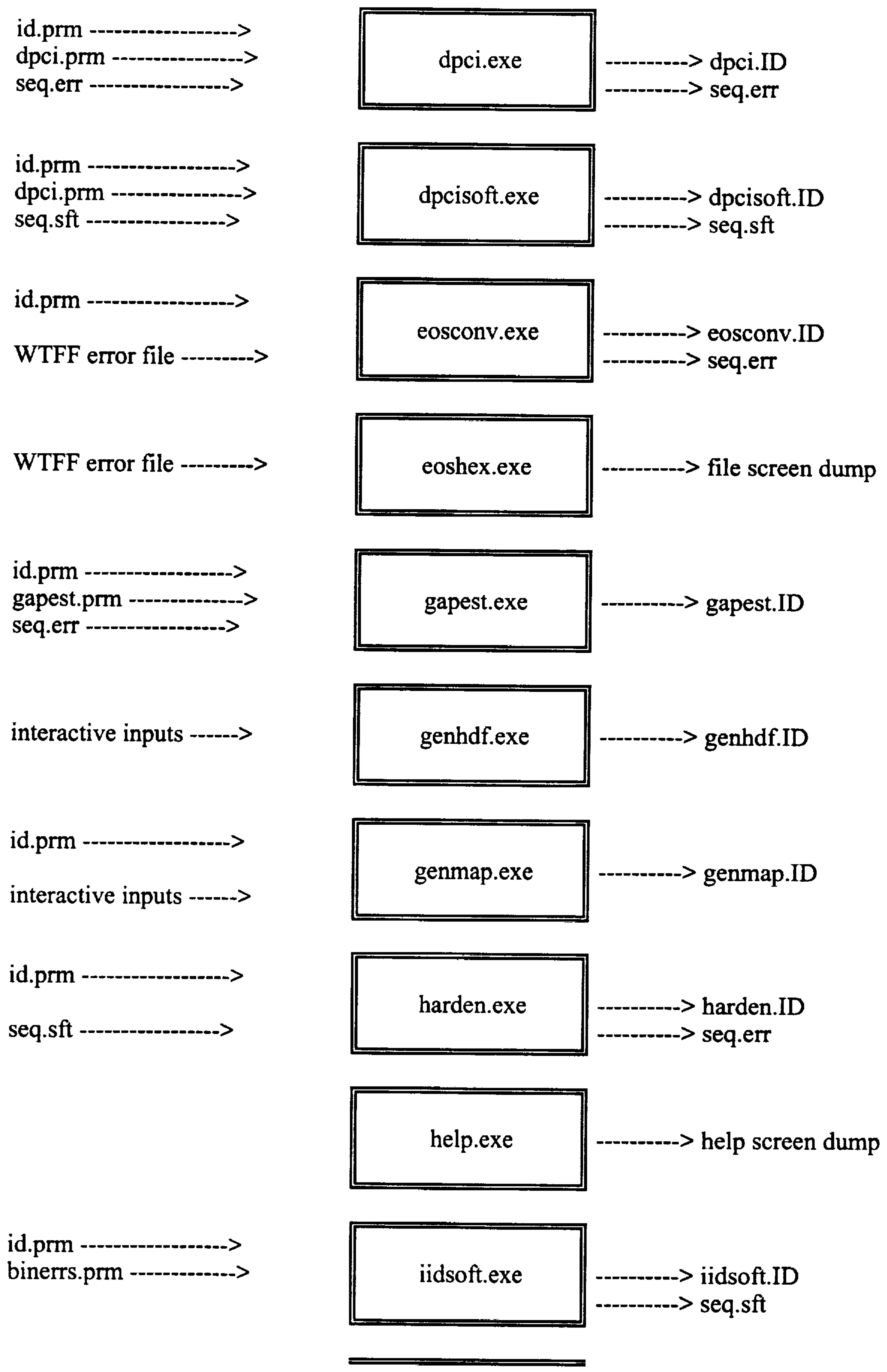
I. Introduction

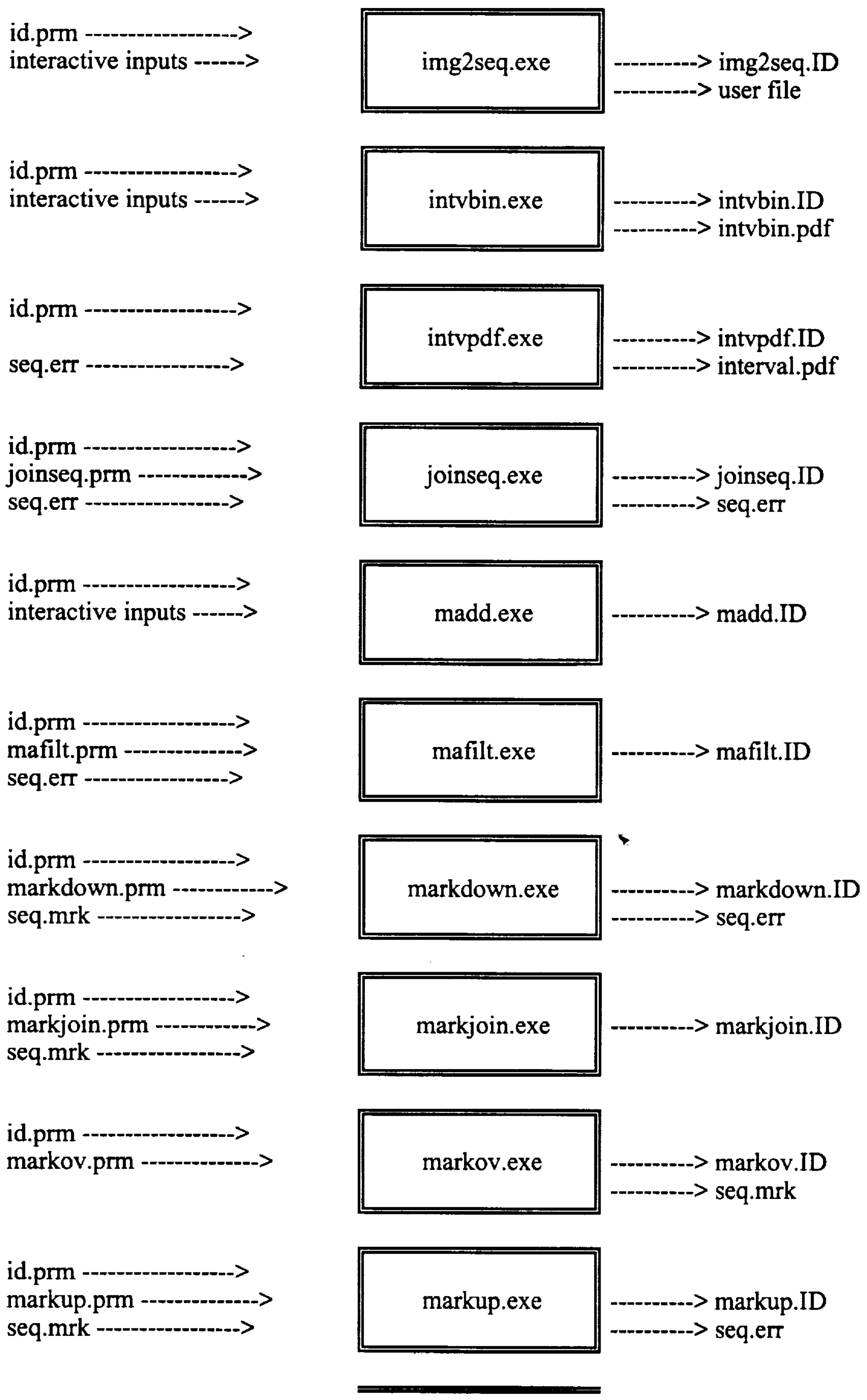
During the past 6 months, CLEAN capabilities have grown substantially. Most of the new programs are briefly described in Section II. Among the developments is the integration of the RICE compression/decompression software into the simulation. In the Appendix, the theory of RICE compression is described 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.

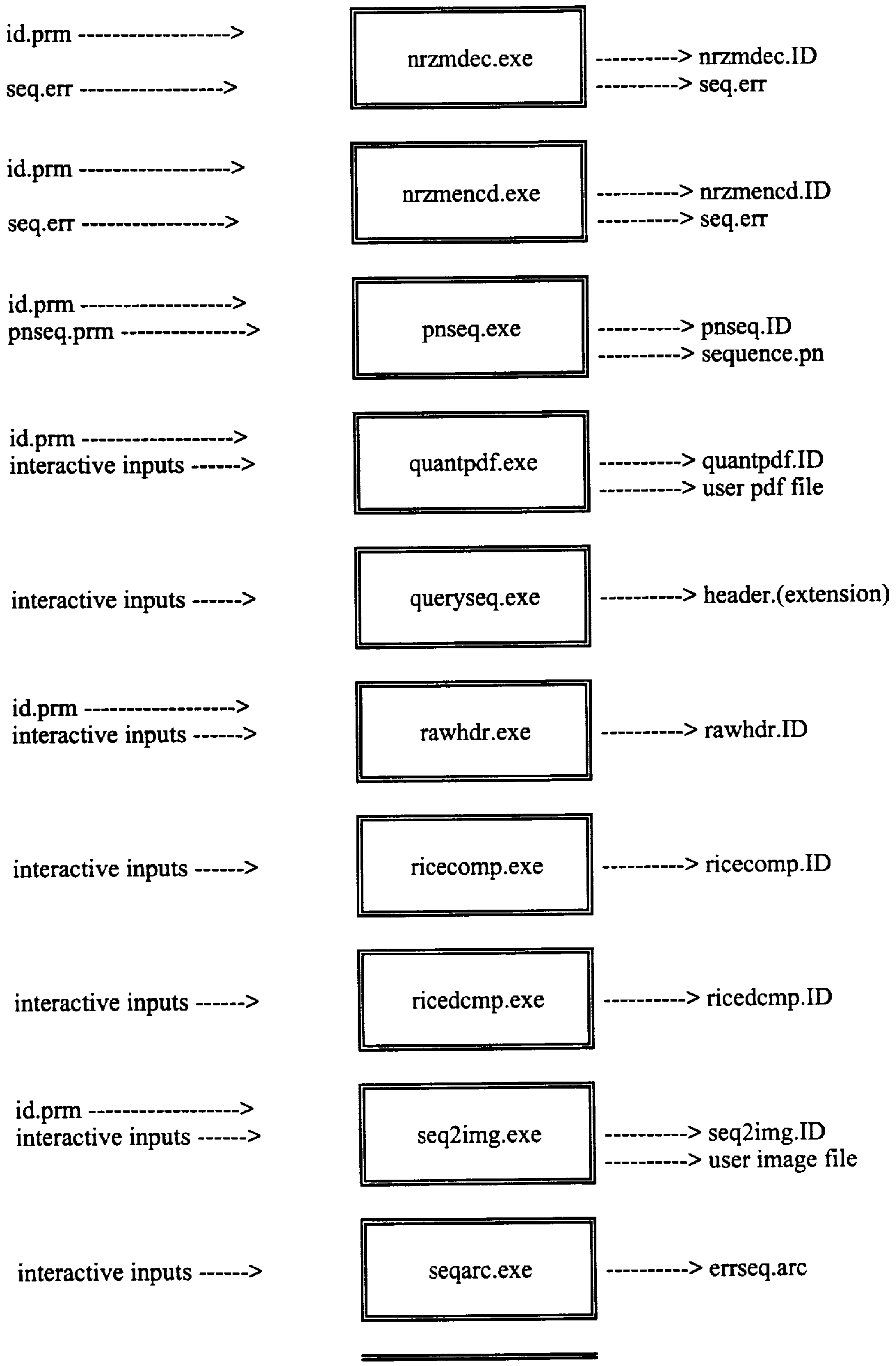
To help run the source code, the following list in given which provides a quick overview of the required input files and the output files which are associated with each program of CLEAN.

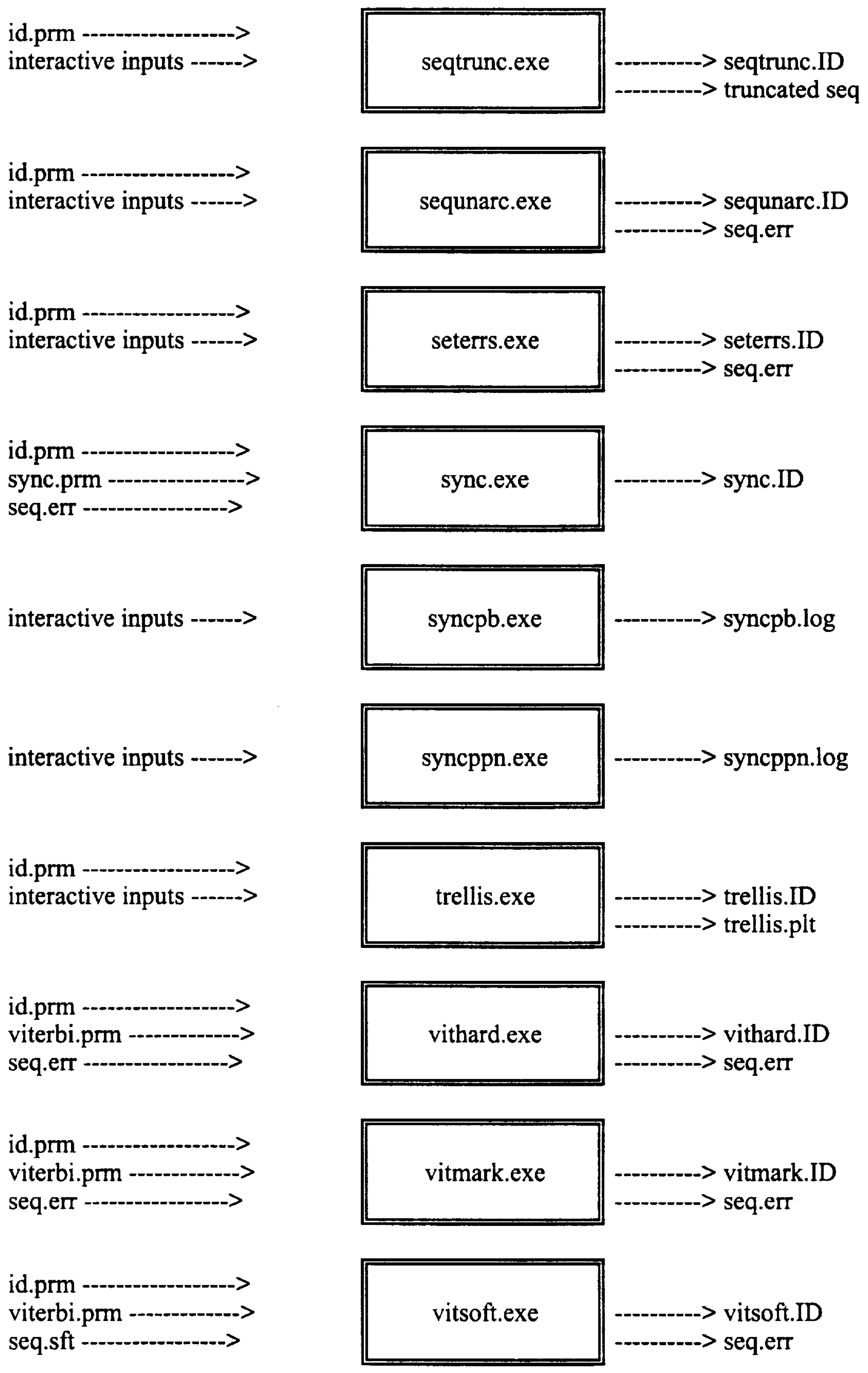
filename(s).pdf --------->	ascii.exe	----------> specified.file
id.prm ------------------> binerrs.prm ------------->	binerrs.exe	----------> binerrs.ID ----------> seq.err
id.prm ------------------> bingap.prm --------------->	bingap.exe	----------> bingap.ID ----------> bingap.pdf
id.prm ------------------> blkdecod.prm ------------> seq.err ------------------>	blkdecod.exe	----------> blkdecod.ID ----------> seq.err
id.prm ------------------> blkdeint.prm ------------> seq.err ------------------>	blkdeint.exe	----------> blkdeint.ID ----------> seq.err
id.prm ------------------> brsterrs.prm ------------>	brsterrs.exe	----------> brsterrs.ID ----------> seq.err
id.prm ------------------> bstyerrs.prm ------------>	bstyerrs.exe	----------> bstyerrs.ID ----------> seq.err
id.prm ------------------> bstyerrs.prm ------------>	bstysoft.exe	----------> softmap.dat ----------> bstysoft.ID ----------> seq.sft


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 | DisplSeg: 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

$$\bar{L} = \sum_{i=1}^{n} L(X_i) p_{X_i} \tag{1}$$

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.

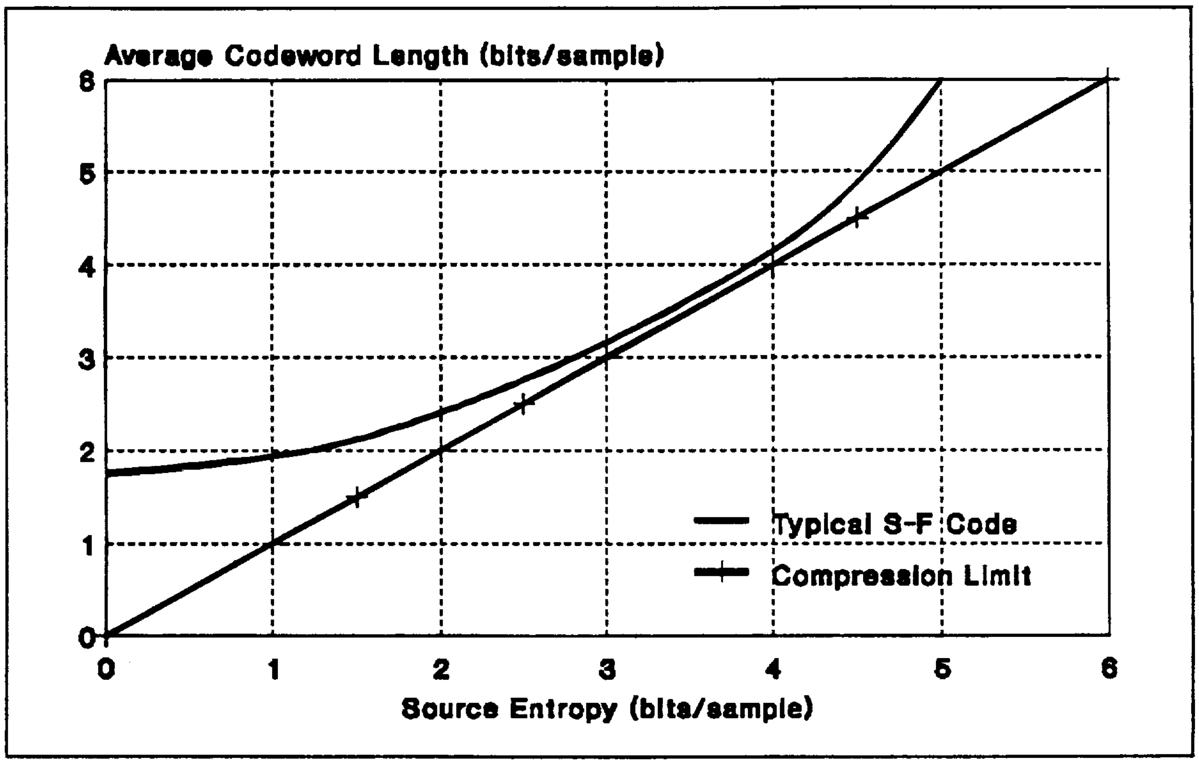


Figure 1. Average Performance for a Typical Shannon-Fanno Code.

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

$$H(X) = -\sum_{i} p_i \log_2 p_i \quad bits/sample \tag{2}$$

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.

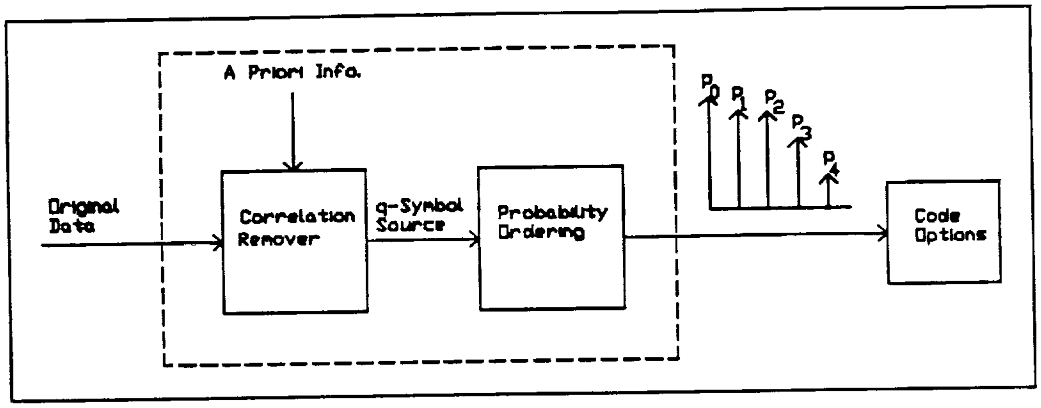


Figure 2. Reversible Preprocessing for RICE Compression.

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

$$fs[i] = 000\ldots0001 \tag{5}$$

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 $C\tilde{FS}$[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

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

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

$$Y = 11010011010100111011010010111 \tag{10}$$

Then the 3rd extension of Y is given as

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

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

$$\psi_2[X] = CFS[X] = cfs[x_1] * cfs[x_2] * \cdots \tag{12}$$

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

$$\psi_0[X] = C\tilde{F}\tilde{S}[X] = c\tilde{f}\tilde{s}[x_1] * c\tilde{f}\tilde{s}[x_2] * \cdots \qquad (13)$$

where $c\tilde{f}\tilde{s}$ comes from Table 2 with the left column complemented. The performance plot for $C\tilde{F}\tilde{S}[X]$ is contained in Figure 4. Note that $C\tilde{F}\tilde{S}[X]$ 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"

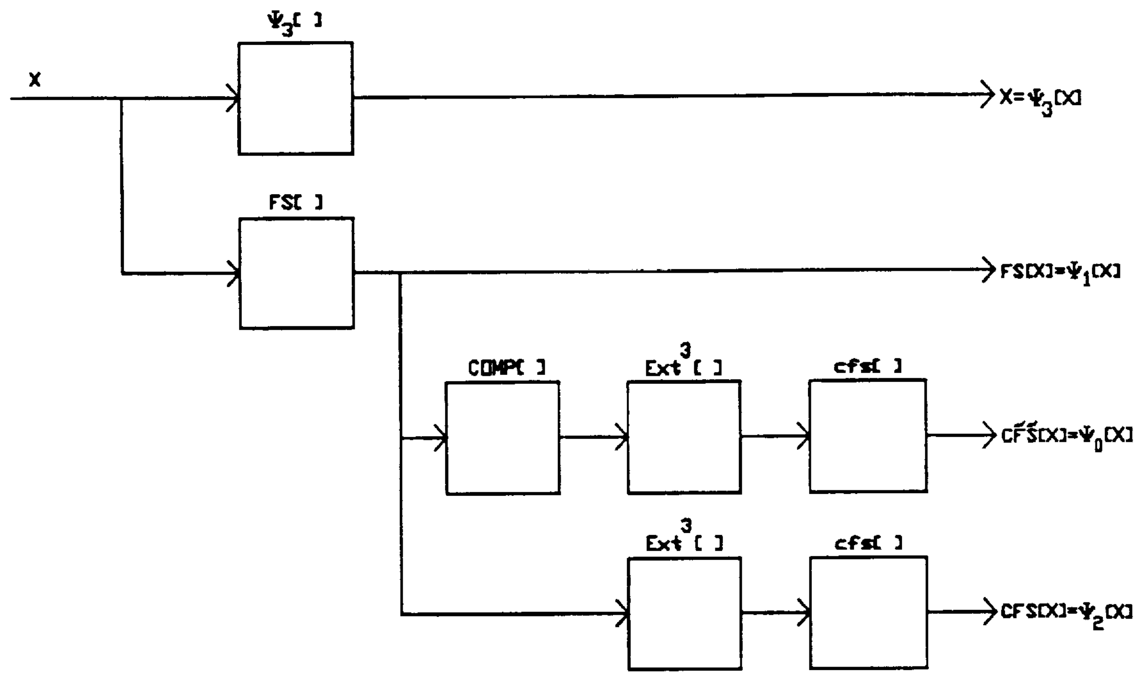


Figure 3. Four Basic Compressor Options.

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

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

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.

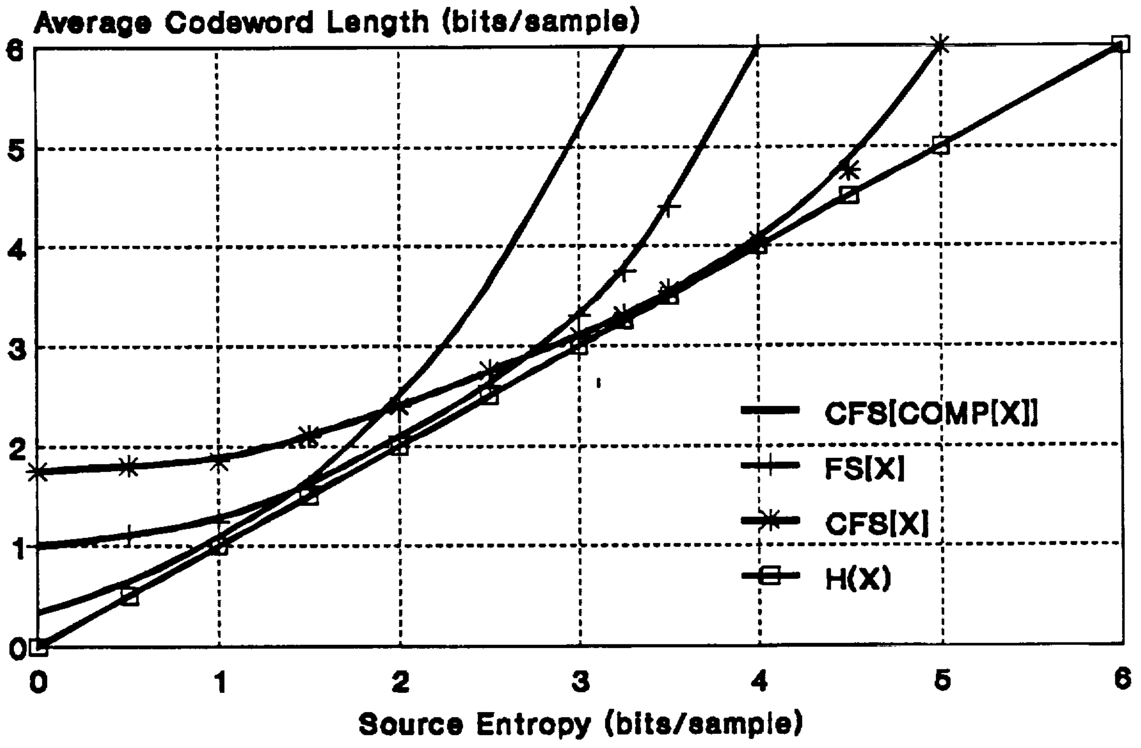


Figure 4: Basic Compressor Performance.

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

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

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.

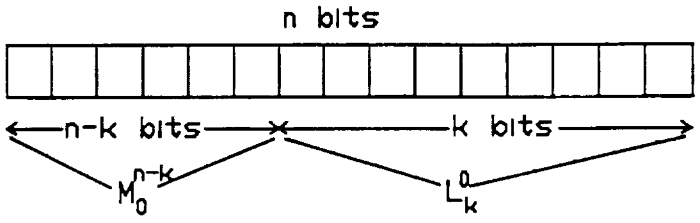


Figure 5. Typical Split-Sample Symbol.

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

$$m_j = b_j^{n-1}2^{n-1}+b_j^{n-2}s^{n-2}+\dots+b_j^0 = \sum_{l=0}^{n-1} b_j^l 2^l \tag{25}$$

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

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

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

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

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

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

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

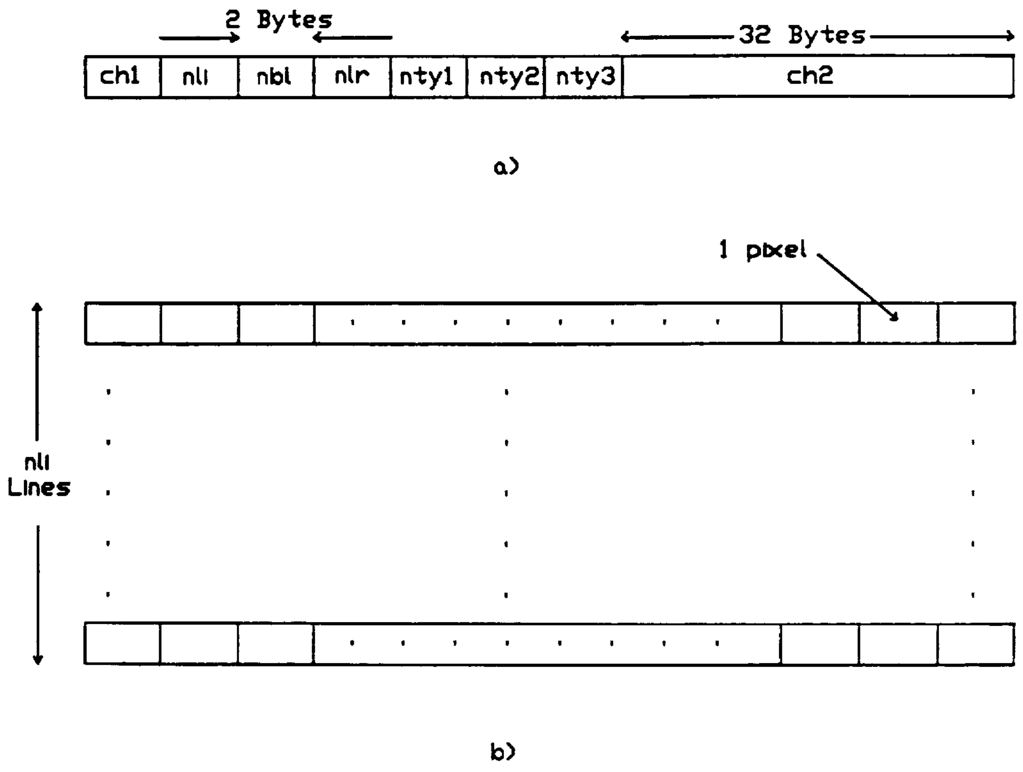


Figure 6. Image File Format.

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

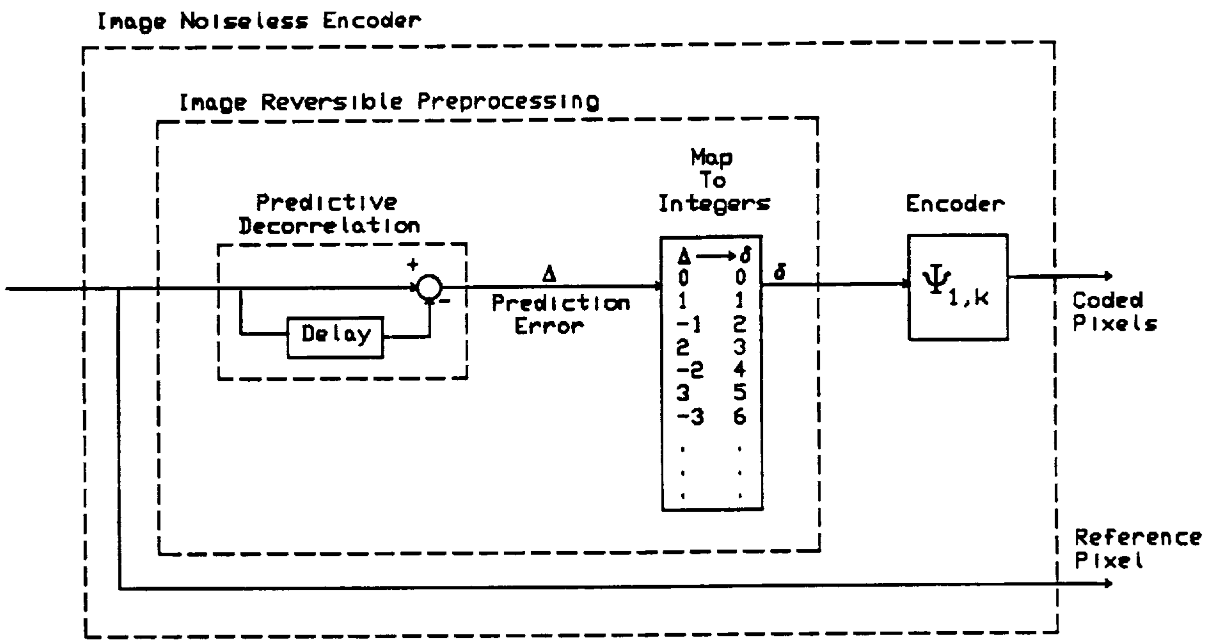


Figure 7. Block Diagram of the RICE Simulator.

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) $\leq \Delta \leq 0$	$2\|\Delta\|-1$
(previous pixel - maximum) $> \Delta > 0$	maximum - pixel value

Table 3. $\Delta \rightarrow \delta$ 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

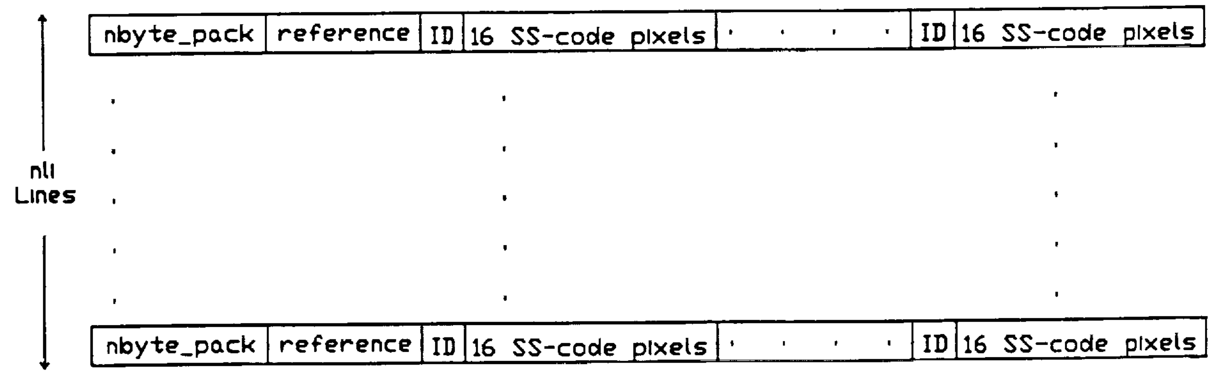


Figure 8. Compressed Image File Format.

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.

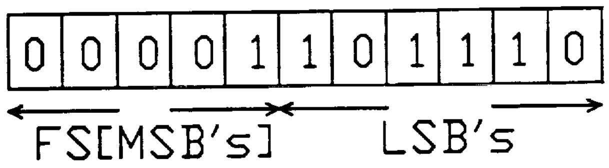


Figure 9. Typical Split-Sample Encoded Pixel.

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.