NATO STANDARD

AComP-4724

VLF/LF MSK MULTI CHANNEL BROADCAST

Edition B Version 1

SEPTEMBER 2021

NORTH ATLANTIC TREATY ORGANIZATION

ALLIED COMMUNICATION PUBLICATION

Published by the
NATO STANDARDIZATION OFFICE (NSO)

INTENTIONALLY BLANK

NORTH ATLANTIC TREATY ORGANIZATION (NATO)

NATO STANDARDIZATION OFFICE (NSO)

NATO LETTER OF PROMULGATION

20 September 2021

1. The enclosed Allied Communication Engineering Publication AComP-4724, Edition B, Version 1, VLF/LF MSK MULTI CHANNEL BROADCAST, which has been approved by the nations in the Consultation, Command, and Control Board (C3B), is promulgated herewith. The agreement of nations to use this publication is recorded in STANAG 4724.

2. AComP-4724, Edition B, Version 1, is effective upon receipt and supersedes AComP-4724, Edition A, Version 1, which shall be destroyed in accordance with the local procedure for the destruction of documents.

3. This NATO standardization document is issued by NATO. In case of reproduction, NATO is to be acknowledged. NATO does not charge any fee for its standardization documents at any stage, which are not intended to be sold. They can be retrieved from the NATO Standardization Document Database ((https://nso.nato.int/nso/) or through your national standardization authorities.

4. This publication shall be handled in accordance with C-M(2002)60.

Dimitrios SIGOULAKIS
Major General, GRC (A)
Director, NATO Standardization Office

INTENTIONALLY BLANK

RESERVED FOR NATIONAL LETTER OF PROMULGATION

INTENTIONALLY BLANK

RECORD OF RESERVATIONS

CHAPTER	RECORD OF RESERVATION BY NATIONS
Note: The reservations listed on this page include only those that were recorded at time of promulgation and may not be complete. Refer to the NATO Standardization Document Database for the complete list of existing reservations.	

INTENTIONALLY BLANK

RECORD OF SPECIFIC RESERVATIONS

[nation]	[detail of reservation]
FRA	As far as the modes N9 and N10 are concerned, the requirement in terms of performance and costs has yet to be confirmed.
POL	The document will be implemented in specific combat platforms during their medernisation and in newly produced platforms.
TUR	The infrastructure of modernized VLF Bafa Station will support STANAG 4724. STANAG 4724 is planned to beimplemented in the manner of ensuring satisfactory technologic development.

Note: The reservations listed on this page include only those that were recorded at time of promulgation and may not be complete. Refer to the NATO Standardization Document Database for the complete list of existing reservations.

INTENTIONALLY BLANK

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1. SCOPE

Communications with submerged submarines is essential for safety reasons and to ensure Command and Control of those submarines. Once submarines are submerged, and no communications means are deployed through floating buoys (SATCOM LOS/BLOS Radios), submarines depend on VLF and/or LF communications to enable communications. Standardization of the VLF/LF Communications was established through the development of STANAG 5030. The operational user community revisited the requirements for the VLF/LF Communications and requested that the STANAG be modified to provide more flexibility and range whilst maintaining the capability to increase data rates. In addition the updated or new STANAG should also support non-VALLOR cryptographic algorithms as the VALLOR algorithm will be phased out.

1.2. AIM

This Standard expands the North Atlantic Treaty Organization (NATO) VLF/LF capability from single channel operations with frequency-shift-keying (FSK) and a multichannel capability which also includes the use of channels of a four channel VLF/LF National transmission to incorporate a co-existent range/rate extension mode with options for increased data rate and/or reduced transmitter radiated power.

This Edition of the Standard re-introduces support for non-VALLOR cryptographic systems into the NATO VLF/LF broadcast. A technical approach for future transition to total compatibility with the multichannel requirement and non-VALLOR crypto systems has been found.

To allow for interoperability with existing VLF systems based on STANAG 5030, this standard should is backwards interoperable with STANAG 5030 based systems, albeit on a limited basis.

1.3. GENERAL

This Standard contains sufficient detail to the extent that Nations can undertake development or procurement of receiver systems and that transmitter and headquarters conversions can be engineered.

This standard introduces performance enhancing Rate/Range Enhancement Mode (REM) capability, based on lossless text compression and Forward Error Correction coding of broadcast data streams.

The technical approach in this document is compatible with ITA No. 2 coding. ITA5 Conversion to 10.0 ITA No. 5 coding in accordance with STANAG 5036 will be considered at a future date. System performance, data throughput and signal processing requirements will be impacted by a change in the coding.

This standard specifies the minimum technical requirements that must be complied with to assure compatibility among components of a broadcast network for relay of messages over the frequency range of 14.0 through 60.00 kHz. The network includes all communication links from the point at which the messages are injected into the network through delivery of the message to the addressee. It also includes the administrative management of the network.

1.3.1. NATO VLF/LF TRANSMITTERS

This standard provides for the operation of the North Atlantic Treaty Organization (NATO) VLF/LF transmitters in the following modes:

a. Single channel with frequency-shift-keying modulation;

b. Two channel with minimum-shift-keying modulation;

c. Four channel with minimum-shift-keying modulation;

d. Flexible multi-channel Rate/Range Enhanced Modes (REM) with minimum-shift-keying modulation.

1.3.2. SHARED CHANNEL OF A NATIONAL FOUR CHANNEL BROADCAST

This standard describes technical characteristics for the utilisation of one or more channels of a National four channel VLF/LF multichannel broadcast for transmission of NATO broadcasts.

1.3.3. SHARED USE OF NATO MULTICHANNEL TRANSMISSION FACILITY

This standard describes technical characteristics for the utilisation of NATO multichannel VLF/LF transmission facilities for transmission of National broadcasts.

1.4. RELATED DOCUMENTS

STANAG 5030 - VLF/LF MSK MULTI CHANNEL BROADCAST.

1.5. DEFINITIONS

ALTERNATE-LEF:

Link Encryption Family; not based upon the TSEC/KW-46 broadcast security equipment, including any suitable NATO or National non-VALLOR link encryption device.

BROADCAST:

A method of transmitting messages on pre-determined schedules wherein no acknowledgment for the message is required.

CHANNEL:

A single signal path, or one which is independent of others sharing a common frequency; e.g., one channel of a Time Division Multiplexed System.

FILLER:

Data transmitted for the purpose of keeping a communications channel active when there are no messages available.

FIBONACCI BITS:

Deterministic unencrypted bits sent in the VALLOR cryptographic system which are the basis for frame synchronization of broadcast data.

MARK:

A condition of the modulated signal corresponding to a binary 1 (condition Z).

NATO MULTICHANNEL:

A broadcast of two or four time division multiplexed channels transmitted over any NATO VLF/LF facility.

ORDERWIRE:

A channel over which information pertaining to signal quality and technical control information is passed.

REM:

Rate/Range Enhancement Mode capability increasing performance by lossless text compression and Forward Error Correction coding of broadcast data streams

SHARED NATO TRANSMISSION FACILITIES:

Use of one or more channels of a NATO multichannel VLF/LF transmission for national broadcast.

SHARED NATIONAL TRANSMISSION FACILITIES:

Use of one or more channels of a national multichannel VLF/LF transmission for NATO broadcast.

SPACE:

A condition of the modulated signal corresponding to a binary 0 (Condition A).

VALLOR:

A cryptographic system based on the TSEC/KW-46 broadcast security equipment.

V_d:

Ratio of RMS to mean absolute value of atmospheric noise in dB.

VLF/LF:

As used herein it is a band of RF frequencies covering from 14kHz to 60kHz.

WAGNER CODE:

An error coding scheme that utilises a single parity bit per block of N information bits and a "soft decision" demodulation process. If a received codeword does not pass the parity check, the least reliable bit is inverted. The Wagner code is designated by an (N+1,N) prefix.

CHAPTER 2 REQUIREMENTS

2.1. GENERAL REQUIREMENTS

2.1.1. NETWORK DEFINITION

The network block diagram is shown in Figure 1. The network shall be defined to include the following:

BROADCAST COORDINATION AUTHORITY (BCOA):
Generally responsible for the management of the network to insure that all components of the network are coordinated such as to be interoperable.

BROADCAST CONTROL STATION (BCS):
This is the point at which messages are entered into the network. If the messages are to be encrypted they are encrypted at these sites. Data generation beyond operational messages may be required to meet traffic flow security requirements.

INTERSITE LINKS:
Intersite links include data transfer links and order wire communication channels. Intersite data links shall interconnect the BCS and the VLF/LF Transmitter Complex for transfer of data to be relayed over the VLF/LF radio frequency. Order wire links shall be those communications channels used for management and coordination of network modes.

VLF/LF TRANSMITTER COMPLEX:
The facility where data for the final relay link (VLF/LF link) is processed and transmitted at the assigned VLF/LF frequency.

RECEIVE PLATFORM:
The platform where the injected message is demodulated, decrypted and otherwise processed to deliver the message to the addressee.

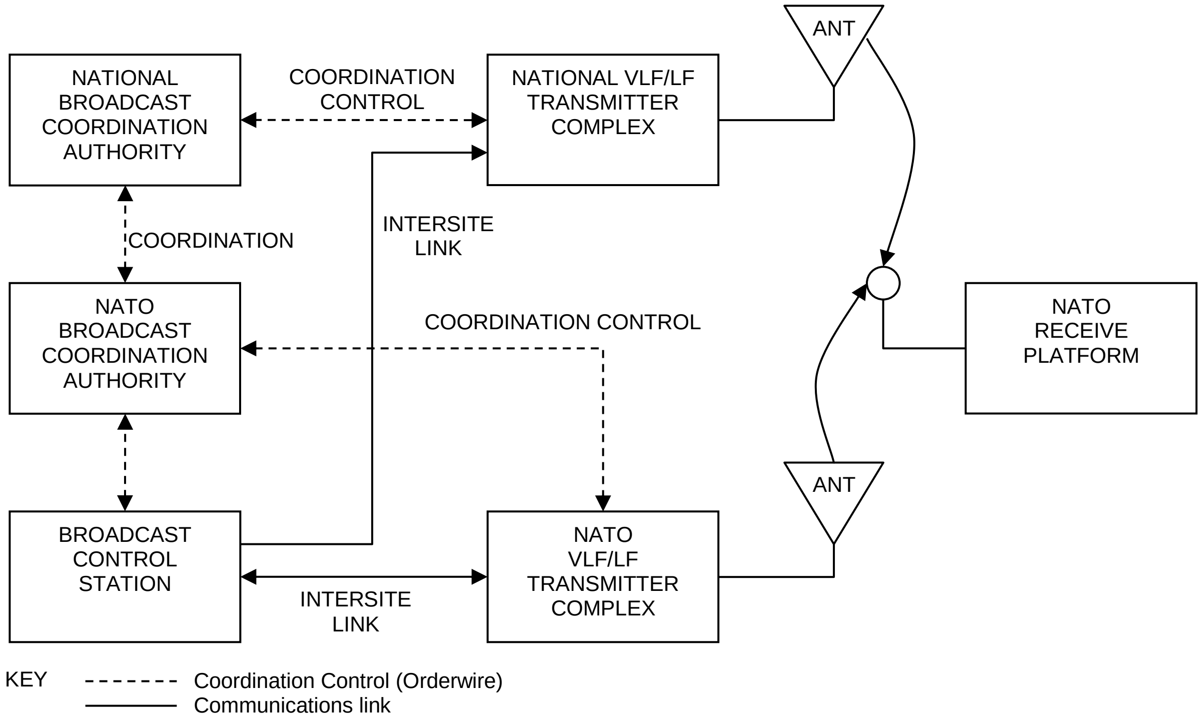

ANT
NATIONAL BROADCAST COORDINATION AUTHORITY
COORDINATION CONTROL
NATIONAL VLF/LF TRANSMITTER COMPLEX
INTERSITE LINK
COORDINATION
NATO BROADCAST COORDINATION AUTHORITY
COORDINATION CONTROL
NATO RECEIVE PLATFORM
ANT
BROADCAST CONTROL STATION
INTERSITE LINK
NATO VLF/LF TRANSMITTER COMPLEX
KEY
Coordination Control (Orderwire)
Communications link

FIGURE 1: NATO VLF/LF NETWORK DIAGRAM.

2.1.1.1. GENERAL DESCRIPTION

The basic requirement of the network is to relay messages from the BCS to the Receiver Platform. A block diagram of the NATO multichannel VLF/LF network is shown in **Figure 2**. Note that the BCS sites are remote from the final link in the network which is a transmission complex for VLF/LF. **Figure 2** shows the addition of RED REM coding processes, but does not show the full RED/BLACK REM capability. This REM functionality is fully detailed in CHAPTER 3.

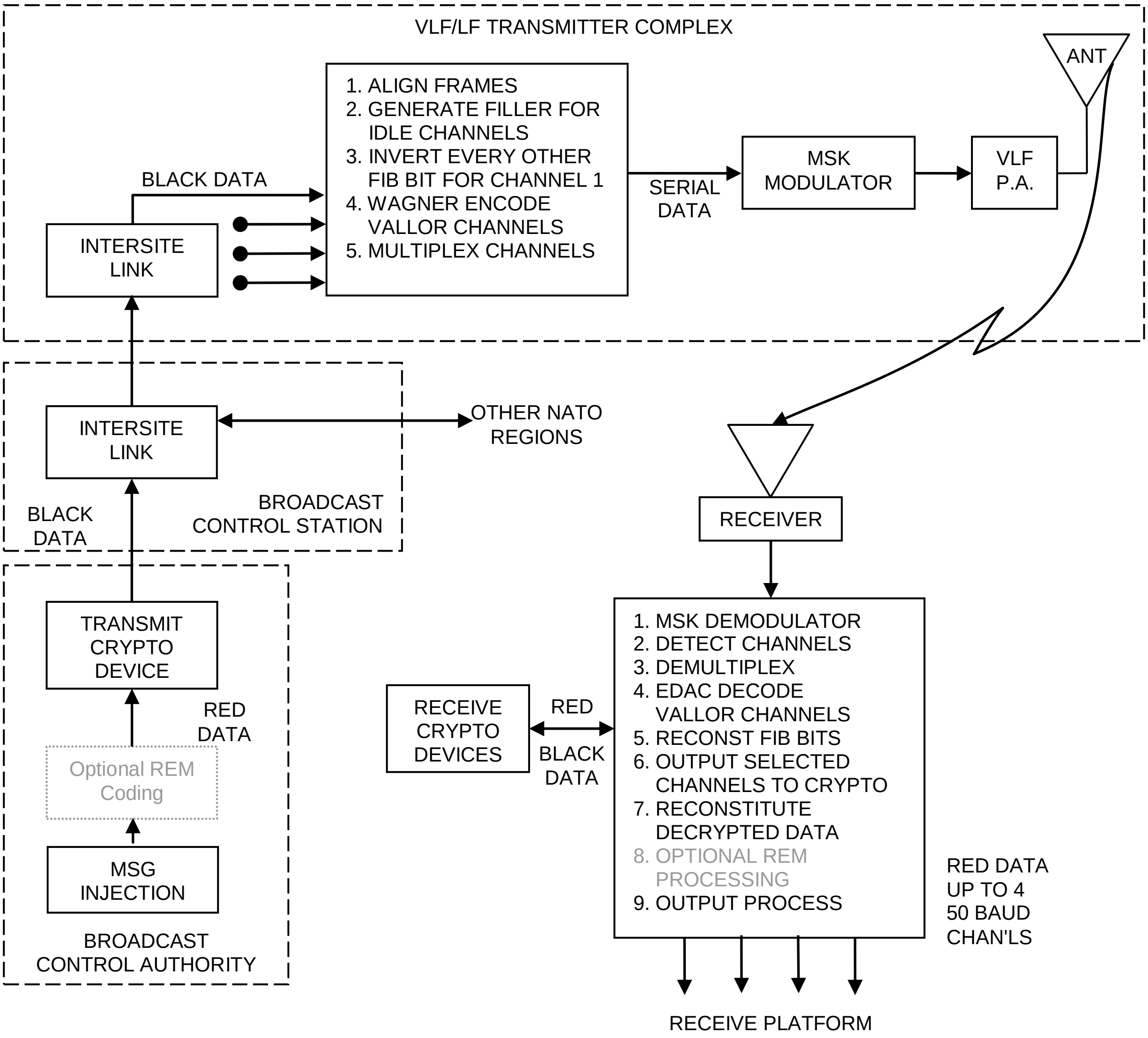

VLF/LF TRANSMITTER COMPLEX
1. ALIGN FRAMES
2. GENERATE FILLER FOR IDLE CHANNELS
3. INVERT EVERY OTHER FIB BIT FOR CHANNEL 1
4. WAGNER ENCODE VALLOR CHANNELS
5. MULTIPLEX CHANNELS
ANT
BLACK DATA
SERIAL DATA
MSK MODULATOR
VLF P.A.
INTERSITE LINK
INTERSITE LINK
OTHER NATO REGIONS
BLACK DATA
BROADCAST CONTROL STATION
RECEIVER
TRANSMIT CRYPTO DEVICE
RED DATA
Optional REM Coding
MSG INJECTION
BROADCAST CONTROL AUTHORITY
RECEIVE CRYPTO DEVICES
RED
BLACK DATA
1. MSK DEMODULATOR
2. DETECT CHANNELS
3. DEMULTIPLEX
4. EDAC DECODE VALLOR CHANNELS
5. RECONST FIB BITS
6. OUTPUT SELECTED CHANNELS TO CRYPTO
7. RECONSTITUTE DECRYPTED DATA
8. OPTIONAL REM PROCESSING
9. OUTPUT PROCESS
RED DATA UP TO 4 50 BAUD CHAN'LS
RECEIVE PLATFORM

FIGURE 2: NATO MULTICHANNEL VLF/LF SYSTEM.

The VLF/LF transmission facility shall be capable of operating in any of the following modes when specified in operational doctrine:

MODE	TDM	BAUD	KEYING	COMSEC
N1	Not required			
N2	1	50	FSK	VALLOR/ALTERNATE-LEF
N3	2	100	MSK	VALLOR/ALTERNATE-LEF
N4	4	200	MSK	VALLOR/ALTERNATE-LEF
N5	1 of 4	200	MSK	VALLOR in designated NATO channel/national COMSEC on remaining channels
N6	1 of 2	100	MSK	VALLOR in designated NATO channel/national COMSEC on remaining channel
N7	4	200	MSK	VALLOR/ALTERNATE-LEF One or more channels containing REM
N8	2	100	MSK	VALLOR/ALTERNATE-LEF One or more channels containing REM
N9	4	200	MSK	ALTERNATE-LEF REM channels
N10	2	100	MSK	ALTERNATE-LEF REM channels

NOTE 1: In mode N2 the VALLOR channel shall not invert the Fibonacci bit and shall not contain WAGNER parity bits.

NOTE 2: REM modes N7, N8, N9 and N10 are defined in CHAPTER 3.

TABLE 1: VLF/LF TRANSMISSION MODES.

2.1.1.1.1. ON-OFF-KEYED CARRIER SINGLE CHANNEL

The single channel mode using on-off-keyed carrier modulation with data encoded in international Morse code is no longer a requirement of this standard.

2.1.1.1.2. FREQUENCY-SHIFT-KEYING SINGLE CHANNEL

There shall be a single channel mode using Frequency-shift-keying (FSK) modulation with 7.0-unit START-STOP International Telegraph Alphabet (ITA) No.2 coded characters at 50 baud. The system shall operate with crypto covered text from either the ALTERNATE-LEF or VALLOR crypto systems. Advance agreement between the cooperating BCS and Receive Platform shall be required as to which crypto system is to be used. In mode N2, the VALLOR channel shall not invert the Fibonacci bit and shall not contain WAGNER parity bits.

2.1.1.1.3. NATO MULTICHANNEL

The NATO VLF/LF network shall operate in Minimum-Shift-Keying (MSK) multichannel modes which shall be of two forms:

a. Two channel operation;

b. Four channel operation.

Each channel shall operate at 50 baud. Each channel carrying data shall be covered with a separate cryptographic unit, using unique crypto keying material. Security among channels shall be maintained from the BCS through to the Receiver Platforms which are subscribers to a particular channel of the VLF/LF transmission. A serial data stream comprised of the channels shall be formed by time division multiplexing at the VLF/LF Transmitter Complex. This stream shall be 100 baud for two channels and 200 baud for four channels. MSK modulation shall be used. Filler data shall be placed on channels for which there are no operational messages to be relayed.

For modes N3, N4, N5 and N6, channel 1 shall be operated as a VALLOR covered channel; Channels 2, 3 and 4 shall operate with VALLOR or ALTERNATE-LEF covered data mixed in any combination. Advanced agreement shall be required between the Broadcast Control Station and the subscriber Receive Platform as to the channel on which the messages are to be placed and the type (VALLOR or ALTERNATE-LEF) of crypto equipment which is to be used. Coding of information with the ALTERNATE-LEF shall be 7.0 unit START-STOP ITA

No.2. Coding of information with the VALLOR shall be 7.0 unit 64-ary ITA No. 2. Inverted FIBONACCI bits, of a VALLOR covered channel are necessary to identify the reference channel of a STANAG 4724 multi-channel MSK transmission. They must only appear in channel one. Other channels must not contain sequences which resemble inverted FIBONACCI bit sequences.

NATO modes N7, N8, N9 and N10 shall operate as defined in CHAPTER 3.

2.1.1.1.4. SHARED NATO TRANSMISSION

The network shall provide for the use of a NATO multichannel facility for the transmission of National broadcasts. The National broadcast shall comply with the transmission characteristics specified herein for a NATO transmission.

2.1.2. CHARACTERISTICS

2.1.2.1. INTERSITE LINKS

This standard introduces no new requirements or functionality to the intersite links, allowing any mode to be implemented within the performance requirements of previous editions.

2.1.2.1.1. DATA LINKS

Intersite links shall be provided between each BCS and the VLF/LF Transmitter Complex. Intersite link communications shall include a primary data channel for transmission of the data for relay over the VLF/LF transmitter and a secondary data channel to replace the primary channel in event of circuit outage. The secondary channel may be dedicated or switched service. The general performance requirements shall be as shown in Table 2 assuming an errorless input at the message injection site.

VLF/LF Mode	Characteristics
FSK	Shall exhibit a character error rate $\leq 10^{-5}$ for a 50 baud data channel
MSK	Shall be as for FSK or for a voice frequency channel or higher data rate channel with error detection and correction to meet the character error rate of $\leq 10^{-5}$

TABLE 2: INTERSITE LINK GENERAL CHARACTERISTICS.

2.1.2.1.2. ORDER WIRE

An order wire shall be provided to connect the Broadcast Co-ordination Authority, the BCS and the VLF/LF Transmitter Complex The order wire may be voice or telegraphic and may be either dedicated or switched (dial up).

2.1.2.2. VLF/LF TRANSMITTER

2.1.2.2.1. RADIO CARRIER FREQUENCY

The VLF/LF Transmitter shall be operated on a radio carrier frequency within the VLF/LF band in 10 Hz increments; e.g. shall be operated only on frequencies compatible with a receiver tuneable in 10 Hz steps.

2.1.2.2.2. FREQUENCY TOLERANCE

For FSK and MSK modes of operation the radio frequency carrier shall have a stability of $\leq 10^{-8}$ and an absolute accuracy of $\leq 10^{-7}$. Stability shall be measured over a one day period and accuracy shall be measured over a one month period.

2.1.2.2.3. PHASE

The following sub-paragraphs shall be applicable to FSK and MSK modes of operation.

2.1.2.2.3.1. PHASE STABILITY

The short term stability of all radio frequencies shall require the phase jitter be not greater than ± 1.0 degree in a 50 Hz bandwidth when averaged over one hundred 20 ms sample periods.

2.1.2.2.3.2. PHASE LINEARITY

Deviation in phase response in the complete VLF/LF transmitter complex (including the antenna) shall not exceed 13^0 from the linear over the radio frequency band about the selected instantaneous transmission frequency as given in the following list:

Condition	Radio Frequency Bandwidth
4-channel MSK	120 Hz
2-channel MSK	60 Hz
FSK	35 Hz

2.1.2.2.4. BANDWIDTH

The minimum transmission bandwidth provided at the 3db points shall be 60 Hz for the FSK or the 2 channel MSK modes and 120 Hz for the 4 channel MSK mode.

2.1.2.3. RECEIVER

The receiver shall process signals in the modes N2, N3, N4, N5, N6, N7 and N8 as specified in paragraph 2.1.1.1. and additionally may process signals in the modes N9 and N10 (as defined in CHAPTER 3). (No additional receiver functionality is required to process modes N7 and N8; additional BLACK REM processing is required in the receiver for modes N9 and N10 as described in CHAPTER 3).

2.1.2.3.1. FREQUENCY SELECTION

The carrier frequency of the receiver shall be tuneable to any 10 Hz step in the VLF/LF range.

2.1.2.3.2. MESSAGE FIDELITY

The minimum signal-to-noise ratio performance of the receiver system for NATO modes N2, N3, N4, N5 and N6 shall be as specified in TABLE 3 Performance requirements for REM operation (modes N7, N8, N9 and N10) are defined in CHAPTER 4.

MODE		Required Signal-to-noise ratio (dB)*
FSK	Non-coherent	+2.0
	Coherent	-4.0
MSK	2 Channel	-12.7
	4 Channel	-9.0
	Shared Transmissions	-9.0
*Character error rate = 10^{-3} for FSK and MSK, SNR measured in 1000 Hz bandwidth. Noise is atmospheric with V_d= 10 dB measured in 1000 Hz bandwidth		

TABLE 3: RECEIVER SYSTEM PERFORMANCE IN ATMOSPHERIC NOISE.

2.2. DETAILED REQUIREMENTS

2.2.1. ON-OFF-KEYED CARRIER MODE

On-off-keyed carrier mode is no longer a requirement of this standard.

2.2.2. FSK SINGLE CHANNEL

There shall be a single channel mode which uses FSK modulation of the VLF/LF radio frequency carrier. The channel shall operate at 50 baud.

2.2.2.1. FUNCTIONAL ALLOCATION

2.2.2.1.1. BROADCAST COORDINATION AUTHORITY (BCOA)

The BCOA shall ensure that the Broadcast Control Station, VLF/LF Transmitter Complex and Receive Platform are advised when FSK single channel mode is to be used and whether the VALLOR or ALTERNATE-LEF crypto equipment is to be used.

2.2.2.1.2. BROADCAST CONTROL STATION

The BCS shall perform the following functions:

a. Inject encrypted messages into the network.

b. Provide for an interface with an intersite link.

2.2.2.1.3. VLF/LF TRANSMITTER COMPLEX

The VLF/LF Transmitter Complex shall perform the following functions:

a. Interface and accept data from an intersite link;

b. Modulate a VLF/LF carrier with the FSK signal;

c. Amplify and transmit the modulated radio frequency.

2.2.2.1.4. RECEIVE PLATFORM

The Receive Platform shall perform the following functions:

a. Receive the VLF/LF signal on a suitable antenna subsystem;

b. Provide a receive terminal capable of receiving and demodulating the VLF/LF signal;

c. Output the demodulated data at 50 baud to either a VALLOR or ALTERNATE-LEF crypto equipment as designated by the Broadcast Coordinating Authority;

d. Process the decrypted data and output the data to a printer or other peripheral unit.

2.2.2.2. CHARACTERISTICS

2.2.2.2.1. DATA FORMAT

Characters in the message shall be coded in 7.0-Unit START-STOP ITA No. 2 code as shown in **TABLE 4**.

Bit numbers 7 6 5 4 3 2 1	Letters Case	Figures Case
1 0 0 0 0 0 0	No Action	No Action
1 0 0 0 0 1 0	E	3
1 0 0 0 1 0 0	Line Feed	Line Feed
1 0 0 0 1 1 0	A	-
1 0 0 1 0 0 0	Space	Space
1 0 0 1 0 1 0	S	(Apos)'
1 0 0 1 1 0 0	I	8
1 0 0 1 1 1 0	U	7
1 0 1 0 0 0 0	Car. Ret	Car. Ret
1 0 1 0 0 1 0	D	WRU
1 0 1 0 1 0 0	R	4
1 0 1 0 1 1 0	J	Aud Sig
1 0 1 1 0 0 0	N	(Comma),
1 0 1 1 0 1 0	F	Unassigned
1 0 1 1 1 0 0	C	:
1 0 1 1 1 1 0	K	(
1 1 0 0 0 0 0	T	5
1 1 0 0 0 1 0	Z	+
1 1 0 0 1 0 0	L	)
1 1 0 0 1 1 0	W	2
1 1 0 1 0 0 0	H	Unassigned
1 1 0 1 0 1 0	Y	6
1 1 0 1 1 0 0	P	0
1 1 0 1 1 1 0	Q	1
1 1 1 0 0 0 0	O	9
1 1 1 0 0 1 0	B	?
1 1 1 0 1 0 0	G	Unassigned
1 1 1 0 1 1 0	Figures	Figures
1 1 1 1 0 0 0	M	.
1 1 1 1 0 1 0	X	/
1 1 1 1 1 0 0	V	;
1 1 1 1 1 1 0	Letters	Letters
Notes: 1. Transmission order is Bit 1 - Bit 7 2. Bit 1 is a START bit and shall be a 0 (space) 3. Bit 7 is a STOP bit and shall be a 1 (Mark)		

TABLE 4: 7.0 UNIT START-STOP ITA NO.

2.2.2.2.2. SECURITY PROVISIONS

Message security shall be provided by on-line encryption of messages at the Broadcast Control Station (BCS) and on-line decryption at the Receive Platform. Security for National originators shall be achieved through distribution control procedures of the on-line crypto keying materials or through prior off-line encryption.

2.2.2.2.2.1. CRYPTOGRAPHIC EQUIPMENT

The FSK single channel mode shall be compatible with both the VALLOR and ALTERNATE-LEF cryptographic equipment. The VALLOR shall be operated in the 6.0 Stepped Digital mode. The ALTERNATE-LEF shall be operated in the clock start mode where applicable.

2.2.2.2.2.2. MESSAGE FLOW SECURITY

Message flow security shall be provided by maintaining a continuous cryptographic covered VLF/LF transmission. It shall be a broadcast management function of the Broadcast

Coordination Authority to place, or cause to be placed from a BCS, symbols encrypted with either VALLOR or ALTERNATE-LEF at all times.

2.2.2.2.3. INTERSITE LINKS

Intersite links shall be in accordance with paragraph 2.1.2.1.

2.2.2.2.4. VLF/LF TRANSMITTER COMPLEX PROCESSING

2.2.2.2.4.1. INTERSITE LINK PHASE DRIFT

The intersite link shall be provided with not less than a ± 8 bit buffer which shall compensate for the phase difference which can be accumulated between the crypto equipment and the transmitter standard during a 24 hour period. The input data shall be retimed at the transmitter complex to cause the transmitted signal to comply with this standard (see 2.1.2.2.). The data shall otherwise be transmitted as received over the intersite link.

2.2.2.2.4.2. FREQUENCY-SHIFT-KEYING MODULATION

Frequency-shift-keying modulation shall use two sinusoids of different frequencies to distinguish between a MARK and a SPACE. The frequency shifts shall be phase continuous and be ± 25 Hertz centred about the carrier frequency. The upper frequency shift shall represent a SPACE.

2.2.2.2.4.3. TRANSMITTER SUBSYSTEM BANDWIDTH

The VLF/LF transmitter subsystem bandwidth shall be in accordance with paragraph 2.1.2.2.4.

2.2.2.2.4.4. RADIO FREQUENCY

The carrier frequency characteristic of the transmitted signal shall be in accordance with paragraph 2.1.2.2.1.

2.2.2.2.4.5. PHASE

The phase stability and linearity of all transmitted radio frequencies shall be in accordance with paragraph 2.1.2.2.3.

2.2.2.2.4.6. RECEIVE SYSTEM

The receiver system shall demodulate the FSK modulated radio frequency and output a signal whose characteristics are compatible with the VALLOR and ALTERNATE-LEF crypto equipment. The receiver shall be in accordance with paragraph 2.1.2.3.

2.2.3. NATO MULTICHANNEL MODES N3, N4, N5 AND N6

There shall be two NATO multichannel modes; a two channel and a four channel mode. Each channel shall operate at 50 baud. The two channel mode shall operate at a VLF/LF transmission rate of 100 baud and the four channel mode shall operate at 200 baud. MSK shall be used on the VLF/LF transmission.

2.2.3.1. NATO MULTICHANNEL NETWORK DIAGRAM

The network operating in the NATO multichannel modes shall be as shown in **Figure 2**.

2.2.3.2. FUNCTIONAL ALLOCATION

2.2.3.2.1. BROADCAST COORDINATION AUTHORITY (BCOA)

The BCOA shall perform the following functions:

a. Control the allocation of channels to the Broadcast Control Stations;

b. Control the assignment of mode characteristics to assure interoperability among the network nodes.

c. Provide for encrypted filler to be transmitted over all channels which are not allocated to a Broadcast Control station for operations;

d. Present the clear text message to the assigned crypto system;

e. Encrypt the message using the VALLOR or ALTERNATE-LEF crypto system as assigned;

f. Provide encrypted filler when no messages are available for injection or when tasked by the Broadcast Coordination Authority.

2.2.3.2.2. BROADCAST CONTROL STATIONS (BCS)

The BCS shall perform the following functions:

a. Provide for an interface with an intersite link.

2.2.3.2.3. VLF/LF TRANSMITTER COMPLEX

The VLF/LF Transmitter Complex shall perform the following functions:

a. Interface with and accept data from the intersite links;

b. Place the data received from the BCS's on the VLF/LF transmitter in accordance with the channel allocation specified by the VLF/LF Broadcast Coordination Authority;

c. Encode VALLOR encrypted data into a (13, 12) Wagner code;

d. Maintained unchanged the signal structure of ALTERNATE-LEF encrypted data received over the intersite link;

e. Multiplex the channels into a serial stream;

f. Operate on (invert sense of) the Fibonacci bits on the data to be transmitted over channel one;

g. Generate an MSK signal;

h. Modulate a VLF/LF carrier with the MSK signal and amplify and transmit this modulated radio frequency;

i. Generate a signal which is in accordance with paragraph 2.2.3.3.5.3 and place this clear text signal on the appropriate channel whenever there is no signal for that channel available from a Broadcast Control Station.

2.2.3.2.4. RECEIVE PLATFORM

The Receive Platform shall perform the following functions:

a. Receive the VLF/LF signal on a suitable antenna subsystem;

b. Provide a receive terminal capable of receiving, demodulating one channel, two and four channel MSK transmissions, in addition, demultiplexing two and four channel transmissions;

c. Automatically identify channels and output these channels corresponding to operator request;

d. Perform error correction on the VALLOR encrypted, WAGNER encoded, channels;

e. Reconstitute the Wagner parity bits to Fibonacci bits prior to outputting the signal to the VALLOR equipment for decryption;

f. Pass ALTERNATE/LEF encrypted channels for decryption;

g. Process the decrypted data and output the data to a printer or other peripheral unit.

2.2.3.3. CHARACTERISTICS

2.2.3.3.1. DATA FORMATS

Characters in the message shall be encoded into the 7.0 unit 64-ary ITA No 2 code as shown in **TABLE 5**.

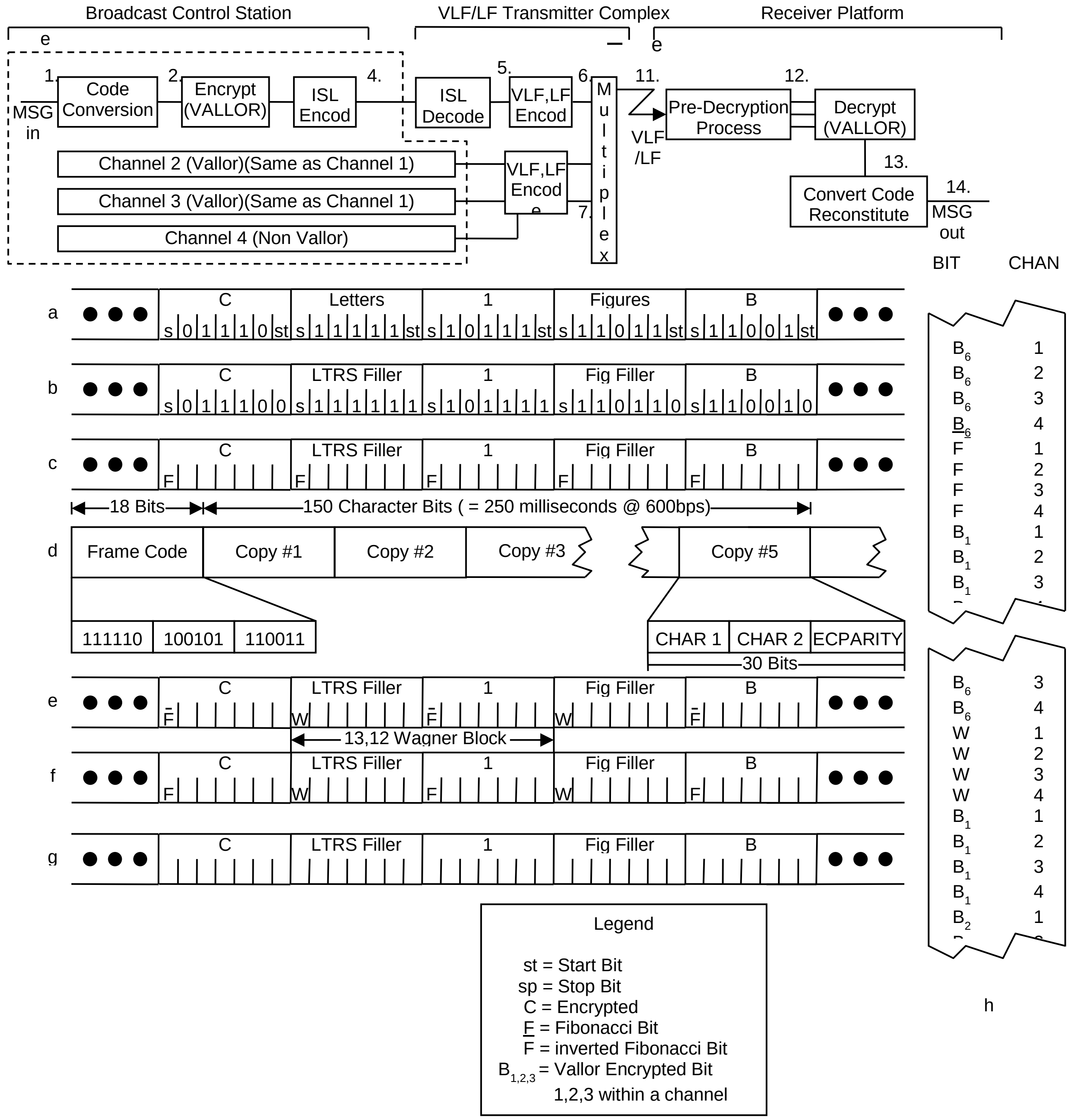
Bit numbers	Letters Case	Bit numbers	Figures Case
7 6 5 4 3 2 1		**7 6 5 4 3 2 1**	
1 0 0 0 0 0 0	No Action	1 0 0 0 0 0 1	No Action
1 0 0 0 0 1 0	E	1 0 0 0 0 1 1	3
1 0 0 0 1 0 0	Line Feed	1 0 0 0 1 0 1	Line Feed
1 0 0 0 1 1 0	A	1 0 0 0 1 1 1	-
1 0 0 1 0 0 0	Space	1 0 0 1 0 0 1	Space
1 0 0 1 0 1 0	S	1 0 0 1 0 1 1	(Apos)'
1 0 0 1 1 0 0	I	1 0 0 1 1 0 1	8
1 0 0 1 1 1 0	U	1 0 0 1 1 1 1	7
1 0 1 0 0 0 0	Car. Ret	1 0 1 0 0 0 1	Car. Ret
1 0 1 0 0 1 0	D	1 0 1 0 0 1 1	WRU
1 0 1 0 1 0 0	R	1 0 1 0 1 0 1	4
1 0 1 0 1 1 0	J	1 0 1 0 1 1 1	Aud Sig
1 0 1 1 0 0 0	N	1 0 1 1 0 0 1	(Comma),
1 0 1 1 0 1 0	F	1 0 1 1 0 1 1	Unassigned
1 0 1 1 1 0 0	C	1 0 1 1 1 0 1	:
1 0 1 1 1 1 0	K	1 0 1 1 1 1 1	(
1 1 0 0 0 0 0	T	1 1 0 0 0 0 1	5
1 1 0 0 0 1 0	Z	1 1 0 0 0 1 1	+
1 1 0 0 1 0 0	L	1 1 0 0 1 0 1	)
1 1 0 0 1 1 0	W	1 1 0 0 1 1 1	2
1 1 0 1 0 0 0	H	1 1 0 1 0 0 1	Unassigned
1 1 0 1 0 1 0	Y	1 1 0 1 0 1 1	6
1 1 0 1 1 0 0	P	1 1 0 1 1 0 1	0
1 1 0 1 1 1 0	Q	1 1 0 1 1 1 1	1
1 1 1 0 0 0 0	O	1 1 1 0 0 0 1	9
1 1 1 0 0 1 0	B	1 1 1 0 0 1 1	?
1 1 1 0 1 0 0	G	1 1 1 0 1 0 1	Unassigned
1 1 1 0 1 1 0	Figures Filler	1 1 1 0 1 1 1	Unassigned
1 1 1 1 0 0 0	M	1 1 1 1 0 0 1	.
1 1 1 1 0 1 0	X	1 1 1 1 0 1 1	/
1 1 1 1 1 0 0	V	1 1 1 1 1 0 1	;
1 1 1 1 1 1 0	Unassigned	1 1 1 1 1 1 1	Letters Filler

Notes:	1. Transmission order is Bit 1 - Bit 7. 2. Bit 7 is a STOP bit and shall be a 1 (Mark). 3. Bit 1 is a START bit and shall be a 0 (space) for the 64-ary letters alphabet and shall be a 1 (Mark) for the 64-ary figures alphabet. 4. The "Figures Filler" and "Letters Filler" characters correspond to the open "Figure Shift" and "Letters Shift" characters of the 7.0 unit 32-ary ITA no. 2 alphabet, respectively.

TABLE 5: 7.0 UNIT 64-ARY ITA NO. 2.

2.2.3.3.2. INTERSITE LINK MESSAGE DATA FORMATS

There are a number of methods to obtain the general performance requirements of paragraph 2.1.2.1. to give for MSK Channels an intersite CER of 10^{-5}. An example of one method is given in **FIGURE 3** showing the Format of Message Data at Network nodal points.



1. The coding of the input message may be in any code but shall be restricted to the symbols available in ITA No. 2. FIGURE 3 a shows the form assuming that a 7.0 unit START-STOP ITA No. 2 code is used (see TABLE 4).
2. The coding of message input into the VALLOR equipment shall be in the 7.0 unit 64-ary ITA no. 2 alphabet (see TABLE 5). The Figures Filler and Letters Filler characters correspond to the 7.0 unit 32-ary ITA no. 2 Figures Shift and Letters Shift characters, respectively. An example of the 64-ary coding with filler characters is shown in FIGURE 3 b.
3. Encryption by the VALLOR equipment operating in the 6.0 Stepped Digital mode will result in bits 1 through 6 being encrypted and bit 7 (STOP) being substituted with an unencrypted and deterministic Fibonacci bit. This form, with filler characters present, is shown in FIGURE 3 c.
4. An example of coding over the intersite link is shown in FIGURE.d. Other schemes which provides a CER equal or better than 10 minus 5 may be used. The requirements of 2.2.3.3.4. apply.
5. The signal received over the intersite link at the VLF/LF Transmitter Complex shall be restored to the form output by the VALLOR equipment (see FIGURE 3 c).
6. The encoding of a channel for VLF/LF transmission shall include blocking the information into two character groups, substituting a parity bit for every second Fibonacci bit to form a (13,12) Wagner odd parity code block (odd number of 1's) over the information bits (Fibonacci bit excluded), and Inverting the sense of the remaining Fibonacci bits. The resulting form with filler characters present is shown in FIGURE 3 e.
7. The encoding of the other VALLOR covered channels (in this example channel 2 and 3) is the same as for channel 1, except that the Fibonacci bits are not inverted. The resulting form with filler characters present is shown in FIGURE 3 f.
8. The coding of messages input into the ALTERNATE-LEF equipment shall be 7.0 unit START-STOP ITA No. 2 (see TABLE 4 and FIGURE 3 a).
9. Processing by the ALTERNATE-LEF equipment will result in all bits being encrypted as shown in FIGURE 3 g.
10. The signal restored to the form output from the ALTERNATE-LEF equipment (see FIGURE 3 g) is left unchanged.
11. Bits from the channels shall be sequentially time division multiplexed in accordance with paragraph 2.2.3.3.5.4. The form and relationship, for this example, is shown in FIGURE 3 h.
12. Pre-decryption processing of the received signal includes demultiplexing (including proper channel numbering and resolution of signal sense ambiguity). For VALLOR covered channels it shall also include performing error correction (using the Wagner code) and restoring the Fibonacci bits such that the outputted signal is in the same form as that generated by the transmit VALLOR equipment as shown in FIGURE 3 c. For ALTERNATE-LEF covered channels there is no code conversion and the form is shown in FIGURE 3 g.
13. Decryption by the VALLOR equipment results in the clear text message in 7.0 unit 64-ary ITA NO. 2 code as shown in FIGURE 3 b.
14. The 7.0 unit 64-ary ITA No. 2 code shall be converted to 7.0 unit START-STOP ITA NO. 2, or other coding compatible with the output peripheral.
15. Following decryption by the ALTERNATE-LEF equipment, the messages will be in clear text 7.0 unit START-STOP ITA NO. 2.

FIGURE 3: NETWORK DATA FORMAT DESCRIPTION.

2.2.3.3.3. SECURITY PROVISIONS

Message security shall be provided on a per channel basis. This message security shall be provided by encryption of messages at the BCS and decryption at the subscriber Receiver Platform. Each channel shall utilise a separate crypto unit and each crypto unit shall have a unique crypto variable. Inter-channel security shall be achieved through distribution control procedures of the crypto keying materials.

2.2.3.3.3.1. CRYPTOGRAPHIC EQUIPMENT

Channel one of two or four shall be compatible with the VALLOR equipment. Channels 2, 3 and 4 shall be compatible with both the VALLOR and the ALTERNATE-LEF cryptographic equipment. The system shall operate with any combination of VALLOR and ALTERNATE-LEF equipment on channels 2, 3 and 4. The VALLOR equipment shall be operated in the 6.0 Stepped Digital mode, the ALTERNATE-LEF equipment shall be operated in the Clock Start mode where applicable.

2.2.3.3.3.2. MESSAGE FLOW SECURITY

Message flow security shall be provided by maintaining a continuous cryptographic covered transmission over all channels normally used for passing operational messages. It shall be a broadcast management function of the Broadcast Coordination Authority to assure that BCSs are tasked at all times to provide crypto covered data to the VLF/LF transmitter complex. The Broadcast Co-ordination Authority shall place, or cause to be placed from BCSs, symbols encrypted with either the VALLOR or ALTERNATE-LEF. A BCS shall provide a continuous flow of encrypted symbols during the entire duration of channel allocation time. In the event there is no operational message for transmission, the channel shall be made continuous with filler symbols encrypted with the same equipment used for operational messages.

2.2.3.3.4. INTERSITE LINKS

Intersite links shall be provided between each BCS and VLF/LF Transmitter site. Intersite link communications shall include a primary data channel for transmission of the data for relay over the VLF/LF transmitter, a secondary channel to replace the primary channel in event of circuit outage, and an order wire for coordination. The secondary data channel may be dedicated or switched service (dial up). The quality of the intersite links shall be such as to exhibit a character error rate less than or equal to 10^{-5} assuming an errorless input at the BCS. Transmitter Complex shall be equipped to receive and process the intersite link.

2.2.3.3.5. VLF/LF TRANSMITTER COMPLEX PROCESSING

2.2.3.3.5.1. INTERSITE LINK PHASE DRIFT

Each intersite link termination shall be provided with not less than an 8 bit buffer which shall compensate for the phase difference which can be accumulated during a 24 hour period among input signals.

2.2.3.3.5.2. INPUT DATA CONTROL

Input data processing shall provide for channel user switching without causing an interruption or incongruity in the multiplexing scheme which would upset the continuity of data transmission or cause loss of data at the receiver.

2.2.3.3.5.3. MODE CAPACITY INTEGRITY

There shall be no interruption of data flow over the VLF/LF multiplexed transmission. If for any reason the transmit site should not receive encrypted data for one or more channels, a signal shall be generated locally. The locally generated signal shall mimic the normal data format with respect to Fibonacci and parity bits but shall not be encrypted. When a channel is not intended to contain Fibonacci bits, for example RED/BLACK REM, an alternate signal shall be used. This shall be as defined in section 3.2 for RED/BLACK REM.

The Information to be placed on an idle MSK channel shall be a constant 64-ary ITA no. 2 Letter Filler, RY, Figure Filler, Channel Number (e.g. 1, 2, 3, 4). The channel number shall represent the channel on which the signal is placed. The idle channel signal shall contain the 64-ary filler characters. The Fibonacci sequence for channel one shall be a string of constant high data states (one’s) and for channels 2, 3 and 4 shall be a string of constant low data states (zero's). Note that the channel one stream represents the Fibonacci sequence in the already inverted state.

2.2.3.3.5.4. CHANNEL MULTIPLEXING

Following processing of the signals received from the Broadcast Control Stations to obtain the code forms specified in FIGURE 3 a and FIGURE 3 f they shall be time division multiplexed into a serial 100 baud stream for a two channel operation and into a 200 baud serial stream for four channel operation. The time division multiplexing shall be on a bit sequential basis and shall follow channel numbering. Data on VALLOR covered channels shall be framed and sequentially aligned in the multiplexing as shown in Figure 4. Signals which are ALTERNATE-LEF encrypted may be multiplexed into the stream without regard to character bit sequencing with respect to channel one. Figure 4 shows the multiplexing with all channels VALLOR covered and Figure 5 where channel 3 is ALTERNATE-LEF covered.

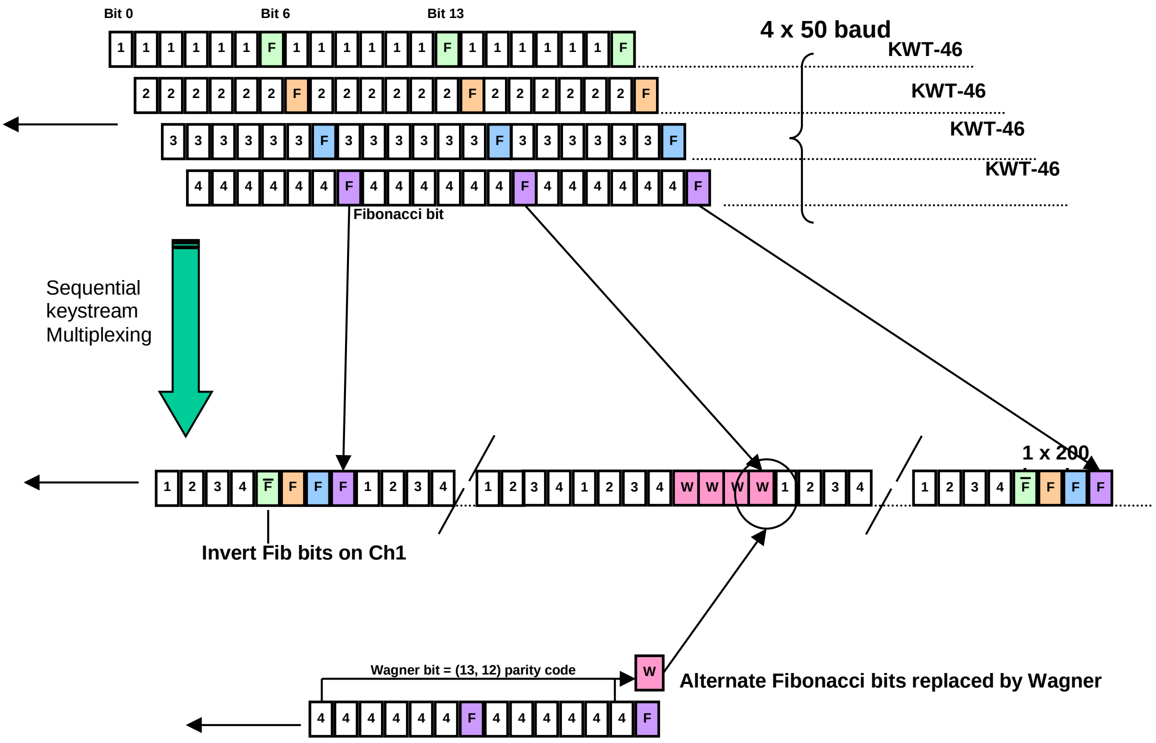

Bit 0
Bit 6
Bit 13
4 x 50 baud
KWT-46
KWT-46
KWT-46
KWT-46
Fibonacci bit
Sequential keystream Multiplexing
1 x 200
Invert Fib bits on Ch1
Wagner bit = (13, 12) parity code
W
Alternate Fibonacci bits replaced by Wagner

FIGURE 4: ALL CHANNELS VALLOR COVERED.

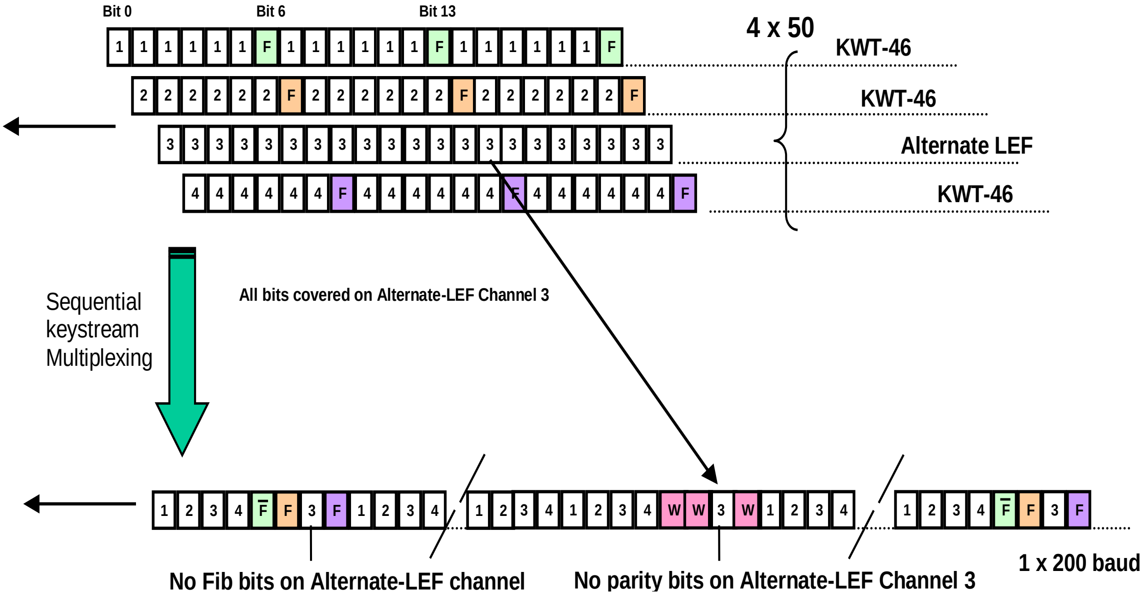

Bit 0
Bit 6
Bit 13
4 x 50
KWT-46
KWT-46
Alternate LEF
KWT-46
Sequential keystream Multiplexing
All bits covered on Alternate-LEF Channel 3
No Fib bits on Alternate-LEF channel
No parity bits on Alternate-LEF Channel 3
1 x 200 baud

FIGURE 5: MULTIPLEXING OF BROADCAST DATA STREAM – ALTERNATE-LEF ON CHANNEL 3.

2.2.3.3.5.5. MINIMUM SHIFT-KEYING MODULATION

MSK is a form of phase shift keying. This modulation technique will allow transmissions of 1.6 bits per Hz of radio frequency bandwidth as defined at the 3dB points. The transmitted signal shall appear as a phase-continuous frequency-shift-keyed signal of constant amplitude. The frequency shift shall be ± ¼ of the modulation rate and shall occur at the modulation rate. This corresponds to a modulation index (m) of ½. The intelligence (mark or space) is contained in the phase shifts and is not consistent with the frequency shifts. The intelligence shall be related to the phase shifts in the following manner. Two sub channels of binary data shall modulate the phase of two sinusoidally weighted components of the reference signal in accordance with the following expression:

$$A\left\{ \overbrace{\left[\sin\left(\omega_0 t+\theta_x\right)\right]\underbrace{\left[\sin\left(\omega_f t\right)\right]}_{\substack{x\ subchannel\\ weighting\ \ function}}}^{x\ component} + \overbrace{\left[\sin\left(\omega_0 t+\theta_y\right)\right]\underbrace{\left[\cos\left(\omega_f t\right)\right]}_{\substack{y\ subchannel\\ weighting\ \ function}}}^{y\ component} \right\}$$

Where:

- *A is the constant amplitude MSK signal;*
- *ω_0 is the frequency of the reference signal (carrier frequency);*
- *ω_f is the frequency of the sinusoidal weighting functions and is equal to π/2T*
- *θ_x shall equal 0 degrees for x sub channel MARKS and 180 degrees for x channel SPACES. These values shall be constant during each ½ cycle of the y sub channel weighting function and equivalent to binary data stream samples at zero crossover instants of sin(ω_ft)*
- *θ_y shall equal 90 degrees for y sub channel MARKS and 270 degrees for y channel SPACES. These values shall be constant during each ½ cycle of the y sub channel weighting function and equivalent to binary data stream samples at zero crossover instants of cos(ω_ft)*
- *T is equal to the time period of the binary data rate.*

2.2.3.3.5.6. VLF/LF TRANSMITTER SUBSYSTEM BANDWIDTH

The VLF/LF transmitter subsystem bandwidth shall be in accordance with paragraph 2.1.2.2.4.

2.2.3.3.5.7. RADIO FREQUENCY

The carrier frequency shall be in accordance with paragraph 2.1.2.2.1 and paragraph 2.1.2.2.2.

2.2.3.3.5.8. PHASE STABILITY

The phase stability and linearity of all radio frequencies shall be in accordance with paragraph 2.1.2.2.3.

2.2.3.3.6. RECEIVER SYSTEM

The receiver system shall operate in accordance with paragraph 2.1.2.3.

2.2.3.3.6.1. SYNCHRONIZATION

In the multichannel modes, the receiver shall synchronize to the incoming signal in not more than 60 seconds with not less than 0.995 probability. This specification shall be exclusive of crypto unit synchronization and shall be at a signal-to-noise ratio of 8dB below the values specified in TABLE.

2.2.3.3.6.2. CHANNEL SELECTION

The receiver system shall provide for operator selection of preferably not less than two channels to be output processed. The processing of these selected channels shall be automatic except that provisions shall be provided for manual reset of the system in the case of false receiver processing. False receiver processing shall occur no more than one time in 100 trials under the conditions specified in TABLE 2.

2.2.3.3.6.3. ALTERNATE-LEF ENCRYPTED SIGNAL PROCESSING

Channels which are received and encrypted with the ALTERNATE-LEF system shall be output to an ALTERNATE-LEF equipment for decryption.

2.2.3.3.6.4. VALLOR ENCRYPTED SIGNAL PROCESSING

Channels which are VALLOR encrypted shall be processed for error correction and reconstitution of the Fibonacci bits prior to routing to the VALLOR equipment for decryption.

Following decryption they shall be further processed for conversion to 7.0 unit START-STOP ITA No. 2 code or other coding which is compatible with output peripheral terminals.

CHAPTER 3 NATO RATE/RANGE EXTENSION (REM)

3.1. NATO Rate/Range Extension Modes

This part defines NATO Rate/Range Extension Modes (REM) for use with VLF broadcasts. REM functionality is incorporated by compressing, protecting, forward error correction coding, interleaving and packetising VLF broadcast streams before multiplexing and transmitting over the air. The objective of REM is to improve broadcast performance over non-REM channels. Different coding schemes are used to optimise how the performance benefit is gained. REM may be used to increase data throughput within a 50 bits per second 'over the air' channel by increasing the effective data rate through compression. Alternatively, REM may be used to increase the broadcast range and coverage area, or to enable a reduction in required transmitter power, without a corresponding change in average throughput. NATO REM will enable VLF reception beyond the traditional NATO Area of Responsibility for existing NATO VLF transmitters.

Two types of REM are available. RED/BLACK REM, offering the greatest performance enhancements but requiring additional RED and BLACK side processing, and RED only REM, which offers some of the enhancements of REM technology whilst retaining full backward compatibility with existing VLF broadcast equipment. RED only REM may be introduced to existing VLF systems by the introduction of RED side processor between the link encryption device and the messaging system. Different packet structures and coding may be used within each of these REM types to balance the performance gain towards data throughput increase or towards range extension.

In this annex RED data and RED processing refers to data and processing on the plain language side of the link encryption device. BLACK data and BLACK processing refers to data and processing on the cipher text side of the link encryption equipment. RED/BLACK processing requires processing on both the plain language and cipher text side of the link encryption device.

3.1.1. NATO MULTICHANNEL REM MODES

NATO VLF multichannel broadcasts containing REM channels shall use modes N7, N8, N9 or N10 as tabulated in paragraph 2.1.1.1. The modes are further defined below.

3.1.1.1. NATO MULTICHANNEL MODE N7

Multichannel mode N7 allows one or more channels of REM to be used in a four channel VLF broadcast. The multi-channel mode N7 shall operate as the NATO multi-channel modes N4 and N5, as defined in paragraph 2.2.3, with the following exceptions: Channel one shall be a VALLOR encrypted channel; this may be either a non REM VALLOR channel or a RED only REM channel. The remaining channels shall comprise of VALLOR or ALTERNATE-LEF channels that may be non-REM, RED REM or RED/BLACK REM encoded. One or more channels may be designated as a NATO channel. A receive platform that is not REM enabled will still be able to decode the non-REM components of a NATO multichannel broadcast containing REM (RED or RED/BLACK) channels. A receive platform that is enabled for RED only REM, shall be able to decode the non-REM and RED REM components of a NATO multichannel broadcast containing RED REM or RED/BLACK REM channels.

3.1.1.2. NATO MULTICHANNEL MODE N8

Multi-channel mode N8 allows one or two channels of REM to be used in a two channel VLF broadcast. The multi-channel mode N8 shall operate as the NATO multi-channel modes N3 and N6, as defined in paragraph 2.2.3, with the following exceptions: Channel one shall be a VALLOR encrypted channel; this may be either a non REM VALLOR channel or a RED only REM channel. The second channel shall be VALLOR or ALTERNATE-LEF, and may be non-REM, RED REM or RED/BLACK REM encoded. Either or both channels may be designated as a NATO channel. A receive platform that is not REM enabled will still be able to decode the non-REM component of a NATO multichannel broadcast containing a REM (RED or RED/BLACK) channel. A receive platform that is enabled for RED only REM, shall be able to decode the non-REM or RED REM component of a NATO multichannel broadcast containing RED REM or a RED/BLACK REM channel.

3.1.1.3. NATO MULTICHANNEL MODE N9

Multichannel mode N9 allows for four channels of full RED/BLACK REM to be used in a four channel VLF broadcast. The multi-channel mode N9 is an optional mode that removes reliance of the multichannel broadcast on the VALLOR encryption equipment. The N9 mode shall operate as the NATO multi-channel modes N4 and N5, as defined in paragraph 2.2.3, but all four channels shall be RED/BLACK REM encoded. Header bits on the channel packets shall be used for channel synchronization in the absence of a VALLOR channel. One or more channels may be designated as a NATO channel.

3.1.1.4. NATO MULTICHANNEL MODE N10

Multichannel mode N10 allows for two channels of full RED/BLACK REM to be used in a two channel VLF broadcast. The multi-channel mode N10 is an optional mode that removes reliance of the multi-channel broadcast on the VALLOR encryption equipment. The N10 mode shall operate as the NATO multi-channel modes N3 and N6, as defined in paragraph 2.2.3, but both channels shall be RED/BLACK REM encoded. Header bits on the channel packets shall be used for channel synchronization in the absence of a VALLOR channel. Either or both channels may be designated as a NATO channel.

3.1.1.5. AUTO-DETECTION OF REM MODE

REM processors shall automatically detect, and correctly output data from non-REM and RED REM channels. When RED/BLACK REM capability is implemented the REM processors shall automatically detect and correctly output data from non-REM, RED REM and RED/BLACK REM channels.

3.1.2. REM PACKET STRUCTURE

Different packet structures are implemented within each type of REM in order to allow the characteristics of the mode to be optimised to the required benefit (e.g. range/throughput). This standard implements two packet structures for each type of REM (RED/BLACK REM and RED REM). For each type of REM, one packet structure has been optimised for increased range and the second packet structure has been optimised for increased throughput. For each type of REM, the packet structure optimised for increased range has been designated as ALPHA, and the packet structure optimised for increased throughput has been designated as BRAVO. Future editions of this standard may implement additional packet structures for either, or both types, of REM.

Figure 6 shows the channel configurations allowed with ACOMP-4724 using the two channel MSK NATO modes (Four channel modes would repeat the second channel for channels three and four. **Figure 6** also shows how ACOMP-4724 systems can be incrementally upgraded to enable full performance gains.

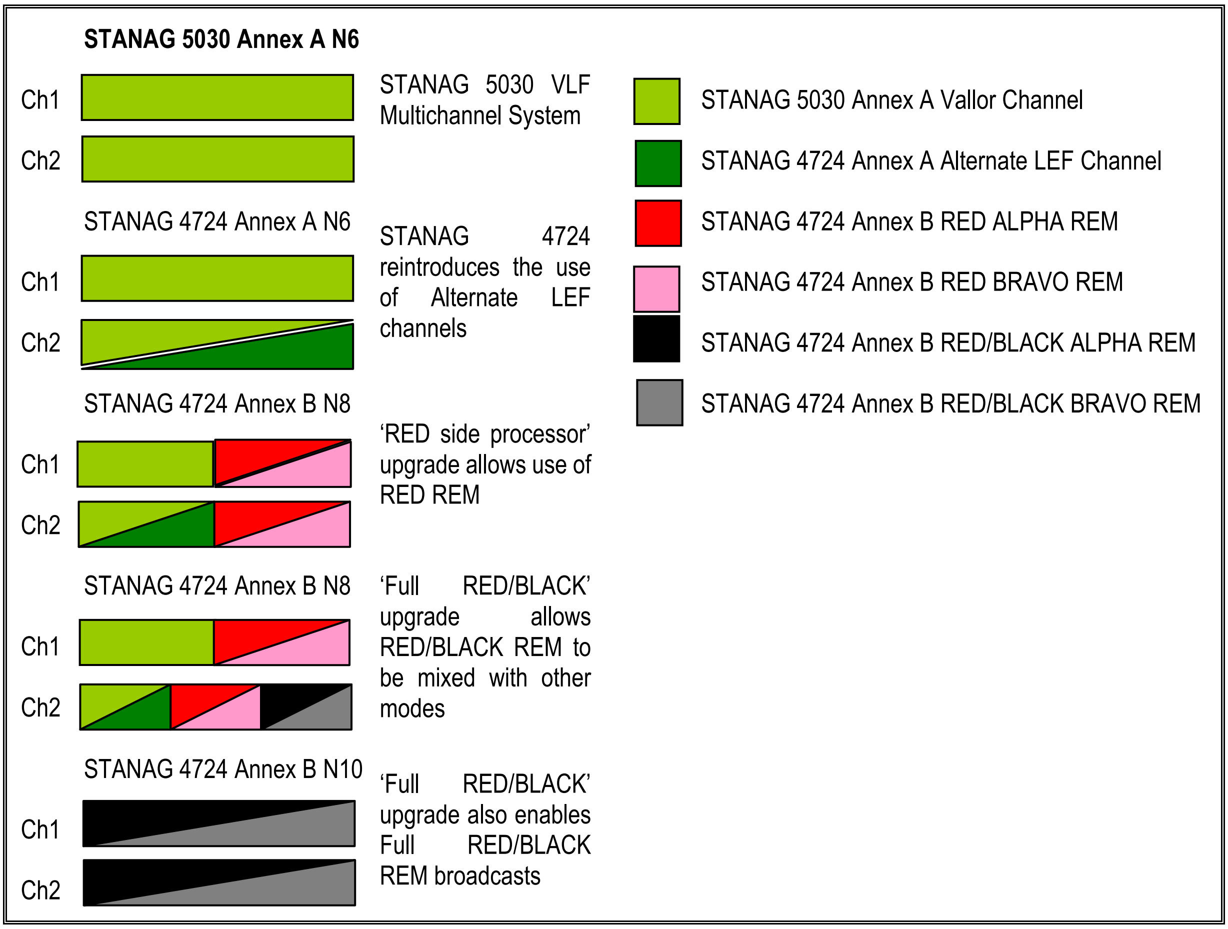

STANAG 5030 Annex A N6
Ch1
Ch2
STANAG 5030 VLF Multichannel System
STANAG 4724 Annex A N6
Ch1
Ch2
STANAG 4724 reintroduces the use of Alternate LEF channels
STANAG 4724 Annex B N8
Ch1
Ch2
'RED side processor' upgrade allows use of RED REM
STANAG 4724 Annex B N8
Ch1
Ch2
'Full RED/BLACK' upgrade allows RED/BLACK REM to be mixed with other modes
STANAG 4724 Annex B N10
Ch1
Ch2
'Full RED/BLACK' upgrade also enables Full RED/BLACK REM broadcasts
STANAG 5030 Annex A Vallor Channel
STANAG 4724 Annex A Alternate LEF Channel
STANAG 4724 Annex B RED ALPHA REM
STANAG 4724 Annex B RED BRAVO REM
STANAG 4724 Annex B RED/BLACK ALPHA REM
STANAG 4724 Annex B RED/BLACK BRAVO REM

FIGURE 6: ACOMP-4724 CHANNEL CONFIGURATION AND UPGRADE OPTIONS.

3.1.2.1. RED/BLACK REM PACKET STRUCTURES

3.1.2.1.1. RED/BLACK REM ALPHA PACKET STRUCTURE

The ALPHA packet structure has been optimized to increase the effective broadcast coverage area, whilst keeping the channel throughput at approximately 50bps. Increased range is obtained by compression, FEC coding, interleaving and packetisation of the broadcast data stream prior to multiplexing and transmission. The ALPHA packet structure uses stronger FEC coding (compared to the BRAVO FEC) to provide an increased broadcast coverage area. If throughput and broadcast coverage are to be kept comparable with the modes defined in CHAPTER 2, the ALPHA packet structure may allow a reduction in transmitter power. The packet structure has been optimised to interface with VALLOR like encryption containing deterministic Fibonacci bits. The BLACK side processing allows for stronger FEC coding compared to the RED ALPHA mode. This provides increased performance compared to the RED ALPHA mode.

3.1.2.1.2. RED/BLACK BRAVO PACKET STRUCTURE

The BRAVO packet structure has been optimised to increase channel throughput whilst keeping the effective range of the VLF broadcast at least comparable with the NATO modes defined in CHAPTER 2. RED/BLACK BRAVO increases the channel throughput to approximately 75bps. Increased throughput is obtained by compression, FEC coding, interleaving and packetisation of the broadcast data stream prior to multiplexing and transmission. The BRAVO packet structure uses weaker FEC coding to enhance throughput (compared to the ALPHA FEC schemes). The packet structure has been optimised to interface with VALLOR like encryption containing deterministic Fibonacci bits.

3.1.2.2. RED REM PACKET STRUCTURES

3.1.2.2.1. RED REM ALPHA PACKET STRUCTURE

The ALPHA packet structure has been optimized to increase the effective broadcast coverage area, whilst keeping the channel throughput at approximately 50bps. Increased range is obtained by compression, FEC coding, interleaving and packetisation of the broadcast data stream prior to multiplexing and transmission. The ALPHA packet structure uses stronger FEC coding (compared to the BRAVO FEC) to provide increased broadcast coverage area. If throughput and broadcast coverage are to be kept comparable with the modes defined in CHAPTER 2, the ALPHA packet structure may allow a reduction in transmitter power. The packet structure has been optimised to interface with VALLOR like encryption containing deterministic Fibonacci bits.

3.1.2.2.2. RED REM BRAVO PACKET STRUCTURE

The BRAVO packet structure has been optimised to increase channel throughput whilst keeping the effective range of the VLF broadcast at least comparable with the NATO modes defined in CHAPTER 2. RED BRAVO increases the channel throughput to approximately 75bps. Increased throughput is obtained by compression, FEC coding, interleaving and packetisation of the broadcast data stream prior to multiplexing and transmission. The BRAVO packet structure uses a weaker FEC to enhance throughput (compared to the ALPHA FEC). The packet structure has been optimised to interface with VALLOR like encryption containing deterministic Fibonacci bits.

3.1.2.2.3. AUTO-DETECTION OF REM PACKET STRUCTURE

REM processors shall use packet headers to automatically detect, and correctly output data from the different REM modes. When RED/BLACK REM capability is implemented the REM processors shall automatically detect and correctly output data from all packet structures defined for both RED/BLACK REM and RED REM. When only RED REM capability is implemented, the REM processors shall automatically detect and correctly output data from all packet structures defined for RED REM only.

3.1.3. NATO RED/BLACK REM

Section 3.2.2 defines the NATO RED/BLACK REM. These modes incorporate a full implementation of REM technology, providing data compression of NATO messages on the RED equipment side and efficient FEC on the BLACK side. Frame synchronisation must be maintained through the packet encryption/decryption process and over the air between the BLACK processors to enable the compressed data to be recovered. This is achieved with reference to accurate time of day information. A RED/BLACK REM channel appears to the broadcast channel multiplexer as an ALTERNATE_LEF channel, regardless of the link encryption employed; this is due to the REM coding process. Figure 7 shows how the NATO RED/BLACK REM can be incorporated into the VLF broadcast system.

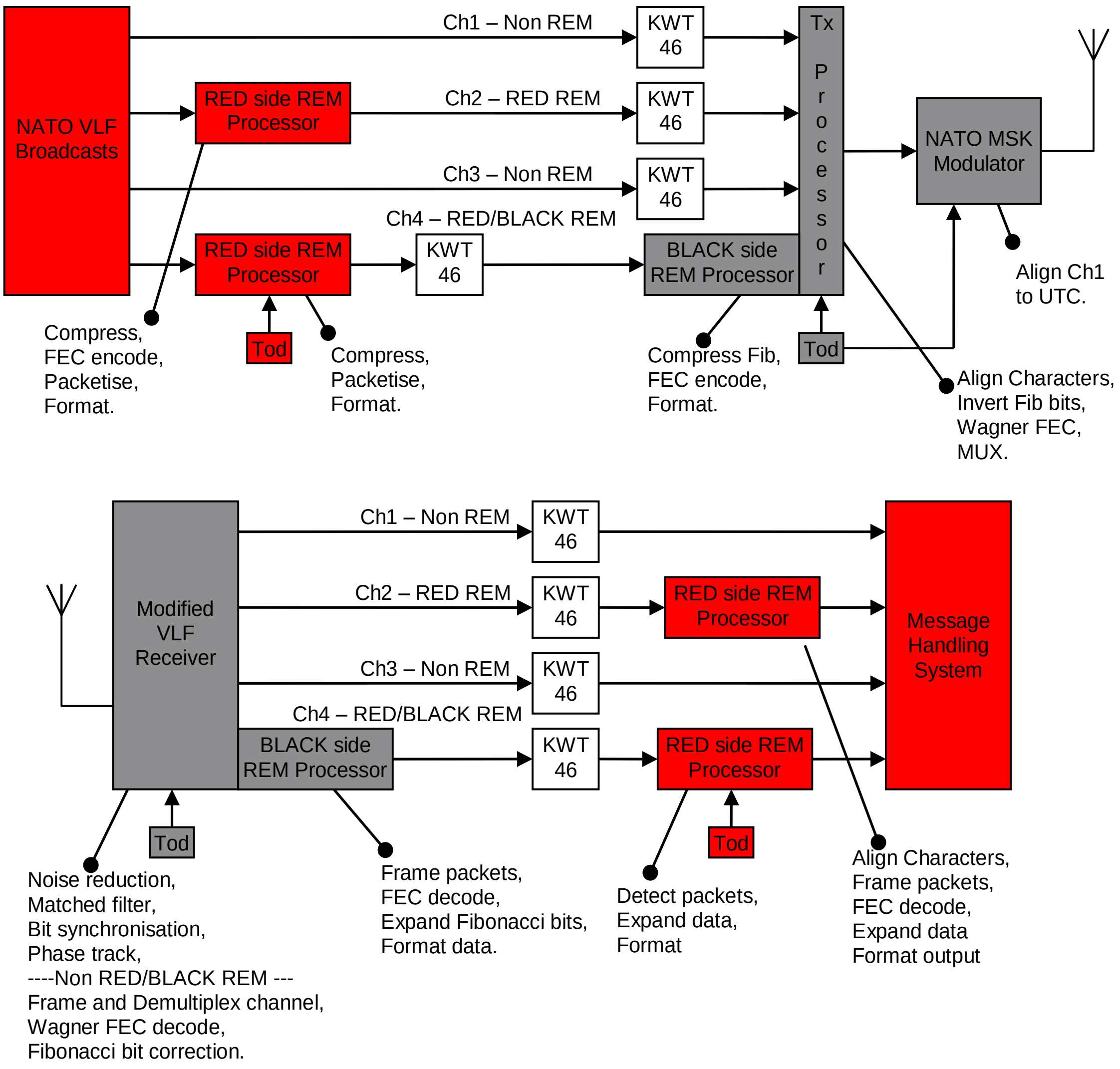

NATO VLF Broadcasts
Ch1 – Non REM
KWT 46
RED side REM Processor
Ch2 – RED REM
KWT 46
Ch3 – Non REM
KWT 46
Ch4 – RED/BLACK REM
RED side REM Processor
KWT 46
BLACK side REM Processor
Tx Processor
NATO MSK Modulator
Tod
Tod
Compress, FEC encode, Packetise, Format.
Compress, Packetise, Format.
Compress Fib, FEC encode, Format.
Align Ch1 to UTC.
Align Characters, Invert Fib bits, Wagner FEC, MUX.
Modified VLF Receiver
Ch1 – Non REM
KWT 46
Ch2 – RED REM
KWT 46
RED side REM Processor
Ch3 – Non REM
KWT 46
Ch4 – RED/BLACK REM
BLACK side REM Processor
KWT 46
RED side REM Processor
Message Handling System
Tod
Tod
Noise reduction, Matched filter, Bit synchronisation, Phase track, ----Non RED/BLACK REM --- Frame and Demultiplex channel, Wagner FEC decode, Fibonacci bit correction.
Frame packets, FEC decode, Expand Fibonacci bits, Format data.
Detect packets, Expand data, Format
Align Characters, Frame packets, FEC decode, Expand data Format output

FIGURE 7: FUNCTIONAL BLOCK DIAGRAM OF REM FOR NATO MULTI-CHANNEL VLF/LF SUBMARINE BROADCAST.

3.1.4. NATO RED REM

Section 3.3.2 defines transition REM offering some of the benefits of REM technology whilst retaining compatibility with equipment designed for the VLF multichannel modes of CHAPTER 2. A RED REM implementation can be wholly contained on the RED equipment side. Specifically the transition modes do not require modification to VLF receivers or the addition of BLACK side processing equipment. **Figure 8** shows how the transition REM can be integrated into the VLF broadcast system.

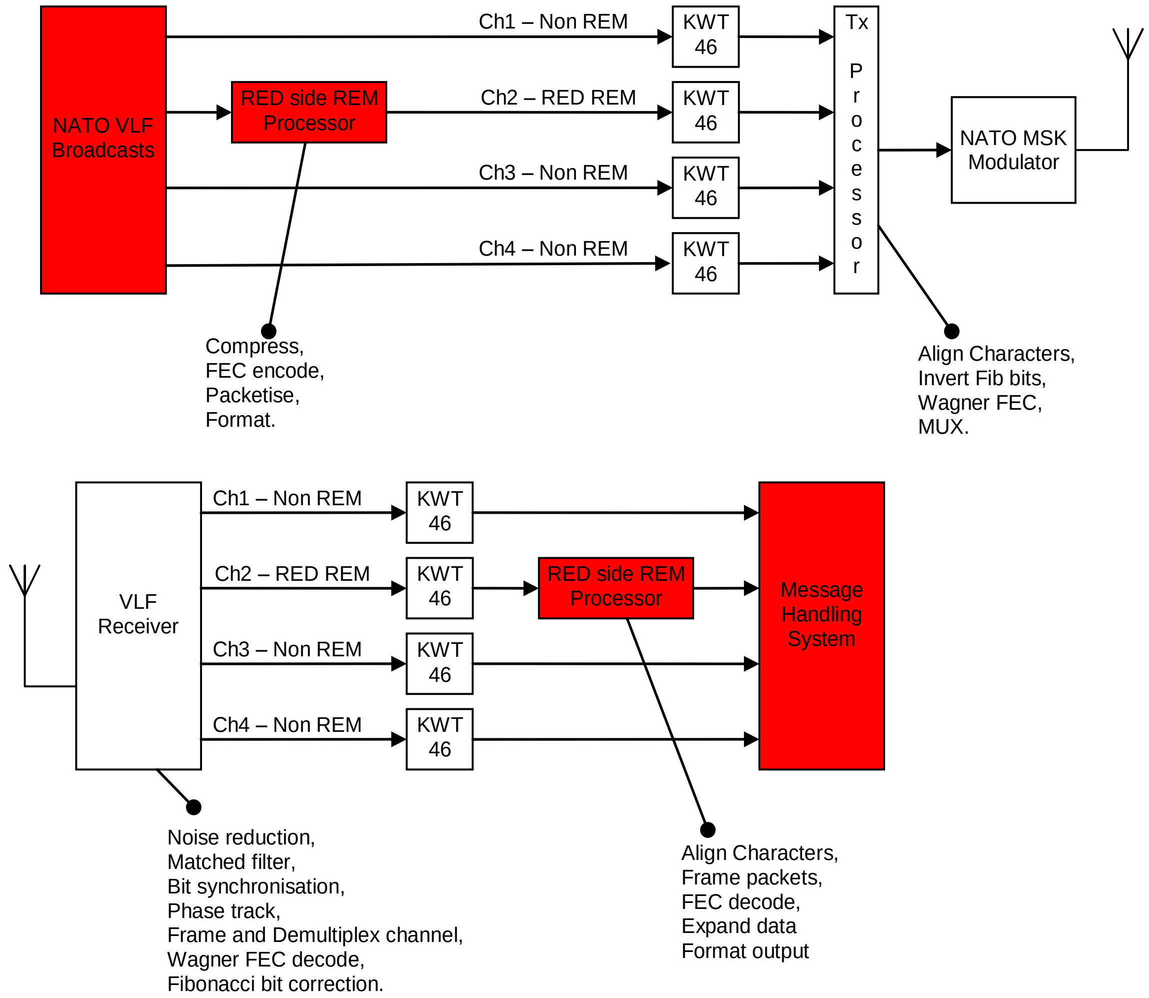


FIGURE 8: FUNCTIONAL BLOCK DIAGRAM OF TRANSITIONAL REM FOR NATO MULTI-CHANNEL VLF/LF SUBMARINE BROADCAST.

The transition modes implement lossless message compression of a VLF broadcast stream, the addition of forward error correction and the packetisation of the data before the link encryption device. As such, all additional processing can be implemented by a processor on the RED side of the link. Similarly on reception, the additional processing is carried out on the RED side of the link encryption device: unpacking, error correcting and uncompressing the channel data.

3.1.5. PERFORMANCE ADVANTAGE OF REM

The design objective of a full RED/BLACK REM ALPHA implementation should provide a performance advantage over a non-REM channel of approximately 13dB. The design objective of a RED REM ALPHA implementation should provide a performance increase over a non-REM channel of approximately 8dB.

The REM BRAVO modes defined in this section should increase the effective channel throughput to approximately 75 bps. The performance of the BRAVO modes would be expected to be approximately 3dB lower than the REM ALPHA modes,

3.1.6. RECEIVER PERFORMANCE

The test methodology and receiver performance of NATO REM modes is defined in CHAPTER 4.

3.2. Processing for NATO RED/BLACK REM

The following defines additional baseband processing necessary to provide NATO VLF MSK RED/BLACK REM. To maximise performance these modes provide data compression on the RED side of the link encryption equipment and efficient forward error correction coding on the BLACK side. The broadcast channel multiplexing and transmissions shall be synchronised with an accurate time of day to ensure frame synchronisation on reception.

Different RED/BLACK REM packet structures are defined to optimise how the REM performance advantage is realised. The following RED/BLACK REM packet structures are defined in this standard:

a. RED/BLACK REM ALPHA defined in paragraph 3.2.3 is optimised for extended range, interfacing with VALLOR like encryption containing deterministic Fibonacci bits;

b. RED/BLACK REM BRAVO defined in paragraph 3.2.4 is optimised to increase throughput to 75bps per channel, interfacing with VALLOR like encryption containing deterministic Fibonacci bits.

3.2.1. NATO RED/BLACK REM

A channel coded with RED/BLACK REM technology has the appearance of an ALTERNATE-LEF encoded channel. A RED/BLACK REM channel may be included as one or more channels of a broadcast operating in modes N7 or N8 (but not on channel one), or on all channels of a broadcast operating in modes N9 or N10. Advanced agreement will be required between the Broadcast Control Station and the subscriber Receive Platform as to the channel(s) on which RED/BLACK REM, RED only REM and NON REM shall be used.

3.2.1.1. FRAME AND TIME SYNCHRONIZATION

To allow correct operation of the RED/BLACK encoding and decoding process it is essential that frame synchronisation is maintained. Frame synchronisation must be maintained across the cryptographic device, and over the air between the transmit and receive systems. Header bits and accurate time of day may be used to maintain frame synchronisation.

To allow the receive systems to reliably search for and detect the packet header symbols without overly large search windows, a NATO multichannel Broadcast containing at least one RED/BLACK REM channel shall be aligned to the UTC day. The first bit of the first packet of channel one (for a RED/BLACK REM channel one), or the first bit of channel one of the broadcast data stream (for non REM or RED REM channel one with RED/BLACK REM on at least one other channel) shall be transmitted at the start of the first second of each standard 24 hour UTC day. Subsequent channel one packets shall be aligned to 21 second intervals thereafter until the start of the next 24 hour UTC day. The first bit of the first packet of channel two shall immediately follow the first bit of channel one. In modes N7 and N9 this shall be followed by the first bit of the first packet of channel three and then channel four.

As 21 second packets to not fit exactly into a 24 hour day, the channels shall be realigned at midnight.

3.2.1.1.1. TIME OF DAY ACCURACY

The time of day used for REM synchronisation shall have an absolute accuracy of plus or minus 0.5 ms at the receiver, relative to the transmitter. The maximum allowable drift rate shall be no greater than 1 part in 10^9 over a 100 day period.

3.2.1.2. RED/BLACK REM FUNCTIONAL ALLOCATIONS

3.2.1.2.1. BROADCAST COORDINATION AUTHORITY (BCA)

a. Control the allocation of channels to the Broadcast Control Stations.

b. Control the assignment of mode characteristics to ensure interoperability among the network nodes.

c. Provide for encrypted filler to be transmitted over all channels which are not allocated to a Broadcast Control station for operations.

d. Carry out RED side REM processing on the broadcast data stream

e. Encrypt the RED REM packaged frame using the VALLOR or ALTERNATE LEF crypto systems as assigned

f. Provide encrypted filler when no messages are available for injection.

3.2.1.2.2. BROADCAST CONTROL STATION (BCS)

The BCS shall perform the following functions:

a. Provide for an interface with an intersite link ensuring time synchronisation is maintained for the encrypted REM frames.

3.2.1.2.3. VLF/LF TRANSMITTER COMPLEX

a. Interface with, accept data and recover REM frame synchronisation from the intersite links.

b. Carry out the BLACK side REM processing of encrypted REM data frames.

c. Place the data on the VLF/LF transmitter in accordance with the channel allocation specified by the VLF/LF Broadcast Coordination Authority.

d. Encode as appropriate non-REM and RED REM channels of the broadcast data stream as per paragraph 2.2.3.2.3 and paragraph 3.3.

e. Multiplex the channels into the serial stream.

f. Align the broadcast with the UTC day as per paragraph 3.2.1.1.

g. Generate the MSK signal.

h. Modulate a VLF/LF carrier with the MSK signal and amplify and transmit this modulated radio frequency

i. Generate a signal which is in accordance with paragraph 2.2.3.3.5.3 and place a clear text signal on the appropriate channel whenever there is no signal for that channel available from a broadcast control station.

3.2.1.2.4. RECEIVE PLATFORM

a. Receive the VLF/LF signal on a suitable antenna subsystem.

b. Provide a receive terminal capable of receiving, demodulating one channel, two and four channel MSK transmissions, in addition, demultiplexing two and four channel transmissions with and without RED/BLACK REM components. RED/BLACK REM channels may need reference to time of day to assist with frame synchronisation.

c. Automatically identify channels and output these channels corresponding to operator request.

d. Carry out BLACK side processing of RED/BLACK REM channels

e. Process as appropriate non-REM and RED REM channels of broadcast as per paragraph 2.2.3.2.4 and section 3.3.

f. Decrypt the channel frames using the appropriate cryptographic equipment

g. Carry out RED side REM processing on decrypted REM data frames

h. Process the decrypted data and output the data to a printer or other peripheral unit.

3.2.2. RED/BLACK REM PROCESSING DEFINITIONS

The following sections define the required REM processing functions for encoding a channel for use with RED/BLACK REM technology.

3.2.2.1. FIXED PACKET STRUCTURE

The RED/BLACK REMs are based on 21 second Channel Packets. For 50 baud broadcast channels this gives 1050 bit packets. These channels have the appearance of ALTERNATE-LEF encoded channels suitable for multiplexing as part of a VLF broadcast, regardless of the actual encryption mode used.

RED REM processing compresses the broadcast data stream (Figure 9 H) using a text data compression technique based on Prediction by Partial Match (PPM) and arithmetic coding technology. RED Channels Frames (Figure 9 G) are built containing 6 padding bits, 6 REM mode identification bits, compressed broadcast stream bits and filler bits. The filler bits allow for the BLACK FEC bits to be added after encryption. The exact number of compressed information bits and filler bits is dependent upon the coding scheme used (ALPHA/BRAVO).

RED Packaged Frames (Figure 9 F) contains the RED Channel Frames (Figure 9 G) which have been packaged into a format suitable for the link encryption used. The 1050 bit RED Packaged frames are sent through the link encryption device for encryption. After encryption the BLACK REM processor takes the 1050 bit Encrypted Channel Frames (Figure 9 E) from the encryption device. Channel Data Frames (Figure 9 D) are made up by extracting the encrypted padding bits, encrypted mode identification bits, encrypted compressed information bits, and 31 Fibonacci bits from the Encrypted Channel Frames, and adding a number of filler bits and the Low Density Parity Check (LDPC) bits. The number of compressed information bits, filler bits and LDPC FEC bits is dependent upon the coding scheme used.

The Channel Interleaved Frame (Figure 9 C) consists of the 980 bit Channel Data Frame whose bits have been shuffled with a fixed, random interleaver as defined in Frame Data Interleaving paragraph 3.2.2.5. Finally the Channel Packet is created by inserting 70 header bits into the Channel Interleaved Frame. The 70 header bits are inserted as 7 sub headers of ten bits each, spread evenly throughout the packet as shown in Figure 9 B.

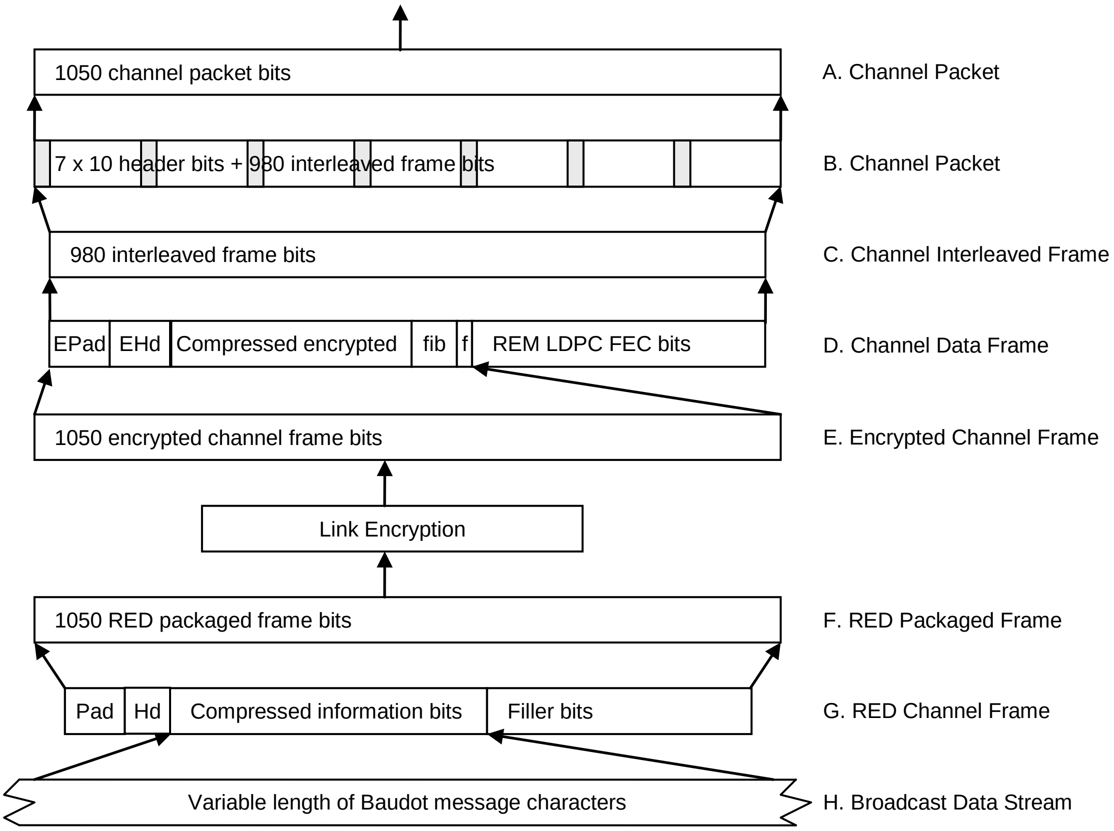

1050 channel packet bits
A. Channel Packet
7 x 10 header bits + 980 interleaved frame bits
B. Channel Packet
980 interleaved frame bits
C. Channel Interleaved Frame
EPad
EHd
Compressed encrypted
fib
f
REM LDPC FEC bits
D. Channel Data Frame
1050 encrypted channel frame bits
E. Encrypted Channel Frame
Link Encryption
1050 RED packaged frame bits
F. RED Packaged Frame
Pad
Hd
Compressed information bits
Filler bits
G. RED Channel Frame
Variable length of Baudot message characters
H. Broadcast Data Stream

FIGURE 9: SCHEMATIC COMPOSITION OF RED/BLACK NATO REM CHANNEL.

3.2.2.2. COMPRESSION ALGORITHM

The NATO REM information frames consist of a compressed representation of a sequence of NATO message text characters. This sequence of characters may represent part of a line of message text or it may represent one or more complete lines of message text (depending on the inherent compressibility of the text). The NATO REM data compression/expansion algorithms translate the text characters into (and out of) frames of compressed information bits without loss of content or meaning.

3.2.2.2.1. PREDICTION BY PARTIAL MATCH ALGORITHMS

Prediction by Partial Match (PPM) algorithms generate adaptive statistical models of conditional probabilities of the next character given the context of a number of preceding characters. For each character context a list of next characters that have been processed with this context and their frequency is assessed. If the next character to be compressed is in this list then the list is used as the conditional probability distribution for the arithmetic encoding of this character. After the encoding, the statistical model is updated with this occurrence of the processed character. Thus the statistical model updates itself while processing the character stream.

Arithmetic encoding of characters in terms of conditional probability distribution is a variable length encoder that matches the number of bits used to encode a character to the entropy of the conditional probability distribution.

3.2.2.2.2. PRE-LOADED STATISTICAL MODEL

Since the compressed packets only represent a small sample of the broadcast text, the adaptive statistical model of the PPM algorithm cannot converge to a useful statistical model if it starts with only a fixed “-1” context. Therefore, fixed, pre-defined initial statistical models are defined for use with the compression and expansion of each compressed information packet.

3.2.2.2.3. STATISTICAL MODEL FORMAT

Details of the statistical model file and format are included in ANNEX A.

3.2.2.2.4. ALTERNATE STATISTICAL MODELS

The statistical models defined in this STANAG shall be used on NATO broadcasts. Alternate statistical models may be generated and used for national purposes.

3.2.2.3. PACKETISATION OF COMPRESSED DATA STREAM

The REM compression algorithm will produce frames of compressed data from the broadcast data stream. The compressed data frames shall be packetised to enable them to be passed to link encryption equipment. RED channel frames shall be produced, consisting of 6 padding bits, 6 REM mode identification bits, compressed information bits, and sufficient filler bits to enable the LDPC FEC coding. The RED packaged frame shall be sent to the link encryption equipment starting with bit position 1.

3.2.2.4. RED/BLACK REM BLACK CODING

The link encryption process will output a stream of encrypted characters. Frame synchronisation between the input and output streams of the link encryption device shall be maintained to enable the encrypted compressed information to be extracted and further processed on the BLACK equipment side.

3.2.2.4.1. BLACK CHANNEL DATA FRAME STRUCTURE

The 980 bit BLACK channel frame shall be made up of the encrypted compressed information bits, extracted from the encrypted channel frame, and the LDPC FEC coding bits. Frame timing shall be maintained between the RED and BLACK data streams to enable the correct portion of the frame to be extracted.

3.2.2.4.2. LDPC FEC CODING

Each RED/BLACK REM channel shall be protected against bit errors in transmission by a FEC code applied to the channel information prior to interleaving.

3.2.2.4.3. LDPC FEC DECODING

Once the FEC parity bits have been incorporated, the channel data frame will consist of an FEC encoded frame with a systematic FEC LDPC code. Using an appropriate decoding algorithm at the receiver, the codes can correct nearly the maximum number of transmission bit errors that are theoretically possible to correct.

A set of parity equations may be used to detect bit errors within the data frame. If the data frame satisfies all the parity check equations there remains only a very small probability that the frame contains undetectable bit errors. ANNEX B describes an example LDPC decoder and defines the parity equations required for the REM modes.

If the FEC decoding process cannot complete successfully (owing to un-correctable bit errors), the expansion of the frame will cause a large amount of garble message text to be generated. Therefore, in this circumstance, the frame should be marked for discarding.

3.2.2.5. FRAME DATA INTERLEAVING

The channel data frame shall be protected from correlated bursts of bit errors in the received data frame by shuffling the data bits in a random (but fixed) order before transmission. This operation spreads clusters of bit errors (that may be due to strong atmospheric noise impulses) more uniformly throughout the data frame. The 980 bit channel data frame shall be shuffled into the 980 bit channel interleaved frame using **TABLE 6** (channel data frame bit position => channel interleaved frame bit position).

1 => 15	2 => 69	3 => 254	4 => 906	5 => 283	6 => 566	7 => 150
8 => 166	9 => 267	10 => 947	11 => 437	12 => 352	13 => 826	14 => 846
15 => 11	16 => 117	17 => 856	18 => 876	19 => 712	20 => 74	21 => 236
22 => 668	23 => 455	24 => 834	25 => 802	26 => 835	27 => 98	28 => 71
29 => 472	30 => 29	31 => 500	32 => 357	33 => 104	34 => 824	35 => 557
36 => 344	37 => 556	38 => 892	39 => 147	40 => 645	41 => 724	42 => 845
43 => 127	44 => 893	45 => 486	46 => 315	47 => 376	48 => 719	49 => 700
50 => 896	51 => 770	52 => 737	53 => 263	54 => 259	55 => 207	56 => 579
57 => 33	58 => 977	59 => 524	60 => 857	61 => 868	62 => 723	63 => 620
64 => 224	65 => 683	66 => 109	67 => 370	68 => 272	69 => 339	70 => 728
71 => 634	72 => 635	73 => 690	74 => 252	75 => 308	76 => 944	77 => 636
78 => 274	79 => 251	80 => 831	81 => 54	82 => 426	83 => 338	84 => 910
85 => 781	86 => 340	87 => 449	88 => 813	89 => 735	90 => 630	91 => 460
92 => 173	93 => 773	94 => 927	95 => 249	96 => 581	97 => 772	98 => 865
99 => 674	100 => 691	101 => 611	102 => 75	103 => 631	104 => 134	105 => 167
106 => 979	107 => 504	108 => 297	109 => 615	110 => 453	111 => 878	112 => 446
113 => 397	114 => 594	115 => 809	116 => 912	117 => 201	118 => 558	119 => 946
120 => 886	121 => 958	122 => 188	123 => 641	124 => 369	125 => 448	126 => 371
127 => 487	128 => 619	129 => 394	130 => 891	131 => 138	132 => 124	133 => 133
134 => 537	135 => 180	136 => 555	137 => 447	138 => 682	139 => 490	140 => 565
141 => 232	142 => 956	143 => 945	144 => 767	145 => 347	146 => 13	147 => 443
148 => 792	149 => 425	150 => 822	151 => 93	152 => 978	153 => 805	154 => 905
155 => 790	156 => 57	157 => 667	158 => 237	159 => 136	160 => 603	161 => 590
162 => 435	163 => 851	164 => 116	165 => 107	166 => 854	167 => 187	168 => 76
169 => 976	170 => 23	171 => 450	172 => 374	173 => 584	174 => 195	175 => 879
176 => 568	177 => 268	178 => 80	179 => 489	180 => 559	181 => 118	182 => 672
183 => 59	184 => 190	185 => 604	186 => 965	187 => 623	188 => 593	189 => 823
190 => 52	191 => 257	192 => 729	193 => 299	194 => 410	195 => 506	196 => 572
197 => 666	198 => 771	199 => 601	200 => 220	201 => 204	202 => 763	203 => 49
204 => 148	205 => 907	206 => 404	207 => 722	208 => 400	209 => 92	210 => 720
211 => 707	212 => 898	213 => 114	214 => 901	215 => 526	216 => 306	217 => 920
218 => 516	219 => 812	220 => 363	221 => 629	222 => 181	223 => 789	224 => 476
225 => 847	226 => 727	227 => 230	228 => 721	229 => 332	230 => 505	231 => 82
232 => 128	233 => 904	234 => 758	235 => 68	236 => 517	237 => 968	238 => 588
239 => 56	240 => 199	241 => 480	242 => 458	243 => 432	244 => 750	245 => 534
246 => 477	247 => 694	248 => 680	249 => 273	250 => 816	251 => 5	252 => 44
253 => 778	254 => 563	255 => 561	256 => 456	257 => 25	258 => 595	259 => 319
260 => 708	261 => 966	262 => 353	263 => 163	264 => 485	265 => 184	266 => 495
267 => 745	268 => 158	269 => 110	270 => 597	271 => 140	272 => 416	273 => 270
274 => 837	275 => 678	276 => 399	277 => 551	278 => 422	279 => 413	280 => 775

281 => 858	282 => 925	283 => 228	284 => 747	285 => 766	286 => 587	287 => 336
288 => 467	289 => 791	290 => 578	291 => 598	292 => 226	293 => 497	294 => 45
295 => 161	296 => 833	297 => 725	298 => 608	299 => 50	300 => 311	301 => 466
302 => 519	303 => 592	304 => 610	305 => 607	306 => 864	307 => 19	308 => 51
309 => 582	310 => 178	311 => 474	312 => 933	313 => 373	314 => 384	315 => 312
316 => 346	317 => 364	318 => 310	319 => 954	320 => 616	321 => 562	322 => 115
323 => 40	324 => 698	325 => 647	326 => 961	327 => 669	328 => 481	329 => 137
330 => 193	331 => 569	332 => 664	333 => 420	334 => 586	335 => 309	336 => 955
337 => 86	338 => 278	339 => 463	340 => 889	341 => 888	342 => 113	343 => 322
344 => 596	345 => 522	346 => 329	347 => 801	348 => 832	349 => 103	350 => 788
351 => 157	352 => 255	353 => 172	354 => 452	355 => 711	356 => 229	357 => 95
358 => 351	359 => 91	360 => 391	361 => 600	362 => 508	363 => 288	364 => 3
365 => 760	366 => 706	367 => 215	368 => 142	369 => 414	370 => 687	371 => 333
372 => 381	373 => 671	374 => 433	375 => 705	376 => 861	377 => 126	378 => 8
379 => 881	380 => 210	381 => 817	382 => 162	383 => 908	384 => 431	385 => 544
386 => 31	387 => 208	388 => 738	389 => 887	390 => 289	391 => 290	392 => 929
393 => 320	394 => 217	395 => 885	396 => 9	397 => 409	398 => 266	399 => 176
400 => 883	401 => 916	402 => 454	403 => 189	404 => 549	405 => 511	406 => 20
407 => 234	408 => 101	409 => 186	410 => 930	411 => 637	412 => 330	413 => 164
414 => 580	415 => 478	416 => 689	417 => 65	418 => 574	419 => 324	420 => 783
421 => 70	422 => 154	423 => 262	424 => 205	425 => 663	426 => 498	427 => 292
428 => 742	429 => 245	430 => 662	431 => 403	432 => 951	433 => 4	434 => 269
435 => 203	436 => 206	437 => 628	438 => 396	439 => 585	440 => 471	441 => 741
442 => 64	443 => 285	444 => 342	445 => 253	446 => 155	447 => 389	448 => 840
449 => 632	450 => 963	451 => 323	452 => 280	453 => 441	454 => 838	455 => 233
456 => 935	457 => 577	458 => 859	459 => 387	460 => 444	461 => 676	462 => 298
463 => 899	464 => 334	465 => 624	466 => 132	467 => 776	468 => 106	469 => 932
470 => 246	471 => 869	472 => 131	473 => 90	474 => 757	475 => 520	476 => 318
477 => 957	478 => 282	479 => 819	480 => 149	481 => 368	482 => 343	483 => 222
484 => 880	485 => 72	486 => 503	487 => 797	488 => 894	489 => 395	490 => 424
491 => 6	492 => 702	493 => 828	494 => 356	495 => 7	496 => 392	497 => 542
498 => 860	499 => 287	500 => 235	501 => 62	502 => 677	503 => 539	504 => 362
505 => 733	506 => 848	507 => 144	508 => 276	509 => 684	510 => 798	511 => 192
512 => 552	513 => 675	514 => 949	515 => 81	516 => 589	517 => 512	518 => 839
519 => 922	520 => 378	521 => 153	522 => 902	523 => 660	524 => 386	525 => 337
526 => 22	527 => 281	528 => 327	529 => 242	530 => 884	531 => 78	532 => 393
533 => 900	534 => 307	535 => 398	536 => 211	537 => 974	538 => 326	539 => 554
540 => 716	541 => 510	542 => 382	543 => 867	544 => 182	545 => 759	546 => 168
547 => 380	548 => 911	549 => 459	550 => 919	551 => 328	552 => 367	553 => 928
554 => 67	555 => 862	556 => 715	557 => 546	558 => 818	559 => 305	560 => 143
561 => 275	562 => 216	563 => 633	564 => 300	565 => 548	566 => 421	567 => 313
568 => 408	569 => 24	570 => 540	571 => 844	572 => 407	573 => 294	574 => 917
575 => 786	576 => 915	577 => 445	578 => 61	579 => 284	580 => 317	581 => 175
582 => 419	583 => 223	584 => 66	585 => 325	586 => 473	587 => 924	588 => 102
589 => 46	590 => 658	591 => 777	592 => 703	593 => 625	594 => 613	595 => 755
596 => 659	597 => 348	598 => 820	599 => 209	600 => 648	601 => 430	602 => 606
603 => 121	604 => 475	605 => 469	606 => 643	607 => 191	608 => 58	609 => 345
610 => 734	611 => 509	612 => 764	613 => 717	614 => 331	615 => 710	616 => 714
617 => 673	618 => 465	619 => 464	620 => 55	621 => 36	622 => 640	623 => 528
624 => 753	625 => 247	626 => 599	627 => 895	628 => 545	629 => 159	630 => 494
631 => 942	632 => 89	633 => 573	634 => 293	635 => 681	636 => 244	637 => 354
638 => 718	639 => 793	640 => 523	641 => 575	642 => 221	643 => 855	644 => 436

645 => 794	646 => 622	647 => 96	648 => 304	649 => 277	650 => 111	651 => 302
652 => 47	653 => 42	654 => 88	655 => 218	656 => 355	657 => 853	658 => 656
659 => 576	660 => 730	661 => 1	662 => 468	663 => 699	664 => 830	665 => 964
666 => 639	667 => 200	668 => 231	669 => 26	670 => 335	671 => 948	672 => 358
673 => 531	674 => 256	675 => 940	676 => 810	677 => 752	678 => 169	679 => 286
680 => 800	681 => 434	682 => 863	683 => 112	684 => 39	685 => 692	686 => 815
687 => 63	688 => 921	689 => 383	690 => 806	691 => 564	692 => 538	693 => 950
694 => 179	695 => 697	696 => 139	697 => 873	698 => 975	699 => 35	700 => 496
701 => 366	702 => 442	703 => 602	704 => 385	705 => 686	706 => 32	707 => 962
708 => 261	709 => 165	710 => 654	711 => 926	712 => 18	713 => 913	714 => 341
715 => 214	716 => 250	717 => 713	718 => 375	719 => 903	720 => 160	721 => 688
722 => 693	723 => 787	724 => 427	725 => 518	726 => 939	727 => 27	728 => 12
729 => 225	730 => 754	731 => 361	732 => 16	733 => 852	734 => 780	735 => 836
736 => 21	737 => 185	738 => 740	739 => 746	740 => 696	741 => 14	742 => 388
743 => 197	744 => 423	745 => 614	746 => 980	747 => 350	748 => 241	749 => 769
750 => 651	751 => 653	752 => 48	753 => 872	754 => 874	755 => 571	756 => 732
757 => 401	758 => 550	759 => 827	760 => 198	761 => 652	762 => 174	763 => 825
764 => 457	765 => 877	766 => 829	767 => 970	768 => 238	769 => 240	770 => 482
771 => 94	772 => 129	773 => 171	774 => 438	775 => 406	776 => 405	777 => 145
778 => 849	779 => 536	780 => 493	781 => 609	782 => 135	783 => 841	784 => 649
785 => 748	786 => 626	787 => 923	788 => 349	789 => 799	790 => 60	791 => 941
792 => 807	793 => 749	794 => 532	795 => 428	796 => 866	797 => 130	798 => 543
799 => 960	800 => 412	801 => 87	802 => 909	803 => 591	804 => 774	805 => 762
806 => 491	807 => 264	808 => 650	809 => 953	810 => 850	811 => 501	812 => 567
813 => 202	814 => 492	815 => 10	816 => 461	817 => 709	818 => 100	819 => 583
820 => 969	821 => 502	822 => 41	823 => 952	824 => 665	825 => 547	826 => 808
827 => 784	828 => 119	829 => 638	830 => 462	831 => 379	832 => 768	833 => 811
834 => 73	835 => 971	836 => 30	837 => 470	838 => 525	839 => 295	840 => 871
841 => 726	842 => 918	843 => 296	844 => 514	845 => 316	846 => 938	847 => 83
848 => 937	849 => 321	850 => 685	851 => 533	852 => 418	853 => 439	854 => 483
855 => 959	856 => 897	857 => 99	858 => 84	859 => 642	860 => 271	861 => 618
862 => 377	863 => 151	864 => 372	865 => 943	866 => 219	867 => 621	868 => 303
869 => 183	870 => 530	871 => 507	872 => 125	873 => 646	874 => 265	875 => 936
876 => 479	877 => 17	878 => 796	879 => 213	880 => 761	881 => 484	882 => 731
883 => 890	884 => 795	885 => 756	886 => 843	887 => 914	888 => 605	889 => 972
890 => 97	891 => 417	892 => 227	893 => 736	894 => 488	895 => 785	896 => 248
897 => 411	898 => 239	899 => 743	900 => 803	901 => 695	902 => 617	903 => 670
904 => 499	905 => 34	906 => 527	907 => 657	908 => 105	909 => 521	910 => 973
911 => 931	912 => 108	913 => 314	914 => 535	915 => 38	916 => 212	917 => 365
918 => 146	919 => 875	920 => 390	921 => 141	922 => 79	923 => 53	924 => 402
925 => 260	926 => 553	927 => 123	928 => 513	929 => 541	930 => 515	931 => 765
932 => 739	933 => 704	934 => 301	935 => 934	936 => 821	937 => 196	938 => 779
939 => 177	940 => 529	941 => 291	942 => 43	943 => 804	944 => 814	945 => 415
946 => 243	947 => 85	948 => 120	949 => 870	950 => 644	951 => 2	952 => 156
953 => 258	954 => 77	955 => 882	956 => 679	957 => 570	958 => 751	959 => 429
960 => 360	961 => 655	962 => 967	963 => 359	964 => 122	965 => 782	966 => 842
967 => 170	968 => 627	969 => 560	970 => 612	971 => 661	972 => 194	973 => 152
974 => 744	975 => 451	976 => 28	977 => 701	978 => 279	979 => 440	980 => 37.

TABLE 6: CHANNEL INTERLEAVING.

At the receiver the data frame shall be un-shuffled (deinterleaved) using the reverse operation.

3.2.2.6. RED/BLACK REM FRAME HEADER

Before transmission, 70 header bits shall be inserted into the 980 bit channel interleaved frame making a channel packet of 1050 bits. The header bits allow the receiver to determine the exact position of the beginning of each REM packet (resolving timing offsets and propagation delays), to estimate the phase of the carrier signal over the REM packet to allow coherent demodulation of the binary symbols, and to identify the type of REM coding used on the channel.

The RED/BLACK REM channel interleaved frame header consists of 70 known bits distributed along the frame in seven 10 bit header segments. The header sequences have low cross correlation and autocorrelation properties to allow for unambiguous detection and framing of the channel packets.

Header sequences shall be used to identify the type of packet structure and coding used on the frames (ALPHA/BRAVO, etc).

3.2.2.7. CHANNEL MULTIPLEXING

3.2.2.7.1. RED/BLACK REM CHANNEL IN NATO MULTICHANNEL MODES N7 AND N8

For NATO modes N7 and N8, RED/BLACK REM channels shall be time division multiplexed onto a broadcast data stream in accordance with paragraph 2.2.3.3.4. To enable frame synchronisation for broadcasts containing non REM or RED REM channels, Channel 1 shall be a VALLOR encrypted channel (or VALLOR like encrypted channel with deterministic Fibonacci bits). This may be non-REM or RED only REM. The multiplexed broadcast shall be aligned to the UTC day in accordance with paragraph 3.2.1.1. if it contains any RED/BLACK REM channels.

3.2.2.7.2. RED/BLACK REM CHANNEL IN NATO MULTICHANNEL MODES N9 AND N10

For NATO modes N9 and N10, RED/BLACK REM channels shall be time division multiplexed onto a broadcast data stream in accordance with paragraph 2.2.3.3.5.4 but Channel 1 does not have to be a VALLOR encrypted channel. The multiplexed broadcast stream shall be aligned to the UTC day in accordance with paragraph 3.2.1.1.

3.2.2.8. MODE CAPACITY INTEGRITY WITH RED/BLACK REM

To prevent interruption of data flow over the VLF/LF multiplexed transmission, a locally generated signal shall be used when a RED/BLACK data stream is not presented to the multiplexer. (This may also be generated by the BLACK processor should it not receive encrypted data.) The signal sequence defined in 2.2.3.3.5.3 is not suitable for RED/BLACK REM. For RED/BLACK REM the sequence shall be constructed as follows:

a. The ‘missing’ Encrypted Channel Frame (Figure 9 E) shall be replaced with the a frame consisting of the repeated sequence defined in 2.2.3.3.5.3 (Letter Filler, R, Y, Figure Filler, Channel Number). The stop bit/Fibonacci bit for all channels shall be set to a low data state (0). Each Frame shall start with a Letter Filler character.

b. This frame shall be processed as a standard Encrypted Channel Frame appropriate to the channel mode being used (ALPHA/BRAVO).

c. The resulting deterministic Channel Packet shall be sent correctly time aligned with the other broadcast channels.

3.2.3. RED/BLACK REM ALPHA PACKET STRUCTURE

The ALPHA packet structure and coding scheme has been optimized to increase the effective broadcast coverage area whilst keeping the channel throughput at approximately 50bps. The packet structure has been optimised to interface with VALLOR like encryption containing deterministic Fibonacci bits.

3.2.3.1. RED/BLACK REM ALPHA PACKET STRUCTURE DETAIL

RED/BLACK ALPHA REM packets are based on 21 second Channel Packets for 50 baud channels,, giving 1050 bit packets as shown in Figure 10.

RED ALPHA REM processing compresses the broadcast data stream (Figure 10 H) using a text data compression technique based on PPM and arithmetic coding technology. RED Channels Frames (Figure 10 G) are built containing 6 padding bits, 6 REM mode identification bits, 444 compressed broadcast stream bits and 444 filler bits. The filler bits allow for the BLACK FEC bits to be added after encryption.

RED Packaged Frames (Figure 10 F) contains the 900 bit RED Channel Frames (Figure 10 G) which have been packaged into 6 bit ITA2 64-ary style characters by inserting a stop bit after each character (6 bits), adding 150 stop bits. The 1050 bit RED Packaged frames are sent through the link encryption device for encryption.

After encryption the BLACK REM processor takes the 1050 bit Encrypted Channel Frames (Figure 10 E) from the encryption device. Channel Data Frames (Figure 10 D) are made up by extracting the 6 encrypted padding bits, 6 encrypted mode identification bits, 444 encrypted compressed information bits, and the first 31 Fibonacci bits from each Encrypted Channel Frame. The ALPHA Channel data frames are completed by adding 3 filler bits and 490 bits of Low Density Parity Check (LDPC) FEC.

The Channel Interleaved Frame (Figure 10 C) consists of the 980 bit Channel Data Frame whose bits have been shuffled with a fixed, random interleaver as defined in Frame Data Interleaving paragraph 3.2.2.5. Finally the Channel Packet is created by inserting 70 header bits into the Channel Interleaved Frame. The 70 header bits are inserted as 7 sub headers of ten bits each, spread evenly throughout the packet as shown in Figure 10 B.

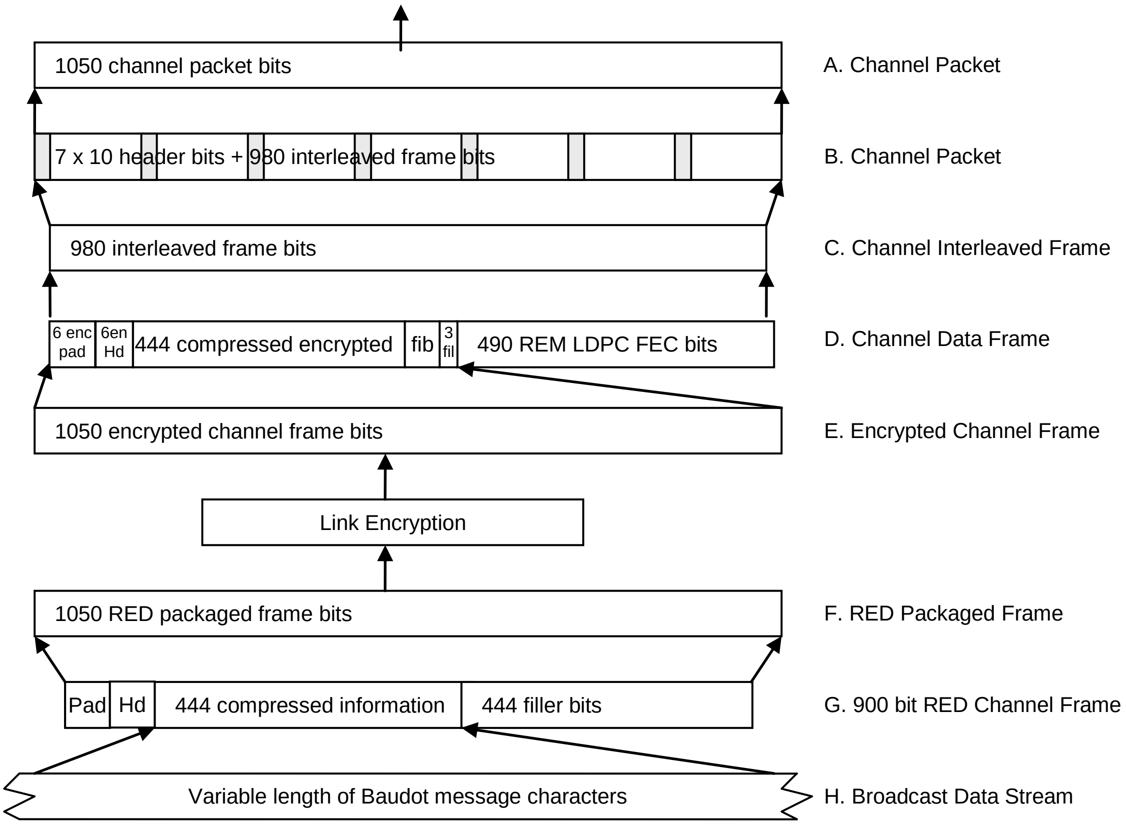

1050 channel packet bits
A. Channel Packet
7 x 10 header bits + 980 interleaved frame bits
B. Channel Packet
980 interleaved frame bits
C. Channel Interleaved Frame
6 enc pad
6en Hd
444 compressed encrypted
fib
3 fil
490 REM LDPC FEC bits
D. Channel Data Frame
1050 encrypted channel frame bits
E. Encrypted Channel Frame
Link Encryption
1050 RED packaged frame bits
F. RED Packaged Frame
Pad
Hd
444 compressed information
444 filler bits
G. 900 bit RED Channel Frame
Variable length of Baudot message characters
H. Broadcast Data Stream

FIGURE 10: SCHEMATIC COMPOSITION OF RED/BLACK ALPHA NATO REM CHANNEL.

3.2.3.2. RED/BLACK REM ALPHA PPM ALGORITHMS

RED/BLACK REM ALPHA uses a PPM algorithm based on an adaptive statistical model of conditional probabilities of the next character given the most recent four character context. The algorithm shall be initialised with a pre-defined, fixed model for each output compressed information frame. Characters shall be assessed one at a time from a sequence of characters of the broadcast stream and be compressed using an arithmetic coding scheme based on the conditional probability distribution of the statistical model given the current context of the four preceding characters.

3.2.3.2.1. RED/BLACK REM ALPHA PPM COMPRESSION

RED/BLACK REM ALPHA PPM compression coding shall be carried out to be interoperable with the code given in ANNEX C.

3.2.3.2.2. RED/BLACK REM ALPHA PPM EXPANSION

The RED/BLACK REM ALPHA shall be expanded to recover the original broadcast data stream. An example of a suitable expansion algorithm is given ANNEX D.

3.2.3.2.3. RED/BLACK REM ALPHA STATISTICAL MODEL

Details of the statistical model file and format are included ANNEX A

3.2.3.3. RED/BLACK REM ALPHA RED SIDE PACKETISATION

The RED/BLACK REM ALPHA compression algorithm will produce 444 bit frames of compressed data from the broadcast data stream, each frame providing the information for a 21 second channel packet. The compressed data frames shall be packetised before being passed to the link encryption equipment.

The first 6 bit positions of each 900 bit RED channel frame shall contain padding bits. These 6 bits shall all be set to zero. The next six bits shall contain a REM mode identification character to allow auto mode detection of RED/BLACK REM frames in the RED REM processor. For RED/BLACK REM ALPHA the mode identification character shall be as defined in **TABLE 7**.

Bit position	7	8	9	10	11	12
Content	1	0	0	0	0	0

TABLE 7: RED/BLACK REM ALPHA MODE IDENTIFICATION CHARACTER

The 444 compressed information bits shall be inserted into the RED channel frame immediately after the mode identification character bits. The compressed information shall be loaded into the 900 bit RED channel frame in the order that it is generated, i.e. the first bit generated shall map to bit 13; the last bit generated shall map to bit 456. The last 444 bits of the RED channel frame are filler bits and shall all be set to one.

This RED channel frame shall be packaged into 6 bit ITA2 64-ary characters with a stop bit inserted after each sixth bit. The inserted stop bits shall all be set to one. A total of 150 stop bits will be required, making a RED packaged frame of 1050 bits.

3.2.3.4. RED/BLACK REM ALPHA BLACK PROCESSING

RED/BLACK REM ALPHA BLACK processing extracts the compressed encrypted bits and Fibonacci sequence from the encrypted channel frames, applies FEC and interleaving to the frames and inserts a header to make the channel packet ready for multiplexing and transmission.

3.2.3.4.1. RED/BLACK REM ALPHA BLACK DATA FRAME

The first 456 bits of the channel data frame shall be made up of the encrypted padding bits, the encrypted mode identification character, and the encrypted compressed information bits extracted from the encrypted channel frame with the Fibonacci bits removed. The next 31 bits shall contain the compressed Fibonacci sequence extracted from the encrypted channel frame in accordance with paragraph 3.2.3.4.2. These shall occupy bit position 457 to 487 of the channel data frame. Bits 488 to 490 shall contain 3 filler bits which shall be set to zero. The final 490 bits of the channel data frame shall contain the LDPC FEC bits coded in accordance with paragraph 3.2.3.4.4.

3.2.3.4.2. RED/BLACK REM ALPHA FIBONACCI BIT COMPRESSION

The Fibonacci sequence shall be extracted from the first 31 characters of the encrypted channel frame. The first 31 Fibonacci bits shall be extracted from the encrypted channel frame and placed in bit positions 457 to 487 of the channel data frame (the first Fibonacci bit shall be placed into position 457). The extracted 31 bit Fibonacci sequence contain sufficient information for the complete Fibonacci sequence to be regenerated by the receive processor. The remaining Fibonacci bits contained within the encrypted channel frame shall be discarded.

3.2.3.4.3. RED/BLACK REM ALPHA FILLER BITS

Three filler bits shall be inserted into the BLACK data frame after the Fibonacci bits. The filler bits shall occupy positions 488 to 490 of the BLACK data frame and shall be set to zero.

3.2.3.4.4. RED/BLACK REM ALPHA LDPC FEC

RED/BLACK REM ALPHA shall use a ½ rate FEC code applied to the channel information prior to interleaving. The FEC code can be decoded using soft decision information from the MSK matched filters during BLACK side processing at the receiver, allowing the correction of bit errors. The FEC coding shall be applied to the 490 information bits contained within the channel data frame (6 encrypted padding bits, 6 encrypted mode identification bits,444 encrypted compressed information bits, 31 Fibonacci sequence bits and 3 filler bits).

3.2.3.4.4.1. RED/BLACK REM ALPHA LDPC FEC CODING

The FEC code shall be generated by the binary multiplication of the information bit frame (considered as a binary column vector) by the binary coding matrix **Equation 1**; this produces a binary column vector (the parity bit vector) which shall be used to fill the FEC parity bit portion of the channel data frame.

$$\begin{bmatrix} M_{1-1}M_{1-2}M_{1-3}.....................M_{1-490} \\ M_{2-1}M_{2-2}M_{2-3}.....................M_{2-490} \\ M_{3-1}M_{3-2}M_{3-3}.....................M_{3-490} \\ . \\ . \\ . \\ M_{490-1}M_{490-2}M_{490-3}.........M_{490-490} \end{bmatrix} \times \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ . \\ . \\ . \\ d_{490} \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ . \\ . \\ . \\ p_{490} \end{bmatrix}$$

EQUATION 1: RED/BLACK ALPHA FEC PARITY GENERATOR MATRIX.

M_{1-1} to $M_{490-490}$ represents the binary generator matrix as defined in ANNEX E. d_1 represents the first bit of information contained within the channel data frame (bit position 1) and d_{490} the last bit of information contained within the channel data frame (the last filler bit at bit position 490). The parity bits generated (p_1 to p_{490}) shall be inserted into the channel data frame immediately after the information bits (p_1 to bit position 491, p_{490} to bit position 980).

3.2.3.4.4.2. RED/BLACK REM ALPHA LDPC FEC DECODING

The receiver may use an LDPC decoder to detect and correct errors. An example decoder is described in ANNEX B and parity equations defined.

3.2.3.4.5. RED/BLACK REM ALPHA CHANNEL DATA FRAME RECEPTION

On reception, the Encrypted Channel Frame shall be built up using the 6 encrypted padding bits, the 6 encrypted mode identification bits, the 444 error corrected Compressed Encrypted Data bits, 444 Filler Bits, all combined with 150 bits of recovered Fibonacci bit sequence. paragraph. The 444 Filler Bits shall all be set to one. The frame shall be packaged into 6 bit characters with a Fibonacci bit inserted in place of a stop bit after each character as shown in Figure 11

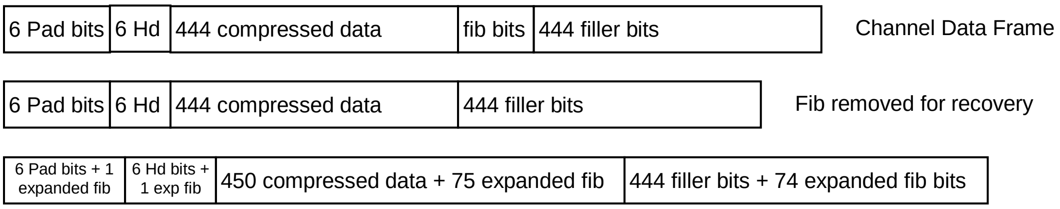


FIGURE 11: SCHEMATIC COMPOSITION OF RED/BLACK BRAVO NATO REM CHANNEL.

3.2.3.5. RED/BLACK REM ALPHA FRAME DATA INTERLEAVING

The channel data frame shall be protected from correlated bursts of bit errors in the received data frame by interleaving the data in accordance with paragraph 3.2.2.5.

3.2.3.6. RED/BLACK REM ALPHA RED/BLACK FRAME HEADER

Before transmission, 70 header bits shall be inserted into the 980 bit channel interleaved frame making a channel packet of 1050 bits. The header bits allow the receiver to determine the exact start of each REM packet (resolving timing offsets and propagation delays) to estimate the phase of the carrier signal over the REM packet to allow coherent demodulation of the binary symbols and to identify the type of REM coding used on the channel.

The RED/BLACK REM ALPHA coding channel interleaved frame header consists of 70 known bits distributed along the frame in seven 10 bit header segments. The values of the seven 10 bit header segments and their position in the channel packet shall be as shown in TABLE 8.

Bit position	1	2	3	4	5	6	7	8	9	10
Content	1	1	1	1	0	0	0	0	1	0
Bit position	151	152	153	154	155	156	157	158	159	160
Content	1	0	0	0	1	1	0	0	0	0
Bit position	301	302	303	304	305	306	307	308	309	310
Content	0	1	0	0	0	0	1	0	1	0
Bit position	451	452	453	454	455	456	457	458	459	460
Content	1	1	0	0	1	0	0	1	0	1
Bit position	601	602	603	604	605	606	607	608	609	610
Content	1	1	0	0	1	1	1	0	1	1
Bit position	751	752	753	754	755	756	757	758	759	760
Content	0	1	1	1	1	1	0	1	1	0

Bit position	901	902	903	904	905	906	907	908	909	910
Content	0	1	0	1	1	1	0	0	0	1

TABLE 8: RED/BLACK ALPHA -BLACK HEADER BIT POSITIONS.

On reception of the channel packet, the 70 header bits shall be removed from the channel packet before the frame is de-interleaved.

3.2.4. RED/BLACK REM BRAVO PACKET STRUCTURE

The BRAVO packet structure and coding scheme has been optimised to increase channel throughput to 75bps whilst keeping the effective range of the VLF broadcast comparable with the NATO modes defined in CHAPTER 2. The coding scheme has been optimised to interface with VALLOR like encryption containing deterministic Fibonacci bits.

3.2.4.1. RED/BLACK REM BRAVO PACKET STRUCTURE DETAIL

RED/BLACK BRAVO REM packets are based on 21 second channel packets for 50 baud channels, giving 1050 bit packets as shown in Figure 12.

RED BRAVO REM processing compresses the broadcast data stream (Figure 12 H) using a text data compression technique based on PPM and arithmetic coding technology. RED Channels Frames (Figure 12 G) are built containing 6 padding bits, 6 REM mode identification bits, 672 compressed broadcast stream bits and 216 filler bits. The filler bits allow for the BLACK FEC bits to be added after encryption.

RED Packaged Frames (Figure 12 F) contains the 900 bit RED Channel Frames (Figure 12 G) which have been packaged into 6 bit ITA2 64-ary style characters by inserting a stop bit after each character (6 bits), adding 150 stop bits. The 1050 bit RED Packaged frames are sent through the link encryption device for encryption.

After encryption the BLACK REM processor takes the 1050 bit Encrypted Channel Frames (Figure 12 E) from the encryption device. Channel Data Frames (Figure 12 D) are made up by extracting the 6 encrypted padding bits, 6 encrypted mode identification bits, 672 encrypted compressed information bits, and the first 31 Fibonacci bits from each Encrypted Channel Frame. The ALPHA Channel data frames are completed by adding 3 filler bits and 263 bits of Low Density Parity Check (LDPC) FEC.

The Channel Interleaved Frame (Figure 12 C) consists of the 980 bit Channel Data Frame whose bits have been shuffled with a fixed, random interleaver as defined in Frame Data Interleaving paragraph 3.2.2.5. Finally the Channel Packet is created by inserting 70 header bits into the Channel Interleaved Frame. The 70 header bits are inserted as 7 sub headers of ten bits each, spread evenly throughout the packet as shown in Figure 12 B.

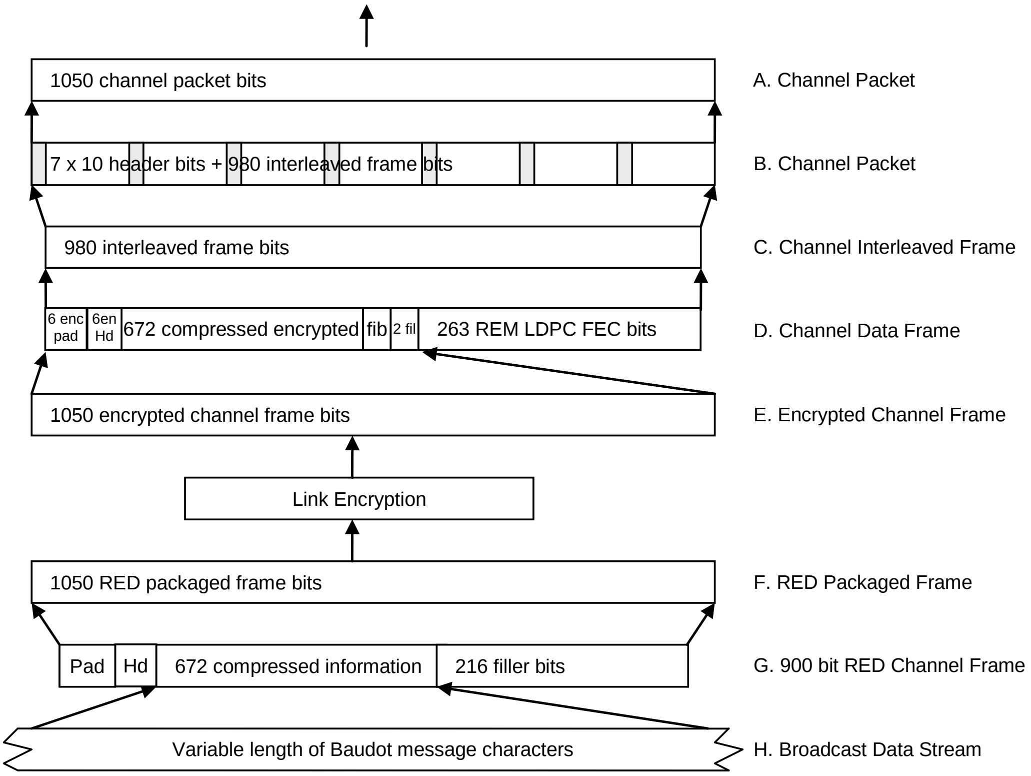

1050 channel packet bits
A. Channel Packet
7 x 10 header bits + 980 interleaved frame bits
B. Channel Packet
980 interleaved frame bits
C. Channel Interleaved Frame
6 enc pad
6en Hd
672 compressed encrypted
fib
2 fil
263 REM LDPC FEC bits
D. Channel Data Frame
1050 encrypted channel frame bits
E. Encrypted Channel Frame
Link Encryption
1050 RED packaged frame bits
F. RED Packaged Frame
Pad
Hd
672 compressed information
216 filler bits
G. 900 bit RED Channel Frame
Variable length of Baudot message characters
H. Broadcast Data Stream

FIGURE 12: SCHEMATIC COMPOSITION OF RED/BLACK BRAVO NATO REM CHANNEL.

3.2.4.2. RED/BLACK REM BRAVO PPM ALGORITHMS

RED/BLACK REM BRAVO uses a PPM algorithm based on an adaptive statistical model of conditional probabilities of the next character given the most recent four character context. The algorithm shall be initialised with a pre-defined fixed model for each output compressed information frame. Characters shall be assessed one at a time from a sequence of characters of the broadcast stream and be compressed using an arithmetic coding scheme based on the conditional probability distribution of the statistical model given the context of the four preceding characters.

3.2.4.2.1. RED/BLACK REM BRAVO PPM COMPRESSION

RED/BLACK REM BRAVO PPM compression coding shall be carried out to be interoperable with the code given in ANNEX C.

3.2.4.2.2. RED/BLACK REM BRAVO PPM EXPANSION

The RED/BLACK REM BRAVO shall be expanded to recover the original broadcast data stream. An example of a suitable expansion algorithm is given in ANNEX D.

3.2.4.2.3. RED/BLACK REM BRAVO STATISTICAL MODEL

Details of the statistical model file and format are included in ANNEX A.

3.2.4.3. RED/BLACK REM BRAVO RED SIDE PACKETISATION

The RED/BLACK REM BRAVO compression algorithm will produce 672 bit frames of compressed data from the broadcast data stream, each frame providing the information for a 21 second channel packet. The compressed data frames shall be packetised before being passed to the link encryption equipment.

The first 6 bit positions of each 900 bit RED channel frame shall contain the paragraph padding bits. These 6 bits shall all be set to zero. The next six bits shall contain a REM mode identification character to allow auto mode detection of RED/BLACK REM frames in the RED REM processor. For RED/BLACK REM BRAVO the mode identification character shall be as defined in **TABLE 9**.

Bit position	7	8	9	10	11	12
Content	0	1	0	0	0	0

TABLE 9: RED/BLACK REM ALPHA MODE IDENTIFICATION CHARACTER

The 672 compressed information bits shall be inserted into the RED channel frame immediately after the mode identification character bits. The compressed information shall be loaded into the 900 bit RED channel frame in the order that it is generated, i.e. the first bit generated shall map to bit 13; the last bit generated shall map to bit 684. The last 216bits of the RED channel frame are filler bits and shall all be set to one.

This RED channel frame shall be packaged into 6 bit ITA2 64-ary characters with a stop bit inserted after each sixth bit. The inserted stop bits shall all be set to one. A total of 150 stop bits will be required making a RED packaged frame of 1050 bits.

3.2.4.4. RED/BLACK REM BRAVO BLACK PROCESSING

RED/BLACK REM BRAVO BLACK processing extracts the compressed encrypted bits and Fibonacci sequence from the encrypted channel frames applies FEC and interleaving to the frames and inserts a header to make the channel packet ready for multiplexing and transmission.

3.2.4.4.1. RED/BLACK REM BRAVO CODING BLACK DATA FRAME

The first 684 bits of the channel data frame shall be made up of the encrypted padding bits, the encrypted mode identification character, and the encrypted compressed information bits extracted from the encrypted channel frame with the Fibonacci bits removed. The next 31 bits shall contain the compressed Fibonacci sequence extracted from the encrypted channel frame in accordance with paragraph 3.2.4.4.2. These shall occupy bit position 685 to 715 of the channel data frame. Bits 716 and 717 shall contain 2 filler bits which shall be set to zero. The final 263 bits of the channel data frame shall contain the LDPC FEC bits coded in accordance with paragraph 3.2.4.4.4.

3.2.4.4.2. RED/BLACK REM BRAVO FIBONACCI BIT COMPRESSION

The Fibonacci sequence shall be extracted from the first 31 characters of the encrypted channel frame. The first 31 Fibonacci bits shall be extracted from the encrypted channel frame and placed in bit positions 685 to 715 of the channel data frame (the first Fibonacci bit shall be placed into position 685). The extracted 31 bit Fibonacci sequence contain sufficient information for the complete Fibonacci sequence to be regenerated by the receive processor. The remaining Fibonacci bits contained within the encrypted channel frame shall be discarded.

3.2.4.4.3. RED/BLACK REM BRAVO FILLER BITS

Two filler bits shall be inserted into the BLACK data frame after the Fibonacci bits. The filler bits shall occupy positions 716 and 717 of the BLACK data frame and shall be set to zero.

3.2.4.4.4. RED/BLACK REM BRAVO LDPC FEC

RED/BLACK REM ALPHA shall use a 0.73 rate FEC code applied to the channel information prior to interleaving. The FEC code can be decoded using soft decision information from the MSK matched filters during BLACK side processing at the receiver, allowing the correction of bit errors. The FEC coding shall be applied to the 717 information bits contained within the channel data frame (6 encrypted padding bits, 6 encrypted mode identification bits,672 encrypted compressed information bits, 31 Fibonacci sequence bits and 2 filler bits).

3.2.4.4.4.1. RED/BLACK REM BRAVO LDPC FEC CODING

The FEC code shall be generated by the binary multiplication of the information bit frame (considered as a binary column vector) by the binary coding matrix shown in **Equation 2**. This produces a binary column vector (the parity bit vector) which shall be used to fill the FEC parity bit portion of the channel data frame.

$$\begin{bmatrix} M_{1-1}M_{1-2}M_{1-3}\ldots\ldots\ldots\ldots\ldots M_{1-717} \\ M_{2-1}M_{2-2}M_{2-3}\ldots\ldots\ldots\ldots\ldots M_{2-717} \\ M_{3-1}M_{3-2}M_{3-3}\ldots\ldots\ldots\ldots\ldots M_{3-717} \\ . \\ . \\ . \\ M_{263-1}M_{263-2}M_{263-3}\ldots\ldots\ldots M_{263-717} \end{bmatrix} \times \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ . \\ . \\ . \\ d_{717} \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ . \\ . \\ . \\ p_{263} \end{bmatrix}$$

EQUATION 2: RED/BLACK BRAVO FEC PARITY GENERATOR MATRIX.

M_{1-1} to $M_{263-717}$ represents the binary generator matrix as defined in ANNEX E. d_1 represents the first bit of information contained within the channel data frame (bit position 1) and d_{717} the last bit of information contained within the channel data frame (the second filler bit at bit position 717) . The parity bits generated (p_1 to p_{263}) shall be inserted into the channel data frame immediately after the information bits (p_1 to bit position 718, p_{263} to bit position 980).

3.2.4.4.4.2. RED/BLACK REM BRAVO LDPC FEC DECODING

The receiver may use an LDPC decoder to detect and correct errors. An example decoder is described in ANNEX B and parity equations defined.

3.2.4.4.5. RED/BLACK REM BRAVO CHANNEL DATA FRAME RECEPTION

On reception, the Encrypted Channel Frame shall be built up using the 6 encrypted padding bits, the 6 encrypted mode identification bits, the 672 error corrected Compressed Encrypted Data bits, 216 Filler Bits combined with 150 bits of recovered Fibonacci bit sequence. paragraph. The 216 Filler Bits shall all be set to one. The frame shall be packaged into 6 bit characters, with a Fibonacci bit inserted in place of a stop bit after each character.

3.2.4.5. RED/BLACK REM BRAVO FRAME DATA INTERLEAVING

The channel data frame shall be protected from correlated bursts of bit errors in the received data frame by interleaving the data in accordance with paragraph 3.2.2.5.

3.2.4.6. RED/BLACK REM BRAVO RED/BLACK FRAME HEADER

Before transmission, 70 header bits shall be inserted into the 980 bit channel interleaved frame making a channel packet of 1050 bits. The header bits allow the receiver to determine the exact start of each REM packet (resolving timing offsets and propagation delays), to estimate the phase of the carrier signal over the REM packet to allow coherent demodulation of the binary symbols, and to identify the type of REM coding used on the channel.

The RED/BLACK REM BRAVO coding channel interleaved frame header consists of 70 known bits distributed along the frame in seven 10 bit header segments. The values of the seven 10 bit header segments and their position in the channel packet shall be as shown in **TABLE 10**.

Bit position	1	2	3	4	5	6	7	8	9	10
Content	1	0	0	0	1	1	1	0	1	0
Bit position	151	152	153	154	155	156	157	158	159	160
Content	0	1	1	0	1	1	1	1	1	0
Bit position	301	302	303	304	305	306	307	308	309	310
Content	1	1	0	1	1	1	0	0	1	1
Bit position	451	452	453	454	455	456	457	458	459	460
Content	1	0	1	0	0	1	0	0	1	1
Bit position	601	602	603	604	605	606	607	608	609	610
Content	0	1	0	1	0	0	0	0	1	0
Bit position	751	752	753	754	755	756	757	758	759	760
Content	0	0	0	0	1	1	0	0	0	1
Bit position	901	902	903	904	905	906	907	908	909	910
Content	0	1	0	0	0	0	1	1	1	1

TABLE 10: RED/BLACK BRAVO BLACK HEADER BIT POSITIONS.

On reception of the channel packet, the 70 header bits shall be removed from the channel packet before the frame is de-interleaved.

3.3. Baseband Processing RED only NATO REM

The following defines additional baseband processing necessary to provide transitional NATO VLF MSK REM. These modes co-exist with existing capability and are compatible with existing in-service equipment, but they do not provide the full performance benefits of a full REM implementation. Access to these modes requires baseband processing functions which are additional to, but do not interfere with, those currently utilized. A channel coded with RED REM technology appears as a VALLOR channel. These REM may be incorporated onto one or more channels of a NATO multichannel broadcast operating in modes N7 or N8.

Different RED only REM packet structures and coding schemes are defined to optimise how the REM performance advantage is realised. The following RED only REM packet structures are defined in this standard:

a. RED only REM ALPHA defined in 3.3.3.3 is optimised for extended range;

b. RED only REM BRAVO defined in 3.3.4.4 is optimised for extended throughput to 75bps per channel.

3.3.1. NATO RED REM MULTICHANNEL MODES

NATO REM VALLOR channels may be used on VLF broadcasts operating multi-channel modes N7 or N8 along with standard non REM channels as defined in Annex A and RED/BLACK REM channels as defined in section 3.2. Advanced agreement will be required between the Broadcast Control Station and the subscriber Receive Platform as to the channel(s) on which RED/BLACK REM, RED only REM and NON REM shall be used.

3.3.1.1. ADDITIONAL RED REM FUNCTIONAL ALLOCATION.

3.3.1.1.1. BROADCAST COORDINATION AUTHORITY (BCA)

The BCA shall perform the functions detailed in paragraph 2.2.3.2.1. In addition, the BCA shall perform the following functions for the RED REM channels:

a. Carry out RED REM processing on broadcast data stream.

b. Present the data packet stream to the assigned crypto system.

3.3.1.1.2. BROADCAST CONTROL STATION (BCS)

The BCS shall perform functions detailed in paragraph 2.2.3.2.2:

3.3.1.1.3. VLF/LF TRANSMITTER COMPLEX

The VLF/LF Transmitter Complex shall perform the functions detailed in paragraph 2.2.3.2.3:

3.3.1.1.4. RECEIVE PLATFORM

The Receive Platform shall perform the following additional functions to those detailed in paragraph 2.2.3.2.4:

a. Identify RED REM coding scheme (if in use) on a channel;

b. Carry out RED REM decoding;

c. Process the decrypted data and output the data to a printer or other peripheral unit.

3.3.2. RED ONLY REM PROCESSING DEFINITIONS

The following sections define the required REM processing functions for encoding a channel with RED only REM for use with NATO multichannel modes N7 and N8.

3.3.2.1. FIXED PACKET STRUCTURE

The RED only REM are based on 21 second packets on 50 baud channels giving 150, VALLOR Encrypted Channel Frame characters per packet as shown in Figure 13 A. The RED Packaged Frame (Figure 13 B) contains a 900 bit Interleaved RED Data Frame (Figure 13 C) which has been packaged into 6 bit ITA2 64-ary characters.

The Interleaved RED Data Frame consists of a 53 bit header and an 847 bit RED Data Frame whose bits have been shuffled with a fixed random interleaver. Contained within the RED Data Frame are the compressed broadcast stream bits (Figure 13 E), generated by a text data compression technique based on Prediction by Partial Match (PPM) and arithmetic coding technology, and the LDPC FEC bits as shown in Figure 13.

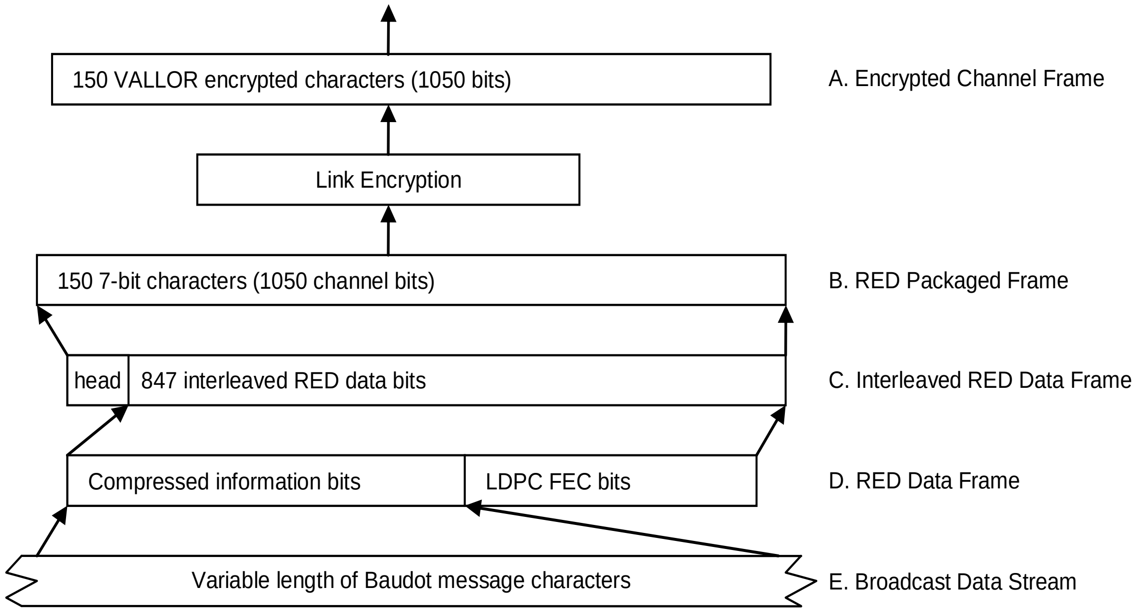

150 VALLOR encrypted characters (1050 bits)
A. Encrypted Channel Frame
Link Encryption
150 7-bit characters (1050 channel bits)
B. RED Packaged Frame
head
847 interleaved RED data bits
C. Interleaved RED Data Frame
Compressed information bits
LDPC FEC bits
D. RED Data Frame
Variable length of Baudot message characters
E. Broadcast Data Stream

FIGURE 13: SCHEMATIC COMPOSITION OF RED NATO REM CHANNEL.

3.3.2.2. COMPRESSION ALGORITHM

The compression algorithm used for the RED REM mode is based upon the algorithm described in paragraph 3.2.2.

3.3.2.3. FORWARD ERROR CORRECTION CODING

Forward error correction coding shall be added to the compressed information bits to protect against bit errors in transmission. The forward error correction code shall be based on LDPC coding.

3.3.2.4. PACKETISATION OF COMPRESSED DATA STREAM

The compressed information bits and the LDPC parity bits shall be packaged into a RED packaged Frame prior to encryption. The RED packaged frame contains a 53 bit header and interleaved information and FEC bits, packaged into 6 bit ITA2 64-ary characters as defined below.

3.3.2.4.1. RED DATA FRAME HEADER

The first 53 bit positions of each 900 bit Interleaved RED Data Frame shall consist of a known frame header sequence. The bit streams have been chosen to have low autocorrelation properties so they can be used to detect and frame channel frames without the need for knowledge of the timing of the channel frame transmission, and to identify the type of RED only REM coding used on the frames. The headers allow the processing of the RED REM frames to be performed off-line in non-real time if needed.

3.3.2.4.2. FRAME DATA INTERLEAVING

The RED data frame is protected from correlated bursts of bit errors in the received data by shuffling the data bits in a random (but fixed) order before encryption. This operation spreads clusters of bit errors (that may be due to strong atmospheric noise impulses) more uniformly throughout the data frame. The 847 bit RED data frame shall be shuffled in accordance with **TABLE 11** (RED data frame bit position => interleaved RED data bit position). Once shuffled the bits shall be inserted into the Interleaved RED Data Frame immediately after the header (bit positions 54 through 900).

1 => 13	2 => 60	3 => 219	4 => 783	5 => 245	6 => 489	7 => 130
8 => 143	9 => 231	10 => 819	11 => 377	12 => 304	13 => 714	14 => 731
15 => 10	16 => 101	17 => 740	18 => 757	19 => 615	20 => 64	21 => 204
22 => 577	23 => 393	24 => 721	25 => 693	26 => 85	27 => 62	28 => 408
29 => 25	30 => 432	31 => 309	32 => 90	33 => 712	34 => 481	35 => 298
36 => 480	37 => 771	38 => 127	39 => 558	40 => 626	41 => 730	42 => 110
43 => 772	44 => 420	45 => 272	46 => 325	47 => 622	48 => 605	49 => 774
50 => 665	51 => 637	52 => 228	53 => 224	54 => 179	55 => 500	56 => 29
57 => 845	58 => 453	59 => 741	60 => 750	61 => 625	62 => 535	63 => 194
64 => 591	65 => 95	66 => 320	67 => 235	68 => 293	69 => 629	70 => 548
71 => 549	72 => 597	73 => 218	74 => 267	75 => 816	76 => 550	77 => 237
78 => 217	79 => 718	80 => 47	81 => 368	82 => 292	83 => 787	84 => 675
85 => 294	86 => 388	87 => 703	88 => 635	89 => 544	90 => 398	91 => 149
92 => 668	93 => 801	94 => 215	95 => 502	96 => 747	97 => 582	98 => 528
99 => 65	100 => 546	101 => 116	102 => 144	103 => 846	104 => 436	105 => 256
106 => 532	107 => 392	108 => 759	109 => 385	110 => 343	111 => 513	112 => 699
113 => 788	114 => 174	115 => 482	116 => 818	117 => 766	118 => 828	119 => 163
120 => 554	121 => 319	122 => 387	123 => 321	124 => 129	125 => 421	126 => 341
127 => 770	128 => 120	129 => 107	130 => 115	131 => 465	132 => 156	133 => 386
134 => 589	135 => 423	136 => 488	137 => 201	138 => 826	139 => 817	140 => 663
141 => 300	142 => 11	143 => 383	144 => 685	145 => 367	146 => 710	147 => 81
148 => 696	149 => 782	150 => 683	151 => 49	152 => 576	153 => 205	154 => 117
155 => 521	156 => 510	157 => 376	158 => 736	159 => 100	160 => 93	161 => 739
162 => 162	163 => 66	164 => 844	165 => 20	166 => 389	167 => 324	168 => 505
169 => 169	170 => 464	171 => 760	172 => 491	173 => 69	174 => 422	175 => 483
176 => 102	177 => 581	178 => 51	179 => 164	180 => 522	181 => 834	182 => 538
183 => 512	184 => 711	185 => 45	186 => 223	187 => 630	188 => 259	189 => 355
190 => 437	191 => 494	192 => 539	193 => 575	194 => 378	195 => 667	196 => 519
197 => 190	198 => 176	199 => 340	200 => 659	201 => 42	202 => 128	203 => 784
204 => 349	205 => 624	206 => 346	207 => 536	208 => 80	209 => 28	210 => 623
211 => 611	212 => 776	213 => 98	214 => 779	215 => 220	216 => 455	217 => 545
218 => 265	219 => 795	220 => 222	221 => 446	222 => 702	223 => 314	224 => 157
225 => 682	226 => 412	227 => 732	228 => 199	229 => 287	230 => 71	231 => 847
232 => 111	233 => 781	234 => 655	235 => 59	236 => 447	237 => 837	238 => 508
239 => 172	240 => 415	241 => 396	242 => 373	243 => 648	244 => 461	245 => 600

246 => 588	247 => 236	248 => 706	249 => 4	250 => 38	251 => 672	252 => 486
253 => 485	254 => 394	255 => 22	256 => 514	257 => 275	258 => 612	259 => 835
260 => 305	261 => 374	262 => 141	263 => 786	264 => 159	265 => 427	266 => 644
267 => 137	268 => 244	269 => 516	270 => 121	271 => 359	272 => 234	273 => 724
274 => 586	275 => 345	276 => 476	277 => 365	278 => 357	279 => 670	280 => 742
281 => 799	282 => 197	283 => 646	284 => 662	285 => 507	286 => 290	287 => 404
288 => 517	289 => 195	290 => 429	291 => 39	292 => 139	293 => 720	294 => 627
295 => 526	296 => 43	297 => 269	298 => 403	299 => 448	300 => 527	301 => 524
302 => 746	303 => 17	304 => 44	305 => 503	306 => 154	307 => 409	308 => 807
309 => 322	310 => 332	311 => 354	312 => 299	313 => 89	314 => 268	315 => 825
316 => 533	317 => 34	318 => 604	319 => 559	320 => 830	321 => 578	322 => 416
323 => 118	324 => 166	325 => 492	326 => 574	327 => 363	328 => 506	329 => 273
330 => 74	331 => 240	332 => 400	333 => 768	334 => 97	335 => 449	336 => 278
337 => 515	338 => 451	339 => 285	340 => 719	341 => 681	342 => 136	343 => 424
344 => 391	345 => 614	346 => 198	347 => 82	348 => 303	349 => 79	350 => 338
351 => 580	352 => 518	353 => 439	354 => 249	355 => 3	356 => 495	357 => 414
358 => 657	359 => 610	360 => 186	361 => 123	362 => 291	363 => 358	364 => 594
365 => 288	366 => 329	367 => 609	368 => 744	369 => 109	370 => 700	371 => 61
372 => 7	373 => 761	374 => 181	375 => 78	376 => 616	377 => 140	378 => 785
379 => 471	380 => 27	381 => 701	382 => 397	383 => 628	384 => 767	385 => 251
386 => 803	387 => 277	388 => 188	389 => 765	390 => 8	391 => 353	392 => 230
393 => 152	394 => 763	395 => 792	396 => 474	397 => 442	398 => 202	399 => 87
400 => 161	401 => 70	402 => 804	403 => 142	404 => 413	405 => 276	406 => 595
407 => 56	408 => 496	409 => 48	410 => 280	411 => 676	412 => 134	413 => 227
414 => 177	415 => 573	416 => 431	417 => 253	418 => 641	419 => 212	420 => 572
421 => 348	422 => 557	423 => 822	424 => 232	425 => 178	426 => 543	427 => 342
428 => 407	429 => 640	430 => 246	431 => 296	432 => 336	433 => 99	434 => 726
435 => 832	436 => 279	437 => 242	438 => 705	439 => 381	440 => 808	441 => 499
442 => 334	443 => 384	444 => 585	445 => 257	446 => 777	447 => 289	448 => 114
449 => 645	450 => 671	451 => 91	452 => 806	453 => 213	454 => 643	455 => 751
456 => 113	457 => 77	458 => 654	459 => 16	460 => 450	461 => 243	462 => 707
463 => 318	464 => 297	465 => 769	466 => 192	467 => 435	468 => 689	469 => 498
470 => 18	471 => 5	472 => 606	473 => 372	474 => 590	475 => 738	476 => 715
477 => 454	478 => 308	479 => 6	480 => 468	481 => 248	482 => 203	483 => 53
484 => 466	485 => 313	486 => 633	487 => 733	488 => 119	489 => 124	490 => 239
491 => 592	492 => 504	493 => 690	494 => 477	495 => 583	496 => 820	497 => 509
498 => 443	499 => 725	500 => 797	501 => 327	502 => 132	503 => 570	504 => 333
505 => 19	506 => 282	507 => 209	508 => 764	509 => 67	510 => 339	511 => 778
512 => 266	513 => 344	514 => 182	515 => 842	516 => 479	517 => 619	518 => 743
519 => 511	520 => 684	521 => 441	522 => 21	523 => 330	524 => 749	525 => 656
526 => 145	527 => 328	528 => 762	529 => 284	530 => 317	531 => 802	532 => 58
533 => 745	534 => 618	535 => 472	536 => 173	537 => 428	538 => 264	539 => 283

540 => 547	541 => 473	542 => 520	543 => 364	544 => 270	545 => 323	546 => 729
547 => 352	548 => 200	549 => 255	550 => 793	551 => 252	552 => 679	553 => 790
554 => 52	555 => 274	556 => 151	557 => 362	558 => 57	559 => 281	560 => 798
561 => 88	562 => 571	563 => 40	564 => 569	565 => 607	566 => 540	567 => 530
568 => 652	569 => 301	570 => 708	571 => 560	572 => 470	573 => 104	574 => 411
575 => 405	576 => 673	577 => 556	578 => 165	579 => 50	580 => 634	581 => 440
582 => 661	583 => 620	584 => 286	585 => 430	586 => 617	587 => 402	588 => 401
589 => 31	590 => 148	591 => 780	592 => 796	593 => 553	594 => 456	595 => 650
596 => 241	597 => 791	598 => 469	599 => 258	600 => 187	601 => 814	602 => 596
603 => 211	604 => 306	605 => 452	606 => 680	607 => 94	608 => 263	609 => 497
610 => 191	611 => 686	612 => 537	613 => 83	614 => 96	615 => 261	616 => 36
617 => 76	618 => 189	619 => 307	620 => 737	621 => 567	622 => 631	623 => 1
624 => 833	625 => 552	626 => 214	627 => 23	628 => 459	629 => 221	630 => 812
631 => 551	632 => 146	633 => 247	634 => 692	635 => 375	636 => 33	637 => 598
638 => 55	639 => 636	640 => 331	641 => 821	642 => 716	643 => 84	644 => 155
645 => 713	646 => 603	647 => 653	648 => 755	649 => 843	650 => 30	651 => 666
652 => 758	653 => 584	654 => 92	655 => 316	656 => 382	657 => 593	658 => 315
659 => 434	660 => 831	661 => 226	662 => 697	663 => 565	664 => 800	665 => 15
666 => 677	667 => 789	668 => 295	669 => 54	670 => 185	671 => 216	672 => 722
673 => 138	674 => 599	675 => 824	676 => 369	677 => 233	678 => 75	679 => 41
680 => 651	681 => 312	682 => 125	683 => 674	684 => 193	685 => 160	686 => 639
687 => 602	688 => 691	689 => 754	690 => 587	691 => 12	692 => 501	693 => 335
694 => 170	695 => 366	696 => 531	697 => 37	698 => 208	699 => 525	700 => 664
701 => 563	702 => 564	703 => 756	704 => 493	705 => 410	706 => 805	707 => 171
708 => 150	709 => 813	710 => 395	711 => 717	712 => 838	713 => 206	714 => 63
715 => 207	716 => 112	717 => 147	718 => 379	719 => 351	720 => 350	721 => 133
722 => 734	723 => 463	724 => 426	725 => 727	726 => 561	727 => 775	728 => 647
729 => 467	730 => 541	731 => 302	732 => 698	733 => 35	734 => 460	735 => 370
736 => 748	737 => 356	738 => 419	739 => 773	740 => 669	741 => 229	742 => 562
743 => 735	744 => 433	745 => 417	746 => 490	747 => 9	748 => 425	749 => 613
750 => 86	751 => 254	752 => 823	753 => 678	754 => 103	755 => 399	756 => 840
757 => 26	758 => 753	759 => 180	760 => 444	761 => 105	762 => 175	763 => 810
764 => 14	765 => 122	766 => 72	767 => 361	768 => 380	769 => 829	770 => 555
771 => 534	772 => 326	773 => 815	774 => 262	775 => 158	776 => 458	777 => 438
778 => 108	779 => 809	780 => 688	781 => 184	782 => 658	783 => 418	784 => 632
785 => 687	786 => 601	787 => 73	788 => 523	789 => 196	790 => 487	791 => 642
792 => 694	793 => 827	794 => 579	795 => 126	796 => 568	797 => 704	798 => 811
799 => 841	800 => 638	801 => 238	802 => 183	803 => 337	804 => 68	805 => 46
806 => 621	807 => 347	808 => 250	809 => 566	810 => 225	811 => 608	812 => 478
813 => 168	814 => 106	815 => 271	816 => 2	817 => 445	818 => 406	819 => 24
820 => 260	821 => 752	822 => 153	823 => 839	824 => 457	825 => 794	826 => 462
827 => 695	828 => 836	829 => 167	830 => 310	831 => 210	832 => 728	833 => 135

834 => 660	835 => 709	836 => 649	837 => 371	838 => 311	839 => 390	840 => 475
841 => 723	842 => 542	843 => 360	844 => 529	845 => 484	846 => 131	847 => 32

TABLE 11: RED CHANNEL INTERLEAVING

At the receiver the interleaved data frame shall be shuffled using the reverse operation once the 53 header bits have been stripped off the frame.

3.3.2.4.3. RED PACKAGED FRAME

The RED Interleaved Data Frame shall be packaged into 6 bit ITA2 64-ary characters, starting at bit position 1, with stop bits inserted after each character (after every six bits). The inserted stop bits shall be set to one. A total of 150 stop bits will be required, taking a RED packaged frame of 1050 bits. The RED Packaged Frame shall be set to the encryption device starting with bit position 1.

3.3.2.5. CHANNEL MULTIPLEXING

3.3.2.5.1. RED ONLY REM CHANNEL IN NATO MULTICHANNEL MODES N7 AND N8

For NATO modes N7 and N8, RED only REM channels shall be time division multiplexed onto a broadcast data stream in accordance with paragraph 2.2.3.3.5.4. To maintain frame synchronisation Channel 1 shall be a VALLOR encrypted channel. This may be non-REM or RED only REM. If a NATO multichannel broadcast contains only non-REM or RED only REM channel, time of day synchronisation of the broadcast, in accordance with paragraph 3.2.1.1, is not required.

3.3.3. RED ONLY REM ALPHA PACKET STRUCTURE

The ALPHA packet structure and coding scheme has been optimised to increase the effective broadcast coverage area, whilst keeping the channel throughput at approximately 50bps. The packet structure has been optimised to interface with VALLOR like encryption containing deterministic Fibonacci bits.

3.3.3.1. RED ONLY REM ALPHA PACKET STRUCTURE DETAIL

The RED only REM ALPHA packet structure is based on 21 second packets on 50 baud channels giving 150 VALLOR Encrypted Channel Frame characters per packet as shown in Figure 12 A.

The 847 bit RED Data Frame contains 484 bits of compressed broadcast stream (Figure 12 E), generated by a text data compression technique based on Prediction by Partial Match (PPM) and arithmetic coding technology, and 363 bits of LDPC FEC bits as shown in Figure 12 D.

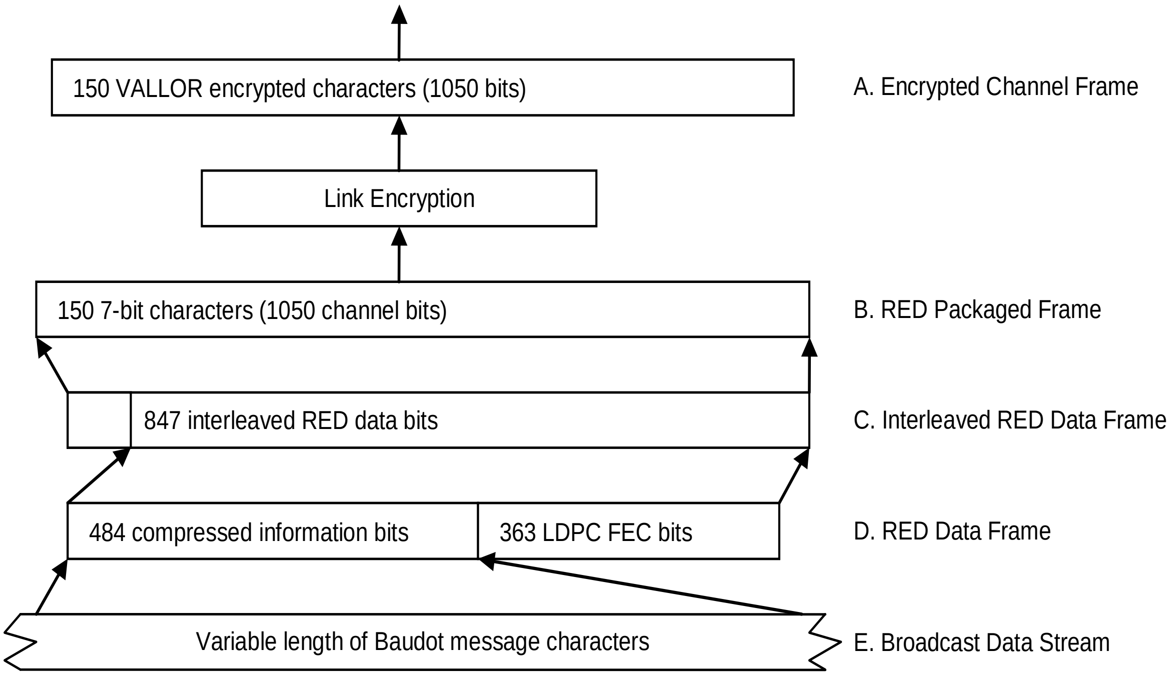

150 VALLOR encrypted characters (1050 bits)
A. Encrypted Channel Frame
Link Encryption
150 7-bit characters (1050 channel bits)
B. RED Packaged Frame
847 interleaved RED data bits
C. Interleaved RED Data Frame
484 compressed information bits
363 LDPC FEC bits
D. RED Data Frame
Variable length of Baudot message characters
E. Broadcast Data Stream

FIGURE 14: SCHEMATIC COMPOSITION OF RED ALPHA NATO REM CHANNEL.

3.3.3.2. RED ONLY REM ALPHA RED DATA FRAME STRUCTURE

The 847 RED Data Frame shall contain the compressed message data and FEC coding bits. The first 484 bits of the RED Data Frame shall be made up of the 484 compressed information bits. These shall be loaded into the frame as they are generated; the first bit generated by the text compression process shall occupy bit one of the RED Data Frame, and the frame shall then be filled sequentially to bit position 484. The FEC coding bits shall follow the compressed information occupying bit positions 485 to 847.

3.3.3.3. RED ONLY REM ALPHA PPM ALGORITHMS

RED only REM ALPHA uses a PPM algorithm based on an adaptive statistical model of conditional probabilities of the next character given the most recent four character context. The algorithm shall be initialised with a pre-defined, fixed model for each compressed information frame. Characters shall be assessed one at a time from a sequence of 6-bit characters on the broadcast stream and be compressed using an arithmetic coding scheme based on the conditional probability distribution of the statistical modem given the context of the four preceding characters.

3.3.3.3.1. RED ONLY REM ALPHA PPM COMPRESSION

The RED only REM ALPHA PPM compression coding shall be carried out to be interoperable with the code given in ANNEX C.

3.3.3.3.2. RED ONLY REM ALPHA PPM EXPANSION

The RED only REM ALPHA packets shall be expanded to recover the original broadcast data stream. An example of a suitable expansion algorithm is given in ANNEX D.

3.3.3.3.3. RED ONLY REM ALPHA STATISTICAL MODEL

Details of the statistical model file and format are included in ANNEX A.

3.3.3.4. RED ONLY REM ALPHA LDPC FEC

The RED only REM ALPHA packet structure shall be protected against bit errors in transmission by a 0.57 rate LDPC FEC code.

3.3.3.4.1. RED ONLY REM ALPHA LDPC FEC CODING

The FEC code shall be applied to the 484 compressed information bits. The FEC code shall be generated by the binary multiplication of the information bits (considered as a binary column vector) by the binary coding matrix defined in **Equation 3**. This produces a binary column vector (the parity bit vector) which shall be used to fill the FEC parity bit portion of the RED data frame.

$$\begin{bmatrix} M_{1-1}M_{1-2}M_{1-3}\ldots\ldots\ldots\ldots\ldots\ldots M_{1-484} \\ M_{2-1}M_{2-2}M_{2-3}\ldots\ldots\ldots\ldots\ldots\ldots M_{2-484} \\ M_{3-1}M_{3-2}M_{3-3}\ldots\ldots\ldots\ldots\ldots\ldots M_{3-484} \\ . \\ . \\ . \\ M_{363-1}M_{363-2}M_{363-3}\ldots\ldots\ldots M_{363-484} \end{bmatrix} \times \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ . \\ . \\ . \\ . \\ . \\ . \\ d_{484} \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ . \\ . \\ . \\ p_{363} \end{bmatrix}$$

EQUATION 3: RED LDPC FEC GENERATOR MATRIX.

M_{1-1} to $M_{363-484}$ represents the binary generator matrix as defined in ANNEX E. d_1 represents the first bit of information contained within the RED data frame (bit position 1) and d_{484} the last information bit contained within the RED data frame (bit position 484). The parity bits generated (p_1 to p_{363}) shall be inserted into the channel data frame immediately after the information bits (p_1 to bit position 485, p_{363} to bit position 847).

3.3.3.4.2. RED ONLY REM ALPHA LDPC FEC DECODING

An LDPC decoder may be used to detect and correct errors. An example decoder is described in ANNEX B and parity equations defined.

3.3.3.5. RED ONLY REM ALPHA RED DATA FRAME HEADER

The first 53 bit positions of each 900 bit Interleaved RED Data Frame shall consist of a known frame header sequence. RED only REM ALPHA coding shall use the header bits defined in **TABLE 12**.

Bit position	1	2	3	4	5	6	7	8	9	10	11
Content	1	1	1	0	0	1	1	1	1	0	1
Bit position	12	13	14	15	16	17	18	19	20	21	22
Content	0	0	1	1	0	1	1	1	0	1	0
Bit position	23	24	25	26	27	28	29	30	31	32	33

Content	1	1	0	1	1	0	1	0	0	0	1
Bit position	34	35	36	37	38	39	40	41	42	43	44
Content	1	0	1	1	1	0	1	0	0	0	0
Bit position	45	46	47	48	49	50	51	52	53		
Content	0	0	1	0	0	1	1	0	0		

TABLE 12: RED ALPHA REM HEADER BIT POSITIONS.

3.3.4. RED only REM BRAVO packet structure

The BRAVO packet structure and coding scheme has been optimised to increase channel throughput to approximately 75bps whilst keeping the effective range of the VLF broadcast comparable with the NATO modes defined in CHAPTER 2. The coding scheme has been optimised to interface with VALLOR like encryption containing deterministic Fibonacci bits.

3.3.4.1. RED ONLY REM BRAVO CODING PACKET STRUCTURE DETAIL

The RED only REM BRAVO packet structure is based on 21 second packets on 50 baud channels giving 150, VALLOR Encrypted Channel Frame characters per packet as shown in Figure 14 A.

The 847 bit RED Data Frame contains 681 bits of compressed broadcast stream (Figure 14 E), generated by a text data compression technique based on Prediction by Partial Match (PPM) and arithmetic coding technology and 166 bits of LDPC FEC bits as shown in Figure 14 D.

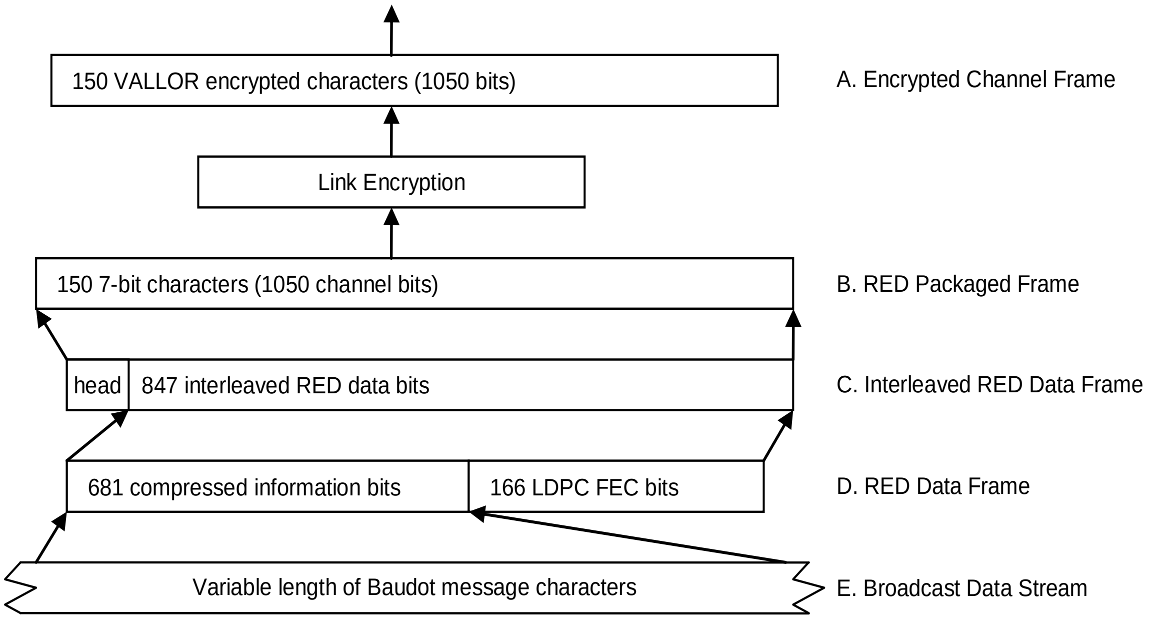

150 VALLOR encrypted characters (1050 bits)
A. Encrypted Channel Frame
Link Encryption
150 7-bit characters (1050 channel bits)
B. RED Packaged Frame
head
847 interleaved RED data bits
C. Interleaved RED Data Frame
681 compressed information bits
166 LDPC FEC bits
D. RED Data Frame
Variable length of Baudot message characters
E. Broadcast Data Stream

FIGURE 15: SCHEMATIC COMPOSITION OF RED BRAVO NATO REM CHANNEL.

3.3.4.2. RED ONLY REM BRAVO RED DATA FRAME STRUCTURE

The 847 RED Data Frame shall contain the compressed message data and FEC coding bits. The first 681 bits of the RED Data Frame shall be made up of the 681 compressed information bits. These shall be loaded into the frame as they are generated; the first bit generated by the text compression process shall occupy bit one of the RED Data Frame and the frame shall then be filled sequentially to bit position 681. The FEC coding bits shall follow the compressed information occupying bit positions 682 to 847.

3.3.4.3. RED ONLY REM BRAVO PPM ALGORITHMS

RED only REM BRAVO uses a PPM algorithm based on an adaptive statistical model of conditional probabilities of the next character given the most recent four character context. The algorithm shall be initialised with a pre-defined fixed model for each compressed information frame. Characters shall be assessed one at a time from a sequence of 6-bit characters on the broadcast stream and be compressed using an arithmetic coding scheme based on the conditional probability distribution of the statistical modem given the context of the four preceding characters.

3.3.4.3.1. RED ONLY REM BRAVO PPM COMPRESSION

The RED only REM BRAVO PPM compression coding shall be carried out to be interoperable with the code given in ANNEX C.

3.3.4.3.2. RED ONLY REM BRAVO PPM EXPANSION

The RED only REM BRAVO packets shall be expanded to recover the original broadcast data stream. An example of a suitable expansion algorithm is given in ANNEX D

3.3.4.3.3. RED ONLY REM BRAVO STATISTICAL MODEL

Details of the statistical model file and format are included in ANNEX A.

3.3.4.4. RED ONLY REM BRAVO LDPC FEC

The RED only REM BRAVO packet structure shall be protected against bit errors in transmission by a 0.78 rate LDPC FEC code.

3.3.4.4.1. RED ONLY REM BRAVO CODING LDPC FEC CODING

The FEC code shall be applied to the 681 compressed information bits. The FEC code shall be generated by the binary multiplication of the information bits (considered as a binary column vector) by the binary coding matrix defined in **Equation 4** This produces a binary column vector (the parity bit vector) which shall be used to fill the FEC parity bit portion of the RED data frame.

$$\begin{bmatrix} M_{1-1}M_{1-2}M_{1-3}\ldots\ldots\ldots\ldots\ldots\ldots M_{1-681} \\ M_{2-1}M_{2-2}M_{2-3}\ldots\ldots\ldots\ldots\ldots\ldots M_{2-681} \\ M_{3-1}M_{3-2}M_{3-3}\ldots\ldots\ldots\ldots\ldots\ldots M_{3-681} \\ . \\ . \\ . \\ M_{166-1}M_{166-2}M_{166-3}\ldots\ldots\ldots M_{166-681} \end{bmatrix} \times \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ . \\ . \\ . \\ . \\ . \\ . \\ d_{681} \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ . \\ . \\ . \\ p_{166} \end{bmatrix}$$

EQUATION 4: RED LDPC FEC GENERATOR MATRIX.

M_{1-1} to $M_{166-681}$ represents the binary generator matrix as defined in ANNEX E. d_1 represents the first bit of information contained within the RED data frame (bit position 1) and d_{681} the last information bit contained within the RED data frame (bit position 681). The parity bits generated (p_1 to p_{166}) shall be inserted into the channel data frame immediately after the information bits (p_1 to bit position 682 p_{166} to bit position 847).

3.3.4.4.2. RED ONLY REM BRAVO LDPC FEC DECODING

An LDPC decoder may be used to detect and correct errors. An example decoder is described in ANNEX B where the necessary parity equations are defined.

3.3.4.5. RED ONLY REM BRAVO RED DATA FRAME HEADER

The first 53 bit positions of each 900 bit Interleaved RED Data Frame shall consist of a known frame header sequence. RED only REM BRAVO coding shall use the header bits defined in **TABLE 13**.

Bit position	1	2	3	4	5	6	7	8	9	10	11
Content	1	0	1	1	0	0	1	0	1	1	1
Bit position	12	13	14	15	16	17	18	19	20	21	22
Content	1	0	0	1	1	1	0	1	1	1	1
Bit position	23	24	25	26	27	28	29	30	31	32	33
Content	1	0	0	0	1	1	1	1	0	1	1
Bit position	34	35	36	37	38	39	40	41	42	43	44
Content	0	0	0	1	0	0	0	0	1	0	1
Bit position	45	46	47	48	49	50	51	52	53		
Content	0	1	1	1	0	0	1	1	0		

TABLE 13: RED BRAVO REM HEADER BIT POSITIONS.

CHAPTER 4 NATO REM PERFORMANCE REQUIREMENTS

This Annex defines the performance required of receive systems operating in NATO REM modes. The chapter defines a standard test environment within which VLF receivers and REM processing systems are to be tested and defines the associated required performance characteristics.

4.1. NON-REM RECEIVER PERFORMANCE

4.1.1. Non-REM receiver performance testing

The REM modes of CHAPTER 3 include performance requirements on the VLF receive system in addition to those defined in CHAPTER 2. These requirements shall be met without the inclusion of REM processing. The performance of the receive system shall be tested with a set of defined test signals.

4.2. Non-REM test signals

The test signals contain a simulated MSK broadcast in a defined noise environment. The signal broadcast contains a VALLOR encrypted channel 1 for receiver synchronisation and Plain Language test messages on the remaining channels for producing the character error rates. Test broadcasts are provided for MSK4 modulation. Test broadcasts are provided for three different noise environments: Gaussian noise, atmospheric noise and a broadcast containing adjacent channel transmissions. In each test environment the test broadcast shall be transmitted on the frequency defined. **Figure 16** shows the test configuration.

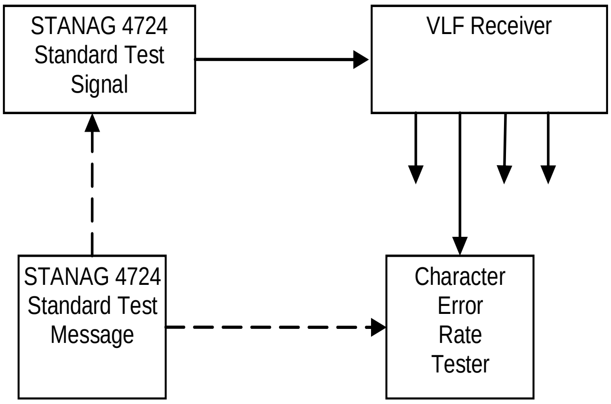


FIGURE 16: NON-REM RECEIVER PERFORMANCE TESTING CONFIGURATION.

The VALLOR channel 1 allows the receiver to synchronise on the broadcast data stream and correctly decode the channels. The plain language channels are to be used for character error rate testing. The test signals are provided as .WAV files encoded at 96kHz / 24bit.

The test signals provided for non-REM receiver performance testing are as defined in **TABLE 14** below.

Filename MD5 Checksum	Mode	Additive Noise	Broadcast test frequency
STANAG4724TestFile1.wav	N5	Gaussian	19.0kHz
STANAG4724TestFile2.wav	N5	Atmospheric	28.0kHz
STANAG4724TestFile3.wav	N5	Adjacent Channel	21.22kHz

TABLE 14: RECEIVER PERFORMANCE TEST MESSAGE.

4.2.1.1. STANAG 4724 STANDARD TEST MESSAGE

The test message used on the plain language channels of the test broadcasts is defined in **TABLE 15** below.

Character	1	2	3	4	5	6	7	8	9	10
Content	S	T	A	N	A	G	FIGS	4	7	2
Character	11	12	13	14	15	16	17	18	19	20
Content	4	space	LTRS	T	E	S	T	space	M	E
Character	21	22	23	24	25	26	27	28	29	30
Content	S	S	A	G	E	space	FIGS	0	1	2
Character	31	32	33	34	35	36	37	38	39	40
Content	3	4	5	6	7	8	9	CR	CR	LF
Character	41	42	43	44	45	46	47	48	49	50
Content	LTRS	T	H	E	space	Q	U	I	C	K
Character	51	52	53	54	55	56	57	58	59	60
Content	space	B	R	O	W	N	space	F	O	X
Character	61	62	63	64	65	66	67	68	69	70
Content	space	J	U	M	P	S	space	O	V	E
Character	71	72	73	74	75	76	77	78	79	80
Content	R	space	T	H	E	space	L	A	Z	Y
Character	80	82	83	84	85	86	87			
Content	space	D	O	G	CR	CR	LF			

TABLE 15: RECEIVER PERFORMANCE TEST MESSAGE.

The characters are 7 unit 32-ary ITA2 characters transmitted in accordance with paragraph 2.2.2.2.1. After the 87 characters defined in TABLE 15 the message shall be repeated starting at character 1.

4.2.1.2. NON-REM RECEIVER PERFORMANCE TEST REQUIREMENTS

Character error rates shall be determined from either channel 2, 3 or 4 of the MSK4 test broadcasts.

4.2.1.2.1. N6 GAUSSIAN NOISE PERFORMANCE

The minimum performance requirements for a STANAG 4724 VLF receiver receiving the STANAG 4724 test file 1 is for a character error rate no worse than 1 in 1000.

4.2.1.2.2. N6 ATMOSPHERIC NOISE PERFORMANCE

The minimum performance requirements for a STANAG 4724 VLF receiver receiving the STANAG 4724 test file 2 is for a character error rate no worse than 1 in 1000.

4.2.1.2.3. N6 ADJACENT CHANNEL PERFORMANCE

The minimum performance requirements for a STANAG 4724 VLF receiver receiving the STANAG 4724 test file 3 is for a character error rate no worse than 1 in 1000.

4.2.2. REM performance testing

4.2.2.1. RED ONLY REM PERFORMANCE TESTING

In addition to the non-REM performance tests of paragraph 4.1.1 above, RED only REM system shall be capable of correctly decoding a set of defined test data streams. These data streams consist of RED Packaged Frames of a defined test message stream, encoded with the RED only REM processing. The data stream shall be injected into the RED REM processor in place of the decrypted data stream as shown in **Figure 17**.

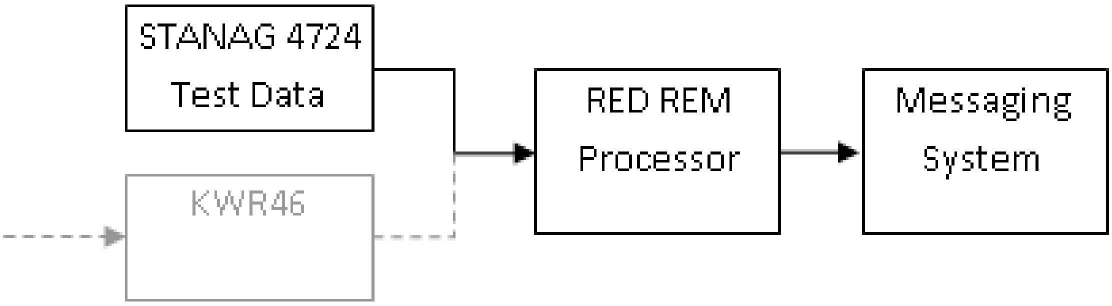


FIGURE 17: RED ONLY REM CER PERFORMANCE TESTING CONFIGURATION.

4.2.2.2. RED ONLY REM TEST DATA STREAM

The RED only test message stream consists of the STANAG 4724, test message defined in paragraph 4.2.1.1, repeated 100 times. The test data streams consist of the message stream compressed, encoded, interleaved and packaged into 64-ary ITA 2 character RED packaged frames using the defined RED only coding schemes. A test data stream is provided for each RED only REM coding scheme defined within the standard. The test data streams are defined in TABLE 16.

File Name MD5 Checksum	RED only coding
REDAlphaTest.dat	Alpha
REDBravoTest.dat	Bravo

TABLE 16: RED ONLY REM STANDARD NOISE FILES.

4.2.2.2.1. RED ONLY TEST FILE FORMAT

The RED only REM test files contain the 64-ary ITA 2 character red data stream stored as 8 bit binary characters. The most significant bit of each 8 bit character is set to 0; the remaining bits contain the 64-ary ITA2 character ordered as per **Figure 18**.

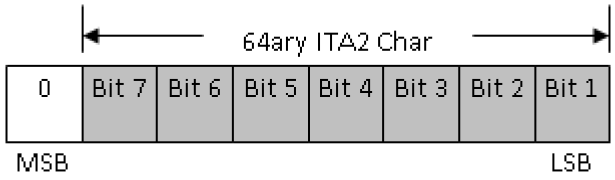


FIGURE 18: RED ONLY TEST FILE FORMAT.

4.2.2.3. RED ONLY REM TESTING

When a test data stream is injected into the RED only REM processor the processor shall decode the data stream using the correct processing definitions and output the RED only REM test data stream without error. This shall be conducted for each RED coding scheme implemented within the processor.

4.2.2.4. RED/BLACK REM TESTING

A similar pre-defined signal method for RED/BLACK REM testing is not currently considered to be practicable due to the link encryption (and need for associated keymat) requirement between the RED and BLACK processors on both APLHA and BRAVO modes.

ANNEX A. STATISTICAL MODEL

A.1. STATISTICAL MODEL FORMAT

A fixed, pre-defined initial statistical model is included with this STANAG. This is a binary file that contains conditional probabilities for arithmetic encoding of characters based on a large representative message data set. The statistical model file is required for the compression and expansion of the compressed data stream. The statistical model is reloaded by the compress/expand process at the start of each packet.

A.2. STATISTICAL MODEL FILE

The following file shall be used as the standard statistical model for RED/BLACK ALPHA REM, RED/BLACK BRAVO REM, RED ALPHA REM and RED BRAVO REM.

- STANAG4724StatisticalModel.cmp
- MD5 Checksum: i.a.w. finalised StatisticalModel.cmp

ANNEX B. LDPC FEC DECODING

B.1. LDPC DECODER OPERATION

A set of parity equations may be used to detect bit errors within the received frames. The input frame to the LDPC decoder is the receiver soft decision likelihood values for each received bit of the FEC encoded frame. The decoder uses this frame and knowledge of the parity equations to iteratively estimate the correct values for the data. If the parity equations can be satisfied there remains only a very small probability that the frame contains undetectable bit errors.

The decoder maintains two versions of the parity matrix elements:

- The Horizontal matrix HL(i,j);
- The Vertical matrix B(I,j), where i indexes the parity equations and j indexes the frame bit.

The decoder iteratively updates the HL matrix and the B matrix until the parity equations are satisfied or a maximum number of iterations have been reached without convergence. Two additional matrices are defined that allow for irregular codes. The Horizontal Index matrix and the Vertical Index matrix define the lengths of each row and column in the horizontal and vertical matrix where the code is irregular. The ALPHA codes defined in this STANAG are regular and the BRAVO codes are only irregular in horizontal matrix; thus only the row index matrix is required for the BRAVO codes. **Figure 19** shows an overview of the operation of the decoder.

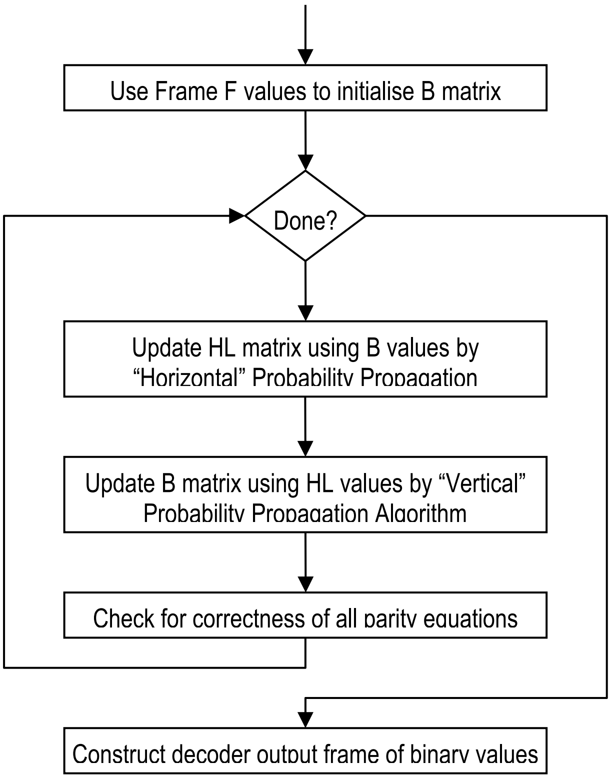

Use Frame F values to initialise B matrix
Done?
Update HL matrix using B values by "Horizontal" Probability Propagation
Update B matrix using HL values by "Vertical" Probability Propagation Algorithm
Check for correctness of all parity equations
Construct decoder output frame of binary values

FIGURE 19: LDPC DECODER OPERATION

The received data frame values initialise the B matrix. The HL matrix is then updated using values of the parity equations in the B matrix. The B matrix is then updated using values of the parity equations in the HL matrix. If the parity equations are satisfied the decoder outputs the frame of binary values; if the equations cannot be satisfied the decoder runs through another iteration. The process is followed until either:

- The parity bit equations are satisfied;
- A maximum number of iterations is reached;
- No changes occur to the tentative bit decisions for 10 iterations (the code stops converging);
- If the code cannot be satisfied the frame is rejected.

To allow a compact representation and rapid access to the appropriate B and HL matrix entries an index of frame positions for each parity equation component (Column matrix) and an index of parity equations that involve each frame (Row matrix) is provided for each code. These are used by the decoder on initialisation to define the parity matrix for the LDPC FEC code.

B.2. LDPC FEC DECODER EXAMPLE

The following code is an example LDPC FEC decoder.

```
{Decode symbol block}
      {L_BLOCK^[]     (data+parity block size)

       Parity_index[] (data+parity block size)
       Hor_index[]    (parity block size)
       COL[]^[]    Max_mult.(# parity entries)
       ROW[]^[]    Max_mult.(# parity entries)

       b[]^[]      Max_mult.(parity index[])
       HL[]^[]     Max_mult.(parity index[])
       State^[]       (data+parity block size)
       d^[]           (hor_index[])

       Decode_block^[](data+parity block size)
      }
      {Init decode}
       FOR J:= 1 TO DATA_BLOCK_SIZE + PARITY_BLOCK_SIZE DO
        BEGIN
         FOR I := 1 TO PARITY_INDEX[J] {NO_PARITIES_PER_SYMBOL} DO
          BEGIN
           HL[I]^[J] := 1.0;
           b[I]^[J] := L_BLOCK^[J];
          END;
        END;
       FOR J := 1 TO DATA_BLOCK_SIZE + PARITY_BLOCK_SIZE DO
         IF L_BLOCK^[J] < 1 THEN STATE^[J]:=1 ELSE STATE^[J]:=0;


      {Iterate decode}
       INDEX := 0; CONVERGENCE:= FALSE; CONVERGENCE_COUNT := 0;
       STATE_CONVERGENCE := FALSE;
       REPEAT
        BEGIN
         INDEX := INDEX + 1;

         {25 June 06 Data Extraction Mod}
         If INDEX = 1 {2} then
          Begin
           {Evaluate the parity equation status; stop if all consistent}
```

```
            CONVERGENCE := TRUE; Parity_errors := 0;
            FOR J:= 1 TO PARITY_BLOCK_SIZE DO
             BEGIN
              PARITY_COUNT := 0;
              FOR L := 1 TO HOR_INDEX[J] {NO_PARITY_ENTRIES} DO
               BEGIN
                IF L_BLOCK^[COL[L]^[J]] < 1 THEN
                 BEGIN
                  IF PARITY_COUNT = 0 THEN PARITY_COUNT := 1 ELSE
PARITY_COUNT := 0;
                 END;
               END;
              IF PARITY_COUNT = 0 THEN
               CONVERGENCE := CONVERGENCE AND TRUE
              ELSE
               Begin
                CONVERGENCE := FALSE;
                Parity_errors := Parity_errors + 1;
               End;
             END;

           WRITELN(Log_file,'  BLOCK INDEX = ',BLOCK_INDEX_'   ITERATION =
',INDEX,'   ',Parity_errors:4:0);

          End;

         WRITELN('   ITERATION = ',INDEX,'   ',Parity_errors:4:0,'
',Convergence_count:4);

        {Horizontal Update}
         FOR I := 1 TO PARITY_BLOCK_SIZE DO
          BEGIN
           FOR K := 1 TO HOR_INDEX[I] {NO_PARITY_ENTRIES} DO
            BEGIN
             d^[K] := (b[ROW[K]^[I]]^[COL[K]^[I]] - 1.0)/
                      (b[ROW[K]^[I]]^[COL[K]^[I]] + 1.0);
            END;
           FOR M := 1 TO HOR_INDEX[I] {NO_PARITY_ENTRIES} DO
            BEGIN
             DELTA:=1.0;
             FOR L := 1 TO HOR_INDEX[I] {NO_PARITY_ENTRIES} DO
              BEGIN
               IF L <> M THEN
                BEGIN
                 DELTA := DELTA*d^[L];
                END;
              END;
               IF DELTA >= 0.999999 THEN TEMP:= TOP
               ELSE TEMP := ((1+DELTA))/
                            ((1-DELTA));
               IF TEMP > TOP THEN TEMP := TOP;
               IF TEMP < BOTTOM THEN TEMP := BOTTOM;
               HL[ROW[M]^[I]]^[COL[M]^[I]] := TEMP;
            END;
          END;
         STATE_CHANGE_COUNT:=0;


        {Vertical Update}
         FOR K := 1 TO DATA_BLOCK_SIZE + PARITY_BLOCK_SIZE DO
          BEGIN
           BL^[K] := L_BLOCK^[K];{Bit estimate for iteration}
           FOR J:= 1 TO PARITY_INDEX[K] {NO_PARITIES_PER_SYMBOL} DO
            BEGIN
             b[J]^[K]:= L_BLOCK^[K];
             FOR L := 1 TO PARITY_INDEX[K] {NO_PARITIES_PER_SYMBOL} DO
              BEGIN
```

```
               IF L <> J THEN
                BEGIN
                 TEMP := b[J]^[K]*HL[L]^[K];
                 IF TEMP > TOP THEN TEMP := TOP;
                 IF TEMP < BOTTOM THEN TEMP := BOTTOM;
                 b[J]^[K] := TEMP;
                END;
               TEMP := BL^[K]*HL[L]^[K];
               IF TEMP > TOP THEN TEMP := TOP;
               IF TEMP < BOTTOM THEN TEMP := BOTTOM;
               BL^[K] := TEMP;
              END;
            END;
           IF BL^[K] < 1 THEN
            BEGIN
             IF STATE^[K] = 0 THEN
              BEGIN
               STATE_CHANGE_COUNT := STATE_CHANGE_COUNT + 1;
               STATE^[K] := 1;
              END
             ELSE
              BEGIN
              END;
            END
           ELSE
            BEGIN
             IF STATE^[K] = 0 THEN
              BEGIN
              END
             ELSE
              BEGIN
               STATE_CHANGE_COUNT := STATE_CHANGE_COUNT + 1;
               STATE^[K] := 0;
              END;
            END;
          END;
        END; {Iteration}

        IF STATE_CHANGE_COUNT = 0 THEN
         BEGIN
          CONVERGENCE_COUNT := CONVERGENCE_COUNT + 1;
          IF CONVERGENCE_COUNT >= CONVERGENCE_LIMIT THEN STATE_CONVERGENCE
:= TRUE;
         END
        ELSE
         BEGIN
          CONVERGENCE_COUNT := 0;
          STATE_CONVERGENCE := FALSE;
         END;
        {Evaluate the parity equation status; stop if all consistent}
         CONVERGENCE := TRUE; Parity_errors := 0;
         FOR J:= 1 TO PARITY_BLOCK_SIZE DO
          BEGIN
           PARITY_COUNT := 0;
           FOR L := 1 TO HOR_INDEX[J] {NO_PARITY_ENTRIES} DO
            BEGIN
             IF BL^[COL[L]^[J]] < 1 THEN
              BEGIN
               IF PARITY_COUNT = 0 THEN PARITY_COUNT := 1 ELSE PARITY_COUNT
:= 0;
              END;
            END;
           IF PARITY_COUNT = 0 THEN
            CONVERGENCE := CONVERGENCE AND TRUE
           ELSE
            Begin
             CONVERGENCE := FALSE;
             Parity_errors := Parity_errors + 1;
            End;
          END;

       UNTIL (CONVERGENCE) OR (INDEX >= MAX_ITERATIONS) OR
(STATE_CONVERGENCE);
```

B.3. LDPC DECODER MATRICES

The files defined in the tables below (TABLE 17 to **Table 20**) are included with the standard and provide the matrices for the LDPC FEC decoder for the REM modes defined in CHAPTER 3.

RED/BLACK ALPHA REM	**Filename**	**MD5 Checksum**
Column Matrix	rb_alpha.col	300a9e2f132d390ca09fee179bf01869
Row Matrix	rb_alpha.row	1d6c9c073925f5e14301f94a62b9a38d
Column Index Matrix	rb_alpha.cdx	8a863f9e572b30def03c67f8f48442fd
Row Index Matrix	rb_alpha.rdx	cbdad0809e1e5109ed7a48189fb9da80

TABLE 17: RED/BLACK ALPHA REM LDPC MATRICES.

RED/BLACK BRAVO REM	**Filename**	**MD5 Checksum**
Column Matrix	rb_bravo.col	3df9e51c29c2aecb1a6410146041cc98
Row Matrix	rb_bravo.row	6e745f849f91bb137819cc34540e1468
Column Index Matrix	rb_bravo.cdx	175fcd8689115f96c00f9dfe4f8030b
Row Index Matrix	rb_bravo.rdx	6e4e49516713f66f5357825eaa9d9bf8

TABLE 18: RED/BLACK ALPHA REM LDPC MATRICES

RED ALPHA REM	**Filename**	**MD5 Checksum**
Column Matrix	r_alpha.col	e9741dfc462be3c22b57f9e77363da4e
Row Matrix	r_alpha.row	970a9a34b55b7bc1c432e438fdd3dc29
Column Index Matrix	r_alpha.cdx	36af90274eb70dd169b059d59abb2553
Row Index Matrix	r_alpha.rdx	c654c2934714597939ceb99132e7ae30

TABLE 19: RED/BLACK ALPHA REM LDPC MATRICES.

RED BRAVO REM	Filename	MD5 Checksum
Column Matrix	r_bravo.col	cb5dff917252f0a61a8585a7c2771282
Row Matrix	r_bravo.row	4fe6e65363e2f002cd6f929b1fda8d3b
Column Index Matrix	r_bravo.cdx	9984e07d8bc1f1b2547549034ee7147f
Row Index Matrix	r_bravo.rdx	bfe8c7d3a80acfd75ecfde375ceb58e2

TABLE 20: RED/BLACK ALPHA REM LDPC MATRICES.

ANNEX C. PPM COMPRESSION CODING

C.1. PPM COMPRESSION CODE

The PPM compression of message text for the NATO REM modes shall be carried out to be interoperable with the code defined below. When build, the compression programme requires three files: the statistical model file (STANAG4724StatisticalModel.cmp), a file containing the message stream to be compressed, and an output file for the compressed message stream.

Whilst this code may be used to compress non-ITA2 data, the statistical model has been built using only the characters contained within the ITA2 character set. If non ITA2 data is compressed using the NATO statistical model, compression performance will be reduced.

The code below defines PACKET_SIZE. This is the size of the compressed broadcast message stream used in each data frame. The value of PACKET_SIZE is dependent on the type of REM coding being implemented. The following values shall be used for PACKET_SIZE when compiling the code for the REM defined in CHAPTER 3:

CODING	PACKET_SIZE
RED/BLACK REM ALPHA coding	444 bytes
RED/BLACK REM BRAVO coding	672 bytes
RED REM ALPHA coding	484 bytes
RED REM BRAVO coding	681 bytes

```
/************************ Start of MAIN-C.C ************************/
/*
 *
 * Compression Test Main(Driver)
 *
 *
 *
 * PPM Context Statistical Model Usage for text compression into binary packets
and subsequent expansion of
 * said binary packets back into text format.
 *
 * The basis of the PPM Context Statistical Model Compression and Expansion
Software is the open source Arith-N.C code  * of Mark Nelson and Jean-loup
Gailly as presented in The Data Compression Book (2nd edition).
 *
 * The software was modified, enhanced, and adapted for packet communications
usage by Technology Service Corporation   * under contract to SPAWAR Systems
Center Pacific.
 *
 * Revision History of this baseline file (Apr2012):
 *
 * This is the driver program used when testing compression algorithms.
 * In order to cut back on repetitive code, this version of main is
 * used with all of the compression routines.  It in order to turn into
 * a real program, it needs to have another module that supplies one
 * routine and two strings, namely:
 *
 *     void CompressFile( FILE *input2, BIT_FILE *output_
 *                        int argc, char *argv );
 *     char *Usage;
 *     char *CompressionName;
 *
 * The main() routine supplied here has the job of checking for valid
 * input and output files, opening them, and then calling the
 * compression routine.  If the files are not present, or no arguments
 * are supplied, it prints out an error message, which includes the
 * Usage string supplied by the compression module.  All of the
 * routines and strings needed by this routine are defined in the
 * main.h header file.
 *
 * After this is built into a compression program of any sort, the
 * program can be called like this:
 *
 *   main-c infile1 infile2 outfile [ options ]
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <math.h>
#include <malloc.h>

typedef struct bit_file {
    FILE *file;
    unsigned char mask;
    int rack;
    int pacifier_counter;
} BIT_FILE;

#ifdef __STDC__

void usage_exit( char *prog_name );
void print_ratios( char *input, char *output );
long file_size( char *name );
long file_size_2( char *name );
```

```
void LoadModel( BIT_FILE *input1, int argc, char *argv[]);
int readTheStatsFile(char* filename);
void LoadStatsModel(BIT_FILE *input1, int argc, char *argv[], char
*statsFilename);
int CompressMessages( FILE *input2, BIT_FILE *output1, int argc, char *argv[] );
int ExpandFile( BIT_FILE *input2, FILE *output, int argc, char *argv[] );

BIT_FILE     *OpenInputBitFile( char *name );
BIT_FILE     *OpenOutputBitFile( char *name );
int          OutputBit( BIT_FILE *bit_file, int bit, int tempwrite );
int          OutputBits( BIT_FILE *bit_file, unsigned long code, int count );
int           InputBit( BIT_FILE *bit_file); //, int tempwrite );
unsigned long InputBits( BIT_FILE *bit_file, int bit_count );
void          CloseInputBitFile( BIT_FILE *bit_file );
void          CloseOutputBitFile( BIT_FILE *bit_file );
void          FilePrintBinary( FILE *file, unsigned int code, int bits );

void fatal_error( char *fmt, ... );

#else

void usage_exit();
void print_ratios();
long file_size();

void LoadModel();
int readTheStatsFile();
void LoadStatsModel();

int  CompressMessages();
int  ExpandFile();

BIT_FILE      *OpenInputBitFile();
BIT_FILE      *OpenOutputBitFile();
int           OutputBit();
int           OutputBits();
int           InputBit();
unsigned long InputBits();
void          CloseInputBitFile();
void          CloseOutputBitFile();
void          FilePrintBinary();

void fatal_error();

#endif

/* Compression procedures */

int main( argc, argv )
int argc;
char *argv[];
{
    BIT_FILE *output1, *input1;
    FILE *input2;
      int status=1;
      char statsFilename[256];
      strcpy(statsFilename, "");

      setbuf(stdout, NULL);

    if ( argc < 4 )
        usage_exit( argv[ 0 ] );

      // Model Creating File
      input1 = OpenInputBitFile( argv[ 1 ] );
      if ( input1 == NULL )
```

```
        fatal_error( "Error opening %s for output\n", argv[ 1 ] );

      input2 = fopen( argv[ 2 ], "rb" );
    if ( input2 == NULL )
        fatal_error( "Error opening %s for input\n", argv[ 2 ] );

      //Binary Model Output
      output1 = OpenOutputBitFile( argv[ 3 ] );
      if ( output1 == NULL )
        fatal_error( "Error opening %s for output\n", argv[ 3 ] );

      if(argv[6] != NULL)
        strcpy(statsFilename, argv[6]);




      if (strlen(statsFilename) > 1){
        printf("\nCompressing %s to %s using STATS %s as initial data\n",
argv[2], argv[3], argv[1]);

        //load statistics in to memory
        LoadStatsModel(input1, argc - 4, argv + 4, statsFilename);

      }
      else{
        printf("\nCompressing %s to %s using MODEL %s as initial data\n",
argv[2], argv[3], argv[1]);

        //Load Model into memory
        LoadModel(input1, argc - 4, argv + 4);
      }




      //Compress Data into Packets as long as input buffer(file) is not empty
(ended)
      while (status > 0)
      {
        status = CompressMessages( input2, output1, argc -4 , argv +4 );
      }

      //Check for error if last return value is not EOF indicator (-1)
      if (status != -1)
        printf("Possible Error Ending File\n");

      CloseOutputBitFile( output1 );
      CloseInputBitFile( input1);
      fclose( input2 ); //fclose( input2 );
      printf( "Message Compression Ratio : ");
      print_ratios( argv[ 2 ], argv[ 3 ] );
    return( 0 );
}

/*
 * This routine just wants to print out the usage message that is
 * called for when the program is run with no parameters.  The first
 * part of the Usage statement is supposed to be just the program
 * name.  argv[ 0 ] generally holds the fully qualified path name
 * of the program being run.  I make a half-hearted attempt to strip
 * out that path info and file extension before printing it.  It should
 * get the general idea across.
 */
void usage_exit( prog_name )
char *prog_name;
{
    char *short_name;
    char *extension;
      char *Usage = "in-file1 in-file2 out-file [ -o order ][stats-in-
file]\n\n";

    short_name = strrchr( prog_name, '\\' );
    if ( short_name == NULL )
        short_name = strrchr( prog_name, '/' );
    if ( short_name == NULL )
        short_name = strrchr( prog_name, ':' );
    if ( short_name != NULL )
        short_name++;
    else
        short_name = prog_name;
    extension = strrchr( short_name, '.' );
    if ( extension != NULL )
        *extension = '\0';
    printf( "\nUsage:  %s %s\n", short_name, Usage );
    exit( 0 );
}

/*
 * This routine is used by main to print out get the size of a file after
 * it has been closed.  It does all the work, and returns a long.  The
 * main program gets the file size for the plain text, and the size of
 * the compressed file, and prints the ratio.
 */
#ifndef SEEK_END
#define SEEK_END 2
#endif

long file_size( name )
char *name;
{
    FILE *file;
      int ch, count = 0;

    file = fopen( name, "r" );
    if ( file == NULL )
        return( 0L );
      while (1) {
        ch = fgetc(file);
        if (ch == EOF)
            break;
        ++count;
    }

      fclose(file);
      return (count);
}

long file_size_2( name )
char *name;
{
    long eof_ftell;
    FILE *file;

    file = fopen( name, "r" );
    if ( file == NULL )
        return( 0L );
    fseek( file, 0L, SEEK_END );
    eof_ftell = ftell( file );
    fclose( file );
    return( eof_ftell );
}

/*
```

```
 * This routine prints out the compression ratios after the input
 * and output files have been closed.
 */
void print_ratios( input, output )
char *input;
char *output;
{
    long input_size;
    long output_size;
    float ratio;
    FILE *summary;

    input_size = file_size( input );
    if ( input_size == 0 )
        input_size = 1;
    output_size = file_size_2( output );
      ratio = (float) output_size*8/(float)input_size;
    printf( "\nInput characters:        %ld\n", input_size );
    printf( "Output bits:       %ld\n", output_size*8 );
    if ( output_size == 0 )
        output_size = 1;
    printf( "Compression ratio:  %g bpc\n", ratio );
    summary = fopen( "summary.out", "w" );
    if ( summary == NULL )
        fatal_error( "Error opening summary.out for output\n");
      fprintf(summary,"Input bytes:      %ld",input_size);
      fprintf(summary,"  Output bytes:     %ld",output_size);
      fprintf(summary,"  Compression ratio:  %d%%\n",ratio);
    fclose(summary);
}

/* Compression procedures
 *
 * Compression and Expansion routines for use with packet communications using
PPM Context Statistical Model
 * as input.
 *
 */
void fatal_error( fmt )
char *fmt;
{
    va_list argptr;

    va_start( argptr, fmt );
    printf( "Fatal error: " );
    vprintf( fmt, argptr );
    va_end( argptr );
    exit( -1 );
}

/*
 * The SYMBOL structure is what is used to define a symbol in
 * arithmetic coding terms.
 */

typedef struct {
    unsigned short int low_count;
    unsigned short int high_count;
    unsigned short int scale;
} SYMBOL;

#define MAXIMUM_SCALE   16383  /* Maximum allowed frequency count */
#define ESCAPE          256    /* The escape symbol               */
#define DONE            (-1)   /* The output stream empty  symbol */
#define FLUSH           (-2)   /* The symbol to flush the model   */
#define END_PACKET      (-3)   /* The symbol to notify end expansion */
```

```
#define PACKET_SIZE     484    /* Packet Size In Bits (RED: 484, RED/BLACK: 444)
*/
#define UPDATE_EXC      1      /* Update Exclusion Flag, 1 = on, 0 = off 0 */

/*
 * Function prototypes.
 */

#ifdef __STDC__

void initialize_options( int argc, char **argv );
int check_compression( FILE *input, BIT_FILE *output );
void initialize_model( void );
void initialize_model2( void );
void initialize_model_load(void);
void update_model( int symbol );
void update_model2( int symbol );
int convert_int_to_symbol( int symbol, SYMBOL *s );
int convert_int_to_symbol2( int symbol, SYMBOL *s );
void get_symbol_scale( SYMBOL *s );
void get_symbol_scale2( SYMBOL *s );
int convert_symbol_to_int( int count, SYMBOL *s );
int convert_symbol_to_int1( int count, SYMBOL *s, int debug );
void add_character_to_model( int c );
void add_character_to_model2( int c, int flag );
void flush_model( void );
void initialize_arithmetic_decoder( BIT_FILE *stream );
void initialize_arithmetic_decoder1( BIT_FILE *stream , int numbits);
int remove_symbol_from_stream( BIT_FILE *stream, SYMBOL *s );
int remove_symbol_from_stream1( BIT_FILE *stream, SYMBOL *s, int bits );
void initialize_arithmetic_encoder( void );
int encode_symbol1( BIT_FILE *stream, SYMBOL *s, int tempwriteflag);
void encode_symbol( BIT_FILE *stream, SYMBOL *s );
int flush_arithmetic_encoder1( BIT_FILE *stream );
void flush_arithmetic_encoder( BIT_FILE *stream );
short int get_current_count( SYMBOL *s );
void save_model(void);
void reset_model(void);

#else /* __STDC_, */

void initialize_options();
int check_compression();
void initialize_model();
void initialize_model2();
void initialize_model_load();
void update_model();
void update_model2();
int convert_int_to_symbol();
int convert_int_to_symbol2();
void get_symbol_scale();
void get_symbol_scale2();
int convert_symbol_to_int();
int convert_symbol_to_int1();
void add_character_to_model();
void add_character_to_model2();
void flush_model();
void initialize_arithmetic_decoder();
void initialize_arithmetic_decoder1();
int remove_symbol_from_stream();
int remove_symbol_from_stream1();
void initialize_arithmetic_encoder();
int encode_symbol1();
void encode_symbol();
int flush_arithmetic_encoder1();
void flush_arithmetic_encoder();
short int get_current_count();
```

```
void save_model();
void reset_model();

#endif  /* __STDC_, */

char *CompressionName = "Adaptive order n model with arithmetic coding";

int max_order = 4;
int bitsinpacket = 0;                   //num compressed bits in packet
int pknum = 0;                          //packet number
int numalloc = 0;                       //number of memory slots allocated
long int input_pos = 0;
int offset = 16;



void initialize_options( argc, argv )
int argc;
char *argv[];
{
    while ( argc-- > 0 ) {
        if ( strcmp( *argv, "-o" ) == 0 ) {
            argc--;
            max_order = atoi( *++argv );
        } else
            printf( "Uknown argument on command line: %s\n", *argv );
        argc--;
        argv++;
    }
}

/*
 * This routine is called once every 256 input symbols.  Its job is to
 * check to see if the compression ratio falls below 10%.  If the
 * output size is 90% of the input size, it means not much compression
 * is taking place, so we probably ought to flush the statistics in the
 * model to allow for more current statistics to have greater impact.
 * This heuristic approach does seem to have some effect.
 */
int check_compression( input, output )
FILE *input;
BIT_FILE *output;
{
    static long local_input_marker = 0L;
    static long local_output_marker = 0L;
    long total_input_bytes;
    long total_output_bytes;
    int local_ratio;

    total_input_bytes  =  ftell( input ) - local_input_marker;
    total_output_bytes = ftell( output->file );
    total_output_bytes -= local_output_marker;
    if ( total_output_bytes == 0 )
        total_output_bytes = 1;
    local_ratio = (int)( ( total_output_bytes * 100 ) / total_input_bytes );

    local_input_marker = ftell( input );
    local_output_marker = ftell( output->file );

    return( local_ratio > (int)90 );
}


//Define context data structures
typedef struct {
    unsigned char symbol;
    unsigned char counts;
```

```
} STATS;

typedef struct {
    struct context *next;
} LINKS;

typedef struct context {
    int max_index;
    LINKS *links;
    STATS *stats;
    struct context *lesser_context;
} CONTEXT;


/*
 * *contexts[] is an array of current contexts.
 */
CONTEXT **contexts;

/*
 * current_order contains the current order of the model.  It starts
 * at max_order, and is decremented every time an ESCAPE is sent.  It
 * will only go down to -1 for normal symbols, but can go to -2 for
 * EOF and FLUSH.
 */
int current_order;

short int totals[ 258 ];
char scoreboard[ 256 ];
char tempscoreboard[ 256 ];

//Global vars now due to functionality break-up
int modtotals[258] = {0};
int modcontexts[20] = {0};

/*
 * Local procedure declarations for modeling routines.
 */
#ifdef __STDC__
void update_table( CONTEXT *table, int symbol );
void rescale_table( CONTEXT *table );
void totalize_table( CONTEXT *table );
void totalize_table2( CONTEXT *table );
CONTEXT *shift_to_next_context( CONTEXT *table, int c, int order);
CONTEXT *shift_to_next_context2( CONTEXT *table, int c, int order);
CONTEXT *shift_to_next_context3( CONTEXT *table, int c, int order);
CONTEXT *allocate_next_order_table( CONTEXT *table, int symbol, CONTEXT
*lesser_context );
void recursive_flush( CONTEXT *table );
CONTEXT* create_model(int ContextT, int Index0, int Index1, int Index2, int
Index3, int Index4, int Order, int Symbol, int Count);
#else
void update_table();
void rescale_table();
void totalize_table();
void totalize_table2();
CONTEXT *shift_to_next_context();
CONTEXT *shift_to_next_context2();
CONTEXT *shift_to_next_context3();
CONTEXT *allocate_next_order_table();
void recursive_flush();
CONTEXT* create_model();

#endif

void initialize_model()
{
    int i;
    CONTEXT *null_table;
    CONTEXT *control_table;

        current_order = max_order;
    contexts = (CONTEXT **) calloc( sizeof( CONTEXT * ), 20 );
    if ( contexts == NULL )
        fatal_error( "Failure #1: allocating context table!" );
    contexts += 2;
    null_table = (CONTEXT *) calloc( sizeof( CONTEXT ), 1 );
    if ( null_table == NULL )
        fatal_error( "Failure #2: allocating null table!" );
    null_table->max_index = -1;
    contexts[ -1 ] = null_table;
    for ( i = 0 ; i <= max_order ; i++ )
        contexts[ i ] = allocate_next_order_table( contexts[ i-1 ], 0, contexts[
i-1 ] );
        null_table->stats =
        (STATS *) realloc( (char *) null_table->stats, sizeof( STATS )*256 );

    if ( null_table->stats == NULL )
        fatal_error( "Failure #3: allocating null table!" );
    null_table->max_index = 255;
    for ( i=0 ; i < 256 ; i++ ) {
        null_table->stats[ i ].symbol = (unsigned char) i;
        null_table->stats[ i ].counts = 1;
    }

    control_table = (CONTEXT *) calloc( sizeof(CONTEXT), 1 );
    if ( control_table == NULL )
        fatal_error( "Failure #4: allocating null table!" );
    control_table->stats =
         (STATS *) calloc( sizeof( STATS ), 2 );
    if ( control_table->stats == NULL )
        fatal_error( "Failure #5: allocating null table!" );
    contexts[ -2 ] = control_table;
    control_table->max_index = 2;
    control_table->stats[ 0 ].symbol = -FLUSH;
    control_table->stats[ 0 ].counts = 1;
    control_table->stats[ 1 ].symbol = -DONE;
    control_table->stats[ 1 ].counts = 1;
        control_table->stats[ 2 ].symbol = -END_PACKET;
    control_table->stats[ 2 ].counts = 1;

    for ( i = 0 ; i < 256 ; i++ )
        scoreboard[ i ] = 0;
}


CONTEXT* create_model(ContextT, Index0, Index1, Index2, Index3, Index4, Order,
Symbol, Count)
int ContextT;
int Index0;
int Index1;
int Index2;
int Index3;
int Index4;
int Order;
int Symbol;
int Count;
{
      int index = 0;
      int new_size;
      CONTEXT *lesser_context = NULL;
      CONTEXT *table = NULL;
      CONTEXT *new_table = NULL;
      struct node *ptr = NULL;
```

```
CONTEXT *found_table = NULL;


if (Order == -1)
{

   if (contexts[ContextT] == NULL){
         new_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
         if (new_table == NULL)
                fatal_error("Failure CONTEXT allocating!");

         new_table->max_index = 0;
         new_table->lesser_context = NULL;

         contexts[ContextT] = new_table;
   }
   table = contexts[ContextT];
   index = ContextT;
   table->max_index = Count;
}
else if (Order == 0){

   table = contexts[ContextT];
   index = Index0;

   if (table->stats == NULL){
         table->stats = (STATS *)calloc(1, sizeof(STATS));
         if (table->stats == NULL)
                fatal_error("Failure STATS allocating!");
   }

   if (table->links == NULL){
         table->links = (LINKS *)calloc(1, sizeof(LINKS));
         if (table->links == NULL)
                fatal_error("Failure LINKS allocating!");

         table->links->next = NULL;
   }
}
else if (Order == 1){

   if (contexts[ContextT]->links[Index0].next == NULL){
         new_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
         if (new_table == NULL)
                fatal_error("Failure CONTEXT allocating!");

         new_table->max_index = -1;
         new_table->lesser_context = NULL;


         contexts[ContextT]->links[Index0].next = new_table;
   }

   table = contexts[ContextT]->links[Index0].next;
   index = Index1;

}
else if (Order == 2){


   if (contexts[ContextT]->links[Index0].next->links[Index1].next == NULL){
         new_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
         if (new_table == NULL)
                fatal_error("Failure CONTEXT allocating!");

         new_table->max_index = -1;
         new_table->lesser_context = NULL;
```

```
               contexts[ContextT]->links[Index0].next->links[Index1].next =
new_table;
        }

        table = contexts[ContextT]->links[Index0].next->links[Index1].next;
        index = Index2;
      }
      else if (Order == 3){

        if (contexts[ContextT]->links[Index0].next->links[Index1].next-
>links[Index2].next == NULL){
               new_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
               if (new_table == NULL)
                      fatal_error("Failure CONTEXT allocating!");

               new_table->max_index = -1;
               new_table->lesser_context = NULL;

               contexts[ContextT]->links[Index0].next->links[Index1].next-
>links[Index2].next = new_table;
        }

        table = contexts[ContextT]->links[Index0].next->links[Index1].next-
>links[Index2].next;
        index = Index3;

      }
      else if (Order == 4){

        if (contexts[ContextT]->links[Index0].next->links[Index1].next-
>links[Index2].next->links[Index3].next == NULL){
               new_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
               if (new_table == NULL)
                      fatal_error("Failure CONTEXT allocating!");

               new_table->max_index = -1;
               new_table->lesser_context = NULL;

               contexts[ContextT]->links[Index0].next->links[Index1].next-
>links[Index2].next->links[Index3].next = new_table;
        }

        table = contexts[ContextT]->links[Index0].next->links[Index1].next-
>links[Index2].next->links[Index3].next;
        index = Index4;

      }

      if (Order >= 0){
        new_size = sizeof(LINKS);
        new_size *= index + 1;
        if (Order <= max_order) {
               if (index == 0) {
                      table->links = (LINKS *)calloc(1, new_size);

               }
               else {
                      table->links = (LINKS *)realloc((char *)table->links,
new_size);
               }


               if (table->links == NULL)
                      fatal_error("Error #9: reallocating table space!");
               table->links[index].next = NULL;
        }
        new_size = sizeof(STATS);
        new_size *= index + 1;
        if (index == 0){
               table->stats = (STATS *)calloc(1, new_size);
        }
        else {
               table->stats = (STATS *)realloc((char *)table->stats, new_size);
        }

        if (table->stats == NULL)
               fatal_error("Error #10: reallocating table space!");
        table->stats[index].symbol = '0';
        table->stats[index].counts = 0;


      }

      if (Order > -1)
      {             //add the stats data
        table->stats[index].symbol = (unsigned char)Symbol;
        table->stats[index].counts = Count;
        if (Order > 0)
               table->max_index++;
      }



      return table;
}


void initialize_model2()
{
    int i;

    current_order = max_order;
    for ( i = 0 ; i < 256 ; i++ )
        scoreboard[ i ] = 0;
}


/*
* This routine has to get everything set up properly so that
* the model can be maintained properly.  The first step is to create
* the *contexts[] array used later to find current context tables.
* The *contexts[] array indices go from -2 up to max_order, so
* the table needs to be fiddled with a little.  This routine then
* has to create the special order -2 and order -1 tables by hand,
* since they aren't quite like other tables.  Then the current
* context is set to \0, \0, \0, ... and the appropriate tables
* are built to support that context.  The current order is set
* to max_order, the scoreboard is cleared, and the system is
* ready to go.
*/
void initialize_model_load()
{
      int i;
      CONTEXT *null_table;
      CONTEXT *control_table;

      current_order = max_order;
      contexts = (CONTEXT **)calloc(20, sizeof(CONTEXT *)); //20
      if (contexts == NULL)
        fatal_error("Failure #1: allocating context table!");
      contexts += 2;
      null_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
      if (null_table == NULL)
        fatal_error("Failure #2: allocating null table!");
      null_table->max_index = -1;

      contexts[-1] = null_table;

      for (i = 0; i <= max_order; i++)
        contexts[i] = allocate_next_order_table(contexts[i - 1], 0, contexts[i -
1]);

      null_table->stats = (STATS *)realloc((char *)null_table->stats,
sizeof(STATS)* 256);
      if (null_table->stats == NULL)
        fatal_error("Failure #3: allocating null table!");

      null_table->max_index = 255;

      for (i = 0; i < 256; i++) {
        null_table->stats[i].symbol = (unsigned char)i;
        null_table->stats[i].counts = 1;
      }

      control_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
      if (control_table == NULL)
        fatal_error("Failure #4: allocating null table!");
      control_table->stats = (STATS *)calloc(3, sizeof(STATS));
      if (control_table->stats == NULL)
        fatal_error("Failure #5: allocating null table!");
      contexts[-2] = control_table;

      control_table->max_index = 2;

      control_table->stats[0].symbol = -FLUSH;
      control_table->stats[0].counts = 1;
      control_table->stats[1].symbol = -DONE;
      control_table->stats[1].counts = 1;
      control_table->stats[2].symbol = -END_PACKET;
      control_table->stats[2].counts = 1;

      for (i = 0; i < 256; i++)
        scoreboard[i] = 0;
}



/*
 * This is a utility routine used to create new tables when a new
 * context is created.
 */
int num = 0;

CONTEXT *allocate_next_order_table( table, symbol, lesser_context )
CONTEXT *table;
int symbol;
CONTEXT *lesser_context;
{
    CONTEXT *new_table;
    int i;
    unsigned int new_size;
        for ( i = 0 ; i <= table->max_index ; i++ )
        if ( table->stats[ i ].symbol == (unsigned char) symbol )
            break;
    if ( i > table->max_index ) {
        table->max_index++;
        new_size = sizeof( LINKS );
        new_size *= table->max_index + 1;
        if ( table->links == NULL )
        {
            table->links = (LINKS *) calloc( new_size, 1 );
        }
        else
        {
            table->links = (LINKS *)realloc( (char *) table->links, new_size );
        }
        if (table->links == NULL)
               fatal_error("Failure #6: allocating new table");

        new_size = sizeof( STATS );
        new_size *= table->max_index + 1;
        if ( table->stats == NULL )
                {
            table->stats = (STATS *) calloc( new_size, 1 );
                }
        else
                {
                        table->stats = (STATS *)
                realloc( (char *) table->stats, new_size );
                }

        if ( table->stats == NULL )
            fatal_error( "Failure #7: allocating new table" );
        table->stats[ i ].symbol = (unsigned char) symbol;
        table->stats[ i ].counts = 0;
    }
      new_size = sizeof(CONTEXT);
    new_table = (CONTEXT *) calloc( sizeof( CONTEXT ), 1 );
    if ( new_table == NULL )
        fatal_error( "Failure #8: allocating new table" );
    new_table->max_index = -1;

    table->links[ i ].next = new_table;
    new_table->lesser_context = lesser_context;
    return( new_table );
}

/*
 * This routine is called to increment the counts for the current
 * contexts.  It is called after a character has been encoded or
 * decoded.
 */
void update_model( symbol )
int symbol;
{
    int i;
    int local_order;
    int loopstart;

    if ( current_order < 0 )
        local_order = 0;
    else
        local_order = current_order;
        //Determines starting point of loop based on UPDATE_EXC flag
        if (UPDATE_EXC)
                loopstart = local_order;
        else
                loopstart = 0;

    if ( symbol >= 0 ) {
        while ( loopstart <= max_order ) {
            if ( symbol >= 0 )
                update_table( contexts[ loopstart ], symbol );
            loopstart++;
        }
    }
    current_order = max_order;
    for ( i = 0 ; i < 256 ; i++ )
        scoreboard[ i ] = 0;
}

void update_model2( symbol )
int symbol;
{
    int i;

    for ( i = 0 ; i < 256 ; i++ )
        scoreboard[ i ] = 0;
}

/*
 * This routine is called to update the count for a particular symbol
 * in a particular table.  The table is one of the current contexts_
 * and the symbol is the last symbol encoded or decoded.
 */


void update_table( table, symbol )
CONTEXT *table;
int symbol;
{
    int i;
    int index;
    unsigned char temp;
    CONTEXT *temp_ptr;
    unsigned int new_size;
    index = 0;
    while ( index <= table->max_index &&
            table->stats[index].symbol != (unsigned char) symbol )
        index++;
    if ( index > table->max_index ) {
        table->max_index++;
        new_size = sizeof( LINKS );
        new_size *= table->max_index + 1;
        if ( current_order < max_order ) {
            if ( table->max_index == 0 ) {
                table->links = (LINKS *) calloc( new_size, 1 );
            }
            else {
                   table->links = (LINKS *)
                   realloc( (char *) table->links, new_size);
            }
            if ( table->links == NULL )
                fatal_error( "Error #9: reallocating table space!" );
            table->links[ index ].next = NULL;
        }
        new_size = sizeof( STATS );
        new_size *= table->max_index + 1;
        if (table->max_index==0){
            table->stats = (STATS *) calloc( new_size, 1 );
        }
        else {
            table->stats = (STATS *)
            realloc( (char *) table->stats, new_size );
        }
        if ( table->stats == NULL )
            fatal_error( "Error #10: reallocating table space!" );
        table->stats[ index ].symbol = (unsigned char) symbol;
        table->stats[ index ].counts = 0;
    }
    i = index;
    while ( i > 0 &&
            table->stats[ index ].counts == table->stats[ i-1 ].counts )
        i--;
    if ( i != index ) {
        temp = table->stats[ index ].symbol;
        table->stats[ index ].symbol = table->stats[ i ].symbol;
        table->stats[ i ].symbol = temp;
        if ( table->links != NULL ) {
            temp_ptr = table->links[ index ].next;
            table->links[ index ].next = table->links[ i ].next;
            table->links[ i ].next = temp_ptr;
        }
        index = i;
    }

    table->stats[ index ].counts++;
    if ( table->stats[ index ].counts == 255 )
        rescale_table( table );
}

/*
 * This routine is called when a given symbol needs to be encoded.
 * It is the job of this routine to find the symbol in the context
 * table associated with the current table, and return the low and
 * high counts associated with that symbol, as well as the scale.
 *
 */
int convert_int_to_symbol( c, s )
int c;
SYMBOL *s;
{
    int i;
    CONTEXT *table;

    table = contexts[ current_order ];
    totalize_table( table );
    s->scale = totals[ 0 ];
    if ( current_order == -2 )
        c = -c;
    for ( i = 0 ; i <= table->max_index ; i++ ) {
        if ( c == (int) table->stats[ i ].symbol ) {
            if ( table->stats[ i ].counts == 0 )
                break;
            s->low_count = totals[ i+2 ];
            s->high_count = totals[ i+1 ];
            return( 0 );
        }
    }
    s->low_count = totals[ 1 ];
    s->high_count = totals[ 0 ];
    current_order--;
    return( 1 );
}

int convert_int_to_symbol2( c, s )
int c;
SYMBOL *s;
{
    int i;
    CONTEXT *table;

    table = contexts[ current_order ];
    totalize_table2( table );
    s->scale = totals[ 0 ];
        if (table->max_index != -1)
        {
                if ( current_order == -2 )
                        c = -c;
                for ( i = 0 ; i <= table->max_index ; i++ ) {
                    if ( c == (int) table->stats[ i ].symbol ) {
                        if ( table->stats[ i ].counts == 0 )
                            break;
                        s->low_count = totals[ i+2 ];
                        s->high_count = totals[ i+1 ];
                        return( 0 );
                    }
                }
        }
    s->low_count = totals[ 1 ];
    s->high_count = totals[ 0 ];
    contexts[current_order] = NULL;
    current_order--;
    return( 1 );
}

/*
 * This routine is called when decoding an arithmetic number.  In
 * order to decode the present symbol, the current scale in the
 * model must be determined.
 */

void get_symbol_scale( s )
SYMBOL *s;
{
    CONTEXT *table;

    table = contexts[ current_order ];
    totalize_table( table );
    s->scale = totals[ 0 ];
}

void get_symbol_scale2( s )
SYMBOL *s;
{
    CONTEXT *table;

    table = contexts[ current_order ];
    totalize_table2( table );
    s->scale = totals[ 0 ];
}

/*
 * This routine is called during decoding.  It is given a count that
 * came out of the arithmetic decoder, and has to find the symbol that
 * matches the count.
 */
int convert_symbol_to_int( count, s )
int count;
SYMBOL *s;
{
    int c;
    CONTEXT *table;

    table = contexts[ current_order ];
    for ( c = 0; count < totals[ c ] ; c++ )
        ;
    s->high_count = totals[ c - 1 ];
    s->low_count = totals[ c ];
    if ( c == 1) {
        current_order--;
        return( ESCAPE );
    }
    if ( current_order < -1 )
        return( (int) -table->stats[ c-2 ].symbol );
    else
        return( table->stats[ c-2 ].symbol );
}

int convert_symbol_to_int1( count, s, debug )
int count;
SYMBOL *s;
int debug;
{
    int c;
    CONTEXT *table;

    table = contexts[ current_order ];
    for ( c = 0; count < totals[ c ] ; c++ )
        ;

    s->high_count = totals[ c - 1 ];
    s->low_count = totals[ c ];
    if ( c == 1 ) {
        current_order--;
        return( ESCAPE );
    }

    if (c==0)
    {
            printf("c = 0\n");
            return(-10);
    }

    if ( current_order < -1 )
        return( (int) -table->stats[ c-2 ].symbol );
    else
        return( table->stats[ c-2 ].symbol );
}

/*
 * After the model has been updated for a new character, this routine
 * is called to "shift" into the new context.
 */

void add_character_to_model( c )
int c;
{
    int i;
    if ( max_order < 0 || c < 0 )
       return;
    contexts[ max_order ] =
            shift_to_next_context( contexts[ max_order ], c, max_order );
    for ( i = max_order-1 ; i > 0 ; i-- )
        contexts[ i ] = contexts[ i+1 ]->lesser_context;
}

void add_character_to_model2( c, flag )
int c;
int flag;
{
    int i,local_order;

    local_order = current_order;
    if ( max_order < 0 || c < 0 )
       return;
    // Attempt to raise order if flag is 1
    if ( flag || local_order == -1)
        {
        contexts[ local_order + 1 ] =
                shift_to_next_context3( contexts[ local_order ], c, local_order
);
        }
        // Stay at current_order if flag is 0 or if attempt to raise order
failed
        if ( (!flag || (flag && contexts[local_order + 1] == NULL)) &&
local_order != -1 )
        {
        contexts[ current_order ] =
                shift_to_next_context2( contexts[ current_order ], c,
current_order );
        }

        for ( i = current_order-1 ; i > 0 ; i-- )
        contexts[ i ] = contexts[ i+1 ]->lesser_context;
}

/*
 * This routine is called when adding a new character to the model. From
 * the previous example, if the current context was "ABC", and the new
 * symbol was 'D', this routine would get called with a pointer to
 * context table "ABC", and symbol 'D', with order max_order.
 */
CONTEXT *shift_to_next_context( table, c, order )
CONTEXT *table;
int c;
int order;
{
    int i;
    CONTEXT *new_lesser;

    table = table->lesser_context;
    if ( order == 0 )
        return( table->links[ 0 ].next );
      for ( i = 0 ; i <= table->max_index ; i++ )
        if ( table->stats[ i ].symbol == (unsigned char) c )
            if ( table->links[ i ].next != NULL )
                return( table->links[ i ].next );
            else
                break;

    new_lesser = shift_to_next_context( table, c, order-1 );
    table = allocate_next_order_table( table, c, new_lesser );
    return( table );
}

CONTEXT *shift_to_next_context2( table, c, order )
CONTEXT *table;
int c;
int order;
{
    int i = 0;

    table = table->lesser_context;
    if ( order < 0 || table->lesser_context == NULL)
        return( table->links[ 0 ].next );

    for ( i = 0 ; i <= table->max_index ; i++ ){
        if ( table->stats[ i ].symbol == (unsigned char) c )
            if ( table->links[ i ].next != NULL )
                return( table->links[ i ].next );
            else
                break;
    }
    return( table );
}
```

```
CONTEXT *shift_to_next_context3( table, c, order )
CONTEXT *table;
int c;
int order;
{
    int i;

    if ( order < 0 ){
        current_order++;
        return( table->links[ 0 ].next );
    }

    for ( i = 0 ; i <= table->max_index ; i++ ){
        if ( table->stats[ i ].symbol == (unsigned char) c )
            if ( table->links[ i ].next != NULL ){
                current_order++;
                return( table->links[ i ].next );
            }
            else
                break;
    }
    return( table = NULL );
}

/*
 * Rescaling the table needs to be done for one of three reasons.
 * First, if the maximum count for the table has exceeded 16383, it
 * means that arithmetic coding using 16 and 32 bit registers might
 * no longer work.  Secondly, if an individual symbol count has
 * reached 255, it will no longer fit in a byte.  Third, if the
 * current model isn't compressing well, the compressor program may
 * want to rescale all tables in order to give more weight to newer
 * statistics.
 */
void rescale_table( table )
CONTEXT *table;
{
    int i;

    if ( table->max_index == -1 )
        return;
    for ( i = 0 ; i <= table->max_index ; i++ )
        table->stats[ i ].counts /= 2;
    if ( table->stats[ table->max_index ].counts == 0 &&
         table->links == NULL ) {
        while ( table->stats[ table->max_index ].counts == 0 &&
                table->max_index >= 0 )
            table->max_index--;
        if ( table->max_index == -1 ) {
            free( (char *) table->stats );
            table->stats = NULL;
        }
        else {
            table->stats = (STATS *)
            realloc( (char *) table->stats,
            sizeof( STATS ) * ( table->max_index + 1 ) );
            if ( table->stats == NULL )
                fatal_error( "Error #11: reallocating stats space!" );
        }
    }
}

/*
 * This routine has the job of creating a cumulative totals table for
 * a given context.
 */
void totalize_table( table )
```

```
CONTEXT *table;
{
    int i;
    unsigned char max;

    for ( ; ; ) {
        max = 0;
        i = table->max_index + 2;
        totals[ i ] = 0;
        for ( ; i > 1 ; i-- ) {
            totals[ i-1 ] = totals[ i ];
            if ( table->stats[ i-2 ].counts )
                if ( ( current_order == -2 ) ||
                     scoreboard[ table->stats[ i-2 ].symbol ] == 0 )
                     totals[ i-1 ] += table->stats[ i-2 ].counts;
            if ( table->stats[ i-2 ].counts > max )
                max = table->stats[ i-2 ].counts;
        }

        if ( max == 0 )
            totals[ 0 ] = 1;
        else {
            totals[ 0 ] = (short int) ( 256 - table->max_index );
            totals[ 0 ] *= table->max_index;
            totals[ 0 ] /= 256;
            totals[ 0 ] /= max;
            totals[ 0 ]++;
            totals[ 0 ] += totals[ 1 ];
        }
        if ( totals[ 0 ] < MAXIMUM_SCALE )
            break;
        rescale_table( table );
    }
    for ( i = 0 ; i < table->max_index ; i++ )
        if (table->stats[i].counts != 0)
    scoreboard[ table->stats[ i ].symbol ] = 1;
}

void totalize_table2( table )
CONTEXT *table;
{
    int i;
    unsigned char max;

    for ( ; ; ) {
        max = 0;
        i = table->max_index + 2;
        totals[ i ] = 0;
        for ( ; i > 1 ; i-- ) {
            totals[ i-1 ] = totals[ i ];
            if ( table->stats[ i-2 ].counts )
                if ( ( current_order == -2 ) ||
                     scoreboard[ table->stats[ i-2 ].symbol ] == 0 )
                     totals[ i-1 ] += table->stats[ i-2 ].counts;
            if ( table->stats[ i-2 ].counts > max )
                max = table->stats[ i-2 ].counts;
        }

        if ( max == 0 )
            totals[ 0 ] = 1;
        else {
            totals[ 0 ] = (short int) ( 256 - table->max_index );
            totals[ 0 ] *= table->max_index;
            totals[ 0 ] /= 256;
            totals[ 0 ] /= max;
            totals[ 0 ]++;
            totals[ 0 ] += totals[ 1 ];
        }
        if ( totals[ 0 ] < MAXIMUM_SCALE )
            break;
    }
    for ( i = 0 ; i < table->max_index ; i++ )
        if (table->stats[i].counts != 0)
            scoreboard[ table->stats[ i ].symbol ] = 1;
}

/*
 * This routine is called when the entire model is to be flushed.
 */
void recursive_flush( table )
CONTEXT *table;
{
    int i;

    if ( table->links != NULL )
        for ( i = 0 ; i <= table->max_index ; i++ )
            if ( table->links[ i ].next != NULL )
                recursive_flush( table->links[ i ].next );
    rescale_table( table );
}

/*
 * This routine is called to flush the whole table, which it does
 * by calling the recursive flush routine starting at the order 0
 * table.
 */
void flush_model()
{
    putc( 'F', stdout );
    recursive_flush( contexts[ 0 ] );
}

static unsigned short int code;  /* The present input code value       */
static unsigned short int low;   /* Start of the current code range    */
static unsigned short int high;  /* End of the current code range      */
long underflow_bits;             /* Number of underflow bits pending   */

/*
 * This routine must be called to initialize the encoding process.
 */
void initialize_arithmetic_encoder()
{
    low = 0;
    high = 0xffff;
    underflow_bits = 0;
}

/*
 * At the end of the encoding process, there are still significant
 * bits left in the high and low registers.  We output two bits_
 * plus as many underflow bits as are necessary.
 */

void flush_arithmetic_encoder( stream )
BIT_FILE *stream;
{
    OutputBit( stream, low & 0x4000 , 0);
    underflow_bits++;
    while ( underflow_bits-- > 0 )
        OutputBit( stream, ~low & 0x4000 , 0 );
    OutputBits( stream, 0L, 16 );
}

/* This is used at end of each packet. It flushes the encoder by
```

```
 * outputting an bits left from the last character and any underflow bits.
 * It then outputs a variable number of zeros so the decoder can sync up
 * and the next packet will start at a predictable location.
 */

int flush_arithmetic_encoder1( stream )
BIT_FILE *stream;
{
    int count = 0;
    int retval = 0;
    int bitsinstream = 0;
    int lastmask = 0;
    int zeros = 0;
    int bisorig = 0;
    int fp = 0;
    int i;

      bitsinstream = bitsinpacket;

    lastmask = stream->mask;
    OutputBit( stream, low & 0x4000,0);
    if (stream -> mask != lastmask){
        bitsinstream++;
        lastmask = stream->mask;
    }

    underflow_bits++;
    while ( underflow_bits > 0 )
      {
        OutputBit( stream, ~low & 0x4000,0 );
        if (stream -> mask != lastmask)
        {
           bitsinstream++;
           lastmask = stream->mask;
        }
        underflow_bits--;
    }
    zeros = PACKET_SIZE - bitsinstream;

    for (i=0;i<zeros;i++)
        OutputBit(stream,0,0);

    return ( zeros + (bitsinstream - bitsinpacket));
}

/*
 * This routine is called to encode a symbol.  The symbol is passed
 * in the SYMBOL structure as a low count, a high count, and a range.
 */
void encode_symbol( stream, s )
BIT_FILE *stream;
SYMBOL *s;
{
    long range;

    range = (long) ( high-low ) + 1;
    high = low + (unsigned short int)
                 (( range * s->high_count ) / s->scale - 1 );
    low = low + (unsigned short int)
                 (( range * s->low_count ) / s->scale );

    for ( ; ; ) {
        if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
            OutputBit( stream, high & 0x8000 , 0);
            while ( underflow_bits > 0 ) {
                OutputBit( stream, ~high & 0x8000 , 0 );
                underflow_bits--;
            }
        }
        else if ( ( low & 0x4000 ) && !( high & 0x4000 )) {
            underflow_bits += 1;
            low &= 0x3fff;
            high |= 0x4000;
        }
        else
            return ;

        low <<= 1;
        high <<= 1;
        high |= 1;
    }
}

//Sames as above, but returns the number of encoded bits
int encode_symbol1( stream, s, tempwriteflag )
BIT_FILE *stream;
SYMBOL *s;
int tempwriteflag;
{
    long range;
    int count = 0;
    int debug = 0;
    int lastmask = 0;
    int addedbits = 0;

    range = (long) ( high-low ) + 1;
    high = low + (unsigned short int)
                 (( range * s->high_count ) / s->scale - 1 );
    low = low + (unsigned short int)
                 (( range * s->low_count ) / s->scale );

    for ( ; ; ) {
        if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
            lastmask = stream->mask;
            debug += OutputBit( stream, high & 0x8000, tempwriteflag );
            if (stream -> mask != lastmask){
                   addedbits++;
                lastmask = stream->mask;
            }
            while ( underflow_bits > 0 ) {
                count += OutputBit( stream, ~high & 0x8000, tempwriteflag );
                if (stream -> mask != lastmask){
                    addedbits++;
                    lastmask = stream->mask;
                }
                underflow_bits--;
            }
        }
        else if ( ( low & 0x4000 ) && !( high & 0x4000 )) {
            underflow_bits += 1;
            low &= 0x3fff;
            high |= 0x4000;
        } else
            return addedbits;
        low <<= 1;
        high <<= 1;
        high |= 1;
    }
        return addedbits;
}

/*
 * When decoding, this routine is called to figure out which symbol
 * is presently waiting to be decoded.
 */
short int get_current_count( s )
SYMBOL *s;
{
    long range;
    short int count;

    range = (long) ( high - low ) + 1;
    count = (short int)
            ((((long) ( code - low ) + 1 ) * s->scale-1 ) / range );

    return( count );
}

/*
 * This routine is called to initialize the state of the arithmetic
 * decoder.
 */
void initialize_arithmetic_decoder( stream )
BIT_FILE *stream;
{
    int i;

    code = 0;
    for ( i = 0 ; i < 16 ; i++ ) {
        code <<= 1;
        code += InputBit( stream );
    }
      low = 0;
    high = 0xffff;
    underflow_bits = 0;
}

/* Starts decoder in correct spot for each new packet
 * Numbits is related to variable number of underflow bits and zeros
 * encoded at end of each packet.
 */

void initialize_arithmetic_decoder1( stream , numbits)
BIT_FILE *stream;
int numbits;
{
    int i;
    code = 0;
    for ( i = 0 ; i < numbits ; i++ ) {
        code <<= 1;
        code += InputBit( stream );
    }
    low = 0;
    high = 0xffff;
}

int remove_symbol_from_stream( stream, s )
BIT_FILE *stream;
SYMBOL *s;
{
    long range;
    int numout = 0;
    range = (long)( high - low ) + 1;
    high = low + (unsigned short int)
                 (( range * s->high_count ) / s->scale - 1 );
    low = low + (unsigned short int)
                 (( range * s->low_count ) / s->scale );
    for ( ; ; ) {
        if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
            if (underflow_bits > 0)
                underflow_bits = 0;
        }
        else if ((low & 0x4000) == 0x4000  && (high & 0x4000) == 0 ) {
            code ^= 0x4000;
            low   &= 0x3fff;
            high  |= 0x4000;
            underflow_bits++;
        }
        else
            return numout;

        low <<= 1;
        high <<= 1;
        high |= 1;
        code <<= 1;
        code += InputBit( stream );
        numout++;
    }
}

int remove_symbol_from_stream1( stream, s, bits )
BIT_FILE *stream;
SYMBOL *s;
int bits;
{
      long range;
      int numout = 0;

      range = (long)(high - low) + 1;
      high = low + (unsigned short int)
         ((range * s->high_count) / s->scale - 1);
      low = low + (unsigned short int)
         ((range * s->low_count) / s->scale);

      for (;;) {
        if ((high & 0x8000) == (low & 0x8000)) {
               if (underflow_bits > 0)
                      underflow_bits = 0;
        }
        else if ((low & 0x4000) == 0x4000 && (high & 0x4000) == 0) {
               code ^= 0x4000;
               low  &= 0x3fff;
               high |= 0x4000;
               underflow_bits++;
        }
        else
               return numout;

        low <<= 1;
        high <<= 1;
        high |= 1;
        code <<= 1;
        if (bits + 16 < PACKET_SIZE) {
               code += InputBit(stream);
               bits++;
        }
        else {
               code += 0;
               bits++;
        }
        numout++;
      }
}

/*
 *   This function is called once at the beginning of compression or expansion
processing to access a standard_
```

```
 * compressed PPM Context Statistical Model and to generate the network
realization of this PPM Context Statistical  * Model in processor memory.  This
memory Model remains unchanged during subsequent calls of compression or
expansion  * functions.
 *
 */

void LoadModel( input1, argc, argv)
BIT_FILE *input1;
int argc;
char *argv[];
{
        SYMBOL s;
        int c, count;
        int thresh=0;
        int i;

        thresh = 0;

        initialize_options(argc, argv);

        initialize_model();
        initialize_arithmetic_decoder( input1 );

        for ( ; ; ) {
            do {
                get_symbol_scale( &s );
                count = get_current_count( &s );
                c = convert_symbol_to_int( count, &s );
                remove_symbol_from_stream( input1, &s );
            } while ( c == ESCAPE );

               if (c == DONE || c == END_PACKET)
                    break;
            if ( c != FLUSH )
            {}
            else
            ;
            update_model( c );
            add_character_to_model( c );
        } //end model read in for loop


        for(i = 0; i< 258; i++)
                modtotals[i] = totals[i];
        for(i = 0; i<20; i++)
                modcontexts[i] = (int)contexts[i-2];
}


int readTheStatsFile(filename)
char* filename;
{
       int Context = 0;

       int    Index = 0;
       int    Index0 = 0;
       int    Index1 = 0;
       int    Index2 = 0;
       int    Index3 = 0;
       int    Index4 = 0;


       int Order = 0;
       int Symbol = 0;
       int Count = 0;
```

```
        int totCount = -1;
        int totVal = 0;
        int curorder = -10;

        int totscore = -1;
        int valscore = 0;

        char    ioBuffer[1024];
        char * tok;
        FILE    *file;

        char    parent[10];
        char    lesser[10];

        int maximum_ID = -1;
        CONTEXT** lookup_table = 0;
        for (int pass = 0; pass < 2; pass++)
        {
                if (pass == 1)
                {
                        lookup_table = (CONTEXT**) calloc(maximum_ID,
sizeof(CONTEXT*));
                        lookup_table[0] = contexts[-1];


                }


        file = fopen(filename, "r");

        current_order = -10;

        if (file != NULL)
        {
                if (pass == 1)
                {
                        printf("\nReading stats parameters\n");
                }
                ioBuffer[0] = ' ';

                while (!feof(file))
                {
                        fgets(ioBuffer, 1024, file);
                        ioBuffer[strlen(ioBuffer) - 1] = '\0';
                        if ((ioBuffer[0] != '\0') && (ioBuffer[0] != '#'))
                        {
                              tok = strtok(ioBuffer, ",");
                              while (tok != NULL)
                              {

                                      if (strstr(tok, "C0:"))
                                      {
                                              // Receive Context
                                              Context = atoi(strchr(tok, ':') + 1);
                                      }
                                      else if (strstr(tok, "I:"))
                                      {
                                              // Receive Order0 set
                                              Index = atoi(strchr(tok, ':') + 1);
                                      }
                                      else if (strstr(tok, "I0:"))
                                      {
                                              // Receive Order0 set
                                              Index0 = atoi(strchr(tok, ':') + 1);
                                      }

                                      else if (strstr(tok, "I1:"))
```

```
                                {
                                        // Receive Order1 set
                                        Index1 = atoi(strchr(tok, ':') + 1);
                                }

                                else if (strstr(tok, "I2:"))
                                {
                                        // Receive Order2 set
                                        Index2 = atoi(strchr(tok, ':') + 1);
                                }

                                else if (strstr(tok, "I3:"))
                                {
                                        // Receive Order3 set
                                        Index3 = atoi(strchr(tok, ':') + 1);
                                }
                                else if (strstr(tok, "I4:"))
                                {
                                        // Receive Order3 set
                                        Index4 = atoi(strchr(tok, ':') + 1);
                                }

                                else if (strstr(tok, "O:"))
                                {
                                        // Order number
                                        Order = atoi(strchr(tok, ':') + 1);
                                }

                                else if (strstr(tok, "S:"))
                                {
                                        // Character or symbol number
                                        Symbol = atoi(strchr(tok, ':') + 1);
                                }

                                else if (strstr(tok, "C:"))
                                {
                                        // Receive count
                                        Count = atoi(strchr(tok, ':') + 1);
                                }
                                else if (strstr(tok, "P:"))
                                {
                                        // Receive count
                                        strcpy(parent, strchr(tok, ':') + 1);

                                }
                                else if (strstr(tok, "PL:"))
                                {
                                        // Receive count
                                        strcpy(lesser, strchr(tok, ':') + 1);
                                }
                                else if (strstr(tok, "totals:"))
                                {
                                        // Receive count
                                        totCount = atoi(strchr(tok, ':') + 1);
                                }
                                else if (strstr(tok, "val:"))
                                {
                                        // Receive count
                                        totVal = atoi(strchr(tok, ':') + 1);
                                }
                                else if (strstr(tok, "current_order:"))
                                {
                                        // Receive count
                                        curorder = atoi(strchr(tok, ':') + 1);
                                        Context = -10;
                                        totscore = -1;
                                        valscore = -1;
                                        }

                                        tok = strtok(NULL, ",");
                                }


                                if (curorder > -10){
                                        current_order = curorder;
                                        curorder = -10;
                                }
                                else if (totCount >= 0){
                                        totals[totCount] = totVal;
                                        totCount = -1;

                                        if (pass==1){
                                                putc('.', stdout);
                                        }
                                }

                                else if (Context > -10){

                                        int parent_as_int = atoi (parent);
                                        if (pass == 0)
                                        {
                                                if (parent_as_int > maximum_ID)
                                                        maximum_ID = parent_as_int;
                                        }
                                        else // pass 1
                                        {
                                                if (Context > -1){
                                                        lookup_table[parent_as_int] =
create_model(Context, Index0, Index1, Index2, Index3, Index4, Order, Symbol,
Count);

                                                        int lesser_as_int =
atoi(lesser);
                                                        if (lesser_as_int >= 0)

lookup_table[parent_as_int]->lesser_context = lookup_table[lesser_as_int];
                                                        else

lookup_table[parent_as_int]->lesser_context = 0;
                                                }
                                        }
                                }
                        }
                }



        }
        else
        {
                fprintf(stderr, "Error opening statistics file %s.\n", filename);
                exit(0);

        }
        fclose(file);
        } // pass 0, 1

        return(1);

}


void LoadStatsModel(input1, argc, argv, statsFilename)
```

```
BIT_FILE *input1;
int argc;
char *argv[];
char *statsFilename;
{
      int thresh = 0;
      int j;

      thresh = 0;

      initialize_options(argc, argv);

      initialize_model_load();

      // this is not required here
      //initialize_arithmetic_decoder_load(input1);

      // populate the data model
      readTheStatsFile(statsFilename);

      for (j = 0; j< 258; j++)
         modtotals[j] = totals[j];

      for (j = 0; j<20; j++)
         modcontexts[j] = (int)contexts[j - 2];

      printf("\nInitialised\n");

}


/*
 *
 * CompressFile routine uses a PPM Context Statistical Model in processor memory
to compress the Input2 text file  * into binary compressed packets in the
output1 file.
 * Once created the model is not changed. Input2 is compressed and packetized
 * based on the static probability model. This precludes the model from adapting
to new
 * input data, but also makes synchronization between tx compression and rx
expansion much more reliable.
 *
 */

int CompressMessages( input2, output1, argc, argv )
FILE *input2;
BIT_FILE *output1;
int argc;
char *argv[];
{
          SYMBOL s;
          int c;
          int escaped;
          int flush = 0;
          long int text_count = 0;
        long int esc_count = 0;
        int i=0,j=0;
        int bitsleft = 0;
        int mem[20] = {0};
        int numchar = 0;
        int newpacket = 0;
        int lastorder = 0;
        int orderflag = 0;
        int ordertime = 0;
        int cocount = 0;
        int biplast=0;
        int order_drop_flag=0;
        long int filepos;
        int highsave, lowsave,bipsave,currentordersave,lastordersave;
        int reset=0;
        int numcharenc=0;
        int tempwrite=0;
        int endpacket=0;
        int orderflagsave, ordertimesave, odfsave;
        char scoreboardsave[ 256 ];
        int contextssave[20] = {0};
        int redo = 0;
        int numcharsave=0;
        int masksave=0;
        int racksave=0;
        int ufbitssave=0;
        int firsttime=0;
//        int backtrack=0;
        int count=0;
        int thresh=0;

//        char inact_text[]="CHANNEL INACTIVE...";
//        int numtext=19;
//        int index5=0;
//        int index5save=0;
        int chanin=0;
        int chaninsave=0;

        FILE *packettext,*packetlength,*pkstats;

        if (input_pos==0)
        {
                packettext = fopen("packettext.dat","w");
                packetlength = fopen("packetlength.dat","w");
                pkstats = fopen("pkstats.dat","w");
        }
        else
        {
                packettext = fopen("packettext.dat","a");
                packetlength = fopen("packetlength.dat","a");
                pkstats = fopen("pkstats.dat","a");
        }

        thresh = 0;
        fprintf(pkstats,"Pknum\tUncompBits\tCompBits\tAvgOrder\n");

        esc_count = 0;

        lastorder = current_order;
        fseek ( input2 , input_pos , SEEK_SET );

        newpacket = 1;
        pknum++;
        bitsinpacket = 0;
        biplast=0;
        numchar = 0;
        current_order = max_order;
        order_drop_flag=0;
        numcharenc=0;
        endpacket=0;
        tempwrite=0;
        firsttime=0;
//        backtrack=0;
        bitsleft = 0;
        fprintf(packettext,"\n\n%d\n",pknum);
        cocount = 0;

        initialize_model2();
        initialize_arithmetic_encoder();
        for(i = 0; i<20; i++)
            contexts[i-2] = (CONTEXT *)modcontexts[i];
            for(i = 0; i< 258; i++)
                    totals[i] = modtotals[i];

                for ( ; ; ) {

                        lastorder = current_order;
                        reset=0;
                      //  if ( !flush ){
                            if (redo && numchar == numcharenc){
                                    c = END_PACKET;
                            }
                            else {
                                if (!chanin){
                                                       c = getc( input2 );
                                                       if (c==EOF)
                                                              chanin=1;
                                }
                            }
                      //    }
                    //    else
                    //        c = FLUSH;

                        if ( chanin && newpacket )
                            c = DONE;

                        do {
                                escaped = convert_int_to_symbol2( c, &s );
                                bitsinpacket += encode_symbol1( output1, &s,
tempwrite);
                                if ( escaped ) esc_count++;
                        } while ( escaped );

                        if (orderflag)
                                ordertime = 1;
                        if (current_order < lastorder){
                                orderflag = 1;
                                ordertime = 0;
                        }

                        numchar++;
                        cocount += current_order;

                      //  if (pknum == 90 && numchar>=120)
                      //      i++;i--;

                        biplast=bitsinpacket;
                        newpacket = 0;


                        fprintf(packettext,"%c",c);

                       //Prepare for End of Packet Processing
                        if( (bitsinpacket >= PACKET_SIZE - 16) || c==END_PACKET)
{
                            if (bitsinpacket + underflow_bits + 1 < PACKET_SIZE)
{
                                endpacket=1;
                                redo=0;
                            }

                                          //Too many bits, try with one less
char and EOP
                            if (!endpacket || (tempwrite && endpacket)) {
                                reset=1;
                                redo=1;
//                                backtrack++;

                                if (!endpacket){
                                    if(numcharenc==0)
                                        numcharenc=numchar-2;
                                    else
                                        numcharenc--;
                                }

                                if (endpacket && tempwrite){
                                    endpacket=0;
                                    tempwrite=0;
                                }

                                //reset everything
                                fseek ( input2 , filepos , SEEK_SET );

                                high=highsave;
                                low=lowsave;
                                bitsinpacket=bipsave;
                                current_order=currentordersave;
                                lastorder=lastordersave;
                                order_drop_flag=odfsave;
                                ordertime=ordertimesave;
                                orderflag=orderflagsave;
                                numchar=numcharsave;
                                output1->mask=masksave;
                                output1->rack=racksave;
                                underflow_bits=ufbitssave;
//                                index5=index5save;
                                chanin=chaninsave;
                                for(i = 0; i<20; i++){
                                    contexts[i-2] = (CONTEXT*)contextssave[i];
                                }

                                for (i=0;i<256;i++){
                                    scoreboard[i] = scoreboardsave[i];
                                }
                            }

                        } //end bip check
                        //Special Case--end file while in tempwrite mode
                        else if (c==DONE){
                            if (tempwrite){
                                reset=1;
                                redo=1;
                                tempwrite=0;
                                //reset everything
                                fseek ( input2 , filepos , SEEK_SET );
                                high=highsave;
                                low=lowsave;
                                bitsinpacket=bipsave;
                                current_order=currentordersave;
                                lastorder=lastordersave;
                                order_drop_flag=odfsave;
                                ordertime=ordertimesave;
                                orderflag=orderflagsave;
                                numchar=numcharsave;
                                output1->mask=masksave;
                                output1->rack=racksave;
                                underflow_bits=ufbitssave;
//                                                index5 = index5save;
                                                chanin = chaninsave;
                                for(i = 0; i<20; i++){
                                        contexts[i-2] =
(CONTEXT*)contextssave[i];
```

```
                                }

                                for (i=0;i<256;i++) {
                                        scoreboard[i] = scoreboardsave[i];
                                }
                            }
                            else
                                          {
                                    endpacket=1;
                            }
                        }

                        //Finalize End of Packet Processing and reset for next
packet
                        if (endpacket){
                            fprintf(packetlength,"%d\t%d\t",pknum,bitsinpacket);
                            bitsinpacket+=flush_arithmetic_encoder1(output1);
                            fprintf(packetlength,"%d\n",bitsinpacket);
                            printf("Packet %d done\n",pknum);


fprintf(pkstats,"%d\t\t%d\t\t%d\t\t%0.2f\n",pknum,numchar*8,bitsinpacket,(float)
cocount/(float)numchar);

                            fclose(packettext);
                            fclose(packetlength);
                            fclose(pkstats);

                            if (c==DONE){
                                input_pos=ftell(input2);
                                return (-1);
                            }
                            else {
                                input_pos=ftell(input2);
                                return (input_pos);
                            }
                        }

                        if (!reset) {
                            if ( c == DONE )
                                break;
                            if ( c == FLUSH ) {
                                flush = 0;
                            }

                            if (!newpacket)
                            {
                                    update_model2( c );
                                    add_character_to_model2( c , ordertime );
                            }

                            if (current_order == max_order)
                            {
                                    ordertime = 0;
                                    orderflag = 0;
                            }

                        } //end reset

                        //Set checkpoint and save encoder state
                        if (bitsinpacket >= PACKET_SIZE - 96 && firsttime==0) {
                            //          if (pknum ==61)
                    //            i++;i--;

                                          firsttime=1;
                            tempwrite=1;
                            filepos=ftell(input2);
                            highsave=high;
                            lowsave=low;
                            bipsave=bitsinpacket;
                            currentordersave=current_order;
                            lastordersave=lastorder;
                            odfsave=order_drop_flag;
                            ordertimesave=ordertime;
                            orderflagsave=orderflag;
                            ufbitssave=underflow_bits;
                            numcharsave=numchar;
                            masksave=output1->mask;
                            racksave=output1->rack;
//                            index5save=index5;
                            chaninsave=chanin;
                            for(i = 0; i<20; i++){
                                contextssave[i] = (int)contexts[i-2];
                            }

                            for (i=0;i<256;i++){
                                scoreboardsave[i] = scoreboard[i];
                            }
                        }
                }
        return(-2);
}

/*
 * Expansion algorithm uses a fixed, memory-based PPM Context Statistical Model
and interprets binary compressed  * packets into output text.
 *
 */


int ExpandFile( input2, output, argc, argv )
BIT_FILE *input2;
FILE *output;
int argc;
char *argv[];
{
    SYMBOL s;
    int c;
    int count;
    int flush = 0;
    long int esc_count = 0;
    int i = 0;
    int bitsread = 0;
    int first = 1;
    FILE *debug;
    int numleft = 0;
    int newpacket = 0;
    int lastorder = 0;
    int orderflag = 0;
    int ordertime = 0;
    int bitsin=0;
    int thresh=0;
    int numchar=0;

    //Input1 is binary model file
    //Input2 is binary compressed file to be expanded
    debug = fopen("debugexpand.dat","w");

    thresh = 0;//_set_sbh_threshold(0);

    initialize_options(argc, argv);

    fseek ( input2->file , input_pos , SEEK_SET );
    initialize_model2();
    initialize_arithmetic_decoder1( input2,offset );

    // If first packet begin reading right away
    for(i = 0; i<20; i++)
        contexts[i-2] = (CONTEXT*)modcontexts[i];

    for(i = 0; i< 258; i++)
        totals[i] = modtotals[i];
    current_order = max_order;
    newpacket = 1;
    offset = 16;
    current_order = max_order;
    pknum++;
    bitsread = 0;

    for ( ; ; ) {
        lastorder = current_order;

        do {
               //dont let it get below -2 as this will fail to find a symbol
               if (current_order < -2) current_order = -2;
            get_symbol_scale2( &s );
            count = get_current_count( &s );
            c = convert_symbol_to_int1( count, &s, debug );

            if (c == -10){
                printf("\n\nBit Error : Packet %d...Skipping to next 
packet\n\n",pknum);
            }
            else
               {
                if ( bitsread < PACKET_SIZE - 100 ){
                    bitsread += remove_symbol_from_stream( input2, &s );
                    newpacket = 0;
                }
                else {
                    bitsin = bitsread;
                    bitsread += remove_symbol_from_stream1( input2, &s, bitsin 
);

                    newpacket = 0;
                }
            }
        } while ( c == ESCAPE );

        if ( orderflag)
            ordertime = 1;
        if ( current_order < lastorder)
        {
            orderflag = 1;
            ordertime = 0;
        }

        numchar++;

        if ( c != FLUSH && c !=END_PACKET && c!=DONE && c!=-10 && c!=13){
            putc( (char) c, output );
        }
        else
        ;       //flush_model();

        if ((bitsread >= PACKET_SIZE - 16 + underflow_bits) || c==END_PACKET || 
c==DONE || c==-10) {
            if (c==-10){
                offset = PACKET_SIZE - bitsread + 16;
            }
            else {
                offset = (PACKET_SIZE - bitsread);
            }

            if (offset < 16)
                offset = 16;

            printf("Packet %d done\n",pknum);

            if (c==DONE) {
                input_pos=ftell(input2->file);
                return (-1);
            }
            else {
                input_pos=ftell(input2->file);
               // if (input_pos != (int)(pknum*PACKET_SIZE/8))
               //     i++;i--;
                //input_pos=(int)(pknum*PACKET_SIZE/8);
                if (c==-10)
                    input_pos = (long int)(pknum*PACKET_SIZE/8);
                return (input_pos);
            }
        }

        if (!newpacket){
            update_model2( c );
            add_character_to_model2( c , ordertime );
        }

        if ( current_order == max_order){
            ordertime = 0;
            orderflag = 0;
        }
    }       // end message for loop
           // If last packet is done, exit loop
}         //End Compression procedures

/* Bit File I/O Procedures
 *
 * The basis of the PPM Context Statistical Model Software is the open source
Arith-N.C code of Mark Nelson and
 * Jean-loup Gailly as presented in The Data Compression Book (2nd edition).
 *
 */

#define PACIFIER_COUNT 2047

BIT_FILE *OpenOutputBitFile( name )
char *name;
{
    BIT_FILE *bit_file;

    bit_file = (BIT_FILE *) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
        return( bit_file );
    bit_file->file = fopen( name, "wb" );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
    return( bit_file );
}

BIT_FILE *OpenInputBitFile( name )
char *name;
{
    BIT_FILE *bit_file;

    bit_file = (BIT_FILE *) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
        return( bit_file );
    bit_file->file = fopen( name, "rb" );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
    return( bit_file );
}

void CloseOutputBitFile( bit_file )
BIT_FILE *bit_file;
{
    if ( bit_file->mask != 0x80 )
        if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
            fatal_error( "Fatal error in CloseBitFile!\n" );
    fclose( bit_file->file );
    free( (char *) bit_file );
}

void CloseInputBitFile( bit_file )
BIT_FILE *bit_file;
{
    fclose( bit_file->file );
    free( (char *) bit_file );
}

int OutputBit( bit_file, bit, tempwrite )
BIT_FILE *bit_file;
int bit, tempwrite;
{
        int bitcount = 0;
        int boolean = 0;


        if ( bit )
        bit_file->rack |= bit_file->mask;
    bit_file->mask >>= 1;
    if ( bit_file->mask == 0 ) {
            if (!tempwrite)
                boolean = putc( bit_file->rack, bit_file->file );
            else
                boolean = bit_file->rack;
            bitcount += 8;
            if ( boolean != bit_file->rack )
                fatal_error( "Fatal error in OutputBit!\n" );
            else {
                if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
                    putc( '.', stdout );
            }
        bit_file->rack = 0;
        bit_file->mask = 0x80;
    }

        return bitcount;
}

int OutputBits( bit_file, code, count )
BIT_FILE *bit_file;
unsigned long code;
int count;
{
    unsigned long mask;
        int bitcount = 0;
        int out = 0;

    mask = 1L << ( count - 1 );
    while ( mask != 0) {
        if ( mask & code )
            bit_file->rack |= bit_file->mask;
        bit_file->mask >>= 1;
        if ( bit_file->mask == 0 ) {
            out = putc( bit_file->rack, bit_file->file );
            bitcount += 8;
            if (  out!= bit_file->rack )
                   fatal_error( "Fatal error in OutputBits!\n" );
        else if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
            putc( '.', stdout );
        bit_file->rack = 0;
        bit_file->mask = 0x80;
        }
        mask >>= 1;
    }
        return out;
}

int InputBit( bit_file )
BIT_FILE *bit_file;
{
    int value;

    if ( bit_file->mask == 0x80 ) {
        bit_file->rack = getc( bit_file->file );
        if ( bit_file->rack == EOF )
            fatal_error( "Fatal error in InputBit!\n" );
    if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
            putc( '.', stdout );
    }
    value = bit_file->rack & bit_file->mask;
    bit_file->mask >>= 1;
    if ( bit_file->mask == 0 )
        bit_file->mask = 0x80;
    return( value ? 1 : 0 );
}

unsigned long InputBits( bit_file, bit_count )
BIT_FILE *bit_file;
int bit_count;
{
    unsigned long mask;
    unsigned long return_value;

    mask = 1L << ( bit_count - 1 );
    return_value = 0;
    while ( mask != 0) {
        if ( bit_file->mask == 0x80 ) {
            bit_file->rack = getc( bit_file->file );
            if ( bit_file->rack == EOF )
                fatal_error( "Fatal error in InputBit!\n" );
        if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
                putc( '.', stdout );
        }
        if ( bit_file->rack & bit_file->mask )
            return_value |= mask;
        mask >>= 1;
        bit_file->mask >>= 1;
        if ( bit_file->mask == 0 )
            bit_file->mask = 0x80;
    }
    return( return_value );
}

void FilePrintBinary( file, code, bits )
FILE *file;
unsigned int code;
```

```
int bits;
{
    unsigned int mask;

    mask = 1 << ( bits - 1 );
    while ( mask != 0 ) {
        if ( code & mask )
            fputc( '1', file );
        else
            fputc( '0', file );
        mask >>= 1;
    }
}
```

/* End BIT I/O procedures */

ANNEX D. PPM EXPANSION CODING

D.1. PPM EXPANSION CODE

The PPM expansion of message text for the NATO REM modes may be carried out using the code defined below. When build, the expansion programme requires three files:

1. the statistical model file (STANAG4724StatisticalModel.cmp);
2. a file containing the compressed message stream;
3. an output file for the decompressed message stream.

The code below defines PACKET_SIZE. This is the size of the compressed broadcast message stream used in each data frame. The value of PACKET_SIZE is dependent on the type of REM coding being implemented. The following values should be used for PACKET_SIZE when compiling the code for the REM defined in CHAPTER 3:

CODING	PACKET_SIZE
RED/BLACK REM ALPHA coding	PACKET_SIZE = 444 bytes
RED/BLACK REM BRAVO coding	PACKET_SIZE = 672 bytes
RED REM ALPHA coding	PACKET_SIZE = 484 bytes
RED REM BRAVO coding	PACKET_SIZE = 681 bytes

```
/************************ Start of MAIN-E.C ************************/
/*
 *
 * Expansion Test Main (Driver)
 *
 *
 * PPM Context Statistical Model Usage for text compression into binary packets and
     subsequent expansion of
 * said binary packets back into text format.
 *
 * The basis of the PPM Context Statistical Model Compression and Expansion
     Software is the open source Arith-N.C code  * of Mark Nelson and Jean-loup
     Gailly as presented in The Data Compression Book (2nd edition).
 *
 * The software was modified, enhanced, and adapted for packet communications usage
     by Technology Service Corporation   * under contract to SPAWAR Systems Center
     Pacific.
 *
 * Revision History of this baseline file (Apr2012):
 *
 *
 * This is the driver program used when testing expansion algorithms.
 * In order to cut back on repetitive code, this version of main is
 * used with all of the compression routines.  It in order to turn into
 * a real program, it needs to have another module that supplies one
 * routine and two strings, namely:
 *
 *     void ExpandFile( BIT_FILE *input2, FILE *output_
 *                        int argc, char *argv );
 *     char *Usage;
 *     char *CompressionName;
 *
 * The main() routine supplied here has the job of checking for valid
 * input and output files, opening them, and then calling the
 * compression routine.  If the files are not present, or no arguments
 * are supplied, it prints out an error message, which includes the
 * Usage string supplied by the compression module.  All of the
 * routines and strings needed by this routine are defined in the
 * main.h header file.
 *
 * After this is built into a compression program of any sort, the
 * program can be called like this:
 *
 *   main-e infile2 outfile [ options ]
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <math.h>
#include <malloc.h>

typedef struct bit_file {
    FILE *file;
    unsigned char mask;
    int rack;
    int pacifier_counter;
} BIT_FILE;

#ifdef __STDC__

void usage_exit( char *prog_name );
void print_ratios( char *input, char *output );
long file_size( char *name );
long file_size_2( char *name );
```

```
void LoadModel( BIT_FILE *input1, int argc, char *argv[]);
int readTheStatsFile(char* filename);
void LoadStatsModel(BIT_FILE *input1, int argc, char *argv[], char *statsFilename);

int CompressMessages( FILE *input2, BIT_FILE *output1, int argc, char *argv[] );
int ExpandFile( BIT_FILE *input2, FILE *output, int argc, char *argv[] );

BIT_FILE     *OpenInputBitFile( char *name );
BIT_FILE     *OpenOutputBitFile( char *name );
int          OutputBit( BIT_FILE *bit_file, int bit, int tempwrite );
int          OutputBits( BIT_FILE *bit_file, unsigned long code, int count );
int           InputBit( BIT_FILE *bit_file); //, int tempwrite );
unsigned long InputBits( BIT_FILE *bit_file, int bit_count );
void          CloseInputBitFile( BIT_FILE *bit_file );
void          CloseOutputBitFile( BIT_FILE *bit_file );
void          FilePrintBinary( FILE *file, unsigned int code, int bits );

void fatal_error( char *fmt, ... );

#else

void usage_exit();
void print_ratios();
long file_size();

void LoadModel();
int readTheStatsFile();
void LoadStatsModel();
int  CompressMessages();
int  ExpandFile();

BIT_FILE      *OpenInputBitFile();
BIT_FILE      *OpenOutputBitFile();
int           OutputBit();
int           OutputBits();
int           InputBit();
unsigned long InputBits();
void          CloseInputBitFile();
void          CloseOutputBitFile();
void          FilePrintBinary();

void fatal_error();

#endif

     char *CompressionName = "Adaptive order n model with arithmetic coding";
/* Extraction procedures */

int main( argc, argv )
int argc;
char *argv[];
{
    BIT_FILE *input1, *input2;
    FILE *output;
     int status=1;
     char statsFilename[256];

     strcpy(statsFilename, "");

     setbuf( stdout, NULL );

    if ( argc < 4 )
        usage_exit( argv[ 0 ] );
     // Model Creating File
     input1 = OpenInputBitFile( argv[ 1 ]);
    if ( input1 == NULL )
```

```
        fatal_error( "Error opening %s for input\n", argv[ 1 ] );

    // File to be compressed/expanded
     input2 = OpenInputBitFile( argv[2]);
    if ( input2 == NULL )
        fatal_error( "Error opening %s for input\n", argv[ 2 ] );

           //RAW Text Output
           output = fopen( argv[ 3 ],"w" );
           if ( output == NULL )
             fatal_error( "Error opening %s for output\n", argv[ 3 ] );

     if(argv[6] != NULL)
           strcpy(statsFilename, argv[6]);

     printf("\nExpanding %s to %s using %s as initial data\n", argv[2], argv[3],
     argv[1]);
     printf("Using %s\n", CompressionName);


     if (strlen(statsFilename) > 1){
           printf("\nCompressing %s to %s using STATS %s as initial data\n",
     argv[2], argv[3], argv[1]);

           //load statistics in to memory
           LoadStatsModel(input1, argc-4, argv+4, statsFilename);
       }
       else{
             printf("\nCompressing %s to %s using MODEL %s as initial data\n",
     argv[2], argv[3], argv[1]);

           //Load Model into memory
           LoadModel(input1, argc-4, argv+4);
      }
     //Compress Data into Packets as long as input buffer(file) is not empty
     (ended)
     while (status > 0)
     {
           status = ExpandFile( input2, output, argc -4 , argv +4 );
     }

     //Check for error if last return value is not EOF indicator (-1)
     if (status != -1)
           printf("Possible Error Ending File\n");
     fclose( output );
     CloseInputBitFile(input1);
     CloseInputBitFile(input2);

     printf("Message Compression Ratio : ");
     print_ratios( argv[ 2 ], argv[ 3 ] );

    return( 0 );
}

/*
 * This routine just wants to print out the usage message that is
 * called for when the program is run with no parameters.  The first
 * part of the Usage statement is supposed to be just the program
 * name.  argv[ 0 ] generally holds the fully qualified path name
 * of the program being run.  I make a half-hearted attempt to strip
 * out that path info and file extension before printing it.  It should
 * get the general idea across.
 */
void usage_exit( prog_name )
char *prog_name;
{
    char *short_name;
    char *extension;
     char *Usage = "in-file1 in-file2 out-file [ -o order ][stats-in-file]\n\n";

    short_name = strrchr( prog_name, '\\' );
    if ( short_name == NULL )
        short_name = strrchr( prog_name, '/' );
    if ( short_name == NULL )
        short_name = strrchr( prog_name, ':' );
    if ( short_name != NULL )
        short_name++;
    else
        short_name = prog_name;
    extension = strrchr( short_name, '.' );
    if ( extension != NULL )
        *extension = '\0';
    printf( "\nUsage:  %s %s\n", short_name, Usage );
    exit( 0 );
}

/*
 * This routine is used by main to print out get the size of a file after
 * it has been closed.  It does all the work, and returns a long.  The
 * main program gets the file size for the plain text, and the size of
 * the compressed file, and prints the ratio.
 */
#ifndef SEEK_END
#define SEEK_END 2
#endif

long file_size( name )
char *name;
{
    long eof_ftell;
    FILE *file;

    file = fopen( name, "r" );
    if ( file == NULL )
        return( 0L );
    fseek( file, 0L, SEEK_END );
    eof_ftell = ftell( file );
    fclose( file );
    return( eof_ftell );
}

/*
 * This routine prints out the compression ratios after the input
 * and output files have been closed.
 */
void print_ratios( input, output )
char *input;
char *output;
{
    long input_size;
    long output_size;
    int ratio;
    FILE *summary;

    input_size = file_size( input );
    if ( input_size == 0 )
        input_size = 1;
    output_size = file_size( output );
    ratio = 100 - (int) ( output_size * 100L / input_size );
    printf( "\nInput bytes:        %ld\n", input_size );
    printf( "Output bytes:       %ld\n", output_size );
    if ( output_size == 0 )
        output_size = 1;
    //printf( "Compression ratio:  %d%%\n", ratio );
    summary = fopen( "summary.out", "w" );
    if ( summary == NULL )
        fatal_error( "Error opening summary.out for output\n");
     fprintf(summary,"Input bytes:      %ld",input_size);
     fprintf(summary,"  Output bytes:     %ld",output_size);
     fprintf(summary,"  Compression ratio:  %d%%\n",ratio);
    fclose(summary);

}


void fatal_error( fmt )
char *fmt;
{
    va_list argptr;

    va_start( argptr, fmt );
    printf( "Fatal error: " );
    vprintf( fmt, argptr );
    va_end( argptr );
    exit( -1 );
}

/* Compression procedures
 * Compression and Expansion routines for use with packet communications using PPM
     Context Statistical Model
 * as input.
 *
 */



/*
 * The SYMBOL structure is what is used to define a symbol in
 * arithmetic coding terms.
 */

typedef struct {
    unsigned short int low_count;
    unsigned short int high_count;
    unsigned short int scale;
} SYMBOL;

#define MAXIMUM_SCALE   16383  /* Maximum allowed frequency count */
#define ESCAPE          256    /* The escape symbol               */
#define DONE            (-1)   /* The output stream empty  symbol */
#define FLUSH           (-2)   /* The symbol to flush the model   */
#define END_PACKET      (-3)
#define PACKET_SIZE     484    /* Packet Size In Bits (RED: 484, RED/BLACK: 444) */
#define UPDATE_EXC      1      /* Update Exclusion Flag, 1 = on, 0 = off 0 */

/*
 * Function prototypes.
 */

#ifdef __STDC__

void initialize_options( int argc, char **argv );
int check_compression( FILE *input, BIT_FILE *output );
void initialize_model( void );
void initialize_model2( void );
void initialize_model_load(void);
void update_model( int symbol );
void update_model2( int symbol );
int convert_int_to_symbol( int symbol, SYMBOL *s );
int convert_int_to_symbol2( int symbol, SYMBOL *s );
void get_symbol_scale( SYMBOL *s );
void get_symbol_scale2( SYMBOL *s );
```

```
int convert_symbol_to_int( int count, SYMBOL *s );
int convert_symbol_to_int1( int count, SYMBOL *s, int debug );
void add_character_to_model( int c );
void add_character_to_model2( int c, int flag );
void flush_model( void );
void initialize_arithmetic_decoder( BIT_FILE *stream );
void initialize_arithmetic_decoder1( BIT_FILE *stream , int numbits);
int remove_symbol_from_stream( BIT_FILE *stream, SYMBOL *s );
int remove_symbol_from_stream1( BIT_FILE *stream, SYMBOL *s, int bits );
void initialize_arithmetic_encoder( void );
int encode_symbol1( BIT_FILE *stream, SYMBOL *s, int tempwriteflag);
void encode_symbol( BIT_FILE *stream, SYMBOL *s );
int flush_arithmetic_encoder1( BIT_FILE *stream );
void flush_arithmetic_encoder( BIT_FILE *stream );
short int get_current_count( SYMBOL *s );
void save_model(void);
void reset_model(void);

#else /* __STDC_, */

void initialize_options();
int check_compression();
void initialize_model();
void initialize_model2();
void initialize_model_load();
void update_model();
void update_model2();
int convert_int_to_symbol();
int convert_int_to_symbol2();
void get_symbol_scale();
void get_symbol_scale2();
int convert_symbol_to_int();
int convert_symbol_to_int1();
void add_character_to_model();
void add_character_to_model2();
void flush_model();
void initialize_arithmetic_decoder();
void initialize_arithmetic_decoder1();
int remove_symbol_from_stream();
int remove_symbol_from_stream1();
void initialize_arithmetic_encoder();
int encode_symbol1();
void encode_symbol();
int flush_arithmetic_encoder1();
void flush_arithmetic_encoder();
short int get_current_count();
void save_model();
void reset_model();

#endif  /* __STDC__ */

int max_order = 4;
int bitsinpacket = 0;                   //num compressed bits in packet
int pknum = 0;                          //packet number
int numalloc = 0;                       //number of memory slots allocated
long int input_pos = 0;
int offset = 16;

void initialize_options( argc, argv )
int argc;
char *argv[];
{
    while ( argc-- > 0 ) {
        if ( strcmp( *argv, "-o" ) == 0 ) {
            argc--;
            max_order = atoi( *++argv );
        } else
```

```
            printf( "Uknown argument on command line: %s\n", *argv );
        argc--;
        argv++;
    }
}

/*
 * This routine is called once every 256 input symbols.  Its job is to
 * check to see if the compression ratio falls below 10%.  If the
 * output size is 90% of the input size, it means not much compression
 * is taking place, so we probably ought to flush the statistics in the
 * model to allow for more current statistics to have greater impact.
 * This heuristic approach does seem to have some effect.
 */
int check_compression( input, output )
FILE *input;
BIT_FILE *output;
{
    static long local_input_marker = 0L;
    static long local_output_marker = 0L;
    long total_input_bytes;
    long total_output_bytes;
    int local_ratio;

    total_input_bytes  =  ftell( input ) - local_input_marker;
    total_output_bytes = ftell( output->file );
    total_output_bytes -= local_output_marker;
    if ( total_output_bytes == 0 )
        total_output_bytes = 1;
    local_ratio = (int)( ( total_output_bytes * 100 ) / total_input_bytes );

    local_input_marker = ftell( input );
    local_output_marker = ftell( output->file );

    return( local_ratio > (int)90 );
}


//Define context data structures
typedef struct {
    unsigned char symbol;
    unsigned char counts;
} STATS;


typedef struct {
    struct context *next;
} LINKS;

typedef struct context {
    int max_index;
    LINKS *links;
    STATS *stats;
    struct context *lesser_context;
} CONTEXT;

/*
 * *contexts[] is an array of current contexts.
 */
CONTEXT **contexts;

/*
 * current_order contains the current order of the model.  It starts
 * at max_order, and is decremented every time an ESCAPE is sent.  It
 * will only go down to -1 for normal symbols, but can go to -2 for
 * EOF and FLUSH.
 */
```

```
int current_order;

short int totals[ 258 ];
char scoreboard[ 256 ];
char tempscoreboard[ 256 ];

//Global vars now due to functionality break-up
int modtotals[258] = {0};
int modcontexts[20] = {0};

/*
 * Local procedure declarations for modeling routines.
 */
#ifdef __STDC__
void update_table( CONTEXT *table, int symbol );
void rescale_table( CONTEXT *table );
void totalize_table( CONTEXT *table );
void totalize_table2( CONTEXT *table );
CONTEXT *shift_to_next_context( CONTEXT *table, int c, int order);
CONTEXT *shift_to_next_context2( CONTEXT *table, int c, int order);
CONTEXT *shift_to_next_context3( CONTEXT *table, int c, int order);
CONTEXT *allocate_next_order_table( CONTEXT *table, int symbol, CONTEXT
      *lesser_context );
void recursive_flush( CONTEXT *table );
CONTEXT* create_model(int ContextT, int Index0, int Index1, int Index2, int Index3,
     int Index4, int Order, int Symbol, int Count);

#else
void update_table();
void rescale_table();
void totalize_table();
void totalize_table2();
CONTEXT *shift_to_next_context();
CONTEXT *shift_to_next_context2();
CONTEXT *shift_to_next_context3();
CONTEXT *allocate_next_order_table();
void recursive_flush();
CONTEXT* create_model();

#endif



void initialize_model()
{
    int i;
    CONTEXT *null_table;
    CONTEXT *control_table;

        current_order = max_order;
    contexts = (CONTEXT **) calloc( sizeof( CONTEXT * ), 20 );
    if ( contexts == NULL )
        fatal_error( "Failure #1: allocating context table!" );
    contexts += 2;
    null_table = (CONTEXT *) calloc( sizeof( CONTEXT ), 1 );
    if ( null_table == NULL )
        fatal_error( "Failure #2: allocating null table!" );
    null_table->max_index = -1;
    contexts[ -1 ] = null_table;
    for ( i = 0 ; i <= max_order ; i++ )
        contexts[ i ] = allocate_next_order_table( contexts[ i-1 ],0, contexts[ i-1
     ] );
         null_table->stats =
           (STATS *) realloc( (char *) null_table->stats, sizeof( STATS )*256 );

    if ( null_table->stats == NULL )
           fatal_error( "Failure #3: allocating null table!" );
     null_table->max_index = 255;
    for ( i=0 ; i < 256 ; i++ ) {
        null_table->stats[ i ].symbol = (unsigned char) i;
        null_table->stats[ i ].counts = 1;
    }

    control_table = (CONTEXT *) calloc( sizeof(CONTEXT), 1 );
    if ( control_table == NULL )
        fatal_error( "Failure #4: allocating null table!" );
    control_table->stats =
         (STATS *) calloc( sizeof( STATS ), 2 );
    if ( control_table->stats == NULL )
        fatal_error( "Failure #5: allocating null table!" );
    contexts[ -2 ] = control_table;
    control_table->max_index = 2;
    control_table->stats[ 0 ].symbol = -FLUSH;
    control_table->stats[ 0 ].counts = 1;
    control_table->stats[ 1 ].symbol = -DONE;
    control_table->stats[ 1 ].counts = 1;
        control_table->stats[ 2 ].symbol = -END_PACKET;
    control_table->stats[ 2 ].counts = 1;

    for ( i = 0 ; i < 256 ; i++ )
        scoreboard[ i ] = 0;
}



CONTEXT* create_model(ContextT, Index0, Index1, Index2, Index3, Index4, Order,
     Symbol, Count)
int ContextT;
int Index0;
int Index1;
int Index2;
int Index3;
int Index4;
int Order;
int Symbol;
int Count;
{
     int index = 0;
     int new_size;
     CONTEXT *lesser_context = NULL;
     CONTEXT *table = NULL;
     CONTEXT *new_table = NULL;
     struct node *ptr = NULL;
     CONTEXT *found_table = NULL;


     if (Order == -1)
     {

           if (contexts[ContextT] == NULL){
             new_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
             if (new_table == NULL)
                    fatal_error("Failure CONTEXT allocating!");

             new_table->max_index = 0;
             new_table->lesser_context = NULL;

             contexts[ContextT] = new_table;
           }
           table = contexts[ContextT];
           index = ContextT;
           table->max_index = Count;
     }
     else if (Order == 0){
           table = contexts[ContextT];
           index = Index0;

           if (table->stats == NULL){
             table->stats = (STATS *)calloc(1, sizeof(STATS));
             if (table->stats == NULL)
                    fatal_error("Failure STATS allocating!");
           }

           if (table->links == NULL){
             table->links = (LINKS *)calloc(1, sizeof(LINKS));
             if (table->links == NULL)
                    fatal_error("Failure LINKS allocating!");

             table->links->next = NULL;
           }
     }
     else if (Order == 1){

           if (contexts[ContextT]->links[Index0].next == NULL){
             new_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
             if (new_table == NULL)
                    fatal_error("Failure CONTEXT allocating!");

             new_table->max_index = -1;
             new_table->lesser_context = NULL;


             contexts[ContextT]->links[Index0].next = new_table;
           }

           table = contexts[ContextT]->links[Index0].next;
           index = Index1;

     }
     else if (Order == 2){


           if (contexts[ContextT]->links[Index0].next->links[Index1].next == NULL){
             new_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
             if (new_table == NULL)
                    fatal_error("Failure CONTEXT allocating!");

             new_table->max_index = -1;
             new_table->lesser_context = NULL;


             contexts[ContextT]->links[Index0].next->links[Index1].next =
     new_table;
           }

           table = contexts[ContextT]->links[Index0].next->links[Index1].next;
           index = Index2;
     }
     else if (Order == 3){

           if (contexts[ContextT]->links[Index0].next->links[Index1].next-
     >links[Index2].next == NULL){
             new_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
             if (new_table == NULL)
                    fatal_error("Failure CONTEXT allocating!");

             new_table->max_index = -1;
             new_table->lesser_context = NULL;
             contexts[ContextT]->links[Index0].next->links[Index1].next-
     >links[Index2].next = new_table;
           }

           table = contexts[ContextT]->links[Index0].next->links[Index1].next-
     >links[Index2].next;
           index = Index3;

     }
     else if (Order == 4){

           if (contexts[ContextT]->links[Index0].next->links[Index1].next-
     >links[Index2].next->links[Index3].next == NULL){
             new_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
             if (new_table == NULL)
                    fatal_error("Failure CONTEXT allocating!");

             new_table->max_index = -1;
             new_table->lesser_context = NULL;

             contexts[ContextT]->links[Index0].next->links[Index1].next-
     >links[Index2].next->links[Index3].next = new_table;
           }

           table = contexts[ContextT]->links[Index0].next->links[Index1].next-
     >links[Index2].next->links[Index3].next;
           index = Index4;

     }

     if (Order >= 0){
           new_size = sizeof(LINKS);
           new_size *= index + 1;
           if (Order <= max_order) {
             if (index == 0) {
                    table->links = (LINKS *)calloc(1, new_size);

             }
             else {
                    table->links = (LINKS *)realloc((char *)table->links,
     new_size);
             }


             if (table->links == NULL)
                    fatal_error("Error #9: reallocating table space!");

             table->links[index].next = NULL;
           }
           new_size = sizeof(STATS);
           new_size *= index + 1;
           if (index == 0){
             table->stats = (STATS *)calloc(1, new_size);
           }
           else {
             table->stats = (STATS *)realloc((char *)table->stats, new_size);
           }

           if (table->stats == NULL)
             fatal_error("Error #10: reallocating table space!");
           table->stats[index].symbol = '0';
           table->stats[index].counts = 0;


     }

     if (Order > -1)
```

```
     {             //add the stats data
           table->stats[index].symbol = (unsigned char)Symbol;
           table->stats[index].counts = Count;
           if (Order > 0)
              table->max_index++;
     }



     return table;
}


void initialize_model2()
{
    int i;

    current_order = max_order;

    for ( i = 0 ; i < 256 ; i++ )
        scoreboard[ i ] = 0;
}


/*
* This routine has to get everything set up properly so that
* the model can be maintained properly.  The first step is to create
* the *contexts[] array used later to find current context tables.
* The *contexts[] array indices go from -2 up to max_order, so
* the table needs to be fiddled with a little.  This routine then
* has to create the special order -2 and order -1 tables by hand,
* since they aren't quite like other tables.  Then the current
* context is set to \0, \0, \0, ... and the appropriate tables
* are built to support that context.  The current order is set
* to max_order, the scoreboard is cleared, and the system is
* ready to go.
*/
void initialize_model_load()
{
     int i;
     CONTEXT *null_table;
     CONTEXT *control_table;

     current_order = max_order;
     contexts = (CONTEXT **)calloc(20, sizeof(CONTEXT *)); //20
     if (contexts == NULL)
           fatal_error("Failure #1: allocating context table!");

     contexts += 2;
     null_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
     if (null_table == NULL)
           fatal_error("Failure #2: allocating null table!");
     null_table->max_index = -1;

     contexts[-1] = null_table;

     for (i = 0; i <= max_order; i++)
           contexts[i] = allocate_next_order_table(contexts[i - 1], 0, contexts[i -
     1]);

     null_table->stats = (STATS *)realloc((char *)null_table->stats, sizeof(STATS)*
     256);
     if (null_table->stats == NULL)
           fatal_error("Failure #3: allocating null table!");

     null_table->max_index = 255;
     for (i = 0; i < 256; i++) {
           null_table->stats[i].symbol = (unsigned char)i;
           null_table->stats[i].counts = 1;
     }

     control_table = (CONTEXT *)calloc(1, sizeof(CONTEXT));
     if (control_table == NULL)
           fatal_error("Failure #4: allocating null table!");
     control_table->stats = (STATS *)calloc(3, sizeof(STATS));
     if (control_table->stats == NULL)
           fatal_error("Failure #5: allocating null table!");
     contexts[-2] = control_table;

     control_table->max_index = 2;

     control_table->stats[0].symbol = -FLUSH;
     control_table->stats[0].counts = 1;
     control_table->stats[1].symbol = -DONE;
     control_table->stats[1].counts = 1;
     control_table->stats[2].symbol = -END_PACKET;
     control_table->stats[2].counts = 1;

     for (i = 0; i < 256; i++)
           scoreboard[i] = 0;
}



/*
 * This is a utility routine used to create new tables when a new
 * context is created.
 */
int num = 0;

CONTEXT *allocate_next_order_table( table, symbol, lesser_context )
CONTEXT *table;
int symbol;
CONTEXT *lesser_context;
{
    CONTEXT *new_table;
    int i;
    unsigned int new_size;
        for ( i = 0 ; i <= table->max_index ; i++ )
        if ( table->stats[ i ].symbol == (unsigned char) symbol )
            break;
    if ( i > table->max_index ) {
        table->max_index++;
        new_size = sizeof( LINKS );
        new_size *= table->max_index + 1;
        if ( table->links == NULL )
        {
            table->links = (LINKS *) calloc( new_size, 1 );
        }
        else
        {
            table->links = (LINKS *)realloc( (char *) table->links, new_size );
        }

        new_size *= table->max_index + 1;
        if ( table->stats == NULL )
                {
            table->stats = (STATS *) calloc( new_size, 1 );
                }
        else
                {
                        table->stats = (STATS *)
                realloc( (char *) table->stats, new_size );
                }

                if ( table->links == NULL )
            fatal_error( "Failure #6: allocating new table" );
        if ( table->stats == NULL )
            fatal_error( "Failure #7: allocating new table" );
        table->stats[ i ].symbol = (unsigned char) symbol;
        table->stats[ i ].counts = 0;
    }
    new_table = (CONTEXT *) calloc( sizeof( CONTEXT ), 1 );
    if ( new_table == NULL )
        fatal_error( "Failure #8: allocating new table" );
    new_table->max_index = -1;
    table->links[ i ].next = new_table;
    new_table->lesser_context = lesser_context;
    return( new_table );
}


/*
 * This routine is called to increment the counts for the current
 * contexts.  It is called after a character has been encoded or
 * decoded.
 */
void update_model( symbol )
int symbol;
{
    int i;
    int local_order;
    int loopstart;

    if ( current_order < 0 )
        local_order = 0;
    else
        local_order = current_order;

    //Determines starting point of loop based on UPDATE_EXC flag
    if (UPDATE_EXC)
            loopstart = local_order;
    else
            loopstart = 0;

    if ( symbol >= 0 ) {
        while ( loopstart <= max_order ) {
            if ( symbol >= 0 )
                update_table( contexts[ loopstart ], symbol );
            loopstart++;
        }
    }
    current_order = max_order;
    for ( i = 0 ; i < 256 ; i++ )
        scoreboard[ i ] = 0;
}


void update_model2( symbol )
int symbol;
{
    int i;

    for ( i = 0 ; i < 256 ; i++ )
        scoreboard[ i ] = 0;
}

/*
 * This routine is called to update the count for a particular symbol
 * in a particular table.  The table is one of the current contexts_
 * and the symbol is the last symbol encoded or decoded.
 */

void update_table( table, symbol )
CONTEXT *table;
int symbol;
{
    int i;
    int index;
    unsigned char temp;
    CONTEXT *temp_ptr;
    unsigned int new_size;

    index = 0;
    while ( index <= table->max_index &&
            table->stats[index].symbol != (unsigned char) symbol )
    index++;
    if ( index > table->max_index ) {
        table->max_index++;
        new_size = sizeof( LINKS );
        new_size *= table->max_index + 1;
        if ( current_order < max_order ) {
             if ( table->max_index == 0 ) {
                table->links = (LINKS *) calloc( new_size, 1 );
            }
            else {
               table->links = (LINKS *)
                realloc( (char *) table->links, new_size);
            }
            if ( table->links == NULL )
                fatal_error( "Error #9: reallocating table space!" );
            table->links[ index ].next = NULL;
        }
        new_size = sizeof( STATS );
        new_size *= table->max_index + 1;
        if (table->max_index==0){
            table->stats = (STATS *) calloc( new_size, 1 );
        }
        else  {
            table->stats = (STATS *)
            realloc( (char *) table->stats, new_size );
        }

        if ( table->stats == NULL )
            fatal_error( "Error #10: reallocating table space!" );
        table->stats[ index ].symbol = (unsigned char) symbol;
        table->stats[ index ].counts = 0;
    }

    i = index;
    while ( i > 0 &&
            table->stats[ index ].counts == table->stats[ i-1 ].counts )
        i--;
    if ( i != index ) {
        temp = table->stats[ index ].symbol;
        table->stats[ index ].symbol = table->stats[ i ].symbol;
        table->stats[ i ].symbol = temp;
        if ( table->links != NULL ) {
            temp_ptr = table->links[ index ].next;
            table->links[ index ].next = table->links[ i ].next;
            table->links[ i ].next = temp_ptr;
        }
        index = i;
    }

    table->stats[ index ].counts++;
    if ( table->stats[ index ].counts == 255 )
```

```
        rescale_table( table );
}

/*
 * This routine is called when a given symbol needs to be encoded.
 * It is the job of this routine to find the symbol in the context
 * table associated with the current table, and return the low and
 * high counts associated with that symbol, as well as the scale.
 *
 */
int convert_int_to_symbol( c, s )
int c;
SYMBOL *s;
{
    int i;
    CONTEXT *table;

    table = contexts[ current_order ];
    totalize_table( table );
    s->scale = totals[ 0 ];
    if ( current_order == -2 )
        c = -c;
    for ( i = 0 ; i <= table->max_index ; i++ ) {
        if ( c == (int) table->stats[ i ].symbol ) {
            if ( table->stats[ i ].counts == 0 )
                break;
            s->low_count = totals[ i+2 ];
            s->high_count = totals[ i+1 ];
            return( 0 );
        }
    }
    s->low_count = totals[ 1 ];
    s->high_count = totals[ 0 ];
    current_order--;
    return( 1 );
}

int convert_int_to_symbol2( c, s )
int c;
SYMBOL *s;
{
    int i;
    CONTEXT *table;

    table = contexts[ current_order ];
    totalize_table2( table );
    s->scale = totals[ 0 ];
        if (table->max_index != -1)
        {
            if ( current_order == -2 )
                c = -c;
            for ( i = 0 ; i <= table->max_index ; i++ ) {
            if ( c == (int) table->stats[ i ].symbol ) {
                if ( table->stats[ i ].counts == 0 )
                    break;
                    s->low_count = totals[ i+2 ];
                    s->high_count = totals[ i+1 ];
                    return( 0 );
                }
            }
        }
    s->low_count = totals[ 1 ];
    s->high_count = totals[ 0 ];
    contexts[current_order] = NULL;
    current_order--;
    return( 1 );
}
```

```
/*
 * This routine is called when decoding an arithmetic number.  In
 * order to decode the present symbol, the current scale in the
 * model must be determined.
 */

void get_symbol_scale( s )
SYMBOL *s;
{
    CONTEXT *table;

    table = contexts[ current_order ];
    totalize_table( table );
    s->scale = totals[ 0 ];
}

void get_symbol_scale2( s )
SYMBOL *s;
{
    CONTEXT *table;

    table = contexts[ current_order ];
    totalize_table2( table );
    s->scale = totals[ 0 ];
}

/*
 * This routine is called during decoding.  It is given a count that
 * came out of the arithmetic decoder, and has to find the symbol that
 * matches the count.
 */
int convert_symbol_to_int( count, s )
int count;
SYMBOL *s;
{
    int c;
    CONTEXT *table;

    table = contexts[ current_order ];
    for ( c = 0; count < totals[ c ] ; c++ )
        ;
    s->high_count = totals[ c - 1 ];
    s->low_count = totals[ c ];
    if ( c == 1) {
        current_order--;
        return( ESCAPE );
    }

    if ( current_order < -1 )
        return( (int) -table->stats[ c-2 ].symbol );
    else
        return( table->stats[ c-2 ].symbol );
}


int convert_symbol_to_int1( count, s, debug )
int count;
SYMBOL *s;
int debug;
{
    int c;
    CONTEXT *table;

    table = contexts[ current_order ];
    for ( c = 0; count < totals[ c ] ; c++ )
        ;

    s->high_count = totals[ c - 1 ];
    s->low_count = totals[ c ];
    if ( c == 1 ) {
        current_order--;
        return( ESCAPE );
    }

    if ( current_order < -1 )
        return( (int) -table->stats[ c-2 ].symbol );
    else
        return( table->stats[ c-2 ].symbol );
}

/*
 * After the model has been updated for a new character, this routine
 * is called to "shift" into the new context.
 */

void add_character_to_model( c )
int c;
{
    int i;
    if ( max_order < 0 || c < 0 )
       return;
    contexts[ max_order ] =
       shift_to_next_context( contexts[ max_order ], c, max_order );
    for ( i = max_order-1 ; i > 0 ; i-- )
        contexts[ i ] = contexts[ i+1 ]->lesser_context;
}

void add_character_to_model2( c, flag )
int c;
int flag;
{
    int i,local_order;

    local_order = current_order;
    if ( max_order < 0 || c < 0 )
       return;
    // Attempt to raise order if flag is 1
    if ( flag || local_order == -1)
     {
        contexts[ local_order + 1 ] =
       shift_to_next_context3( contexts[ local_order ], c, local_order );
    }
    // Stay at current_order if flag is 0 or if attempt to raise order failed
    if ( (!flag || (flag && contexts[local_order + 1] == NULL)) && local_order != -
     1 )
     {
        contexts[ current_order ] =
                shift_to_next_context2( contexts[ current_order ], c, current_order
     );
    }

    for ( i = current_order-1 ; i > 0 ; i-- )
        contexts[ i ] = contexts[ i+1 ]->lesser_context;
}

/*
 * This routine is called when adding a new character to the model. From
 * the previous example, if the current context was "ABC", and the new
 * symbol was 'D', this routine would get called with a pointer to
 * context table "ABC", and symbol 'D', with order max_order.
 */
```

```
CONTEXT *shift_to_next_context( table, c, order )
CONTEXT *table;
int c;
int order;
{
    int i;
    CONTEXT *new_lesser;

    table = table->lesser_context;
    if ( order == 0 )
        return( table->links[ 0 ].next );
    for ( i = 0 ; i <= table->max_index ; i++ )
        if ( table->stats[ i ].symbol == (unsigned char) c )
            if ( table->links[ i ].next != NULL )
                return( table->links[ i ].next );
            else
                break;

    new_lesser = shift_to_next_context( table, c, order-1 );
    table = allocate_next_order_table( table, c, new_lesser );
    return( table );
}

CONTEXT *shift_to_next_context2( table, c, order )
CONTEXT *table;
int c;
int order;
{
    int i = 0;

    table = table->lesser_context;
    if ( order < 0 || table->lesser_context == NULL)
        return( table->links[ 0 ].next );

    for ( i = 0 ; i <= table->max_index ; i++ ){
        if ( table->stats[ i ].symbol == (unsigned char) c )
            if ( table->links[ i ].next != NULL )
                return( table->links[ i ].next );
            else
                break;
    }
    return( table );
}

CONTEXT *shift_to_next_context3( table, c, order )
CONTEXT *table;
int c;
int order;
{
    int i;

    if ( order < 0 ){
        current_order++;
        return( table->links[ 0 ].next );
    }

    for ( i = 0 ; i <= table->max_index ; i++ ){
        if ( table->stats[ i ].symbol == (unsigned char) c )
            if ( table->links[ i ].next != NULL ){
                current_order++;
                return( table->links[ i ].next );
            }
            else
                break;
    }
    return( table = NULL );
}
```

```
/*
 * Rescaling the table needs to be done for one of three reasons.
 * First, if the maximum count for the table has exceeded 16383, it
 * means that arithmetic coding using 16 and 32 bit registers might
 * no longer work.  Secondly, if an individual symbol count has
 * reached 255, it will no longer fit in a byte.  Third, if the
 * current model isn't compressing well, the compressor program may
 * want to rescale all tables in order to give more weight to newer
 * statistics.
 */
void rescale_table( table )
CONTEXT *table;
{
    int i;

    if ( table->max_index == -1 )
        return;
    for ( i = 0 ; i <= table->max_index ; i++ )
        table->stats[ i ].counts /= 2;
    if ( table->stats[ table->max_index ].counts == 0 &&
         table->links == NULL ) {
        while ( table->stats[ table->max_index ].counts == 0 &&
                table->max_index >= 0 )
            table->max_index--;
        if ( table->max_index == -1 ) {
            free( (char *) table->stats );
            table->stats = NULL;
        }
           else {
            table->stats = (STATS *)
            realloc( (char *) table->stats,
            sizeof( STATS ) * ( table->max_index + 1 ) );

            if ( table->stats == NULL )
                fatal_error( "Error #11: reallocating stats space!" );
        }
    }
}

/*
 * This routine has the job of creating a cumulative totals table for
 * a given context.
 */
void totalize_table( table )
CONTEXT *table;
{
    int i;
    unsigned char max;

    for ( ; ; ) {
        max = 0;
        i = table->max_index + 2;
        totals[ i ] = 0;
        for ( ; i > 1 ; i-- ) {
            totals[ i-1 ] = totals[ i ];
            if ( table->stats[ i-2 ].counts )
                if ( ( current_order == -2 ) ||
                     scoreboard[ table->stats[ i-2 ].symbol ] == 0 )
                     totals[ i-1 ] += table->stats[ i-2 ].counts;
            if ( table->stats[ i-2 ].counts > max )
                max = table->stats[ i-2 ].counts;
        }

        if ( max == 0 )
            totals[ 0 ] = 1;
        else {
            totals[ 0 ] = (short int) ( 256 - table->max_index );
            totals[ 0 ] *= table->max_index;
            totals[ 0 ] /= 256;
            totals[ 0 ] /= max;
            totals[ 0 ]++;
            totals[ 0 ] += totals[ 1 ];
        }
        if ( totals[ 0 ] < MAXIMUM_SCALE )
            break;
        rescale_table( table );
    }

    for ( i = 0 ; i < table->max_index ; i++ )
        if (table->stats[i].counts != 0)
    scoreboard[ table->stats[ i ].symbol ] = 1;
}

void totalize_table2( table )
CONTEXT *table;
{
    int i;
    unsigned char max;

    for ( ; ; ) {
        max = 0;
        i = table->max_index + 2;
        totals[ i ] = 0;
        for ( ; i > 1 ; i-- ) {
            totals[ i-1 ] = totals[ i ];
            if ( table->stats[ i-2 ].counts )
                if ( ( current_order == -2 ) ||
                     scoreboard[ table->stats[ i-2 ].symbol ] == 0 )
                     totals[ i-1 ] += table->stats[ i-2 ].counts;
            if ( table->stats[ i-2 ].counts > max )
                max = table->stats[ i-2 ].counts;
        }

        if ( max == 0 )
            totals[ 0 ] = 1;
        else {
            totals[ 0 ] = (short int) ( 256 - table->max_index );
            totals[ 0 ] *= table->max_index;
            totals[ 0 ] /= 256;
            totals[ 0 ] /= max;
            totals[ 0 ]++;
            totals[ 0 ] += totals[ 1 ];
        }
        if ( totals[ 0 ] < MAXIMUM_SCALE )
            break;
    }
    for ( i = 0 ; i < table->max_index ; i++ )
        if (table->stats[i].counts != 0)
            scoreboard[ table->stats[ i ].symbol ] = 1;
}

/*
 * This routine is called when the entire model is to be flushed.
 */
void recursive_flush( table )
CONTEXT *table;
{
    int i;

    if ( table->links != NULL )
        for ( i = 0 ; i <= table->max_index ; i++ )
            if ( table->links[ i ].next != NULL )
                recursive_flush( table->links[ i ].next );
    rescale_table( table );
}

/*
 * This routine is called to flush the whole table, which it does
 * by calling the recursive flush routine starting at the order 0
 * table.
 */
void flush_model()
{
    putc( 'F', stdout );
    recursive_flush( contexts[ 0 ] );
}


static unsigned short int code;  /* The present input code value       */
static unsigned short int low;   /* Start of the current code range    */
static unsigned short int high;  /* End of the current code range      */
long underflow_bits;             /* Number of underflow bits pending   */

/*
 * This routine must be called to initialize the encoding process.
 */
void initialize_arithmetic_encoder()
{
    low = 0;
    high = 0xffff;
    underflow_bits = 0;
}

/*
 * At the end of the encoding process, there are still significant
 * bits left in the high and low registers.  We output two bits_
 * plus as many underflow bits as are necessary.
 */

void flush_arithmetic_encoder( stream )
BIT_FILE *stream;
{
    OutputBit( stream, low & 0x4000 , 0);
    underflow_bits++;
    while ( underflow_bits-- > 0 )
        OutputBit( stream, ~low & 0x4000 , 0 );
    OutputBits( stream, 0L, 16 );
        //low = 0;
        //high = 0xfff;
}

/* This is used at end of each packet. It flushes the encoder by
 * outputting an bits left from the last character and any underflow bits.
 * It then outputs a variable number of zeros so the decoder can sync up
 * and the next packet will start at a predictable location.
 */

int flush_arithmetic_encoder1( stream )
BIT_FILE *stream;
{
           int count = 0;
        int retval = 0;
        int bitsinstream = 0;
        int lastmask = 0;
        int zeros = 0;
        int bisorig = 0;
        int fp = 0;
        int i;

        bitsinstream = bitsinpacket;
        lastmask = stream->mask;
        OutputBit( stream, low & 0x4000,0);
        if (stream -> mask != lastmask){
            bitsinstream++;
            lastmask = stream->mask;
        }

        underflow_bits++;
        while ( underflow_bits > 0 )
        {
            OutputBit( stream, ~low & 0x4000,0 );
            if (stream -> mask != lastmask)
            {
                bitsinstream++;
                lastmask = stream->mask;
            }
            underflow_bits--;
        }
        zeros = PACKET_SIZE - bitsinstream;

        for (i=0;i<zeros;i++)
            OutputBit(stream,0,0);

        return ( zeros + (bitsinstream - bitsinpacket));
}

/*
 * This routine is called to encode a symbol.  The symbol is passed
 * in the SYMBOL structure as a low count, a high count, and a range.
 */
void encode_symbol( stream, s )
BIT_FILE *stream;
SYMBOL *s;
{
    long range;

    range = (long) ( high-low ) + 1;
    high = low + (unsigned short int)
                 (( range * s->high_count ) / s->scale - 1 );
    low = low + (unsigned short int)
                 (( range * s->low_count ) / s->scale );

    for ( ; ; ) {
        if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
            OutputBit( stream, high & 0x8000 , 0);
            while ( underflow_bits > 0 ) {
                OutputBit( stream, ~high & 0x8000 , 0 );
                underflow_bits--;
            }
        }
        else if ( ( low & 0x4000 ) && !( high & 0x4000 )) {
            underflow_bits += 1;
            low &= 0x3fff;
            high |= 0x4000;
        }
          else
            return ;
        low <<= 1;
        high <<= 1;
        high |= 1;
    }
}

//Sames as above, but returns the number of encoded bits
int encode_symbol1( stream, s, tempwriteflag )
```

```
BIT_FILE *stream;
SYMBOL *s;
int tempwriteflag;
{
    long range;
    int count = 0;
    int debug = 0;
    int lastmask = 0;
    int addedbits = 0;

    range = (long) ( high-low ) + 1;
    high = low + (unsigned short int)
                 (( range * s->high_count ) / s->scale - 1 );
    low = low + (unsigned short int)
                 (( range * s->low_count ) / s->scale );

    for ( ; ; ) {
        if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
            lastmask = stream->mask;
            debug += OutputBit( stream, high & 0x8000, tempwriteflag );
            if (stream -> mask != lastmask){
                    addedbits++;
                    lastmask = stream->mask;
            }
            while ( underflow_bits > 0 ) {
                count += OutputBit( stream, ~high & 0x8000, tempwriteflag );
                if (stream -> mask != lastmask){
                    addedbits++;
                    lastmask = stream->mask;
                }
                underflow_bits--;
            }
        }
        else if ( ( low & 0x4000 ) && !( high & 0x4000 )) {
            underflow_bits += 1;
            low &= 0x3fff;
            high |= 0x4000;
        } else
            return addedbits;
        low <<= 1;
        high <<= 1;
        high |= 1;
    }
        return addedbits;
}

/*
 * When decoding, this routine is called to figure out which symbol
 * is presently waiting to be decoded.
 */
short int get_current_count( s )
SYMBOL *s;
{
    long range;
    short int count;

    range = (long) ( high - low ) + 1;
    count = (short int)
            ((((long) ( code - low ) + 1 ) * s->scale-1 ) / range );

    return( count );
}

/*
 * This routine is called to initialize the state of the arithmetic
 * decoder.
 */
```

```
void initialize_arithmetic_decoder( stream )
BIT_FILE *stream;
{
    int i;

    code = 0;
    for ( i = 0 ; i < 16 ; i++ ) {
        code <<= 1;
        code += InputBit( stream );
    }
    low = 0;
    high = 0xffff;
    underflow_bits = 0;
}

/* Starts decoder in correct spot for each new packet
 * Numbits is related to variable number of underflow bits and zeros
 * encoded at end of each packet.
 */

void initialize_arithmetic_decoder1( stream , numbits)
BIT_FILE *stream;
int numbits;
{
    int i;

    code = 0;
    for ( i = 0 ; i < numbits ; i++ ) {
        code <<= 1;
        code += InputBit( stream );
    }
    low = 0;
    high = 0xffff;
}

int remove_symbol_from_stream( stream, s )
BIT_FILE *stream;
SYMBOL *s;
{
    long range;
    int numout = 0;

    range = (long)( high - low ) + 1;
    high = low + (unsigned short int)
                 (( range * s->high_count ) / s->scale - 1 );
    low = low + (unsigned short int)
                 (( range * s->low_count ) / s->scale );

    for ( ; ; ) {
        if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
           if (underflow_bits > 0)
               underflow_bits = 0;
        }
        else if ((low & 0x4000) == 0x4000  && (high & 0x4000) == 0 ) {
            code ^= 0x4000;
            low   &= 0x3fff;
            high  |= 0x4000;
            underflow_bits++;
        }
           else
            return numout;

        low <<= 1;
        high <<= 1;
        high |= 1;
        code <<= 1;
        code += InputBit( stream );
        numout++;
    }
}

int remove_symbol_from_stream1( stream, s, bits )
BIT_FILE *stream;
SYMBOL *s;
int bits;
{
    long range;
    int numout = 0;

    range = (long)(high - low) + 1;
    high = low + (unsigned short int)
                 (( range * s->high_count) / s->scale - 1);
    low = low + (unsigned short int)
                 (( range * s->low_count) / s->scale);

    for ( ; ; ) {
        if ((high & 0x8000) == (low & 0x8000)) {
            if (underflow_bits > 0)
                underflow_bits = 0;
        }
        else if ((low & 0x4000) == 0x4000 && (high & 0x4000) == 0 ) {
            code ^= 0x4000;
            low  &= 0x3fff;
            high |= 0x4000;
            underflow_bits++;
        }
           else
            return numout;

        low <<= 1;
        high <<= 1;
        high |= 1;
        code <<= 1;
        if (bits + 16 < PACKET_SIZE) {
            code += InputBit(stream);
            bits++;
        }
        else {
            code += 0;
            bits++;
        }
        numout++;
    }
}

/*
 *   This function is called once at the beginning of compression or expansion
     processing to access a standard_
 * compressed PPM Context Statistical Model and to generate the network realization
     of this PPM Context Statistical  * Model in processor memory.  This memory
     Model remains unchanged during subsequent calls of compression or expansion  *
     functions.
 *
 */

void LoadModel( input1, argc, argv)
BIT_FILE *input1;
int argc;
char *argv[];
{
        SYMBOL s;
        int c, count;
        int thresh=0;
        int i;
        thresh = 0;

        initialize_options(argc, argv);

        initialize_model();
        initialize_arithmetic_decoder( input1 );

        for ( ; ; ) {
            do {
                get_symbol_scale( &s );
                count = get_current_count( &s );
                c = convert_symbol_to_int( count, &s );
                remove_symbol_from_stream( input1, &s );
            } while ( c == ESCAPE );

            if (c == DONE || c == END_PACKET)
                    break;
            if ( c != FLUSH )
            {}
            else
            ;       //flush_model();
            update_model( c );
            add_character_to_model( c );
        } //end model read in for loop

        for(i = 0; i< 258; i++)
                modtotals[i] = totals[i];
        for(i = 0; i<20; i++)
                modcontexts[i] = (int)contexts[i-2];
}


int readTheStatsFile(filename)
char* filename;
{
       int Context = 0;

       int    Index = 0;
       int    Index0 = 0;
       int    Index1 = 0;
       int    Index2 = 0;
       int    Index3 = 0;
       int    Index4 = 0;


       int Order = 0;
       int Symbol = 0;
       int Count = 0;

       int totCount = -1;
       int totVal = 0;
       int curorder = -10;

       int totscore = -1;
       int valscore = 0;

       char   ioBuffer[1024];
       char * tok;
       FILE   *file;

       char   parent[10];
       char   lesser[10];

       int maximum_ID = -1;
       CONTEXT** lookup_table = 0;
       for (int pass = 0; pass < 2; pass++)
```

```
  {
          if (pass == 1)
          {
                  lookup_table = (CONTEXT**) calloc(maximum_ID,
sizeof(CONTEXT*));
                  lookup_table[0] = contexts[-1];


          }


  file = fopen(filename, "r");

  current_order = -10;

  if (file != NULL)
  {
          if (pass == 1)
          {
                  printf("\nReading stats parameters\n");
          }
          ioBuffer[0] = ' ';

          while (!feof(file))
          {
                  fgets(ioBuffer, 1024, file);
                  ioBuffer[strlen(ioBuffer) - 1] = '\0';
                  if ((ioBuffer[0] != '\0') && (ioBuffer[0] != '#'))
                  {
                        tok = strtok(ioBuffer, ",");
                        while (tok != NULL)
                        {

                                if (strstr(tok, "C0:"))
                                {
                                        // Receive Context
                                        Context = atoi(strchr(tok, ':') + 1);
                                }
                                else if (strstr(tok, "I:"))
                                {
                                        // Receive Order0 set
                                        Index = atoi(strchr(tok, ':') + 1);
                                }
                                else if (strstr(tok, "I0:"))
                                {
                                        // Receive Order0 set
                                        Index0 = atoi(strchr(tok, ':') + 1);
                                }

                                else if (strstr(tok, "I1:"))
                                {
                                        // Receive Order1 set
                                        Index1 = atoi(strchr(tok, ':') + 1);
                                }

                                else if (strstr(tok, "I2:"))
                                {
                                        // Receive Order2 set
                                        Index2 = atoi(strchr(tok, ':') + 1);
                                }

                                else if (strstr(tok, "I3:"))
                                {
                                        // Receive Order3 set
                                        Index3 = atoi(strchr(tok, ':') + 1);
                                }
                                else if (strstr(tok, "I4:"))
```

```
                                        {
                                                // Receive Order3 set
                                                Index4 = atoi(strchr(tok, ':') + 1);
                                        }

                                        else if (strstr(tok, "O:"))
                                        {
                                                // Order number
                                                Order = atoi(strchr(tok, ':') + 1);
                                        }

                                        else if (strstr(tok, "S:"))
                                        {
                                                // Character or symbol number
                                                Symbol = atoi(strchr(tok, ':') + 1);
                                        }

                                        else if (strstr(tok, "C:"))
                                        {
                                                // Receive count
                                                Count = atoi(strchr(tok, ':') + 1);
                                        }
                                        else if (strstr(tok, "P:"))
                                        {
                                                // Receive count
                                                strcpy(parent, strchr(tok, ':') + 1);

                                        }
                                        else if (strstr(tok, "PL:"))
                                        {
                                                // Receive count
                                                strcpy(lesser, strchr(tok, ':') + 1);
                                        }
                                        else if (strstr(tok, "totals:"))
                                        {
                                                // Receive count
                                                totCount = atoi(strchr(tok, ':') + 1);
                                        }
                                        else if (strstr(tok, "val:"))
                                        {
                                                // Receive count
                                                totVal = atoi(strchr(tok, ':') + 1);
                                        }
                                        else if (strstr(tok, "current_order:"))
                                        {
                                                // Receive count
                                                curorder = atoi(strchr(tok, ':') + 1);
                                                Context = -10;
                                                totscore = -1;
                                                valscore = -1;

                                        }

                                        tok = strtok(NULL, ",");
                                }


                                if (curorder > -10){
                                        current_order = curorder;
                                        curorder = -10;
                                }
                                else if (totCount >= 0){
                                        totals[totCount] = totVal;
                                        totCount = -1;

                                        if (pass==1){
                                                putc('.', stdout);
```

```
                                  }
                           }

                           else if (Context > -10){

                                  int parent_as_int = atoi (parent);
                                  if (pass == 0)
                                  {
                                         if (parent_as_int > maximum_ID)
                                                maximum_ID = parent_as_int;
                                  }
                                  else // pass 1
                                  {
                                         if (Context > -1){
                                                lookup_table[parent_as_int] =
     create_model(Context, Index0, Index1, Index2, Index3, Index4, Order, Symbol,
     Count);

                                                int lesser_as_int = atoi(lesser);
                                                if (lesser_as_int >= 0)
                                                       lookup_table[parent_as_int]-
     >lesser_context = lookup_table[lesser_as_int];
                                                else
                                                       lookup_table[parent_as_int]-
     >lesser_context = 0;
                                         }
                                  }
                           }
                     }
              }



       }
       else
       {
              fprintf(stderr, "Error opening statistics file %s.\n", filename);
              exit(0);

       }
       fclose(file);
       } // pass 0, 1

       return(1);

}


void LoadStatsModel(input1, argc, argv, statsFilename)
BIT_FILE *input1;
int argc;
char *argv[];
char *statsFilename;
{
     int thresh = 0;
     int j;

     thresh = 0;

     initialize_options(argc, argv);

     initialize_model_load();

     // this is not required here
     //initialize_arithmetic_decoder_load(input1);

     // populate the data model
     readTheStatsFile(statsFilename);

     for (j = 0; j< 258; j++)
           modtotals[j] = totals[j];

     for (j = 0; j<20; j++)
           modcontexts[j] = (int)contexts[j - 2];

     printf("\nInitialised\n");

}


/*
 *
 * CompressFile routine uses a PPM Context Statistical Model in processor memory to
     compress the Input2 text file  * into binary compressed packets in the output1
     file.
 * Once created the model is not changed. Input2 is compressed and packetized
 * based on the static probability model. This precludes the model from adapting to
     new
 * input data, but also makes synchronization between tx compression and rx
     expansion much more reliable.
 *
 */

int CompressMessages( input2, output1, argc, argv )
FILE *input2;
BIT_FILE *output1;
int argc;
char *argv[];
{
           SYMBOL s;
           int c;
           int escaped;
           int flush = 0;
           long int text_count = 0;
        long int esc_count = 0;
        int i=0,j=0;
        int bitsleft = 0;
        int mem[20] = {0};
        int numchar = 0;
        int newpacket = 0;
        int lastorder = 0;
        int orderflag = 0;
        int ordertime = 0;
        int cocount = 0;
        int biplast=0;
        int order_drop_flag=0;

        long int filepos;
        int highsave, lowsave,bipsave,currentordersave,lastordersave;
        int reset=0;
        int numcharenc=0;
        int tempwrite=0;
        int endpacket=0;
        int orderflagsave, ordertimesave, odfsave;
        char scoreboardsave[ 256 ];
        int contextssave[20] = {0};
        int redo = 0;
        int numcharsave=0;
        int masksave=0;
        int racksave=0;
        int ufbitssave=0;
        int firsttime=0;
        int backtrack=0;
        int count=0;
        int thresh=0;

        char inact_text[]="CHANNEL INACTIVE...";
        int numtext=19;
        int index5=0;
        int index5save=0;
        int chanin=0;
        int chaninsave=0;

        FILE *packettext,*packetlength,*pkstats;

        if (input_pos==0)
           {
            packettext = fopen("packettext.dat","w");
            packetlength = fopen("packetlength.dat","w");
            pkstats = fopen("pkstats.dat","w");
        }
           else
           {
            packettext = fopen("packettext.dat","a");
            packetlength = fopen("packetlength.dat","a");
            pkstats = fopen("pkstats.dat","a");
        }

        thresh = 0;
         fprintf(pkstats,"Pknum\tUncompBits\tCompBits\tAvgOrder\n");

           esc_count = 0;

        lastorder = current_order;
        fseek ( input2 , input_pos , SEEK_SET );

        newpacket = 1;
        pknum++;
        bitsinpacket = 0;
        biplast=0;
        numchar = 0;
        current_order = max_order;
        order_drop_flag=0;
        numcharenc=0;
        endpacket=0;
        tempwrite=0;
        firsttime=0;
        backtrack=0;
        bitsleft = 0;
        fprintf(packettext,"\n\n%d\n",pknum);
        cocount = 0;

        initialize_model2();
        initialize_arithmetic_encoder();

        for(i = 0; i<20; i++)
            contexts[i-2] = (CONTEXT *)modcontexts[i];
        for(i = 0; i< 258; i++)
                totals[i] = modtotals[i];

           for ( ; ; ) {

            lastorder = current_order;
            reset=0;
//            if ( !flush ){
                if (redo && numchar == numcharenc){
                    c = END_PACKET;
                }
                else {
                    if (!chanin){
                        c = getc( input2 );
                        if (c==EOF)
                            chanin=1;
                    }
                }
//            }
//            else
//                c = FLUSH;

             if ( chanin && newpacket )
                c = DONE;

            do {
                escaped = convert_int_to_symbol2( c, &s );
                bitsinpacket += encode_symbol1( output1, &s, tempwrite);
                if ( escaped ) esc_count++;
            } while ( escaped );

            if (orderflag)
                ordertime = 1;
            if (current_order < lastorder){
                orderflag = 1;
                ordertime = 0;
            }

            numchar++;
            cocount += current_order;

//            if (pknum == 90 && numchar>=120)
//                    i++;i--;

            biplast=bitsinpacket;
            newpacket = 0;


            fprintf(packettext,"%c",c);

            //Prepare for End of Packet Processing
            if( (bitsinpacket >= PACKET_SIZE - 16) || c==END_PACKET) {
                if (bitsinpacket + underflow_bits + 1 < PACKET_SIZE) {
                    endpacket=1;
                    redo=0;
                }

                //Too many bits, try with one less char and EOP
                if (!endpacket || (tempwrite && endpacket)) {
                    reset=1;
                    redo=1;
                    backtrack++;

                    if (!endpacket){
                        if(numcharenc==0)
                            numcharenc=numchar-2;
                        else
                            numcharenc--;
                    }

                    if (endpacket && tempwrite){
                        endpacket=0;
                        tempwrite=0;
                    }

                    //reset everything
                    fseek ( input2 , filepos , SEEK_SET );

                    high=highsave;
                    low=lowsave;
                    bitsinpacket=bipsave;
```

```
               current_order=currentordersave;
               lastorder=lastordersave;
               order_drop_flag=odfsave;
               ordertime=ordertimesave;
               orderflag=orderflagsave;
               numchar=numcharsave;
               output1->mask=masksave;
               output1->rack=racksave;
               underflow_bits=ufbitssave;
               index5=index5save;
               chanin=chaninsave;
               for(i = 0; i<20; i++){
                   contexts[i-2] = (CONTEXT*)contextssave[i];
               }

               for (i=0;i<256;i++){
                   scoreboard[i] = scoreboardsave[i];
               }
           }

       } //end bip check
          //Special Case--end file while in tempwrite mode
       else if (c==DONE){
           if (tempwrite){
               reset=1;
               redo=1;
               tempwrite=0;
               //reset everything
               fseek ( input2 , filepos , SEEK_SET );
               high=highsave;
               low=lowsave;
               bitsinpacket=bipsave;
               current_order=currentordersave;
               lastorder=lastordersave;
               order_drop_flag=odfsave;
               ordertime=ordertimesave;
               orderflag=orderflagsave;
               numchar=numcharsave;
               output1->mask=masksave;
               output1->rack=racksave;
               underflow_bits=ufbitssave;
                      index5 = index5save;
                      chanin = chaninsave;
               for(i = 0; i<20; i++){
                       contexts[i-2] = (CONTEXT*)contextssave[i];
               }

               for (i=0;i<256;i++) {
                       scoreboard[i] = scoreboardsave[i];
               }
            }
           else
           {
               endpacket=1;
           }
       }

       //Finalize End of Packet Processing and reset for next packet
       if (endpacket){
           fprintf(packetlength,"%d\t%d\t",pknum,bitsinpacket);
           bitsinpacket+=flush_arithmetic_encoder1(output1);
           fprintf(packetlength,"%d\n",bitsinpacket);
           printf("Packet %d done\n",pknum);


fprintf(pkstats,"%d\t\t%d\t\t%d\t\t%0.2f\n",pknum,numchar*8,bitsinpacket,(floa
t)cocount/(float)numchar);
```

```
                fclose(packettext);
                fclose(packetlength);
                fclose(pkstats);

                if (c==DONE){
                    input_pos=ftell(input2);
                    return (-1);
                }
                else {
                    input_pos=ftell(input2);
                    return (input_pos);
                }
            }

            if (!reset) {
                    if ( c == DONE )
                            break;
                    if ( c == FLUSH ) {
                            flush = 0;
                    }

                    if (!newpacket)
                           {
                        update_model2( c );
                        add_character_to_model2( c , ordertime );
                    }

                    if (current_order == max_order)
                           {
                        ordertime = 0;
                        orderflag = 0;
                    }

            } //end reset

                        //Set checkpoint and save encoder state
                        if (bitsinpacket >= PACKET_SIZE - 96 && firsttime==0) {
//                                      if (pknum ==61)
//                                 i++;i--;

                            firsttime=1;
                            tempwrite=1;
                            filepos=ftell(input2);
                            highsave=high;
                            lowsave=low;
                            bipsave=bitsinpacket;
                            currentordersave=current_order;
                            lastordersave=lastorder;
                            odfsave=order_drop_flag;
                            ordertimesave=ordertime;
                            orderflagsave=orderflag;
                            ufbitssave=underflow_bits;
                            numcharsave=numchar;
                            masksave=output1->mask;
                            racksave=output1->rack;
                            index5save=index5;
                            chaninsave=chanin;
                            for(i = 0; i<20; i++){
                                contextssave[i] = (int)contexts[i-2];
                            }

                            for (i=0;i<256;i++){
                                scoreboardsave[i] = scoreboard[i];
                            }
                        }
              }
```

```
           return(-2);
}

/*
 * Expansion algorithm uses a fixed, memory-based PPM Context Statistical Model and
     interprets binary compressed  * packets into output text.
 *
 */

int ExpandFile( input2, output, argc, argv )

BIT_FILE *input2;
FILE *output;
int argc;
char *argv[];
{
           SYMBOL s;
           int c;
           int count;
           int flush = 0;
        long int esc_count = 0;
        int i = 0;
        int bitsread = 0;
        int first = 1;
        FILE *debug;
        int numleft = 0;
        int newpacket = 0;
        int lastorder = 0;
        int orderflag = 0;
        int ordertime = 0;
        int bitsin=0;
        int thresh=0;
        int numchar=0;

        //Input1 is binary model file
        //Input2 is binary compressed file to be expanded
        debug = fopen("debugexpand.dat","w");

        thresh = 0;

        initialize_options(argc, argv);

        fseek ( input2->file , input_pos , SEEK_SET );

        initialize_model2();
        initialize_arithmetic_decoder1( input2,offset );

        // If first packet begin reading right away
        for(i = 0; i<20; i++)
                contexts[i-2] = (CONTEXT*)modcontexts[i];

        for(i = 0; i< 258; i++)
                totals[i] = modtotals[i];
        current_order = max_order;
        newpacket = 1;
        offset = 16;
        current_order = max_order;
        pknum++;
        bitsread = 0;

           for ( ; ; ) {
               lastorder = current_order;

                 do {
                           //dont let it get below -2 as this will fail to find a
     symbol
                           if (current_order < -2) current_order = -2;
                    get_symbol_scale2( &s );
                    count = get_current_count( &s );
                    c = convert_symbol_to_int1( count, &s, debug );

                           if (c == -10){
                            printf("\n\nBit Error : Packet %d...Skipping to next
     packet\n\n",pknum);
                    }
                    else
                    {
                        if ( bitsread < PACKET_SIZE - 100 ){
                                bitsread += remove_symbol_from_stream( input2, &s
     );
                                newpacket = 0;
                        }
                        else {
                                bitsin = bitsread;
                                bitsread += remove_symbol_from_stream1( input2, &s,
     bitsin );
                                newpacket = 0;
                        }
                    }
                    } while ( c == ESCAPE );

                    if ( orderflag)
                                 ordertime = 1;
                    if ( current_order < lastorder)
                    {
                           orderflag = 1;
                           ordertime = 0;
                    }

                    numchar++;

             if ( c != FLUSH && c !=END_PACKET && c!=DONE && c!=-10 && c!=13) {
                                 putc( (char) c, output );
                    }
                    else
                    ;       //flush_model();

                    if ((bitsread >= PACKET_SIZE - 16 + underflow_bits) ||
     c==END_PACKET || c==DONE || c==-10) {
                           if (c==-10){
                                  offset = PACKET_SIZE - bitsread + 16;
                           }
                           else {
                                  offset = (PACKET_SIZE - bitsread);
                           }

                           if (offset < 16)
                                  offset = 16;

                           printf("Packet %d done\n",pknum);

                           if (c==DONE) {
                                  input_pos=ftell(input2->file);
                                  return (-1);
                           }
                           else {
                                  input_pos=ftell(input2->file);
                           //     if (input_pos != (int)(pknum*PACKET_SIZE/8))
                    //                     i++;i--;

                                  if (c==-10)
                                          input_pos = (long
     int)(pknum*PACKET_SIZE/8);
                                  return (input_pos);
                           }
                    }

                    if (!newpacket){
                           update_model2( c );
                           add_character_to_model2( c , ordertime );
                    }

                    if ( current_order == max_order){
                           ordertime = 0;
                           orderflag = 0;
                    }
           }       // end message for loop
        // If last packet is done, exit loop
}       //End Compression procedures

/* Bit File I/O Procedures
 *
 * The basis of the PPM Context Statistical Model Software is the open source
     Arith-N.C code of Mark Nelson and
 * Jean-loup Gailly as presented in The Data Compression Book (2nd edition).
 *
 */

#define PACIFIER_COUNT 2047

BIT_FILE *OpenOutputBitFile( name )
char *name;
{
    BIT_FILE *bit_file;

    bit_file = (BIT_FILE *) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
        return( bit_file );
    bit_file->file = fopen( name, "wb" );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
    return( bit_file );
}

BIT_FILE *OpenInputBitFile( name )
char *name;
{
    BIT_FILE *bit_file;

    bit_file = (BIT_FILE *) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
        return( bit_file );
    bit_file->file = fopen( name, "rb" );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
    return( bit_file );
}

void CloseOutputBitFile( bit_file )
BIT_FILE *bit_file;
{
    if ( bit_file->mask != 0x80 )
        if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
            fatal_error( "Fatal error in CloseBitFile!\n" );
    fclose( bit_file->file );
    free( (char *) bit_file );
}

void CloseInputBitFile( bit_file )
```

```
BIT_FILE *bit_file;
{
    fclose( bit_file->file );
    free( (char *) bit_file );
}

int OutputBit( bit_file, bit, tempwrite )
BIT_FILE *bit_file;
int bit, tempwrite;
{
    int bitcount = 0;
    int boolean = 0;


    if ( bit )
    bit_file->rack |= bit_file->mask;
    bit_file->mask >>= 1;
    if ( bit_file->mask == 0 ) {
        if (!tempwrite)
                boolean = putc( bit_file->rack, bit_file->file );
        else
                boolean = bit_file->rack;
        bitcount += 8;
        if ( boolean != bit_file->rack )
                fatal_error( "Fatal error in OutputBit!\n" );
        else {
                if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
                    putc( '.', stdout );
        }
        bit_file->rack = 0;
        bit_file->mask = 0x80;
    }
        return bitcount;
}

int OutputBits( bit_file, code, count )
BIT_FILE *bit_file;
unsigned long code;
int count;
{
    unsigned long mask;
    int bitcount = 0;
    int out = 0;

    mask = 1L << ( count - 1 );
    while ( mask != 0) {
        if ( mask & code )
            bit_file->rack |= bit_file->mask;
        bit_file->mask >>= 1;
        if ( bit_file->mask == 0 ) {
            out = putc( bit_file->rack, bit_file->file );
            bitcount += 8;
            if (  out!= bit_file->rack )
                  fatal_error( "Fatal error in OutputBits!\n" );
        else if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
            putc( '.', stdout );
        bit_file->rack = 0;
        bit_file->mask = 0x80;
        }
        mask >>= 1;
    }
        return out;
}

int InputBit( bit_file )
BIT_FILE *bit_file;
{
    int value;

    if ( bit_file->mask == 0x80 ) {
        bit_file->rack = getc( bit_file->file );
        if ( bit_file->rack == EOF )
            fatal_error( "Fatal error in InputBit!\n" );
    if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
            putc( '.', stdout );
    }
    value = bit_file->rack & bit_file->mask;
    bit_file->mask >>= 1;
    if ( bit_file->mask == 0 )
        bit_file->mask = 0x80;
    return( value ? 1 : 0 );
}

unsigned long InputBits( bit_file, bit_count )
BIT_FILE *bit_file;
int bit_count;
{
    unsigned long mask;
    unsigned long return_value;

    mask = 1L << ( bit_count - 1 );
    return_value = 0;
    while ( mask != 0) {
        if ( bit_file->mask == 0x80 ) {
            bit_file->rack = getc( bit_file->file );
            if ( bit_file->rack == EOF )
                fatal_error( "Fatal error in InputBit!\n" );
           if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
                putc( '.', stdout );
           }
        if ( bit_file->rack & bit_file->mask )
            return_value |= mask;
        mask >>= 1;
        bit_file->mask >>= 1;
        if ( bit_file->mask == 0 )
            bit_file->mask = 0x80;
    }
    return( return_value );
}

void FilePrintBinary( file, code, bits )
FILE *file;
unsigned int code;
int bits;
{
    unsigned int mask;

    mask = 1 << ( bits - 1 );
    while ( mask != 0 ) {
        if ( code & mask )
            fputc( '1', file );
        else
            fputc( '0', file );
        mask >>= 1;
    }
}

/* End BIT I/O procedures */
```

ANNEX E. LDPC FEC GENERATION MATRICES

E.1. LDPC FEC GENERATOR MATRIX FORMAT

The files referenced in this appendix define the LDPC FEC generator matrices. Each file contains entries for the required number of rows and columns for each generator matrix. The first character in the file represents M1-1 of the generator matrix. The last character in the file represents Ma-b of the generator matrix, where a and b are the maximum row and column sizes of the matricies respectively.

E.2. FIBONACCI BIT PARITY FEC GENERATOR MATRIX

The files defined in Table 21 below are included with the standard and provide the matrices for the LDPC FEC generators.

	Filename	**MD5 Checksum**
RED/BLACK ALPHA REM	rb_alpha.GEN	7734517453151bbbeb77ac1342a1972a
RED/BLACK BRAVO REM	rb_bravo.GEN	9a48116c5e9b59826ba04f12edf2c28e
RED ALPHA REM	r_alpha.GEN	ec89abe6cf13866716731957eb0af48b
RED BRAVO REM	r_bravo.GEN	1e678fe91d7f3e869efffe6bfe55561f

TABLE 21: LDPC FEC GENERATOR MATRICES.

INTENTIONALLY BLANK

AComP-4724(B)(1)