Office de la Propriété Intellectuelle du Canada
Un organisme d'Industrie Canada

Canadian Intellectual Property Office
An agency of Industry Canada

CA 2313330 C 2005/02/15

(11)(21) 2 313 330

(12) BREVET CANADIEN
CANADIAN PATENT

(13) C

(86) Date de dépôt PCT/PCT Filing Date: 1998/12/11
(87) Date publication PCT/PCT Publication Date: 1999/06/17
(45) Date de délivrance/Issue Date: 2005/02/15
(85) Entrée phase nationale/National Entry: 2000/06/07
(86) N° demande PCT/PCT Application No.: US 1998/026421
(87) N° publication PCT/PCT Publication No.: 1999/030462
(30) Priorités/Priorities: 1997/12/12 (08/989,616) US;
1997/12/12 (08/989,483) US

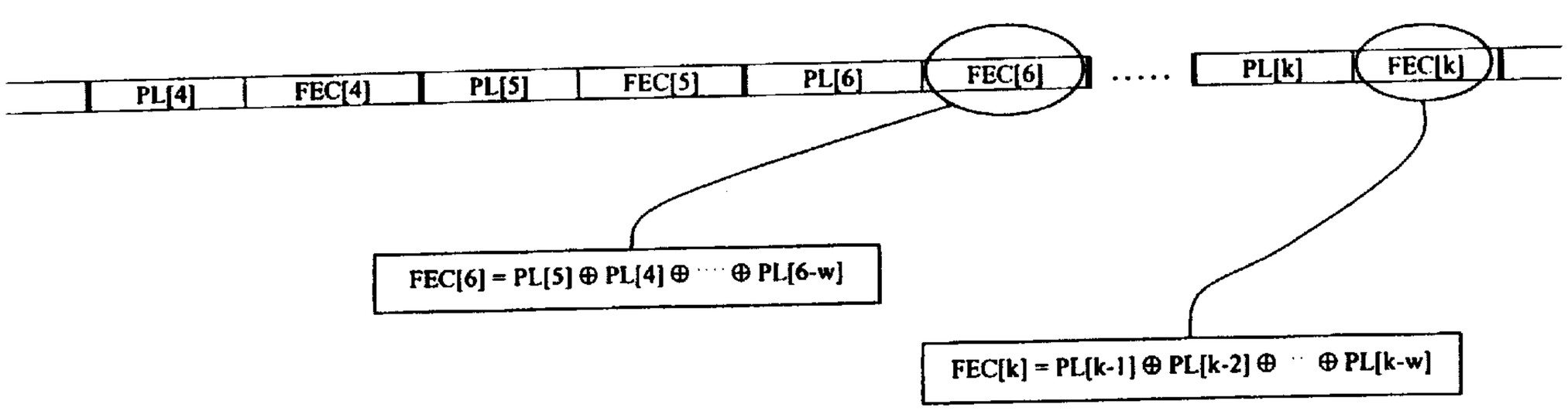
(51) Cl.Int.^{6}/Int.Cl.^{6} H04L 1/00

(72) Inventeurs/Inventors:
SCHUSTER, GUIDO M., US;
MAHLER, JERRY J., US;
SIDHU, IKHLAQ S., US;
BORELLA, MICHAEL S., US

(73) Propriétaire/Owner:
3COM CORPORATION, US

(74) Agent: SMART & BIGGAR

(54) Titre : SYSTEME DE CORRECTION AVAL DES ERREURS POUR MEDIA EN TEMPS REEL A BASE DE PAQUETS
(54) Title: A FORWARD ERROR CORRECTION SYSTEM FOR PACKET BASED REAL-TIME MEDIA



(57) Abrégé/Abstract:
A computationally simple yet powerful forward error correction code scheme for transmission of real-time media signals, such as digitized voice, video or audio, in a packet switched network, such as the Internet. According to one aspect, an encoder at the sending end derives p redundancy blocks from each group of a k payload blocks and concatenates the redundancy blocks, respectively, with payload blocks in the next group of k payload blocks. At the receiving end, a decoder may recover up to p missing packets in a group of k packets, provided with the p redundancy blocks carried by the next group of k packets. The invention may, for instance, append to each of a series of payload packets a single forward error correction code that is defined by taking the XOR sum of a preceding specified number of payload packets. The invention thereby enables correction from the loss of multiple packets in a row, without significantly increasing the data rate or otherwise delaying transmission.

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : H04L 12/00	A2	(11) International Publication Number: WO 99/30462
		(43) International Publication Date: 17 June 1999 (17.06.99)

(21) International Application Number: PCT/US98/26421

(22) International Filing Date: 11 December 1998 (11.12.98)

(30) Priority Data:

08/989,483	12 December 1997 (12.12.97)	US
08/989,616	12 December 1997 (12.12.97)	US

(71) Applicant *(for all designated States except US)*: 3COM CORPORATION [US/US]; 3800 Golf Road, Rolling Meadows, IL 60008 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: SCHUSTER, Guido, M. [CH/US]; Apartment 408, 1433 Perry Street, Des Plaines, IL 60016 (US). MAHLER, Jerry, J. [US/US]; 20 Country Club Drive #B, Prospect Heights, IL 60070 (US). SIDHU, Ikhlaq, S. [US/US]; 403 River Grove Lane, Vernon Hills, IL 60061 (US). BORELLA, Michael, S. [US/US]; 1208 Haverhill Circle, Naperville, IL 60563 (US).

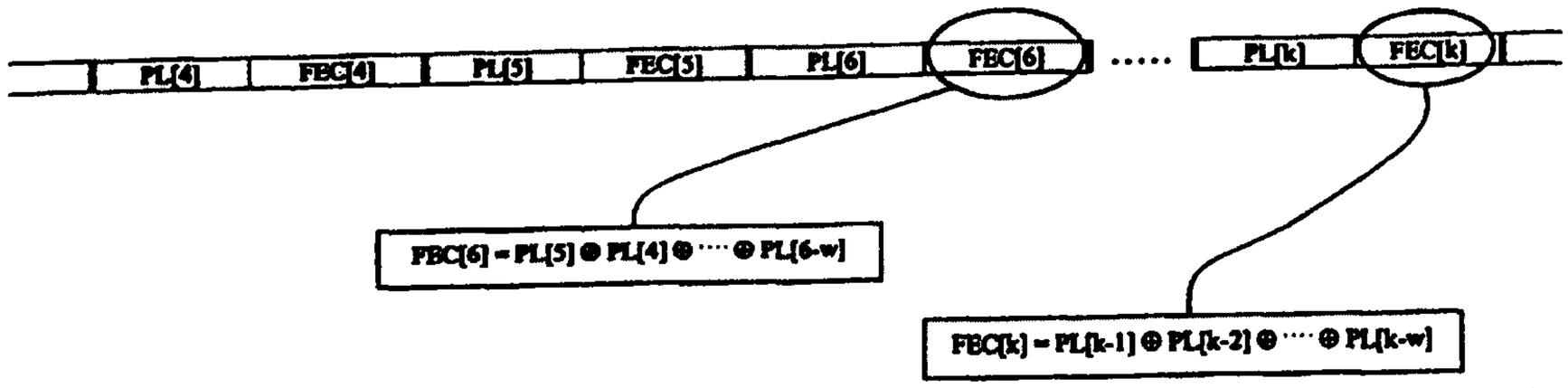
(74) Agent: AARONSON, Lawrence, H.; McDonnell Boehnen Hulbert & Berghoff, 300 South Wacker Drive, Chicago, IL 60606 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, HR, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published

Without international search report and to be republished upon receipt of that report.

(54) Title: A FORWARD ERROR CORRECTION SYSTEM FOR PACKET BASED REAL–TIME MEDIA



(57) Abstract

A computationally simple yet powerful forward error correction code scheme for transmission of real–time media signals, such as digitized voice, video or audio, in a packet switched network, such as the Internet. According to one aspect, an encoder at the sending end derives p redundancy blocks from each group of a k payload blocks and concatenates the redundancy blocks, respectively, with payload blocks in the next group of k payload blocks. At the receiving end, a decoder may recover up to p missing packets in a group of k packets, provided with the p redundancy blocks carried by the next group of k packets. The invention may, for instance, append to each of a series of payload packets a single forward error correction code that is defined by taking the XOR sum of a preceding specified number of payload packets. The invention thereby enables correction from the loss of multiple packets in a row, without significantly increasing the data rate or otherwise delaying transmission.

A FORWARD ERROR CORRECTION SYSTEM FOR PACKET BASED REAL TIME MEDIA

COPYRIGHT

A portion of the disclosure of this patent document contains material that is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to packet switched telecommunications networks and more particularly to a system for correcting loss of data packets in such a network.

2. Description of the Related Art

In a packet switched network, a message to be sent is divided into blocks, or data packets, of fixed or variable length. The packets are then sent individually over the network through multiple locations and then reassembled at a final location before being delivered to a user at a receiving end. To ensure proper transmission and re-assembly of the blocks of data at the receiving end, various control data, such as sequence and verification information, is typically appended to each packet in the form of a packet header. At the receiving end, the packets are then reassembled and transmitted to an end user in a format compatible with the user's equipment.

A variety of packet switching protocols are available, and these protocols range in degree of efficiency and reliability. Those skilled in the art are familiar, for instance, with the TCP/IP suite of protocols, which is used to manage transmission of packets throughout the Internet. Two of the protocols within the TCP/IP suite, as examples, are TCP and UDP.

TCP is a reliable connection-oriented protocol, which includes intelligence necessary to confirm successful transmission between sending and receiving ends in the network. According to TCP, each packet is marked in its header with a sequence number to allow the receiving end to properly reassemble the packets into the original message. The receiving end is then typically configured to acknowledge receipt of

packets and expressly request the sending end to re-transmit any lost packets. UDP, in contrast, is an unreliable connectionless protocol, which facilitates sending and receiving of packets but does not include any intelligence to establish that a packet successfully reached its destination.

In the Internet, loss of entire packets has been found to occur at a rate of over 20% when the network is very congested. Typically, this packet loss occurs one packet at a time. However, at times, multiple sequential packets in a row may be lost. In either case, as those skilled in the art will appreciate, a connection-oriented protocol such as TCP introduces delay into packet transmission, due to its need to confirm successful transmission and to request retransmission of these lost packets. While this delay may not be a significant problem in the transmission of pure data signals (such as an e-mail message), the delay can unacceptably disrupt the transmission of real-time media signals (such as digitized voice, video or audio).

Carle et al., International Publication No. WO 97/38549, discloses a method and apparatus for error control in an Asynchronous Transfer Mode (ATM) network, a fast packet switched network that use short packets of fixed size called cells. For each frame of data cells D_1 ... D_m, a forward error code consisting of redundancy cells R_1 ... R_h, is generated using a double matrix interleaving XOR scheme. The cells of each CPCS-PDU (Protocol Data Unit of the Common Part Convergence Sublayer of ATM Adaptation Layer 5) are arranged to include cells representing user data for frame i and cells representing the forward error correction code for frame i-1. A disadvantage with this approach, however, is that the additional redundancy cells that are created and transmitted increase the packet rate and, thus, the work load on the packet switching elements in the network. This, in turn, can cause transmission delays.

Matshushita, U.S. Patent No. 5,608,738, discloses a method of forming m forward error correction code packets from n data packets, by using one or more error correction frames as intermediate data structures. After the m forward error correction code packets are created, they are transmitted sequentially with the n data packets. Because additional error correction packets are created and transmitted, thereby increasing the packet rate, the Matsushita approach has a similar drawback.

Therefore, a need exists for an improved system of responding to and correcting packet loss errors without introducing significant transmission delays.

SUMMARY OF THE INVENTION

The present invention provides a computationally simple yet powerful system for handling packet loss that may arise in the communication of real time media signals, such as digitized voice, video or audio, in a packet switched network. According to one aspect, an encoder at the sending end derives p redundancy blocks from each group of a k payload blocks and concatenates the redundancy blocks, respectively, with payload blocks in the next group of k payload blocks. According to a further aspect, the invention generates and appends to each of a series of payload packets a forward error correction code that is defined by taking the XOR sum of a predetermined number of preceding payload packets. In this way, a receiving end may extract lost payload from the redundant error correction codes carried by succeeding packets and may correct for the loss of multiple packets in a row.

Beneficially, regardless of the number of packets in a row to be recovered by this correction scheme, the size of the forward error correction code employed by the present invention is of the same order as thepayload itself. The present invention

does not increase the packet rate and may perform its function without introducing significant delay into the transmission process.

The invention may be summarized according to one aspect as in a packet switched telecommunications network in which routers route packets based on information contained in the packet headers, a method of encoding packets from a sequence of payload blocks to enable recovery of lost packets, said sequence of payload blocks including a sequential group of *k* payload blocks and a subsequent sequential group of payload blocks, said method characterized by: deriving *p* redundancy blocks from said sequential group of *k* payload blocks; and generating *p* packets by a process comprising combining each of said *p* redundancy blocks, respectively, with a payload block in said subsequent sequential group of payload blocks, whereby each one of said *p* packets includes a payload block and one of said *p* redundancy blocks.

According to another aspect the invention provides in a packet switched telecommunications network in which routers route packets based on information contained in the packet headers, an apparatus for encoding packets from a sequence of payload blocks to enable recovery of lost packets, said sequence of payload blocks including a sequential group of *k* payload blocks and a subsequent sequential group of payload blocks, said apparatus having a computer processor and a memory, said apparatus charaterized by: a first set of machine language instructions stored in said memory and executed by said processor for deriving *p* redundancy blocks from said sequential group of *k* payload blocks; and a second set of machine language instructions stored in said memory and executed by said processor for generating *p* packets by a process comprising combining each

of said *p* redundancy blocks, respectively, with a payload block in said subsequent sequential group of payload blocks, whereby each one of said *p* packets includes a payload block and one of said *p* redundancy blocks.

According to yet another aspect of the invention provides in a packet switched telecommunications network in which routers route packets based on information contained in the packet headers, a method for communicating a payload, said payload being divided into a sequence of payload blocks, PL[k-w], ..., PL[k-2], PL[k-1], PL[k], PL[k+1], …, PL[k+u], said method characterized by: for each payload block PL[i] in said sequence of payload blocks, generating a forward error correction code FEC[i] comprising an XOR sum of a predetermined number of preceding payload blocks and appending said forward error correction code FEC[i] to said payload block PL[i], said predetermined number being greater than one; transmitting a sequence of packets P[k], P[k+1], …, P[k+u] from a first device in said network for receipt by a second device in said network, wherein each packet P[i] in said sequence of packets includes payload block PL[i] and forward error code FEC[i].

According to still another aspect the invention provides in a packet switched telecommunications network in which routers route packets based on information contained in the packet headers, a method of recovering a lost packet in a sequence of packets transmitted in said network, each packet in said sequence defining a sequence number and carrying a payload block and a redundancy block, said method characterized by: (a) receiving an incoming packet of said sequence of packets; (b) establishing a window of analysis beginning with said incoming packet and extending for a predetermined number of packets of said sequence following said incoming packet; and (c) if only one payload block in

said window of analysis has not yet been received, recovering said one payload block by taking and XOR sum of a plurality of payload blocks within said window of analysis.

According to a further aspect the invention provides in a packet switched telecommunications network in which routers route packets based on information contained in the packet headers, an apparatus for communicating a payload, said payload being divided into a sequence of payload blocks, PL[k-w], …, PL[k-2], PL[k-1] PL[k], PL[k+1], …, PL[k+u], said apparatus having a computer processor and a memory, said apparatus characterized by: a first segment for generating, for each payload block PL[i] in said sequence of payload blocks, a forward error correction code FEC[i] comprising an XOR sum of a predetermined number of preceding payload blocks and appending said forward error correction code FEC[i] to said payload block PL[i], said predetermined number being greater than one; and a second segment for transmitting a sequence of packets P[k], P[k+1], …, P[k+u] from a first device in said network for receipt by a second device in said network, wherein each packet P[i] in said sequence of packets includes payload block PL[i] and forward error code FEC[i].

These as well as other advantages of the present invention will become apparent to those of ordinary skill in the art by reading the following detailed description, with appropriate reference to the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

A preferred embodiment of the present invention is described herein with reference to the drawings, in which:

Figure 1 depicts a payload data stream divided into payload packets PL[k];

Figure 2 depicts an encoder operating in accordance with a preferred embodiment of the present invention;

Figure 3 depicts a packet encoded in accordance with a preferred embodiment of the present invention;

Figure 4 depicts a decoder operating in accordance with a preferred embodiment of the present invention;

Figure 5 depicts an output data stream encoded in accordance with a preferred embodiment of the present invention, including packets of the form P[k] = {PL[k], FEC[k]};

Figure 6 depicts several stages of an encoder operating in a preferred embodiment of the present invention;

Figure 7 depicts a packet encoded in accordance with a preferred embodiment of the present invention;

Figure 8 depicts a link list compiled by a decoder operating in accordance with a preferred embodiment of the present invention;

Figure 9 depicts the analysis steps performed by a decoder operating in accordance with a preferred embodiment of the present invention; and

Figure 10 lists the steps depicted in Figure 9.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

The present invention provides an improved system for communicating substantially real time media signals through an unreliable digital transmission channel. The invention may find particular use for transmission of digitized video or audio (including, for instance, voice) signals over the Internet. In the preferred embodiment, the invention employs a computationally simple error correction scheme that enables the recovery of lost data packets within specified limits, while beneficially adding very little delay in transmission time.

For purposes of illustration, the following description will assume that an audio or video signal has been converted into a digital data stream and is to be transmitted in a network from a first node to a second node. It will be appreciated, of course, that the invention is not restricted to use in a traditional network configuration but may extend to any communication path through which a sequence of packets are transmitted, including, for instance, a direct path. In the preferred embodiment, however, the signal at issue is to be transmitted between nodes of a network such as the Internet.

The description will further assume by way of example that the digital data stream, or payload, has been divided into a sequence of frames or payload packets, PL[1], PL[2], PL[3], PL[4], PL[5], PL[6], . . . , PL[k]. A source data stream divided into these packets is illustrated for example in Figure 1. In this example, each of these payload packets consists of a set number of bytes or bits and preferably represents a portion of a signal to be transmitted through a network.

This description additionally assumes that the packets will be transmitted in a network according to a packet switching protocol that employs a bit-wise or other high resolution verification scheme such as a checksum or parity bit. Therefore, it will be assumed that a technique is in place to respond to loss of some portion of a successfully transmitted packet. Remaining at issue, however, is how to correct for loss of one or more entire packets.

As discussed in the Background section above, the TCP protocol offers one method for responding to loss of packets in a digital transmission network. According to TCP, the receiving node may be configured to acknowledge receipt of packets and expressly request the transmitting node to retransmit any lost packets. This request

and retransmission system is generally accurate. However, as noted above, the system is not well suited for use in the context of real-time media transmissions, because the transmission of such signals is very sensitive to the delay introduced by retransmission requests.

Rather than employing a request and retransmission system, greater efficiency in packet loss correction may be achieved by transmitting a correction code of some sort concurrently with the payload data, thereby providing the receiving end with sufficient information to recover lost packets. Several error correction code mechanisms are available for this purpose. These mechanisms include, for instance, interleaving and block coding.

Interleaving is commonly used in wireless communications, providing an effective method to combat error bursts that occur on a fading channel. Interleaving generally calls for spreading the bits of each codeword (or payload packet) apart from each other so they experience independent fading and so that the original data may be more readily recovered.

Block coding, in turn, calls for mapping a frame of binary source data into a coded block of data that includes a set of redundant parity symbols. By conventional terminology, a block coder typically converts a group of k payload units (such as bytes or bits) into a larger group of n units by adding or appending $p = n - k$ parity units or forward error correction (FEC) codes to the group. Each parity packet is generated through a predetermined coding technique based on all or some subset of the k payload units. One of the simplest forms of a block code is a repetition code, in which the binary source data is repeated as a set of parity bits. One of the more popular but complex block codes is the Reed-Solomon (RS) class of codes over the 2^8 Galois field. These codes are optimal in their ability to correct erased bytes. For example, provided that 8 bytes are protected with 3 parity bytes (a total of 11 bytes), any three bytes can be lost, and the original 8 bytes may still be recovered.

Unfortunately, however, the addition of redundant parity packets gives rise to increased complexity and delays in transmission. In a packet switched network, routers route packets based on information contained in the packet headers. Therefore, the amount of work performed by a packet router generally relates directly to the number of packets being routed. Because each packet, whether payload or

parity, contains its own header, an increase in packet rate consequently increases the burden on network routers and could delay transmission time or, in theory, cause a network to fail.

Adding separate parity packets to the transmission sequence is a particular waste of resources when done in the context of some of the more common real-time media transmission protocols, because many of the existing protocols add substantial overhead to each packet. For instance, the G.723.1 voice coder provides 24 bytes of payload every 30 milliseconds, but RTP/UDP/IP adds a 50 byte header to each payload packet. A parity block destined for the same receiving end as an adjacent payload block would have a header almost identical to the header of the payload block. Yet the parity packet would still require the full 44 bytes of overhead, resulting in a waste of bandwidth. As this example illustrates, traditional block coding techniques are therefore not well suited for correcting packet loss in real time media transmissions.

To avoid an increase in packet rate, another technique for providing parity information is to *append* redundant parity information to existing payload packets. For instance, as an offshoot of traditional repetition codes, the transmitting node may append to each payload packet redundant copies of the preceding n number of payload packets. In this way, the receiving end may readily recover a lost packet PL[k] from one of the n subsequent packets PL[k+1] . . . PL[k+n]. For instance, with n=2, payload packets PL[k-1] and PL[k-2] would be redundantly appended to and transmitted together with payload packet PL[k], providing the following transmission packets P[k], for example:

P[1]	=	{ PL[1],	PL[0],	PL[-1] }
P[2]	=	{ PL[2],	PL[1],	PL[0] }
P[3]	=	{ PL[3],	PL[2],	PL[1] }
P[4]	=	{ PL[4],	PL[3],	PL[2] }
P[5]	=	{ PL[5],	PL[4],	PL[3] }
P[6]	=	{ PL[6],	PL[5],	PL[4] }
P[7]	=	{ PL[7],	PL[6],	PL[5] }
P[8]	=	{ PL[8],	PL[7],	PL[6] }
P[9]	=	{ PL[9],	PL[8],	PL[7] }

* * *

P[k] = { PL[k], PL[k-1], PL[k-2]}

With this scheme, in the event a payload packet is lost in transmission, the receiving end may simply extract a redundant copy of the payload from one of the *n* subsequent data packets.

By appending *n* preceding payload packets to each current data packet P[k], it becomes possible to recover *n* lost packets in a row without having to request retransmission. As more preceding packets are concatenated with each current packet, the network can tolerate a higher rate of packet loss. Additionally, this method will not affect the packet rate or routing rate, since, as noted above, the routing rate is concerned principally with the header information in a given packet, rather than with the size of each packet. Unfortunately, however, by concatenating multiple payload packets, this method necessarily and substantially increases the data rate. For instance, to be able to correct for two lost packets in a row (n=2) this method nearly triples the size of every packet. Therefore, this method is also not desirable.

Instead, the present invention beneficially provides a suitable and less costly scheme of correcting for packet loss, adding very little delay to transmission time. The invention preferably employs a coding technique in which parity bits associated with current packets are piggy-backed onto future packets. Rather than concatenating multiple previous payload packets with each current payload packet (and thus greatly raising the data rate), the preferred embodiment of the present invention calls for concatenating a single forward error correction (FEC) code (block code, or redundancy block) with specific payload packets or with each payload packet in a way that enables the recovery of multiple lost packets in a row.

1. Concatenated Parity Blocks

According to a preferred embodiment of the present invention, as a sequence of payload blocks is being transmitted, every k payload blocks in the sequence are fed through a block coder to create p = n - k forward error correction (FEC) codes or redundancy blocks, where $p \leq k$. (As shown by way of example below, p could be 1; however, this description may refer to p in plural for sake of illustration.) These p redundancy blocks are then concatenated respectively with the next p payload blocks

being transmitted. In turn, at the receiving end, if a packet is lost, the associated payload may be extracted from the redundancy blocks carried by the appropriate packets.

The present invention may employ substantially any block coder now known or later developed in order to create the required p redundancy blocks. Of course, the choice of block coder, including the choice of the (n, k) values used by the coder, may depend on a number of factors, including the efficiency required for the specified application. In the preferred embodiment, for transmission of real-time media signals over the Internet, the invention employs the well known Reed-Solomon (RS) class of codes over the 2^8 Galois field. Also referred to as RS erasure codes, these codes are optimal in their ability to correct erased bytes. For example, provided that 8 bytes are protected with 3 parity bytes (a total of 11 bytes) any three bytes can be lost and the original 8 bytes may still be recovered.

Figure 2 illustrates by way of example an encoder 10 operating in accordance with the present invention. In the example shown, a stream of fixed length packets or payload blocks arrives. The first six of these payload blocks, PL[1] through PL[6], are shown. For purposes of this example, the encoder employed by the invention is an RS block coder having (n, k) values of (5, 3). Thus, for every three payload blocks in the incoming sequence, the coder derives two FEC codes or redundancy blocks, FEC0 and FEC1. According to the invention, the encoder then appends theses redundancy blocks, respectively, to the next two payload blocks, which are the first two payload blocks in the next group of three.

Referring to Figure 2, as each of the first three payload blocks, PL[1], PL[2] and PL[3], arrives, encoder 10 writes copies of these payload blocks into memory for use in creating the required redundancy blocks. The encoder then forwards each of these three payload blocks to a packetizer 12, which adds header information and passes resulting packets, P[1], P[2] and P[3], along for transmission to the network 14. (For purposes of example, Figure 2 does not show the header information or other overhead information included with these output packets.) Because these are the first three payload blocks, they are transmitted without any added redundancy blocks.

Using the copies of payload blocks PL[1], PL[2] and PL[3] that it stored in memory, the encoder then derives two redundancy blocks, FEC0 and FEC1, as

indicated at section 14 in Figure 2. In particular, the RS block coder operates symbol-wise (for example, byte-wise) on the payload symbols to create corresponding bits of the FEC symbols. The left most symbol of each of the payload blocks PL[1], PL[2] and PL[3] is used to derive the left most symbols of the redundancy blocks, the next symbol of each of the payload blocks PL[1], PL[2] and PL[3] is used to derive the next most symbols of the redundancy blocks, and so forth. As a result, the size of the redundancy blocks employed by the present invention is of the same order as that of the payload blocks with which they will be concatenated.

After generating redundancy blocks FEC0 and FEC1, the encoder preferably purges payload blocks PL[1], PL[2] and PL[3], writes copies of the next three payload blocks, PL[4], PL[5] and PL[6], to memory, and concatenates with payload blocks PL[4] and PL[5], respectively, the two redundancy blocks that it derived from payload blocks PL[1], PL[2] and PL[3]. To continue steady transmission, the encoder then passes the concatenated symbols {PL[4], FEC0} and {PL[5], FEC1} as well as payload block PL[6] to the packetizer 12 to generate packets P[4], P[5] and [P6] for transmission to the network.

While Figure 1 shows only the first six payload blocks and resulting packets, it will be appreciated that this process continues as long as a payload stream continues to arrive at the encoder. Thus, using a block coder with (n, k) values of (5, 3), the first two packets in every group of three packets transmitted to the network will each preferably include a redundancy block derived from the payload blocks in the previous three packets.

Of course, it will also be appreciated that the present invention is not restricted to appending the redundancy blocks specifically to the next payload packets following the group from which the redundancy blocks were derived, but other arrangements may be used. For instance, using a (5, 3) block coder as discussed above, the two redundancy blocks FEC0 and FEC1 derived from payload blocks PL[i], PL[i+1] and PL[i+2] could regularly be concatenated, respectively, with PL[i+4] and PL[i+5] rather than with PL[i+3] and PL[i+4].

Because an encoder operating according to the preferred embodiment of the present invention is an RS block coder, the invention presumes that the payload blocks being combined are all of the same length (for instance, the same number of

bits). With a slight adjustment, however, the invention will work equally well in a network of variable length packets (and/or fixed length packets). Provided with payload blocks of various lengths, the invention contemplates padding the shorter payload blocks in each group of k payload blocks (every three payload blocks in the above example) with zeros, so that all k blocks are the same length. For instance, with a (5, 3) block coder, if the first two of three incoming payload blocks are 16 bits long but the third is 24 bits long, the encoder will pad the first two blocks with 8 zero bits so that all three blocks are 24 bits long. The encoder will then derive the necessary redundancy blocks FEC0 and FEC1, each of which, in this example, will also be 24 bits long.

In order to conserve bandwidth, the encoder of the present invention then preferably strips payload blocks of any padded zeros before passing the blocks to the packetizer for packetizing and transmission. At the same time, however, in order to facilitate recovery of lost packets (i.e., decoding) at the receiving end, the encoder preferably concatenates with each payload block an indication of how long the payload block needs to be for decoding. Additionally, if a redundancy block is to be included in a packet, the encoder of the present invention preferably includes in the concatenated symbol an indication of how long the redundancy block is, in order to indicate where the redundancy block ends. Thus, for instance, in the above example, the encoder preferably adds to the symbol extra bits indicating that the payload block needs to be 24 bits long for decoding, and the encoder adds to the symbol extra bits indicating that the redundancy block is 24 bits long.

Consequently, to facilitate decoding in the preferred embodiment, a packet that is transmitted according to the present invention preferably includes indications of the sequence number or packet number, the (n, k) values, the payload/data length, the payload/data block, the redundancy block length (if any) and a redundancy block (if any). Figure 3 illustrates an example of a packet containing this information, where the encoder employs a block coder using (n, k) values (6, 5) (*i.e.*, p=1), and where a redundancy block (FEC) has been concatenated with a payload block. Alternatively, it will be appreciated that the (n, k) values could be agreed upon in advance so that the values of n and k need not be included in each packet header.

In most cases, the packets of information encoded according to the present invention are successfully transmitted through a network from the sending end to a receiving end. As discussed above, however, a number of these packets may be lost along the way and never make it to the receiving end. In the Internet, for instance, it is normal to lose 3% to 5%, and even up to 20%, of packets. Therefore, at the receiving end, a decoder is should be in place to recover missing packets.

Figure 4 illustrates a decoder 20 operating in accordance with a preferred embodiment of the invention. As shown in Figure 4, a stream of packets arrives at the receiving end. These packets are illustrated arriving in sequential order. However, it will be appreciated that the present invention is not limited to sequential packet transmission. Rather, depending on the packet switching protocol in use, these packets may arrive in sequential order or out of order. A packet switching protocol such as ATM, for instance, transmits packets in sequence. However, in other transmission systems, sequentially numbered packets might be routed differently through a network and therefore do not arrive at the receiving end in their original sequence.

As shown in the example of Figure 4, all of the packets P[1] through P[6] have arrived successfully at the receiving end except for packet P[3]. According to the present invention, each of these packets is parsed by a de-packetizer 22 to remove payload or redundancy blocks from packet header information. Those payload blocks that arrived successfully are preferably forwarded directly to a dynamic buffer 24, which serves to put the payload blocks in proper sequence for receipt by an end user.

Additionally, the parsed payload blocks and redundancy blocks are stored as necessary in memory, with memory space preferably allocated according to the payload length and redundancy length that were indicated in the packet. To facilitate decoding and recovery of any lost packets, the number of blocks stored in memory may depend on the (n, k) values of the block coder originally used for encoding. In the example shown, packet P[3] (and therefore payload block PL[3]) is missing. Further, assume that the (n, k) values of the encoder are (5, 3). Accordingly, the decoder 20 stores in memory at least payload blocks PL[1] and PL[2] and redundancy blocks FEC0 and FEC1. Decoder 20 need not store payload blocks PL[4], PL[5] and PL[6] in memory in this example, because no payload in that group of three (k) is

missing. Nevertheless, to illustrate context, packets P[4], P[5] and P[6] are depicted in the decoder.

In the preferred embodiment, every time decoder 20 receives new data, it determines whether the new data can help to recover missing information. Thus, in the example shown in Figure 4, after decoder 20 has received payload blocks PL[1], PL[2] and PL[4] and redundancy block FEC0 (which was transmitted with payload block PL[4]), it recognizes that payload block PL[3] is missing. Given the (5, 3) RS block coder used in this example, decoder 20 can recover the missing payload block PL[3]. In particular, decoder 20 may employ a (5, 3) RS decoder to derive PL[3], given PL[1], PL[2] and FEC0.

Once decoder 20 recovers the missing payload block PL[3], it passes PL[3] to the dynamic buffer 24. Dynamic buffer 24 in turn places PL[3] in sequential order between payload blocks PL[2] and PL[4] and forwards the ordered payload to the end user.

As those skilled in the art will appreciate from the foregoing, in the example shown, if any two of three sequential payload blocks are then lost, the RS block decoder will be able to recover the lost data as long as the necessary redundancy block(s) arrive. In particular, in the example, if up to two of packets P[1], P[2] and P[3] are lost, decoder 20 can recover the lost payload as long as packets P[4] and P[5], and therefore redundancy blocks FEC0 and FEC1, arrive successfully. Note that if only one packet P[1], P[2] or P[3] was lost, only one packet P[4] or P[5] needs to arrive, since one FEC block is enough to reconstruct one lost data block.

The present invention thus conveniently enables the correction of burst errors (the loss of multiple sequential packets in a row), as long as the required redundancy blocks arrive successfully. In this regard, it is further contemplated that a still more robust solution to packet loss may be achieved by widely distributing the redundancy blocks, or portions of the redundancy blocks, within the k packets following those from which the redundancy blocks were derived, rather than placing the redundancy blocks in adjacent packets. By doing so, there may be less chance that the redundancy blocks themselves will be lost as a result of a burst error.

As noted above, by selecting desired (n, k) values, it is possible to vary the efficiency of the correction mechanism provided by the present invention. In this

regard, it will be understood that the choice of these values is based on balance between delay and burst recovery. As higher values of p are used, the decoder will be able to recover more packets lost at once. At the same time, as higher values of k are used, the decoder will have to wait longer to recover lost packets, which, as explained above, would be undesirable for transmission of real-time media signals such as voice, video or audio.

For instance, using (n, k) values of (10, 9), an encoder would derive one redundancy block from every nine payload blocks and would include that redundancy block in one of the next nine packets. If exactly one of nine sequential packets is then lost in transmission, the lost payload can be recovered as described above. If, however, more than one of nine packets is lost in transmission, the one redundant block would not enable recovery of all of the lost packets.

Using (n, k) values of (10, 8) instead, the encoder of the preferred embodiment would create two RS-coded redundancy blocks from every eight payload blocks and would include those redundancy blocks in two of the next eight payload blocks. With this setting, it is possible according to the present invention to lose up to any two of eight sequential packets and to recover those lost packets using the two RS-coded redundant blocks.

For transmissions of coded voice signals over the Internet, an RS-coder using (n, k) values of (4, 3) is preferred. With these values, the transmission system can survive a loss of one packet in every four, which would be a 25% loss rate.

In contrast, for transmissions of coded video signals over the Internet, an RS-coder using (n, k) values of (18, 9) is preferred. These values provide nine redundancy blocks and therefore allow for recovery from a loss of nine packets out of every eighteen. These (n, k) values would work particularly well in the context of the H.263 low bit rate video coding standard currently recommended by the International Telecommunications Union, because the H.263 standard calls for dividing a single video frame into 9 pieces or packets. Therefore, each video frame could be separately corrected according to the present invention as long as the next video frame arrives completely.

While an encoder or decoder operating in accordance with the present invention may take any of a variety of forms (such as hardware, software or

firmware), both the encoding and decoding functions are preferably carried out by a computer processor operating a set of machine language instructions that are stored in a memory. As an example, Appendix A to this description sets forth a C++ source code listing, which can be compiled by any standard C++ compiler and executed by an appropriate processor. In this listing, the modules rs.c and rs.h are taken from the public domain web site http://hideki.iis.u-tokyo.ac.ip/~robert/rs.tar.

As the foregoing illustrates, the present invention provides a computationally simple mechanism for encoding and decoding a sequence of packets in order to recover lost packets. Beneficially, the invention accomplishes this function without increasing packet rate and without substantially increasing the data rate of transmission beyond the single FEC block (and trivial amount of other overhead) added to specified packets. As the value of p is increased, the present invention conveniently enables the recovery from larger burst errors. The present invention thus provides a powerful mechanism for reducing the effect of packet loss.

2. **Concatenated XOR Parity Blocks**

According to another preferred embodiment of the present invention, a single FEC block is appended to each payload packet. This single FEC block is about the same size as the payload packet and is computed by taking the XOR (exclusive-or, or $\oplus$) product of a predetermined number w of preceding payload packets, where w is preferably more than 1. In turn, at the receiving end, if a packet is lost, the associated payload may be extracted from the XOR sum carried by one or more subsequent data packets.

In general, the present invention therefore calls for building a forward error correction code FEC[k] for each payload packet P[k], where FEC[k] = PL[k-1] $\oplus$ PL[k-2] $\oplus$, . . . , $\oplus$ PL[k-w], and where w is a positive integer generally greater than 1. This FEC[k] is then piggy-backed onto the payload PL[j], where j > k. The resulting packet P[k] is therefore the concatenation of the payload and the FEC: P[k] = {PL[k], FEC[k]}. Figure 5 illustrates a data stream containing a sequence of packets concatenated with their associated forward error correction codes in this fashion.

The predetermined number w defines a sliding window over which the XOR operation is taken and, as a result, defines the length of a burst error, or number of lost packets in a row, from which the system is able to recover. As a block coder that adds

a single redundancy block to each data packet based on the previous *w* data packets, the present invention may be understood to employ (n, k) values of

$$k = w,$$

and

$$n = k + 1.$$

Thus, for instance, if the sliding window *w* is 5, then the (n, k) values of the XOR block coder according to the present invention are (6, 5).

By repeatedly generating block codes according to a sliding window over a sequence of packets, the packet data is used in several block codes rather than in only a single block code. Consequently, the present invention provides a high order redundancy and enables the recovery of multiple lost packets in a row, while requiring only a single redundancy block per packet.

To better understand the operation of this sliding window, Figure 6 depicts several stages of an encoder operating in a preferred embodiment of the invention. Figure 6 assumes that 10 packets (numbered 1-10) are to be transmitted in a network and that the window size *w* is 3. In the first stage shown, the window *w* covers payload packets PL[1] - PL[3], so that the encoder computes FEC[4] = PL[1] $\oplus$ PL[2] $\oplus$ PL[3]. The encoder then appends this FEC[4] redundancy block to payload packet PL[4] and outputs the resulting packet P[4] = {PL[4], FEC[4]} for transmission to the receiving end. In turn, in the next stage shown, the window *w* covers payload packets P[2] - P[4], so that the encoder computes FEC[5] = PL[2] $\oplus$ PL[3] $\oplus$ PL[4]. The encoder then appends this FEC[5] redundancy block to payload packet PL[5] and outputs the resulting packet P[5] = {PL[5], FEC[5]} for transmission to the receiving end. As partially illustrated in the figure, this process continues as long as the sequence of payload packets continues.

The use of the XOR operation presumes that the packets being combined are of the same length (for instance, the same number of bits). The present invention, however, will work equally well in a network of variable length packets. Provided with packets of various lengths, the invention contemplates padding the shorter packets with zeros, so that all packets combined in a single XOR operation, as well as the resulting XOR sum, will be the length of the longest among them. Once the XOR coding operation is complete, the extra zeros are dropped from each padded packet,

and the unpadded packet is output for transmission to the network (together with a redundancy block based on the previous *w* packets). In this variation, extra codes may be added to the header to indicate the lengths of the data block and the redundancy block, in order to facilitate decoding in accordance with the invention.

It should be further appreciated, of course, that the FEC contemplated by the present invention is not restricted to being equal to the above-described XOR sum, but may include other modifications as necessary. For instance, the present invention would extend to the use of an FEC that is computed by taking an XOR sum of the previous *w* payload packets and then inverting one or more predetermined digits or cyclically shifting the code or resulting packet. Additionally, it will be understood that the above equation is set forth for purposes of illustration only and that the present invention is not necessarily limited to employing the XOR sum of 3 or more (k-1, k-2, . . ., k-w) preceding payload packets; the invention may, for instance, extend to the XOR sum of only the previous two payload packets as well. Still further, it will be appreciated that the sequence illustrated in Figure 1 is shown for example only; the present invention may equally extend to separate and independent transmission of packets P[k] = {PL[k], FEC[k]} through a packet switched network.

As a general example of the encoding and decoding process contemplated by the present invention, the following illustrates the structure of a series of payload packets P[k] generated by an encoder, given payload frame PF[k] and a window *w* = 3:

$$P[1] = \{ PL[1], FEC[1] \} = \{ PL[1], \quad PL[0] \oplus PL[-1] \oplus PL[-2] \}$$

$$P[2] = \{ PL[2], FEC[2] \} = \{ PL[2], \quad PL[1] \oplus PL[0] \oplus PL[-1] \}$$

$$P[3] = \{ PL[3], FEC[3] \} = \{ PL[3], \quad PL[2] \oplus PL[1] \oplus PL[0] \}$$

$$P[4] = \{ PL[4], FEC[4] \} = \{ PL[4], \quad PL[3] \oplus PL[2] \oplus PL[1] \}$$

$$P[5] = \{ PL[5], FEC[5] \} = \{ PL[5], \quad PL[4] \oplus PL[3] \oplus PL[2] \}$$

$$P[6] = \{ PL[6], FEC[6] \} = \{ PL[6], \quad PL[5] \oplus PL[4] \oplus PL[3] \}$$

$$P[7] = \{ PL[7], FEC[7] \} = \{ PL[7], \quad PL[6] \oplus PL[5] \oplus PL[4] \}$$

$$P[8] = \{ PL[8], FEC[8] \} = \{ PL[8], \quad PL[7] \oplus PL[6] \oplus PL[5] \}$$

$$P[9] = \{ PL[9], FEC[9] \} = \{ PL[9], \quad PL[8] \oplus PL[7] \oplus PL[6] \}$$

* * *

$$P[k] = \{ PL[k], FEC[k] \} = \{ PL[k], \quad PL[k-1] \oplus PL[k-2] \oplus PL[k-3] \}$$

Assume that packet P[5] was lost in transmission. With the present invention, a decoder may recreate packet P[5] by using the FEC of the packets in which packet P[5] was included. In this case, since w=3, the three packets following packet P[5] are each based in part on the value of payload packet PL[5]. Consequently, payload packet PL[5] may be recovered by solving any of the equations defining these three packets. For instance, using packet P[8],

$$FEC[8] = PL[7] \oplus PL[6] \oplus PL[5],$$

and $$PL[5] = FEC[8] \oplus PL[7] \oplus PL[6].$$

This example may be extended to illustrate that, provided with a window w=3, the present invention enables the recovery of three packets lost in a row. Assume, for instance, that packets P[4], P[5] and P[6] are lost. In order to recover the payload carried by these lost packets, the subsequent three packets, P[7], P[8] and P[9], will need to have arrived successfully. Provided with these three packets, the receiving end may first extract payload packet PL[6] from FEC[9] as follows:

$$FEC[9] = PL[8] \oplus PL[7] \oplus PL[6],$$

and $$PL[6] = FEC[9] \oplus PL[8] \oplus PL[7].$$

Next, the receiving end may extract payload packet PL[5] from FEC[8] as follows:

$$FEC[8] = PL[7] \oplus PL[6] \oplus PL[5],$$

and $$PL[5] = FEC[8] \oplus PL[7] \oplus PL[6].$$

Finally, the receiving end may extract payload packet PL[4] from FEC[7] as follows:

$$FEC[7] = PL[6] \oplus PL[5] \oplus PL[4],$$

and $$PL[4] = FEC[7] \oplus PL[6] \oplus PL[5].$$

The foregoing illustrates a straightforward mechanism for recovering lost data within the present invention. This mechanism works well when the transmitted packets arrive in sequence, as would occur with the transmission of ATM cells. With packets arriving in sequence, a lost packet can be recovered as soon as the next w packets successfully arrive. In many transmission systems, however, sequentially numbered packets that are sent in order through a network do not arrive at the receiving end in their original sequence.

To enable more robust recovery of lost packets, a preferred embodiment of the invention preferably operates as follows. At the sending end, an encoder translates

incoming data into packets, using the XOR encoding mechanism of the present invention. As described above, this XOR encoder employs a sliding window *w*, such that the n and k values of the block coder are (n=k+1, k=*w*). Each packet preferably includes an indication of the sequence number (packet number), the (n, k) values, a payload/data block and a redundancy block. Additionally, to account for possible variation in packet length, each packet also preferably includes an indication of data length and an indication of redundancy length. Figure 7 illustrates an example of a packet containing this information, where the encoder employs a sliding window *w* of 4 (and, therefore, k = 4).

At the receiving end, a decoder is in place to receive these packets and recover any lost packets that can be recovered. According to the preferred embodiment of the invention, as the decoder receives a packet, it stores the packet in memory, parses the packet into its components, and creates a "bubble" of information (for instance, a "c"-structure), containing the sequence number, and pointers to the data block and the redundancy block in the packet. The decoder then places the bubble into a doubly linked list by storing a pointer to the bubble. Additionally, the decoder preferably passes the data block downstream for use by other elements in the transmission system.

In the preferred embodiment, each time the decoder receives new information and adds a bubble to the link list, it determines whether the information in the bubble can help to recover any missing information. For example, Figure 8 illustrates a series of bubbles 2-17 compiled by a sample decoder. As shown by Figure 8, assume that bubbles 9 and 12 are missing, but the decoder has so far received packets sufficient to provide the information needed for bubbles 2-8, 10-11 and 13-17. Assume now that packet 9 arrives. The decoder receives packet 9 and inserts bubble 9 in the link list, to obtain a revised link list shown in Figure 9.

Because the sliding window *w* in this example is 4 (as established by the value of k included in each received packet), the decoder knows that, in an ordered sequence of packets, each packet is related through an XOR operation with the preceding 4 packets as well as with the following 4 packets. In the present example, therefore, the decoder knows that the information newly received from packet number 9 can help to recover lost data, if any, in bubbles 5-8 and 10-13. This range may be referred to as

the relevant range with respect to newly received packet number 9. Knowing this relevant range conveniently limits the work that the decoder needs to do when it receives a new packet in order to decide whether the information can help in some way.

In the preferred embodiment, the decoder analyzes the relevant range (5-13 in this example) as follows. The decoder begins at the end of the range by considering bubbles 9-13, which may be referred to as a window of analysis. In this window of analysis, the decoder first determines whether there is exactly one data block missing in the range 9-12 that the redundancy block 13, which needs to be in the linked list, can help recover (through the XOR operation). If more than one data block is missing in the range 9-12, the decoder knows that it cannot, with the XOR operation, recover any data in the window of analysis. In that case, or in the event the decoder finds no missing data in the window of analysis, then the decoder moves the window of analysis up one notch and repeats this analysis, now for the window of analysis defined by bubbles 8-12. The decoder continues this process until it completes its analysis with the top of the window of analysis at the top of the relevant range, which is bubble 5 in this example.

If the decoder determines that exactly one data block is missing in a given window of analysis, and the last bubble in the analysis window contains a redundancy block, then the decoder may employ the XOR operation as discussed above to recover the missing data block. Once it does so, it will have received a new piece of information, namely, the recovered data block, which should give rise to a new relevant range. As a result, the decoder may jump back to a higher number bubble (i.e., down in the link list), moving the window of analysis to a higher number starting point. The decoder may then continue moving the window of analysis up the link list until the top of the window of analysis has reached the top of the original relevant range.

In this example, for instance, once the window of analysis is over bubbles 9-13, the missing data block 12 can be recovered by taking the XOR sum of data blocks 9, 10 and 11 and redundancy block 13. The decoder will therefore have the newly received information of data block 12, which gives rise to a new relevant range of 8-16 (i.e., 8-12 and 12-16). As a result, the decoder positions a new window of analysis

to start over bubbles 12-16 and to move upward bubble-by-bubble as discussed above. For instance, if packet 15 had not yet arrived, but data blocks 12, 13 and 14 and redundancy block 16 had arrived, then the decoder could compute data block 15 using the XOR operation.

It will be appreciated that two window of analysis loops are thus in operation. A first window of analysis loop, or "outer loop," is operating as a result of the receipt of data block 9. A second window of analysis loop, or "inner loop," is operating as a result of the recovery of data block 12. The outer loop is geared to move a window of analysis up the link list until the decoder has completed its analysis of bubbles 5-9. The inner loop, in comparison, is geared to move a window of analysis up the link list until the decoder has completed its analysis of bubbles 8-12.

In order to avoid unnecessary repetition, a decoder operating according to the preferred embodiment of the present invention will complete the inner loop and will then continue moving up the link list to complete the outer loop, without jumping back to where it left off before it began the inner loop. Thus, in the above example, the steps performed by the decoder are illustrated by the brackets in Figure 9 and by the rows in Figure 10. As shown in these Figures, at step 1, the decoder first analyzes bubbles 9-13 and recovers lost data block 12. At step 2, the decoder therefore jumps to a window of analysis over bubbles 12-16, where the decoder recovers no new data. At steps 3 through 6, the decoder moves the window of analysis bubble-by-bubble up the link list, until it has completed the inner loop at bubbles 8-12, recovering no new data at each step.

Next, rather than recursively returning to the outer loop and repeating an analysis of bubbles 8-12, the decoder continues moving up the link list from where it left off with the inner loop, analyzing bubbles 7-11 at step 7, bubbles 6-10 at step 8, and bubbles 5-9 at step 9. In this example, the decoder does not recover any additional data along the way. After completing its analysis of bubbles 5-9, the decoder's analysis is complete, and the decoder waits for the arrival of a new packet to begin another analysis.

While an encoder or decoder operating in accordance with the present invention may take any of a variety of forms (such as hardware, software or firmware), both the encoding and decoding functions are preferably carried out by a

computer processor or microprocessor operating a set of machine language instructions that are stored in a memory. As an example, Appendix B to this description sets forth a C++ source code listing, which can be compiled by any standard C++ compiler and executed by an appropriate processor.

The computer processor and/or machine language instructions that carries out the encoding function according to the present invention may be referred to as a first segment of a communication apparatus, and a device that transmits the resulting encoded packets according to the present invention may be referred to as a second segment of the communication apparatus. In turn, the computer processor and/or machine language instructions that recreates a lost payload packet by extracting information from other packets may be referred to as a third segment of the communication apparatus. Alternatively, it will be appreciated that the various components required to carry out the present invention may be divided into other segments of a communication apparatus.

As the foregoing illustrates, the present invention provides a computationally simple mechanism for encoding and decoding a sequence of packets in order to recover lost packets. Beneficially, the invention accomplishes this function without increasing packet rate and without substantially increasing the data rate of transmission beyond the single FEC block (and trivial amount of other overhead) added to each payload packet. As the window size w is increased, provided with successful transmission and receipt of sufficient adjacent data blocks, the present invention conveniently enables the recovery from larger burst errors. The present invention thus provides a powerful mechanism for reducing the effect of packet loss, by establishing high order redundancy through the use of a sliding window and an efficient forward error correction code as described above.

APPENDIX A

```
/* CODER.C */

/*****************************************************************************/
/*                                                                           */
/* coder.c                                                                   */
/*                                                                           */
/* 10/7/97                                                                   */
/*                                                                           */
/* Guido Schuster                                                            */
/* Advanced Technologies                                                     */
/* Carrier Systems Division                                                  */
/* 3COM                                                                      */
/* 1800 W. Central Rd.                                                       */
/* Mount Prospect, Il  60056                                                 */
/*                                                                           */
/* This is a test program for the forward error correction modules.  Twenty packets of      */
/* variable length are created.  These packets are then put into the channel coder.  Then   */
/* some of the output packets of the channel coder are  dropped and/or rearranged. This     */
/* results in a packet  stream with out of order packets and lost packets. This stream is   */
/* then fed into the channel decoder which attempts to recover the dropped packets. Finally */
/* the resulting packet stream is displayed                                  */
/*****************************************************************************/


/********************************* The Include Files *************************************/
#include "stdlib.h" // the basic libraries
#include "malloc.h"
#include "string.h"
#include "windows.h"

#include "types.h"  // this is where the general types are defined
#include "RTOS.h"   // the real time operating system (rtos),
                    // basically the messageboxes

#include "rs_coder.h"       // the reed solomon channel coder
#include "rs_decoder.h"     // the reed solomon channel decoder
#include "xor_coder.h"      // the XOR based channel coder
#include "xor_decoder.h"    // the XOR based channel decoder
/*****************************************************************************/


/******************************** The Define Statements **********************************/
```

```
#define RS      0   // the reed solomon coder is used instead of the XOR coder
#define N_RS    6   // the block length for the FEC (forward error correction)
#define K_RS    4   // the number of information bytes
#define MEM_RS  10000   // the target number of bytes used for storing the packets

#define XOR     1   // the XOR coder is used instead of the RS coder
#define N_XOR   6   // the block length for the XOR
#define MEM_XOR 10000   // the target number of bytes used for storing the packets
/*****************************************************************************/


/****************************** The Main Program *****************************/
/* No Inputs, the only output is what is printed to the screen, which is the packet stream  */
/* exiting the decoder                                                                      */
void main (void )
 {

    PKT *pCoderInPacket[20]; // used to store the packets
    PKT *pCoderOutPacket[20];
    PKT *pDeCoderInPacket[20];
    PKT *pDeCoderOutPacket[20];

    MESSAGEBOX *pCoderInBox = CreateEmptyMessageBox(); // used to pass the packets
    MESSAGEBOX *pCoderOutBox = CreateEmptyMessageBox();
    MESSAGEBOX *pDeCoderInBox = CreateEmptyMessageBox();
    MESSAGEBOX *pDeCoderOutBox = CreateEmptyMessageBox();

    // init the coder and the decoder
    if (XOR)
    {
        init_xor_coder(N_XOR);
        init_xor_decoder(N_XOR,MEM_XOR);
    }
    if (RS)
    {
        init_rs_coder(N_RS,K_RS);
        init_rs_decoder(N_RS,K_RS,MEM_RS);
    }

    // Initialize the test packets
    for(int i=0;i<20;i++)
    {
```

```
        pCoderInPacket[i] = new PKT;
        pCoderInPacket[i]->nLengthBytes=i;
        pCoderInPacket[i]->pData=malloc(pCoderInPacket[i]->nLengthBytes);
        for (int j=0;j<pCoderInPacket[i]->nLengthBytes;j++)
        {
            *(((unsigned char*)pCoderInPacket[i]->pData)+j)='0'+i%10;
        }
    }

    // Send the test packets into the coder and run the coder
    for(i=0;i<20;i++)
    {
        AddMessageToBox(pCoderInBox,(void*)pCoderInPacket[i]);
        if(XOR)
        {
            xor_coder(pCoderInBox,pCoderOutBox);
        }
        if(RS)
        {
            rs_coder(pCoderInBox,pCoderOutBox);
        }
    }

    // Simulate the Internet, i.e., packets can be lost or arrive out of sequence
    // first read the packets
    for(i=0;i<20;i++)
    {
        pCoderOutPacket[i]=(PKT*)GetMessageFromBox(pCoderOutBox);
    }

    // drop some packets
    for (i=0;i<20;i++)
    {
        if (i==1||i==7||i==13||i==14)
        {
            pCoderOutPacket[i]=NULL;
        }
    }

    // change the sequence
    int from[20]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19};
    int to[20]  ={1,2,0,3,4,5,9,6,7,8,10,11,12,13,14,18,15,16,17,19};
```

```
for(i=0;i<20;i++)
{
    pDeCoderInPacket[to[i]]=pCoderOutPacket[from[i]];
}

// pass the packets to the decoder
for(i=0;i<20;i++)
{
    if(pDeCoderInPacket[i])
    {
        AddMessageToBox(pDeCoderInBox,pDeCoderInPacket[i]);
        // run the decoder
        if(RS)
        {
            rs_decoder(pDeCoderInBox,pDeCoderOutBox);
        }
        if(XOR)
        {
            xor_decoder(pDeCoderInBox,pDeCoderOutBox);
        }
    }

    // display the result
    for(i=0;i<20;i++)
    {
        while(pDeCoderOutPacket[i]=(PKT*)GetMessageFromBox(pDeCoderOutBox))
        {
            PrintPacket(pDeCoderOutPacket[i]);
        }
    }

}
/*****************************************************************************************/


/* DLLIST.C */

#include <stdio.h>
#include <stdlib.h>
#include "dllist.h"
```

```
static int nNextDLLId = 0;

/*************************************************************************/
/**                                                                     **/
/**  Call this function to create a doubly linked list and to get a     **/
/**  pointer to the list.  This pointer is needed to do all operations  **/
/**  on the list.                                                       **/
/**                                                                     **/
/**  When a list is done being used (such as at the end of the          **/
/**  program), call DLLDestroy() with this pointer.                     **/
/**                                                                     **/
/**  Returns:  pointer to the linked list structure.                    **/
/**                                                                     **/
/*************************************************************************/
DLLType* DLLCreate()
{
    DLLType* pDLL;

    /* generate a new ID for the list */
    nNextDLLId++;

    pDLL = (DLLType*)malloc(sizeof(DLLType));
    if( pDLL == NULL )
    {
        /* error -- not enough memory */
        return (NULL);
    }

    /* fill in fields in DLL structure */
    pDLL->pHead   = NULL;
    pDLL->pTail   = NULL;
    pDLL->ulCount = 0;
    pDLL->nID     = nNextDLLId;
    pDLL->pPtr    = NULL;
    return (pDLL);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to destroy a doubly linked list that was        **/
/**  created with the DLLCreate() function.  The pointer obtained from  **/
/**  DLLCreate() must be supplied to this function.                     **/
```

```
/**                                                                    **/
/**  Note that only the nodes of the linked list and the data          **/
/**  structure holding information about the linked list is destroyed. **/
/**  If any nodes are in the list when this function is called, the    **/
/**  data pointed to by the nodes is NOT cleared.  The clearing of     **/
/**  that data is the responsibility of the program that allocated     **/
/**  that memory.                                                      **/
/**                                                                    **/
/************************************************************************/
void DLLDestroy( DLLType* pDLL )
{
    DLLNodeType* pDLLNode;
    DLLNodeType* pDLLNextNode;

    if( DLLVerifyIntegrity(pDLL) )
    {
        /* DLL integrity not violated.  */
        /* Erase all nodes      */
        pDLLNode = pDLL->pHead;
        while( pDLLNode )
        {
            pDLLNextNode = pDLLNode->pNextNode;
            free( pDLLNode );
            pDLLNode = pDLLNextNode;
        }
    }

    free( pDLL );
}

/************************************************************************/
/**                                                                    **/
/**  Call this function to insert a node into the linked list at the   **/
/**  head of the list.  This node will point to the data stored at     **/
/**  pData.                                                            **/
/**                                                                    **/
/**  Returns:  0 if operation failed (out of memory)                   **/
/**            1 if operation successful                               **/
/**                                                                    **/
/************************************************************************/
int DLLInsertAtHead( DLLType* pDLL, void* pData )
{
```

```
DLLNodeType* pDLLNode;

pDLLNode = (DLLNodeType*)malloc(sizeof(DLLNodeType));
if( pDLLNode == NULL )
{
    /* error -- not enough memory */
    return (0);
}

/* update all pointers */
if( pDLL->pHead == NULL )
{
    /* list empty */
    pDLL->pHead = pDLLNode;
    pDLL->pTail = pDLLNode;
    pDLLNode->pNextNode = NULL;
    pDLLNode->pPrevNode = NULL;
}
    else
    {
        /* list not empty */
        pDLL->pHead->pPrevNode = pDLLNode;
        pDLLNode->pNextNode = pDLL->pHead;
        pDLLNode->pPrevNode = NULL;
        pDLL->pHead = pDLLNode;
    }

    /* change pointer to data */
    pDLLNode->pData = pData;

    /* update counter information */
    pDLL->ulCount++;

    return (1);
}

/*****************************************************************************/
/**                                                                         **/
/**  Call this function to insert a node into the linked list at the        **/
/**  tail of the list.  This node will point to the data stored at          **/
/**  pData.                                                                 **/
```

```
/**                                                                    **/
/**  Returns:  0 if operation failed (out of memory)                   **/
/**            1 if operation successful                               **/
/**                                                                    **/
/************************************************************************/
int DLLInsertAtTail( DLLType* pDLL, void* pData )
{
    DLLNodeType* pDLLNode;

    pDLLNode = (DLLNodeType*)malloc(sizeof(DLLNodeType));
    if( pDLLNode == NULL )
    {
        /* error -- not enough memory */
        return (0);
    }

    /* update all pointers */
    if( pDLL->pTail == NULL )
    {
        /* list empty */
        pDLL->pTail = pDLLNode;
        pDLL->pHead = pDLLNode;
        pDLLNode->pNextNode = NULL;
        pDLLNode->pPrevNode = NULL;
    }

    else
    {
        /* list not empty */
        pDLL->pTail->pNextNode = pDLLNode;
        pDLLNode->pPrevNode = pDLL->pTail;
        pDLLNode->pNextNode = NULL;
        pDLL->pTail = pDLLNode;
    }

    /* change pointer to data */
    pDLLNode->pData = pData;

    /* update counter information */
    pDLL->ulCount++;

    return (1);
```

```
}

/*****************************************************************************/
/**                                                                         **/
/**  Call this function to remove a node from the linked list at the        **/
/**  head of the list.  A pointer to the data pointed to by that node       **/
/**  is returned.                                                           **/
/**                                                                         **/
/**  Returns:  NULL            if operation failed (list empty)             **/
/**            pointer to data if operation successful                      **/
/**                                                                         **/
/*****************************************************************************/
void* DLLRemoveFromHead( DLLType* pDLL )
{
    void* pData;
    DLLNodeType* pNewHead;

    /* check for empty list */
    if( pDLL->pHead == NULL )
    {
        /* list empty   */
        return (NULL);
    }

    /* remember where data is */
    pData = pDLL->pHead->pData;

    /* check for future validity of DLL Pointer */
    if( pDLL->pPtr == pDLL->pHead )
    {
        /* DLL Pointer points to head node  */
        /* DLL Pointer becomes invalid now  */
        pDLL->pPtr = NULL;
    }

    /* update all pointers */
    if( pDLL->pHead == pDLL->pTail )
    {
        /* only one element in list */
        free( pDLL->pHead );
        pDLL->pHead = NULL;
        pDLL->pTail = NULL;
```

```
    }

    else
    {
        /* not the only element in list */
        pNewHead = pDLL->pHead->pNextNode;
        pNewHead->pPrevNode = NULL;
        free( pDLL->pHead );
        pDLL->pHead = pNewHead;
    }

    /* update counter information */
    pDLL->ulCount--;

    return (pData);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to remove a node from the linked list at the    **/
/**  tail of the list.  A pointer to the data pointed to by that node   **/
/**  is returned.                                                       **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLRemoveFromTail( DLLType* pDLL )
{
    void* pData;
    DLLNodeType* pNewTail;

    /* check for empty list */
    if( pDLL->pTail == NULL )
    {
        /* list empty   */
        return (NULL);
    }

    /* remember where data is */
    pData = pDLL->pTail->pData;
```

```
    /* check for future validity of DLL Pointer */
    if( pDLL->pPtr == pDLL->pTail )
    {
        /* DLL Pointer points to head node  */
        /* DLL Pointer becomes invalid now  */
        pDLL->pPtr = NULL;
    }

    /* update all pointers */
    if( pDLL->pHead == pDLL->pTail )
    {
        /* only one element in list */
        free( pDLL->pTail );
        pDLL->pHead = NULL;
        pDLL->pTail = NULL;
    }

    else
    {
        /* not the only element in list */
        pNewTail = pDLL->pTail->pPrevNode;
        pNewTail->pNextNode = NULL;
        free( pDLL->pTail );
        pDLL->pTail = pNewTail;
    }

    /* update counter information */
    pDLL->ulCount--;

    return (pData);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to remove a node from the linked list at the    **/
/**  specified node.                                                    **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLRemove( DLLType* pDLL, DLLNodeType* pDLLNode )
```

```
{
    void* pData;

    if( pDLLNode == NULL )
    {
        /* error -- supplied pointer invalid */
        return (NULL);
    }

    if( pDLLNode == pDLL->pHead )
    {
        /* removing from head */
        return (DLLRemoveFromHead(pDLL));
    }

    if( pDLLNode == pDLL->pTail )
    {
        /* removing from tail */
        return (DLLRemoveFromTail(pDLL));
    }

    /* removing one in middle */
    if( pDLLNode == pDLL->pPtr )
    {
        /* remove node pointed to by DLL Pointer */
        /* this will invalidate that pointer     */
        pDLL->pPtr = NULL;
    }

    if( pDLLNode->pPrevNode == NULL )
    {
        /* error -- supplied pointer not    */
        /* part of this DLL         */
        return (NULL);
    }

    if( pDLLNode->pNextNode == NULL )
    {
        /* error -- supplied pointer not    */
        /* part of this DLL         */
        return (NULL);
    }
```

```
    pDLLNode->pPrevNode->pNextNode = pDLLNode->pNextNode;
    pDLLNode->pNextNode->pPrevNode = pDLLNode->pPrevNode;

    /* remember where data is */
    pData = pDLLNode->pData;

    /* free node */
    free( pDLLNode );

    /* update counter information */
    pDLL->ulCount--;

    return (pData);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to peek at the head node in the linked list.    **/
/**  The node is not removed from the list, the pointer to the data     **/
/**  is simply returned.                                                **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLPeekAtHead( DLLType* pDLL )
{
    return( DLLPeek(pDLL, pDLL->pHead) );
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to peek at the tail node in the linked list.    **/
/**  The node is not removed from the list, the pointer to the data     **/
/**  is simply returned.                                                **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLPeekAtTail( DLLType* pDLL )
```

```
{
    return( DLLPeek(pDLL, pDLL->pTail) );
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to peek at the specified node in the linked     **/
/**  list.  The node is not removed from the list, the pointer to the   **/
/**  data is simply returned.                                           **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLPeek( DLLType* pDLL, DLLNodeType* pDLLNode )
{
    /* prevent compiler complaining */
    pDLL->ulCount += 0;

    if( pDLLNode == NULL )
    {
        /* error -- supplied pointer invalid */
        return (NULL);
    }

    return (pDLLNode->pData);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to insert a node pointing to the specified      **/
/**  data before the specified node in the list.                        **/
/**                                                                     **/
/**  Returns:  0 if operation failed (out of memory, etc)               **/
/**            1 if operation successful                                **/
/**                                                                     **/
/*************************************************************************/
int DLLInsertBefore( DLLType* pDLL, DLLNodeType* pDLLNode, void* pData )
{
    DLLNodeType* pDLLNewNode;

    if( pDLLNode == NULL )
```

```
{
    /* means put at end */
    return (DLLInsertAtTail(pDLL,pData));
}

if( pDLL->pHead == pDLLNode )
{
    /* inserting at beginning */
    return (DLLInsertAtHead(pDLL,pData));
}

if( pDLLNode->pPrevNode == NULL )
{
    /* error -- supplied pointer is not */
    /* head of list, but previous node  */
    /* is NULL -- perhaps node not part */
    /* of this linked list              */
    return (0);
}

pDLLNewNode = (DLLNodeType*)malloc(sizeof(DLLNodeType));
if( pDLLNewNode == NULL )
{
    /* error -- not enough memory */
    return (0);
}

/* inserting in middle */
pDLLNewNode->pNextNode = pDLLNode;
pDLLNewNode->pPrevNode = pDLLNode->pPrevNode;
pDLLNewNode->pPrevNode->pNextNode = pDLLNewNode;
pDLLNewNode->pNextNode->pPrevNode = pDLLNewNode;

/* change pointer to data */
pDLLNewNode->pData = pData;

/* update counter information */
pDLL->ulCount++;

return (1);
}
```

```
/*************************************************************************/
/**                                                                     **/
/**  Call this function to insert a node pointing to the specified      **/
/**  data after the specified node in the list.                         **/
/**                                                                     **/
/**  Returns:  0 if operation failed (out of memory, etc)               **/
/**            1 if operation successful                                **/
/**                                                                     **/
/*************************************************************************/
int DLLInsertAfter( DLLType* pDLL, DLLNodeType* pDLLNode, void* pData )
{
    DLLNodeType* pDLLNewNode;

    if( pDLLNode == NULL )
    {
        /* means put at beginning */
        return (DLLInsertAtHead(pDLL,pData));
    }

    if( pDLL->pTail == pDLLNode )
    {
        /* inserting at end */
        return (DLLInsertAtTail(pDLL,pData));
    }

    if( pDLLNode->pNextNode == NULL )
    {
        /* error -- supplied pointer is not */
        /* tail of list, but next node is   */
        /* NULL -- perhaps node not part of */
        /* this linked list         */
        return (0);
    }

    pDLLNewNode = (DLLNodeType*)malloc(sizeof(DLLNodeType));
    if( pDLLNewNode == NULL )
    {
        /* error -- not enough memory */
        return (0);
    }

    /* inserting in middle */
```

```
    pDLLNewNode->pPrevNode = pDLLNode;
    pDLLNewNode->pNextNode = pDLLNode->pNextNode;
    pDLLNewNode->pNextNode->pPrevNode = pDLLNewNode;
    pDLLNewNode->pPrevNode->pNextNode = pDLLNewNode;

    /* change pointer to data */
    pDLLNewNode->pData = pData;

    /* update counter information */
    pDLL->ulCount++;

    return (1);
}

/*****************************************************************************/
/**                                                                         **/
/**  Call this function to move the DLL Pointer to the head of the          **/
/**  linked list.                                                           **/
/**                                                                         **/
/**  Returns:  no return value                                              **/
/**                                                                         **/
/*****************************************************************************/
void DLLMovePtrToHead( DLLType* pDLL )
{
    pDLL->pPtr = pDLL->pHead;
}

/*****************************************************************************/
/**                                                                         **/
/**  Call this function to move the DLL Pointer to the tail of the          **/
/**  linked list.                                                           **/
/**                                                                         **/
/**  Returns:  no return value                                              **/
/**                                                                         **/
/*****************************************************************************/
void DLLMovePtrToTail( DLLType* pDLL )
{
    pDLL->pPtr = pDLL->pTail;
}

/*****************************************************************************/
/**                                                                         **/
```

```
/**  Call this function to move the DLL Pointer to the specified node   **/
/**  of the linked list.                                                **/
/**                                                                     **/
/**  Returns:  no return value                                          **/
/**                                                                     **/
/*************************************************************************/
void DLLSetPtrToPtr( DLLType* pDLL, DLLNodeType* pDLLNode )
{
    pDLL->pPtr = pDLLNode;
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to remove the node pointed to by the DLL        **/
/**  Pointer.                                                           **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLRemoveAtPtr( DLLType* pDLL )
{
    return DLLRemove( pDLL, pDLL->pPtr );
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to peek at the node pointed to by the DLL       **/
/**  Pointer.  The node is not removed from the list, the pointer to    **/
/**  the data is simply returned.                                       **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLPeekAtPtr( DLLType* pDLL )
{
    return DLLPeek( pDLL, pDLL->pPtr );
}

/*************************************************************************/
/**                                                                     **/
```

```
/**  Call this function to peek at the node pointed to by the DLL       **/
/**  Pointer, and then to advance the DLL Pointer to the next node.     **/
/**  If the DLL Pointer is NULL (such as is the case when the previous  **/
/**  call to this function read in the last node), then this function   **/
/**  returns NULL.                                                      **/
/**                                                                     **/
/**  Returns:  NULL            if list empty or if already read the     **/
/**                               last node previously                  **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLPeekNextPtr( DLLType* pDLL )
{
    void* pData;

    pData = DLLPeekAtPtr( pDLL );

    if( pDLL->pPtr != NULL )
    {
        pDLL->pPtr = pDLL->pPtr->pNextNode;
    }

    return (pData);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to advance the DLL Pointer to the next node     **/
/**  in the list (the node in the direction of the tail node).  If the  **/
/**  pointer currently points to the last node, the pointer does not    **/
/**  take on a NULL value; the pointer does not change, and a 0 is      **/
/**  returned.                                                          **/
/**                                                                     **/
/**  Returns:  0 if operation failed (list empty or already at last     **/
/**              node                                                   **/
/**            1 if operation successful                                **/
/**                                                                     **/
/*************************************************************************/
int DLLAdvancePtr( DLLType* pDLL )
{
    if( pDLL->pPtr == NULL )
    {
```

```
        pDLL->pPtr = pDLL->pHead;
        return (1);
    }

    else if( pDLL->pPtr == pDLL->pTail )
    {
        /* already at end */
        return (0);
    }

    else
    {
        pDLL->pPtr = pDLL->pPtr->pNextNode;
        return (1);
    }
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to retreat the DLL Pointer to the previous node **/
/**  in the list (the node in the direction of the head node).  If the  **/
/**  pointer currently points to the first node, the pointer does not   **/
/**  take on a NULL value; the pointer does not change, and a 0 is      **/
/**  returned.                                                          **/
/**                                                                     **/
/**  Returns:  0 if operation failed (list empty or already at first    **/
/**              node                                                   **/
/**            1 if operation successful                                **/
/**                                                                     **/
/*************************************************************************/
int DLLRetreatPtr( DLLType* pDLL )
{
    if( pDLL->pPtr == NULL )
    {
        pDLL->pPtr = pDLL->pTail;
        return (1);
    }

    else if( pDLL->pPtr == pDLL->pHead )
    {
        /* already at end */
        return (0);
```

```
    }

    else
    {
        pDLL->pPtr = pDLL->pPtr->pPrevNode;
        return (1);
    }
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to insert a node pointing to the specified      **/
/**  data before the node pointed to by the DLL Pointer.                **/
/**                                                                     **/
/**  Returns:  0 if operation failed (out of memory, etc)               **/
/**            1 if operation successful                                **/
/**                                                                     **/
/*************************************************************************/
int DLLInsertBeforePtr( DLLType* pDLL, void* pData )
{
    return DLLInsertBefore( pDLL, pDLL->pPtr, pData );
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to insert a node pointing to the specified      **/
/**  data after the node pointed to by the DLL Pointer.                 **/
/**                                                                     **/
/**  Returns:  0 if operation failed (out of memory, etc)               **/
/**            1 if operation successful                                **/
/**                                                                     **/
/*************************************************************************/
int DLLInsertAfterPtr( DLLType* pDLL, void* pData )
{
    return DLLInsertAfter( pDLL, pDLL->pPtr, pData );
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to get a pointer to the node pointed to by the  **/
/**  DLL Pointer.                                                       **/
/**                                                                     **/
```

```
/**  Returns:  NULL            if pointer not valid                    **/
/**        pointer to node if pointer valid                           **/
/**                                                                   **/
/***********************************************************************/
DLLNodeType* DLLGetPtrToPtr( DLLType* pDLL )
{
    return pDLL->pPtr;
}

/***********************************************************************/
/**                                                                   **/
/**  Call this function to determine whether the specified node is    **/
/**  contained in the specified linked list.                          **/
/**                                                                   **/
/**  Returns:  0 if no node specified or if node not in linked list   **/
/**        1 if node in linked list                                   **/
/**                                                                   **/
/***********************************************************************/
int DLLIsNodeInDLL( DLLType* pDLL, DLLNodeType* pDLLNode )
{
    unsigned long ulNumNodes;
    unsigned long ulCounter;
    DLLNodeType* pCurrentNode;

    if( pDLLNode == NULL )
    {
        /* error -- supplied pointer invalid */
        return (0);
    }

    ulNumNodes = pDLL->ulCount;

    ulCounter = 0;
    pCurrentNode = pDLL->pHead;

    while( (ulCounter<ulNumNodes)    &&
           (pCurrentNode!=NULL)      &&
           (pCurrentNode!=pDLLNode)     )
    {
        pCurrentNode = pCurrentNode->pNextNode;
        ulCounter++;
    }
```

```
    if( pCurrentNode == pDLLNode )
        return (1);
    else
        return (0);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to determine the number of nodes in a linked    **/
/**  list.                                                              **/
/**                                                                     **/
/**  Returns:  the number of nodes in linked list                       **/
/**                                                                     **/
/*************************************************************************/
unsigned long DLLCount( DLLType* pDLL )
{
    return (pDLL->ulCount);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to check the integrity of a linked list.  This  **/
/**  function checks certain criteria about the linked list to make     **/
/**  certain the structure remains valid.  If one suspects that a       **/
/**  linked list has become corrupt, this function should be used.      **/
/**                                                                     **/
/**  Returns:  0 if linked list corrupt                                 **/
/**            1 if linked list okay                                    **/
/**                                                                     **/
/*************************************************************************/
int DLLVerifyIntegrity( DLLType* pDLL )
{
    /* get number of nodes                                          */
    /* start from head and go until                                 */
    /*  (1) null, or                                                */
    /*  (2) check "number of nodes"                                 */
    /*  --- if reached end too soon or didn't reach end             */
    /*          after checking "number of nodes", violation         */
    /*      --- if last node != tail, violation                      */
    /* start from tail and go until                                 */
    /*  (1) null, or                                                */
```

```
/*  (2) check "number of nodes"                                  */
/*  --- if reached end too soon or didn't reach end              */
/*          after checking "number of nodes", violation          */
/*      --- if last node != head, violation                      */

unsigned long ulNumNodes;
unsigned long ulCounter;
DLLNodeType*  pCurrentNode;
DLLNodeType*  pLastNode;
int       Violation1 = 0;
int       Violation2 = 0;
int       Violation3 = 0;
int       Violation4 = 0;
int       Violation5 = 0;
int       Violation6 = 0;

ulNumNodes = pDLL->ulCount;

ulCounter = 0;
pCurrentNode = pDLL->pHead;
pLastNode = NULL;
while( (ulCounter<ulNumNodes) &&
       (pCurrentNode!=NULL)       )
{
    ulCounter++;
    pLastNode = pCurrentNode;
    pCurrentNode = pCurrentNode->pNextNode;
}

if( ulCounter != pDLL->ulCount )
    Violation1 = 1;

if( pLastNode != pDLL->pTail )
    Violation2 = 1;

if( pCurrentNode != NULL )
    Violation3 = 1;

ulCounter = 0;
pCurrentNode = pDLL->pTail;
pLastNode = NULL;
while( (ulCounter<ulNumNodes) &&
```

```
        (pCurrentNode!=NULL)        )
    {
        ulCounter++;
        pLastNode = pCurrentNode;
        pCurrentNode = pCurrentNode->pPrevNode;
    }

    if( ulCounter != pDLL->ulCount )
        Violation4 = 1;

    if( pLastNode != pDLL->pHead )
        Violation5 = 1;

    if( pCurrentNode != NULL )
        Violation6 = 1;

    if( Violation1 ||
        Violation2 ||
        Violation3 ||
        Violation4 ||
        Violation5 ||
        Violation6     )
    {
        return (0);
    }

    else
    {
        return (1);
    }

}

/*****************************************************************************/
/**                                                                         **/
/**  Call this function to determine whether the DLL Pointer is valid       **/
/**  or not.  The DLL Pointer is invalid under two conditions:  (1) at      **/
/**  startup (because the pointer has not yet been defined), and (2)        **/
/**  when the node pointed to by the DLL Pointer is removed.                **/
/**                                                                         **/
/**  Returns:  0 if DLL Pointer invalid                                     **/
/**            1 if DLL Pointer valid                                       **/
```

```
/**                                                                    **/
/************************************************************************/
int DLLIsPtrValid( DLLType* pDLL )
{
    if( pDLL->pPtr == NULL )
        return (0);
    else
        return (1);
}

/************************************************************************/
/**                                                                    **/
/**  Call this function to perform a very simple dump of the linked    **/
/**  list.  The ASCII value of the first byte of the data pointed to   **/
/**  by each node is printed to the screen, starting from the head and **/
/**  ending at the tail.                                               **/
/**                                                                    **/
/**  Returns:  no return value                                         **/
/**                                                                    **/
/************************************************************************/
/**                                                                    **/
/************************************************************************/
void DLLSimpleDump( FILE* pfileOut, DLLType* pDLL )
{
    void* pData;
    DLLNodeType* pOldPtr;

    /* remember where point used to be */
    pOldPtr = DLLGetPtrToPtr( pDLL );

    /* move pointer to head */
    DLLMovePtrToHead( pDLL );

    /* continue going through list until end */
    while( (pData=DLLPeekNextPtr(pDLL)) != NULL )
        fprintf( pfileOut, "%c ", *((unsigned char*)pData) );

    /* move pointer back to original spot */
    DLLSetPtrToPtr( pDLL, pOldPtr );
}


/* DLLIST.H */
```

```
#define _DLList_h_

struct structDLLNodeType
{
    void*                       pData;
    struct structDLLNodeType* pNextNode;
    struct structDLLNodeType* pPrevNode;
};
typedef struct structDLLNodeType DLLNodeType;

struct structDLLType
{
    DLLNodeType*  pHead;
    DLLNodeType*  pTail;
    unsigned long ulCount;
    int           nID;
    DLLNodeType*  pPtr;
};
typedef struct structDLLType DLLType;

DLLType* DLLCreate();
void DLLDestroy( DLLType* pDLL );

int DLLInsertAtHead( DLLType* pDLL, void* pData );
int DLLInsertAtTail( DLLType* pDLL, void* pData );
void* DLLRemoveFromHead( DLLType* pDLL );
void* DLLRemoveFromTail( DLLType* pDLL );
void* DLLRemove( DLLType* pDLL, DLLNodeType* pDLLNode );
void* DLLPeek( DLLType* pDLL, DLLNodeType* pDLLNode );
void* DLLPeekAtHead( DLLType* pDLL );
void* DLLPeekAtTail( DLLType* pDLL );
int DLLInsertBefore( DLLType* pDLL, DLLNodeType* pDLLNode, void* pData );
int DLLInsertAfter( DLLType* pDLL, DLLNodeType* pDLLNode, void* pData );

void DLLMovePtrToHead( DLLType* pDLL );
void DLLMovePtrToTail( DLLType* pDLL );
void DLLSetPtrToPtr( DLLType* pDLL, DLLNodeType* pDLLNode );
void* DLLRemoveAtPtr( DLLType* pDLL );
void* DLLPeekAtPtr( DLLType* pDLL );
void* DLLPeekNextPtr( DLLType* pDLL );
int DLLAdvancePtr( DLLType* pDLL );
int DLLRetreatPtr( DLLType* pDLL );
```

```
int DLLInsertBeforePtr( DLLType* pDLL, void* pData );
int DLLInsertAfterPtr( DLLType* pDLL, void* pData );
DLLNodeType* DLLGetPtrToPtr( DLLType* pDLL );

int DLLIsNodeInDLL( DLLType* pDLL, DLLNodeType* pDLLNode );
unsigned long DLLCount( DLLType* pDLL );
int DLLVerifyIntegrity( DLLType* pDLL );
int DLLIsPtrValid( DLLType* pDLL );

/*int DLLSwapNodes( DLLType* pDLL, DLLNodeType* pDLLNode1, DLLNodeType* pDLLNode2 );*/
/*void DLLSimpleDump( FILE* pfileOut, DLLType* pDLL );*/


/* RS.C */

#include <stdio.h>
#include "rs.h"

#define MM  8       /* RS code over GF(2**MM) i.e., we use Bytes GF(256) */
#define NN 255      // Fixed since MM=8 -> NN=2^8-1
                    // Max length of block including padding

/* This defines the type used to store an element of the Galois Field
 * used by the code. Make sure this is something larger than a char if
 * if anything larger than GF(256) is used.
 *
 * Note: unsigned char will work up to GF(256) but int seems to run
 * faster on the Pentium.
 */
typedef unsigned char gf;

//Global variables
int KK;  // number of info symbols including padding
int n;   // length of block
int k;   // number of info bytes before padding

/* Primitive polynomial */
/* 1+x^2+x^3+x^4+x^8 */
int Pp[MM+1] = { 1, 0, 1, 1, 1, 0, 0, 0, 1 };

/* Alpha exponent for the first root of the generator polynomial */
#define B0  1
```

```
/* index->polynomial form conversion table */
gf Alpha_to[NN + 1];

/* Polynomial->index form conversion table */
gf Index_of[NN + 1];

/* No legal value in index form represents zero, so
 * we need a special value for this purpose
 */
#define A0   (NN)

/* Generator polynomial g(x)
 * Degree of g(x) = 2*TT
 * has roots @**B0, @**(B0+1), ... ,@^(B0+2*TT-1)
 */
//gf Gg[NN - KK + 1];
gf Gg[NN]; // we don't  know KK in advance, hence assume KK=1

/* Compute x % NN, where NN is 2**MM - 1,
 * without a slow divide
 */
/* static inline */
static __inline gf
modnn(int x)
{
    while (x >= NN) {
        x -= NN;
        x = (x >> MM) + (x & NN);
    }
    return x;
}

#define min(a,b)     ((a) < (b) ? (a) : (b))

#define CLEAR(a,n) {\
    int ci;\
    for(ci=(n)-1;ci >=0;ci--)\
        (a)[ci] = 0;\
    }

#define COPY(a,b,n) {\
```

```
    int ci;\
    for(ci=(n)-1;ci >=0;ci--)\
        (a)[ci] = (b)[ci];\
    }
#define COPYDOWN(a,b,n) {\
    int ci;\
    for(ci=(n)-1;ci >=0;ci--)\
        (a)[ci] = (b)[ci];\
    }

/* These two functions *must* be called in this order (e.g.,
 * by init_rs()) before any encoding/decoding
 */

void generate_gf(void); /* Generate Galois Field */
void gen_poly(void);    /* Generate generator polynomial */

void init_rs(int n_init, int k_init)
{
    n=n_init; k=k_init;
    KK = NN-(n-k);
    generate_gf();
    gen_poly();
}

/* generate GF(2**m) from the irreducible polynomial p(X) in p[0]..p[m]
   lookup tables:  index->polynomial form   alpha_to[] contains j=alpha**i;
                   polynomial form -> index form  index_of[j=alpha**i] = i
   alpha=2 is the primitive element of GF(2**m)
   HARI's COMMENT: (4/13/94) alpha_to[] can be used as follows:
        Let @ represent the primitive element commonly called "alpha" that
   is the root of the primitive polynomial p(x). Then in GF(2^m), for any
   0 <= i <= 2^m-2,
        @^i = a(0) + a(1) @ + a(2) @^2 + ... + a(m-1) @^(m-1)
   where the binary vector (a(0),a(1),a(2),...,a(m-1)) is the representation
   of the integer "alpha_to[i]" with a(0) being the LSB and a(m-1) the MSB. Thus for
   example the polynomial representation of @^5 would be given by the binary
   representation of the integer "alpha_to[5]".
                   Similarily, index_of[] can be used as follows:
        As above, let @ represent the primitive element of GF(2^m) that is
   the root of the primitive polynomial p(x). In order to find the power
   of @ (alpha) that has the polynomial representation
```

```
        a(0) + a(1) @ + a(2) @^2 + ... + a(m-1) @^(m-1)
    we consider the integer "i" whose binary representation with a(0) being LSB
    and a(m-1) MSB is (a(0),a(1),...,a(m-1)) and locate the entry
    "index_of[i]". Now, @^index_of[i] is that element whose polynomial
     representation is (a(0),a(1),a(2),...,a(m-1)).
    NOTE:
        The element alpha_to[2^m-1] = 0 always signifying that the
    representation of "@^infinity" = 0 is (0,0,0,...,0).
        Similarily, the element index_of[0] = A0 always signifying
    that the power of alpha which has the polynomial representation
    (0,0,...,0) is "infinity".

*/

void
generate_gf(void)
{
    register int i, mask;

    mask = 1;
    Alpha_to[MM] = 0;
    for (i = 0; i < MM; i++) {
        Alpha_to[i] = mask;
        Index_of[Alpha_to[i]] = i;
        /* If Pp[i] == 1 then, term @^i occurs in poly-repr of @^MM */
        if (Pp[i] != 0)
            Alpha_to[MM] ^= mask;   /* Bit-wise EXOR operation */
        mask <<= 1; /* single left-shift */
    }
    Index_of[Alpha_to[MM]] = MM;
    /*
     * Have obtained poly-repr of @^MM. Poly-repr of @^(i+1) is given by
     * poly-repr of @^i shifted left one-bit and accounting for any @^MM
     * term that may occur when poly-repr of @^i is shifted.
     */
    mask >>= 1;
    for (i = MM + 1; i < NN; i++) {
        if (Alpha_to[i - 1] >= mask)
            Alpha_to[i] = Alpha_to[MM] ^ ((Alpha_to[i - 1] ^ mask) << 1);
        else
            Alpha_to[i] = Alpha_to[i - 1] << 1;
        Index_of[Alpha_to[i]] = i;
```

```
        }
        Index_of[0] = A0;
        Alpha_to[NN] = 0;
    }

    /*
     * Obtain the generator polynomial of the TT-error correcting, length
     * NN=(2**MM -1) Reed Solomon code from the product of (X+@**(B0+i)), i = 0,
     * ... ,(2*TT-1)
     *
     * Examples:
     *
     * If B0 = 1, TT = 1. deg(g(x)) = 2*TT = 2.
     * g(x) = (x+@) (x+@**2)
     *
     * If B0 = 0, TT = 2. deg(g(x)) = 2*TT = 4.
     * g(x) = (x+1) (x+@) (x+@**2) (x+@**3)
     */
    void
    gen_poly(void)
    {
        register int i, j;

        Gg[0] = Alpha_to[B0];
        Gg[1] = 1;      /* g(x) = (X+@**B0) initially */
        for (i = 2; i <= NN - KK; i++) {
            Gg[i] = 1;
            /*
             * Below multiply (Gg[0]+Gg[1]*x + ... +Gg[i]x^i) by
             * (@**(B0+i-1) + x)
             */
            for (j = i - 1; j > 0; j--)
                if (Gg[j] != 0)
                    Gg[j] = Gg[j - 1] ^ Alpha_to[modnn((Index_of[Gg[j]]) + B0 + i - 1)];
                else
                    Gg[j] = Gg[j - 1];
            /* Gg[0] can never be zero */
            Gg[0] = Alpha_to[modnn((Index_of[Gg[0]]) + B0 + i - 1)];
        }
        /* convert Gg[] to index form for quicker encoding */
        for (i = 0; i <= NN - KK; i++)
            Gg[i] = Index_of[Gg[i]];
```

```
    }

    /*
     * take the string of symbols in data[i], i=0..(k-1) and encode
     * systematically to produce NN-KK parity symbols in bb[0]..bb[NN-KK-1] data[]
     * is input and bb[] is output in polynomial form. Encoding is done by using
     * a feedback shift register with appropriate connections specified by the
     * elements of Gg[], which was generated above. Codeword is   c(X) =
     * data(X)*X**(NN-KK)+ b(X)
     */
    int encode_rs(unsigned char data[], unsigned char bb[])
    {
        register int i, j;
        gf feedback;

        CLEAR(bb,NN-KK);
        for (i = k - 1; i >= 0; i--) {
#if (MM != 8)
            if(data[i] > NN)
                return -1;  /* Illegal symbol */
#endif
            feedback = Index_of[data[i] ^ bb[NN - KK - 1]];
            if (feedback != A0) {   /* feedback term is non-zero */
                for (j = NN - KK - 1; j > 0; j--)
                    if (Gg[j] != A0)
                        bb[j] = bb[j - 1] ^ Alpha_to[modnn(Gg[j] + feedback)];
                    else
                        bb[j] = bb[j - 1];
                bb[0] = Alpha_to[modnn(Gg[0] + feedback)];
            } else {    /* feedback term is zero. encoder becomes a
                     * single-byte shifter */
                for (j = NN - KK - 1; j > 0; j--)
                    bb[j] = bb[j - 1];
                bb[0] = 0;
            }
        }

        return 0;
    }

    /*
```

```
 * Performs ERRORS+ERASURES decoding of RS codes. If decoding is successful,
 * writes the codeword into data[] itself. Otherwise data[] is unaltered.
 *
 * Return number of symbols corrected, or -1 if codeword is illegal
 * or uncorrectable.
 *
 * First "no_eras" erasures are declared by the calling program. Then, the
 * maximum # of errors correctable is t_after_eras = floor((NN-KK-no_eras)/2).
 * If the number of channel errors is not greater than "t_after_eras" the
 * transmitted codeword will be recovered. Details of algorithm can be found
 * in R. Blahut's "Theory ... of Error-Correcting Codes".
 */
int eras_dec_rs(unsigned char data[], int eras_pos[], int no_eras)
{
    int deg_lambda, el, deg_omega;
    int i, j, r;
    gf u,q,tmp,num1,num2,den,discr_r;
    gf recd[NN];
    gf lambda[NN], s[NN];   /* Err+Eras Locator poly
                             * and syndrome poly */
    gf b[NN], t[NN], omega[NN];
    gf root[NN-1], reg[NN], loc[NN-1];
    int syn_error, count;

    // move the data arry into the padded form
    int delta = KK-k;
    for (i=k; i<n;i++)
    {
        data[i+delta]=data[i];
        data[i]=0;
    }
    for(i=n;i<KK;i++)
    {
        data[i]=0;
    }
    for(i=0;i<no_eras;i++)
    {
        if(eras_pos[i]>=k) eras_pos[i]+=delta;
    }

    /* data[] is in polynomial form, copy and convert to index form */
    for (i = NN-1; i >= 0; i--){
```

```
#if (MM != 8)
        if(data[i] > NN)
            return -1;  /* Illegal symbol */
#endif
        recd[i] = Index_of[data[i]];
    }
    /* first form the syndromes; i.e., evaluate recd(x) at roots of g(x)
     * namely @**(B0+i), i = 0, ... ,(NN-KK-1)
     */
    syn_error = 0;
    for (i = 1; i <= NN-KK; i++) {
        tmp = 0;
        for (j = 0; j < NN; j++)
            if (recd[j] != A0)  /* recd[j] in index form */
                tmp ^= Alpha_to[modnn(recd[j] + (B0+i-1)*j)];
        syn_error |= tmp;   /* set flag if non-zero syndrome =>
                             * error */
        /* store syndrome in index form  */
        s[i] = Index_of[tmp];
    }
    if (!syn_error) {
        /*
         * if syndrome is zero, data[] is a codeword and there are no
         * errors to correct. So return data[] unmodified
         */

        // move the data arry into the unpadded form
        for (i=k; i<n;i++)
        {
            data[i]=data[i+delta];
            data[i+delta]=0;
        }
        for(i=0;i<no_eras;i++)
        {
            if(eras_pos[i]>=KK) eras_pos[i]-=delta;
        }


        return 0;
    }
    CLEAR(&lambda[1],NN-KK);
    lambda[0] = 1;
```

```
    if (no_eras > 0) {
        /* Init lambda to be the erasure locator polynomial */
        lambda[1] = Alpha_to[eras_pos[0]];
        for (i = 1; i < no_eras; i++) {
            u = eras_pos[i];
            for (j = i+1; j > 0; j--) {
                tmp = Index_of[lambda[j - 1]];
                if(tmp != A0)
                    lambda[j] ^= Alpha_to[modnn(u + tmp)];
            }
        }
#ifdef ERASURE_DEBUG
        /* find roots of the erasure location polynomial */
        for(i=1;i<=no_eras;i++)
            reg[i] = Index_of[lambda[i]];
        count = 0;
        for (i = 1; i <= NN; i++) {
            q = 1;
            for (j = 1; j <= no_eras; j++)
                if (reg[j] != A0) {
                    reg[j] = modnn(reg[j] + j);
                    q ^= Alpha_to[reg[j]];
                }
            if (!q) {
                /* store root and error location
                 * number indices
                 */
                root[count] = i;
                loc[count] = NN - i;
                count++;
            }
        }
        if (count != no_eras) {
            printf("\n lambda(x) is WRONG\n");
            return -1;
        }
#ifndef NO_PRINT
        printf("\n Erasure positions as determined by roots of Eras Loc Poly:\n");
        for (i = 0; i < count; i++)
            printf("%d ", loc[i]);
        printf("\n");
#endif
```

```
#endif
    }
    for(i=0;i<NN-KK+1;i++)
        b[i] = Index_of[lambda[i]];

    /*
     * Begin Berlekamp-Massey algorithm to determine error+erasure
     * locator polynomial
     */
    r = no_eras;
    el = no_eras;
    while (++r <= NN-KK) {  /* r is the step number */
        /* Compute discrepancy at the r-th step in poly-form */
        discr_r = 0;
        for (i = 0; i < r; i++){
            if ((lambda[i] != 0) && (s[r - i] != A0)) {
                discr_r ^= Alpha_to[modnn(Index_of[lambda[i]] + s[r - i])];
            }
        }
        discr_r = Index_of[discr_r];     /* Index form */
        if (discr_r == A0) {
            /* 2 lines below: B(x) <-- x*B(x) */
            COPYDOWN(&b[1],b,NN-KK);
            b[0] = A0;
        } else {
            /* 7 lines below: T(x) <-- lambda(x) - discr_r*x*b(x) */
            t[0] = lambda[0];
            for (i = 0 ; i < NN-KK; i++) {
                if(b[i] != A0)
                    t[i+1] = lambda[i+1] ^ Alpha_to[modnn(discr_r + b[i])];
                else
                    t[i+1] = lambda[i+1];
            }
            if (2 * el <= r + no_eras - 1) {
                el = r + no_eras - el;
                /*
                 * 2 lines below: B(x) <-- inv(discr_r) *
                 * lambda(x)
                 */
                for (i = 0; i <= NN-KK; i++)
                    b[i] = (lambda[i] == 0) ? A0 : modnn(Index_of[lambda[i]] - discr_r + NN);
            } else {
```

```
            /* 2 lines below: B(x) <-- x*B(x) */
            COPYDOWN(&b[1],b,NN-KK);
            b[0] = A0;
        }
        COPY(lambda,t,NN-KK+1);
    }
}

/* Convert lambda to index form and compute deg(lambda(x)) */
deg_lambda = 0;
for(i=0;i<NN-KK+1;i++){
    lambda[i] = Index_of[lambda[i]];
    if(lambda[i] != A0)
        deg_lambda = i;
}
/*
 * Find roots of the error+erasure locator polynomial. By Chien
 * Search
 */
    COPY(&reg[1],&lambda[1],NN-KK);
    count = 0;      /* Number of roots of lambda(x) */
    for (i = 1; i <= NN; i++) {
        q = 1;
        for (j = deg_lambda; j > 0; j--)
            if (reg[j] != A0) {
                reg[j] = modnn(reg[j] + j);
                q ^= Alpha_to[reg[j]];
            }
        if (!q) {
            /* store root (index-form) and error location number */
            root[count] = i;
            loc[count] = NN - i;
            count++;
        }
    }

#ifdef DEBUG
    printf("\n Final error positions:\t");
    for (i = 0; i < count; i++)
        printf("%d ", loc[i]);
    printf("\n");
#endif
```

```
if (deg_lambda != count) {
    /*
     * deg(lambda) unequal to number of roots => uncorrectable
     * error detected
     */
    return -1;
}
/*
 * Compute err+eras evaluator poly omega(x) = s(x)*lambda(x) (modulo
 * x**(NN-KK)). in index form. Also find deg(omega).
 */
deg_omega = 0;
for (i = 0; i < NN-KK;i++){
    tmp = 0;
    j = (deg_lambda < i) ? deg_lambda : i;
    for(;j >= 0; j--){
        if ((s[i + 1 - j] != A0) && (lambda[j] != A0))
            tmp ^= Alpha_to[modnn(s[i + 1 - j] + lambda[j])];
    }
    if(tmp != 0)
        deg_omega = i;
    omega[i] = Index_of[tmp];
}
omega[NN-KK] = A0;

/*
 * Compute error values in poly-form. num1 = omega(inv(X(l))), num2 =
 * inv(X(l))**(B0-1) and den = lambda_pr(inv(X(l))) all in poly-form
 */
for (j = count-1; j >=0; j--) {
    num1 = 0;
    for (i = deg_omega; i >= 0; i--) {
        if (omega[i] != A0)
            num1  ^= Alpha_to[modnn(omega[i] + i * root[j])];
    }
    num2 = Alpha_to[modnn(root[j] * (B0 - 1) + NN)];
    den = 0;

    /* lambda[i+1] for i even is the formal derivative lambda_pr of lambda[i] */
    for (i = min(deg_lambda,NN-KK-1) & ~1; i >= 0; i -=2) {
        if(lambda[i+1] != A0)
            den ^= Alpha_to[modnn(lambda[i+1] + i * root[j])];
```

```
        }
        if (den == 0) {
#ifdef DEBUG
            printf("\n ERROR: denominator = 0\n");
#endif
            return -1;
        }
        /* Apply error to data */
        if (num1 != 0) {
            data[loc[j]] ^= Alpha_to[modnn(Index_of[num1] + Index_of[num2] + NN - Index_of[den])];
        }
    }

    // move the data back into the unpadded form
    for (i=k; i<n;i++)
        {
            data[i]=data[i+delta];
            data[i+delta]=0;
        }
        for(i=0;i<no_eras;i++)
        {
            if(eras_pos[i]>=KK) eras_pos[i]-=delta;
        }

    return count;
}

// this is my stuff to test the above routines
/*
void main()
{
    int data[NN]={1,2,3,4,5,6,7,8,9};
    int i, corrected,j;
    int eras_pos[NN];
    int n = 50; k=40;

    for (i=9;i<NN;i++) data[i]=0;

    init_rs(n,k);
    encode_rs(&data[0],&data[k]);

    j=0;
```

```
    for (i=0;i<4;i++)
    {
        data[i]=0;
        eras_pos[j++]=i;
    }
    for (i=n-5;i<n;i++)
    {
        data[i]=0;
        eras_pos[j++]=i;
    }
    corrected = eras_dec_rs(data, eras_pos,9);
}
*/

/* RS.H */

#ifndef RS
#define RS

/* Initialization function, this function needs to be called before any encoding
or decoding */
/* the resulting RS code is (n,k) over GF(2^8) */
/* where n is the length of the block (0..n-1) and */
/* k is the number of data bytes used (k<n)  */
/* note that the number of parity bytes is n-k */
/* therefore n-k erasures can be fixed */
/* Example init_rs(18,9) */
void init_rs(int n, int k);

/* Reed-Solomon encoding*/
/* data[] is the input block, parity symbols are placed in bb[]*/
/*  encode_rs(&data[0],&data[223]);*/
/* Note that the data array has to be of dimension 255*/
/* int data[255]*/
/* even when shortened (n<255) RS codes are used*/
/* Example encode_rs(&data[0],&data[9]) */

int encode_rs(unsigned char data[], unsigned char bb[]);

/* Reed-Solomon erasures-and-errors decoding*/
/* The received block goes into data[], and a list of zero-origin*/
```

```
/* erasure positions, if any, goes in eras_pos[] with a count in no_eras.*/
/* */
/* The decoder corrects the symbols in place, if possible and returns*/
/* the number of corrected symbols. If the codeword is illegal or*/
/* uncorrectible, the data array is unchanged and -1 is returned*/

int eras_dec_rs(unsigned char data[], int eras_pos[], int no_eras);

#endif
```

/* RS_CODER.CPP */

```
/*****************************************************************************/
/*                                                                           */
/* rs_coder.c                                                                */
/*                                                                           */
/* 10/7/97                                                                   */
/*                                                                           */
/* Guido Schuster                                                            */
/* Advanced Technologies                                                     */
/* Carrier Systems Division                                                  */
/* 3COM                                                                      */
/* 1800 W. Central Rd.                                                       */
/* Mount Prospect, Il  60056                                                 */
/*                                                                           */
/* This module implements a reed solomon (RS) based channel coder for packet streams.       */
/* Before the the main function rs_coder() can be called, the initialization function       */
/* init_rs_coder(int n, int k) needs to be called the parameters are the (n,k) of the       */
/* resulting rs block code. I.e., n is the block size and k is the number of data bytes per */
/* n block bytes. The main function rs_coder(MESSAGEBOX* pInBox, MESSAGEBOX* pOutBox), uses */
/* an in-box and an out-box (see RTOS) to receive packets and to return packets.  The       */
/* incoming packet stream should be put into the in-box, the coder then calculates the      */
/* parity information and piggy-backs that onto the incoming packet to create the outgoing  */
/* packet stream.  This program is intended to work hand in hand with "rs_decoder.c"        */
/*****************************************************************************/


/********************************** The Include Files ************************************/
#include "stdlib.h" // the standard files
#include "malloc.h"
#include "string.h"
```

```
#include "windows.h"

#include "types.h"  // this is where the common types are defined
#include "RTOS.h"   // the implementation of the messageboxes
#include "rs_coder.h"   // the header file

// linking of these files uses the old c-style
extern "C"
{
#include "rs.h" // the definition of the rs-engine
#include "dllist.h" // an all purpose doubly linked list
}
/*****************************************************************************/


/************************** The Global Variables For This File ***************************/
static int n_old=0; // the n from the rs (n,k) block code is stored here
static int k_old=0; // the k of the (n,k) code.
/*****************************************************************************/


/***************************** Initialization Routine ************************************/
/* init_rs_coder needs to be called before the rs_coder is used as expected, it takes the   */
/* (n,k) parameters of the rs block code as input it then sets the global variables n_old   */
/* and k_old to n and k and also initializes the rs engine                                  */
void init_rs_coder(int n, int k)
{
    init_rs(n, k);
    n_old=n; k_old=k;
}
/*****************************************************************************/


/********************************** The Main Routine *************************************/
/* rs_coder takes as its two parameters the names of the inbox and outbox when it is called,*/
/* it expects a packet in its inbox. It will then generate an outgoing packet, adding a     */
/* header and a trailer to the inpacket. The header contains the sequence number, the block */
/* code used, and the lenght of the inpacket. The trailer contains the length of the added  */
/* redundancy and the redundancy.  The redundancy is calculated using a rs_engine (rs.c).   */
/* The previous packets, which are required to calculate the redundancy are stored in a     */
```

```
/* doubly linked list (dll) and so are the parity packets.                    */
void rs_coder(MESSAGEBOX* pInBox, MESSAGEBOX* pOutBox)
{
    /* The Static Variables */
    static int i_p=n_old-k_old; // the parity packet counter (0..n_old-k_old-1),
                                // points to the parity line packetized next
                                // initilizd to point behind the last parity packet
    static unsigned char SeqNum=0; // The sequence number starting at 0 (0,..,255,0,..,255,..)
    static DLLType* pDataList=DLLCreate();      // Store the incoming data packets
    static DLLType* pParityList=DLLCreate();    // Store the generated parity bytes (Not packets)
    static int nMaxBytes=0; // the max length of the data packets in a block,
                            // this is changed for every block

    /* All Purpose Variables */
    int i;      // general counter
    int nByte;  // counts through the bytes in a packet
    unsigned char *pParity; // temp pointer to a party byte array
    PKT* pPacket;           // temp pointer to a packet, mostly used for dll access

    /* The Code */
    //get the packet
    PKT* pInPacket=(PKT*)GetMessageFromBox(pInBox);
    if (!pInPacket) return; // no packet, no processing

    // create the outgoing data packet
    PKT* pOutPacket = new PKT;
    // 7 Bytes are added 1. SeqNum, 2.n, 3.k, 4&5.  Data Length
    // and after the data, 2 Bytes are added for the Parity Length
    if(i_p<n_old-k_old) // parity bytes available ?
    {   // yes
        pOutPacket->nLengthBytes=5+pInPacket->nLengthBytes+2+nMaxBytes;
    }
    else
    {   // no
        pOutPacket->nLengthBytes=5+pInPacket->nLengthBytes+2;
    }
    pOutPacket->pData=malloc(pOutPacket->nLengthBytes * sizeof(unsigned char));
    // write SeqNum
    *((unsigned char*)pOutPacket->pData)=SeqNum;
    // also add the SeqNum into the data structures
    pOutPacket->SeqNum=SeqNum;
    pInPacket->SeqNum=SeqNum;
```

```
SeqNum++;
// write n_old,k_old
*(((unsigned char*)pOutPacket->pData)+1)=n_old;
*(((unsigned char*)pOutPacket->pData)+2)=k_old;
// write data Length
*((WORD*)(((unsigned char*)pOutPacket->pData)+3))=(WORD) pInPacket->nLengthBytes;
// copy the incomming data to the outgoing packet
memmove(((unsigned char*)pOutPacket->pData)+5, pInPacket->pData,pInPacket->nLengthBytes );
// if the parity data is available, add that to the outgoing packet
if(i_p<n_old-k_old)
{
    // write parity length
    *((WORD*)(((unsigned char*)pOutPacket->pData)+5+pInPacket->nLengthBytes))=
        (WORD) nMaxBytes;

    // copy parity
    DLLMovePtrToHead(pParityList);
    for(nByte=0; nByte<nMaxBytes; nByte++)
    {
        pParity=(unsigned char*)DLLPeekNextPtr(pParityList);
        *(((unsigned char*)pOutPacket->pData)+5+pInPacket->nLengthBytes+2+nByte)=
            *(pParity+i_p);
    }
    i_p++;

    // the last parity bytes have been read, we free the memory
    if(i_p==n_old-k_old)
    {
        for( i=0;i<nMaxBytes;i++)
        {
            pParity=(unsigned char*)DLLRemoveFromHead(pParityList);
            free(pParity);
        }
    }

}
else
{
    // write parity length of zero
    *((WORD*)(((unsigned char*)pOutPacket->pData)+5+pInPacket->nLengthBytes))=
        (WORD) 0;
}
```

```
// we can now put the data into the out box
AddMessageToBox(pOutBox,(void*) pOutPacket);

// put packet pointer into LL
DLLInsertAtTail(pDataList,(void*)pInPacket);

// do we have k data packets to run the (n,k) RS coder?
if (DLLCount(pDataList)== (unsigned long int)k_old)
{
    // reset the parity counter
    i_p=0;

    // used to pass the data to the RS encoder
    unsigned char *pData=(unsigned char*) malloc(k_old);

    // do the two data legth bytes first
    for(nByte=0; nByte<2; nByte++)
    {
        // allocate the space for the parity
        pParity=(unsigned char*)malloc(n_old-k_old);

        // the inner loop going through the data packets
        DLLMovePtrToHead(pDataList);
        for (i=0; i<k_old; i++)
        {
            pPacket=(PKT*)DLLPeekNextPtr(pDataList);
            *(pData+i)=
                *(((unsigned char*)(&(pPacket->nLengthBytes)))+nByte);
        }

        // call the rs encoder, this creates the Parity
        encode_rs(pData,pParity);
        // Enter it into the parity DLL
        DLLInsertAtTail(pParityList,(void*)pParity);

}

// now do the data bytes
// First we need to find out nMaxBytes
nMaxBytes=0;
DLLMovePtrToHead(pDataList);
```

```
for (i=0; i<k_old; i++)
{
    pPacket=(PKT*)DLLPeekNextPtr(pDataList);
    if(pPacket->nLengthBytes>nMaxBytes)
    {
        nMaxBytes=pPacket->nLengthBytes;
    }
}
// add the two data length bytes
nMaxBytes+=2;

// Now, go through the data bytes (i.e., 2 bytes less than nMaxBytes)
for(nByte=0; nByte<nMaxBytes-2; nByte++)
{
    // the inner loop going through the packets
    DLLMovePtrToHead(pDataList);
    for (i=0; i<k_old; i++)
    {
        pPacket=(PKT*)DLLPeekNextPtr(pDataList);
        // is zero padding required?
        if (pPacket->nLengthBytes<=nByte)
        { // yes
            *(pData+i)=0;
        }
        else
        { // no
            *(pData+i)=*((unsigned char*)(pPacket->pData)+nByte);
        }
    }

    // allocate the space for the parity
    pParity=(unsigned char*)malloc(n_old-k_old);

    // call the rs encoder, this creates the Parity
    encode_rs(pData,pParity);
    // Enter it into the parity DLL
    DLLInsertAtTail(pParityList,(void*)pParity);

}

// free the memory used for the passing the data
free(pData);
```

```
        // erase the DataList
        for( i=0;i<k_old;i++)
        {
            pPacket=(PKT*)DLLRemoveFromHead(pDataList);
            free(pPacket->pData);
            delete(pPacket);
        }

    }

}
/*****************************************************************************/


/* RS_CODER.H */

#ifndef RS_CODER
#define RS_CODER

#include "types.h"

void init_rs_coder(int n, int k);

void rs_coder(MESSAGEBOX* pInBox, MESSAGEBOX* pOutBox);

#endif


/* RS_DECODER.CPP */

/*****************************************************************************************/
/*                                                                                       */
/* rs_decoder.c                                                                          */
/*                                                                                       */
/* 10/8/97                                                                               */
/*                                                                                       */
/* Guido Schuster                                                                        */
/* Advanced Technologies                                                                 */
/* Carrier Systems Division                                                              */
/* 3COM                                                                                  */
/* 1800 W. Central Rd.                                                                   */
```

```
/* Mount Prospect, Il  60056                                                    */
/*                                                                              */
/* This module implements a reed solomon (RS) based channel decoder for packet streams. This*/
/* program is intended to work hand in hand with "rs_encoder.c". Before the the main        */
/* function rs_decoder() can be called, the initialization function                         */
/* init_rs_decoder(int n, int k, int BufferLength) needs to be called. The parameters are   */
/* the (n,k) of the rs block code used in the encoder. I.e., n is the block size and k is   */
/* the number of data bytes per n block bytes. The BufferLength is in bytes and represents  */
/* the target number of bytes the decoder should use to store previous packets. If this is  */
/* too small, then there are not enough packets in memory to run the rs decoder sucessfully.*/
/* Too large of a value does not make much sense either since packets recovered would arrive*/
/* very late. The main function rs_decoder(MESSAGEBOX* pInBox, MESSAGEBOX* pOutBox), uses   */
/* an in-box and an out-box (see RTOS) to receive packets and to return packets.            */
/* The basic operation is as follows. A packet is read from the inbox. The original payload */
/* is recovered and put into the outbox.  Then the data and the redundancy are put into     */
/* a buffer structure and when k packets of a block are available and some data in this     */
/* block is missing, the rs_decode engine is run to recover the missing data.  This         */
/* recovered data is then put into the outbox.                                              */
/****************************************************************************************/


/****************************** The Include Files ***************************************/
#include "stdlib.h"     // the standard include files
#include "stdio.h"
#include "malloc.h"
#include "string.h"
#include "windows.h"
#include "math.h"

#include "types.h"  // the commonly used types
#include "RTOS.h"   // the messageboxes
#include "rs_decoder.h" // the declaration
#include "tools.h"  // useful routines

extern "C" // link these files using the "C" linkage
{
#include "rs.h" // the rs engine
#include "dllist.h" // the implementation of the doubly linked list
}
/****************************************************************************************/
```

```
/****************************** The Global Variables ***************************************/
static unsigned int n_old=0, k_old=0; // the(n,k) of the rs code
static int BufferLength_old=0; // the depth of the buffer data structure in bytes
static DLLType *pBlockList; // the buffer data structure
static unsigned __int32 PrevSeqNum=0; // the extended sequence number of the previously received packet
MESSAGEBOX *pOutBox; // declared a global variable, instead of passing it through 3 functions
/*******************************************************************************************/


/**************** Types And Their Printing Routines Used In This File ***********************/
struct structExtPacket_t // the extended packet is a packet with a 32 bit sequence number
{
    unsigned __int32 ExtSeqNum;
    PKT* pPacket;
};
typedef struct structExtPacket_t ExtPacket_t;

// a simple printing routine for the extended packet
void PrintExtPacket(ExtPacket_t *pExtPacket)
{
    printf("--- ExtPacket ---\n");
    if (pExtPacket==NULL)
    {
        printf("NULL pointer \n");
    }
    else
    {
        printf("ExtSeqNum: %d\n",pExtPacket->ExtSeqNum);
        PrintPacket(pExtPacket->pPacket);
    }
}

struct structBlockListData_t // the node type for the block dll
{
    unsigned char NumOfPackets; // number of packets in this block
    unsigned __int32 StartExtSeqNum; // extendeded sequence number of the first packet in block
    DLLType *pBlock; // pointer to the block, which is a dll containing extended packets
};
typedef struct structBlockListData_t BlockListData_t;

// a simple printing routine for a block
void PrintBlockListData(BlockListData_t *pBlockListData)
```

```
{
    printf("*** BlockListData ***\n");
    if (pBlockListData==NULL)
    {
        printf("NULL pointer \n");
    }
    else
    {
        printf("StartExtSeqNum: %d\n",pBlockListData->StartExtSeqNum);
        printf("NumOfPackets: %d\n",pBlockListData->NumOfPackets);
        if(pBlockListData->pBlock==NULL)
        {
            printf("NULL pointer \n");
        }
        else
        {
            DLLMovePtrToHead(pBlockListData->pBlock);
            ExtPacket_t *pExtPacket;
            while(pExtPacket=(ExtPacket_t*)DLLPeekNextPtr(pBlockListData->pBlock))
            {
                PrintExtPacket(pExtPacket);
            }
        }
    }
}
/*****************************************************************************************/



/********************************** Functions ********************************************/

/********************************** PrintBlockList ***************************************/
/* printing routine which prints the entire buffer structure, which is a list of blocks,    */
/* each block is a list of  extended packets where each extended packet is an extended      */
/* sequence number and a packet.  Since the buffer structure pBlockList is a global variable*/
/* this routine has no input parameters                                                     */
void PrintBlockList(void)
{
    BlockListData_t *pBlockListData;

    printf("=== BlockList ===\n");
    DLLMovePtrToHead(pBlockList); // pBlockList is a global variable
```

```
    while(pBlockListData=(BlockListData_t*)DLLPeekNextPtr(pBlockList))
    {
        PrintBlockListData(pBlockListData);
    }
}
/*****************************************************************************************/



/************************************ My_malloc ***************************************/
/* This routine, together with "AdjustTheMemory", is used to control the amount of memory   */
/* used for storing past data packets and past redundancy.  During initialization, the      */
/* desired number of bytes is stored in the global variable BufferLength_old.Whenever memory*/
/* is allocated to store the data or the redundancy, my_malloc instead of malloc is called. */
/* My_malloc behaves like malloc and in addition, it subtracts the used bytes from the      */
/* available bytes.                                                                         */
static void *my_malloc(size_t size)
{
    BufferLength_old-=size;
    return malloc(size);
}
/*****************************************************************************************/



/************************************ Init RS Decoder *********************************/
/* As the title suggests, this is the initialization routine for the reed solomon decoder.  */
/* Hence it needs to be called before the decoder is called. Its main job is to set the     */
/* global variables n_old and k_old and BufferLength_old and call the init routine of the   */
/* rs engine                                                                                */
void init_rs_decoder(unsigned char n, unsigned char k, int BufferLength)
{
    // Does the rs_decoder need to be initilized?
    if (n!=n_old || k!=k_old)
    {
        n_old=n;
        k_old=k;
    }

    // Adjust Buffer Size
    if (BufferLength_old!=BufferLength)
    {
```

```
        BufferLength_old=BufferLength;
    }

    // initilize the rs engine
    init_rs(n_old, k_old);

    // create the Buffer structure
    pBlockList = DLLCreate();
}
/*****************************************************************************************/


/************************* ReconstructDataPacket ***************************************/
/* This is the workhorse routine of this file.  It expects a pointer to a Block (i.e., the  */
/* the data of a block list) and then figures out if any data packets in that block are     */
/* missing.  If so, it checks if enough parity packets are available to reconstruct the     */
/* missing data packet.  Note that if a block has been reconstructed, than we could remove  */
/* it from the memory, since it will never be needed again.  This is not done yet           */
void ReconstructDataPacket(BlockListData_t *pBlockListData)
{
    // function prototyp, puts a packet into the buffer structure
    void PutExtPacket(ExtPacket_t*, unsigned char);

    // are there any packets missing?
    if (pBlockListData->NumOfPackets < n_old)
    {   // Yes
        // could we reconstruct data packets?
        if( pBlockListData->NumOfPackets>=k_old)
        {   // we could!
            // this is where we store the erasure positions
            int *eras_pos= (int*)malloc(n_old*sizeof(int));
            // this is used to remember the lost packets
            unsigned char *lost=(unsigned char*)malloc(n_old*sizeof(unsigned char));
            int no_eras=0;  // count the number of lost packets
            // the extended sequence number with which the block starts
            unsigned __int32 StartExtSeqNum=pBlockListData->StartExtSeqNum;

            // which packets are missig?
            for(unsigned char i=0;i<n_old;i++) lost[i]=1; // all to start with
            DLLMovePtrToHead(pBlockListData->pBlock);
            ExtPacket_t *pExtPacket;
            while(pExtPacket=(ExtPacket_t*)DLLPeekNextPtr(pBlockListData->pBlock))
```

```
{
    lost[pExtPacket->ExtSeqNum-StartExtSeqNum]=0; // got this one
}
// record which packets are lost
unsigned char j=0;
for(i=0;i<n_old;i++)
{
    if(lost[i])
    {
        eras_pos[j]=i;
        j++;
    }
}
no_eras=j;

// are any of the data packets missing?
if(eras_pos[0]<(int)k_old)
{   // Yes
    // First we run the rs-decoder to recover the two length bytes
    unsigned char Byte0[255], Byte1[255];
    DLLMovePtrToHead(pBlockListData->pBlock);
    while(pExtPacket=(ExtPacket_t*)DLLPeekNextPtr(pBlockListData->pBlock))
    {
        Byte0[pExtPacket->ExtSeqNum-StartExtSeqNum]=
            *((unsigned char*)(pExtPacket->pPacket->pData));
        Byte1[pExtPacket->ExtSeqNum-StartExtSeqNum]=
            *((unsigned char*)(pExtPacket->pPacket->pData)+1);

    }
    eras_dec_rs(Byte0,eras_pos,no_eras); // run the rs engine
    eras_dec_rs(Byte1,eras_pos,no_eras);

    // store the recovered data packets in this DLL
    DLLType *pRecoveredExtPackets=DLLCreate();
    for(i=0;i<k_old;i++)
    {   // was this data packet lost?
        if(lost[i])
        {   //create an empty packet, i.e., no data yet
            pExtPacket=new ExtPacket_t;
            pExtPacket->ExtSeqNum=i+StartExtSeqNum;
            pExtPacket->pPacket=new PKT;
            pExtPacket->pPacket->SeqNum=(unsigned char)(pExtPacket->ExtSeqNum%256);
```

```
            // use the length bytes we just recovered
            pExtPacket->pPacket->nLengthBytes=Byte0[i]+256*Byte1[i];
            pExtPacket->pPacket->pData=malloc(pExtPacket->pPacket->nLengthBytes);
            // put it into the DLL
            DLLInsertAtTail(pRecoveredExtPackets,(void*)pExtPacket);
        }
    }

    // now we recover the data
    unsigned char data[255];
    int Byte=2;// the first two bytes are the length bytes
    unsigned char DataAvailable;
    do
    {
        DataAvailable=0;
        DLLMovePtrToHead(pBlockListData->pBlock);
        while(pExtPacket=(ExtPacket_t*)DLLPeekNextPtr(pBlockListData->pBlock))
        {
            if(Byte<pExtPacket->pPacket->nLengthBytes)
            {
                DataAvailable=1; // copy the data
                data[pExtPacket->ExtSeqNum-StartExtSeqNum]=
                    *((unsigned char*)(pExtPacket->pPacket->pData)+Byte);
            }
            else
            {   // padding with 0's
                data[pExtPacket->ExtSeqNum-StartExtSeqNum]=0;
            }
    }

    if (DataAvailable)
    {   // run the rs decoder
        eras_dec_rs(data,eras_pos,no_eras);

        // put the recovered data into the packets
        DLLMovePtrToHead(pRecoveredExtPackets);
        while(pExtPacket=(ExtPacket_t*)DLLPeekNextPtr(pRecoveredExtPackets))
        {   // the 2 data length bytes are not included in the data
            if(Byte-2<pExtPacket->pPacket->nLengthBytes)
            {
                *((unsigned char*)(pExtPacket->pPacket->pData)+Byte-2)=
                    data[pExtPacket->ExtSeqNum-StartExtSeqNum];
```

```
                }
            }

            Byte++;
        }
    }
    while(DataAvailable);

    // put copies of the recovered data into the buffer structure
    DLLMovePtrToHead(pRecoveredExtPackets);
    while(pExtPacket=(ExtPacket_t*)DLLPeekNextPtr(pRecoveredExtPackets))
    {
        ExtPacket_t *pCopy=new ExtPacket_t;
        pCopy->ExtSeqNum=pExtPacket->ExtSeqNum;
        pCopy->pPacket=new PKT;
        pCopy->pPacket->SeqNum=pExtPacket->pPacket->SeqNum;
        // the 2 length bytes are data in the buffer
        pCopy->pPacket->nLengthBytes=pExtPacket->pPacket->nLengthBytes+2;
        pCopy->pPacket->pData=my_malloc(pCopy->pPacket->nLengthBytes);
        *(WORD*)(pCopy->pPacket->pData)=pExtPacket->pPacket->nLengthBytes;
        memmove(((unsigned char*)(pCopy->pPacket->pData)+2),
            pExtPacket->pPacket->pData,pExtPacket->pPacket->nLengthBytes);
        PutExtPacket(pCopy,0);
    }


    // send the recovered data packets to the outbox
    DLLMovePtrToHead(pRecoveredExtPackets);
    while(pExtPacket=(ExtPacket_t*)DLLPeekNextPtr(pRecoveredExtPackets))
    {
        AddMessageToBox(pOutBox, (void*) pExtPacket->pPacket);
    }

    // destroy the DLL
    DLLDestroy(pRecoveredExtPackets);

}

// clean up
free(eras_pos);
free(lost);
}
```

```
    }
}
/*****************************************************************************************/


/********************************* Recover ***********************************************/
/* This routine is called after a new packet has been inserted into the buffer.  Clearly    */
/* that packet can only contribute to two possible blocks, the one its data packet belongs  */
/* to and the one its redundancy packet belongs to.  This is not considered in this routine */
/* but we simply go through all the blocks and try to reconstruct missing packets           */
void Recover(void)
{
    BlockListData_t *pBlockListData;
    DLLMovePtrToHead(pBlockList);
    while(pBlockListData=(BlockListData_t*)DLLPeekNextPtr(pBlockList))
    {
        ReconstructDataPacket(pBlockListData);
    }
}
/*****************************************************************************************/


/********************************* Put Packet Into Block *********************************/
/* As the name suggests, given a pointer to a block, a packet and its extended sequence     */
/* number, the packet is insert into that block at the correct location.                     */
void PutPacketIntoBlock(BlockListData_t *pData, PKT* pPacket, unsigned __int32 ExtSeqNum)
{
    // function prototyp
    void KillExtPacket(ExtPacket_t *pExtPacket);

    // variables used
    ExtPacket_t *pExtPacket, *pTempPacket;

    // create the extended packet
    pExtPacket = new ExtPacket_t;
    pExtPacket->ExtSeqNum=ExtSeqNum;
    pExtPacket->pPacket = pPacket;

    // does the Block exist?
    if(pData->pBlock==NULL)
    {   // No,create a DLL and insert the node
        pData->pBlock=DLLCreate();
```

```
        DLLInsertAtHead(pData->pBlock,(void*)pExtPacket);
        // one packet has been added, update the counter
        pData->NumOfPackets+=1;
    }
    else
    {   // Yes, the Block exists, find the packet
        // is it to the right of the Head (earlier?)
        pTempPacket=(ExtPacket_t*)DLLPeekAtHead(pData->pBlock);
        if (pTempPacket->ExtSeqNum>ExtSeqNum)
        {   //put the node to the right of the head
            DLLInsertAtHead(pData->pBlock,(void*)pExtPacket);
            // one packet has been added, update the counter
            pData->NumOfPackets+=1;
        }
        else
        {   // to the left of Tail? (later)
            pTempPacket=(ExtPacket_t*)DLLPeekAtTail(pData->pBlock);
            if (pTempPacket->ExtSeqNum<ExtSeqNum)
            {   //put the node to the left of the tail
                DLLInsertAtTail(pData->pBlock,(void*)pExtPacket);
                // one packet has been added
                pData->NumOfPackets+=1;
            }
            else
            {   // between Head and Tail
                // Does it already exist?
                DLLMovePtrToHead(pData->pBlock);
                pTempPacket=(ExtPacket_t*)DLLPeekAtPtr(pData->pBlock);
                while(pTempPacket->ExtSeqNum < ExtSeqNum)
                {
                    DLLAdvancePtr(pData->pBlock);
                    pTempPacket=(ExtPacket_t*)DLLPeekAtPtr(pData->pBlock);
                }
                // two options, we found it, or we are too far
                if (pTempPacket->ExtSeqNum == ExtSeqNum)
                {   // This packet already arrived OR has already been
                    // reconstructed, hence we just free the memory
                    KillExtPacket(pExtPacket);
                }
                else
                {   // node does not exist yet, hence we are
                    // too far to the left, need to insert it
```

```
                    DLLInsertBeforePtr(pData->pBlock,(void*)pExtPacket);
                    // one packet has been added, update the counter
                    pData->NumOfPackets+=1;
                }
            }
        }
    }
}
/*****************************************************************************************/


/********************************** Kill Packet ******************************************/
void KillPacket(PKT *pPacket)
{
    BufferLength_old += pPacket->nLengthBytes; // give the bytes back
    free(pPacket->pData);
    free(pPacket);
}
/*****************************************************************************************/


/********************************** Kill Extended Packet *********************************/
void KillExtPacket(ExtPacket_t *pExtPacket)
{
    KillPacket(pExtPacket->pPacket);
    free(pExtPacket);
}
/*****************************************************************************************/


/********************************** Kill Block *******************************************/
void KillBlock(DLLType *pBlock)
{
    ExtPacket_t *pExtPacket;
    while( pExtPacket=(ExtPacket_t*)DLLRemoveFromHead(pBlock))
    {
        KillExtPacket(pExtPacket);
    }
}
/*****************************************************************************************/
```

```
/********************************** Adjust The Memory ***************************************/
/* This function is called after all the processing for a newly arrived packet has been done*/
/* It simply goes through the list of blocks, erasing blocks at the head (past) until the   */
/* size of the buffer (pBlockList) is just below the set target BufferLength_old            */
void AdjustTheMemory(void)
{
    BlockListData_t *pBlockListData;
    while(BufferLength_old<=0)
    {
        pBlockListData = (BlockListData_t *) DLLRemoveFromHead(pBlockList);
        KillBlock(pBlockListData->pBlock);
        free(pBlockListData);
    }
}
/*****************************************************************************************/


/********************************** Put Extended Packet *************************************/
/* This is the main routine for entering packets into the buffer. The inputs are a pointer  */
/* to the extended packet to be inserted, and a flag if the packet is a redundancy packet,  */
/* or not.  In the case the packet is a redundancy packet, the extended sequence number is  */
/* adjusted temporarily, so that the same routine can be used to find the corresponding     */
/* block as for the data packets.  First we find the corresponding block, and then we enter */
/* the packet using the PutPAcketIntoBlock function.                                        */
void PutExtPacket(ExtPacket_t *pExtPacket, unsigned char RedundancyPacket)
{
    BlockListData_t *pData, *pTempData;
    PKT *pPacket=pExtPacket->pPacket;

    // Store the ExtSeqNum
    unsigned __int32 ExtSeqNum=pExtPacket->ExtSeqNum;
    // cheat for the Redundancy case
    // so that the redundancy packets are put into the correct block
    if (RedundancyPacket)
    {
        ExtSeqNum-=k_old;
    }

    // Figure out the starting ExtSeqNum of the Block
    unsigned __int32 StartExtSeqNum=ExtSeqNum/k_old*k_old;

    // create an empty node
```

```
pData=new BlockListData_t;
pData->StartExtSeqNum=StartExtSeqNum;
pData->NumOfPackets=0;
pData->pBlock=NULL;

// Does a Block list entry for this StartExtSeqNum exist?
pTempData=(BlockListData_t*)DLLPeekAtHead(pBlockList);
if (pTempData==NULL)
{   // new DLL
    DLLInsertAtHead(pBlockList,(void*)pData);
    // put the packet into the block
    if(RedundancyPacket) ExtSeqNum+=k_old;
    PutPacketIntoBlock(pData,pPacket,ExtSeqNum);
}
else
{   // to the right of the Head (earlier?)
    if (pTempData->StartExtSeqNum>StartExtSeqNum)
    {   // enter a node to the right of the head
        DLLInsertAtHead(pBlockList,(void*)pData);
        // put the packet into the block
        if(RedundancyPacket) ExtSeqNum+=k_old;
        PutPacketIntoBlock(pData,pPacket,ExtSeqNum);
    }
    else
    {   // to the left of Tail? (later)
        pTempData=(BlockListData_t*)DLLPeekAtTail(pBlockList);
        if (pTempData->StartExtSeqNum<StartExtSeqNum)
        {   // enter it to the left of the tail
            DLLInsertAtTail(pBlockList,(void*)pData);
            // put the packet into the block
            if(RedundancyPacket) ExtSeqNum+=k_old;
            PutPacketIntoBlock(pData,pPacket,ExtSeqNum);
        }
        else
        {   // between Head and Tail
            // Does it already exist?
            DLLMovePtrToHead(pBlockList);
            pTempData=(BlockListData_t*)DLLPeekAtPtr(pBlockList);
            while(pTempData->StartExtSeqNum < StartExtSeqNum)
            {
                DLLAdvancePtr(pBlockList);
                pTempData=(BlockListData_t*)DLLPeekAtPtr(pBlockList);
```

```
                    }
                    // two options, we found it, or we are too far
                    if (pTempData->StartExtSeqNum == StartExtSeqNum)
                    {   // found it, hence we can free the memory used by pData
                        delete pData;
                        // put the packet into the block pointed to by pTempData
                        if(RedundancyPacket) ExtSeqNum+=k_old;
                        PutPacketIntoBlock(pTempData,pPacket,ExtSeqNum);
                    }
                    else
                    {   // node does not exist yet, hence we are
                        // too far to the left, need to insert it
                        DLLInsertBeforePtr(pBlockList,(void*)pData);
                        // put the packet into the block
                        if(RedundancyPacket) ExtSeqNum+=k_old;
                        PutPacketIntoBlock(pData,pPacket,ExtSeqNum);
                    }
                }
            }
        }
    }
}
/*****************************************************************************/


/********************************* Reed Solomon Decoder *************************************/
/* This is the main routine of this file.  The input parameters are two pointers to Message */
/* boxes.  the first one is the inbox and the second one the outbox.  Packets are passed to */
/* the rs_decoder using the inbox and the resulting decoded packets are passed on through   */
/* the outbox.  The inbox packets must be created with rs_encode, using the same (n,k).     */
/* the outbox packets will then be excatly the same as the packets given to the inbox of    */
/* the rs_encoder, except that the SeqNum field is now valid and reflects the SeqNum of the */
/* packet.  The flow of the routine is as follows.  First the data of the incomming packet  */
/* stripped out and an new outgoing packet is created and put into the outbox.  I.e., the   */
/* the data is just copied and passed through, while the redundancy is just copied but not  */
/* passed on. Then extended packets are created for the data and if the redundancy is       */
/* also for the redundancy.  Then the data packet is put into the buffer and we run the     */
/* recover routine which tries to use this new piece of information to recover some missing */
/* data packets.  If redundancy is available, we put that into the buffer and again run the */
/* recover routine.  Finally, we run the AdjustTheMemory routine which, as the name suggests*/
/* erases the tail of the buffer until it is of the desired length.                         */
void rs_decoder(MESSAGEBOX* pInBox, MESSAGEBOX* pOutlBox)
{
```

```
pOutBox=pOut1Box; // set the global outbox to the passed pointer
PKT *pInPacket;
PKT *pOutDataPacket=new PKT;
PKT *pRedundancyPacket;
PKT *pDataPacket=new PKT;

// First thing we do is to get the data (i.e., the redundancy is not copied)
// and put it into the OutBox
pInPacket = (PKT*) GetMessageFromBox(pInBox);
if (pInPacket==NULL) return;  // there was nothing in the inbox
// explicitly add the sequence number
pOutDataPacket->SeqNum=*((unsigned char*)(pInPacket->pData));
// figure out the Extended Sequence number and store it as the previous one
PrevSeqNum=ExtendSeqNum(pOutDataPacket->SeqNum,PrevSeqNum);
// read the (n,k)
n_old =*(((unsigned char*)(pInPacket->pData))+1);
k_old =*(((unsigned char*)(pInPacket->pData))+2);
// read the  data length
pOutDataPacket->nLengthBytes=
    *((WORD*)(((unsigned char*)(pInPacket->pData))+3));
// Read the length of the redundancy
int RedundancyLength=
    *((WORD*)(((unsigned char*)(pInPacket->pData))+5+pOutDataPacket->nLengthBytes));
if (RedundancyLength==0)
{ // No redundancy sent
    pRedundancyPacket=NULL;
}
else
{
    // copy the redundancy
    pRedundancyPacket = new PKT;
    pRedundancyPacket->nLengthBytes=RedundancyLength;
    pRedundancyPacket->SeqNum=pOutDataPacket->SeqNum;
    pRedundancyPacket->pData=my_malloc(pRedundancyPacket->nLengthBytes * sizeof(unsigned char));
    memmove(pRedundancyPacket->pData,(((unsigned char*)(pInPacket->pData))+5+
    pOutDataPacket->nLengthBytes+2),pRedundancyPacket->nLengthBytes);
}
// copy the data to the out packet
pOutDataPacket->pData=malloc(pOutDataPacket->nLengthBytes * sizeof(unsigned char));
memmove(pOutDataPacket->pData,(((unsigned char*)(pInPacket->pData))+5),pOutDataPacket->nLengthBytes);

// Before we place the data into the outbox, we need a local
```

```
        // copy, which includes the two data length bytes
        pDataPacket->nLengthBytes=pOutDataPacket->nLengthBytes+2;
        pDataPacket->SeqNum = pOutDataPacket->SeqNum;
        pDataPacket->pData=my_malloc(pDataPacket->nLengthBytes * sizeof(unsigned char));
        memmove(pDataPacket->pData,(((unsigned char*)(pInPacket->pData))+3),pDataPacket->nLengthBytes);

        // put the data into the OutBox
        AddMessageToBox(pOutBox, (void*) pOutDataPacket);

        // first we put the data packet into the buffer structure
        // Create an extended packet
        ExtPacket_t *pExtPacket=new ExtPacket_t;
        pExtPacket->ExtSeqNum=ExtendSeqNum(pDataPacket->SeqNum,PrevSeqNum);
        pExtPacket->pPacket=pDataPacket;
        // put the extended packet into the buffer
        PutExtPacket(pExtPacket,0);
        // if this piece of data allows other data to be recovered,
        // this whill be done in this routine
        Recover();

        // put the redundancy data into the buffer structure
        if (RedundancyLength!=0)
        {
            // Create an extended packet
            ExtPacket_t *pExtPacket=new ExtPacket_t;
            pExtPacket->ExtSeqNum=ExtendSeqNum(pRedundancyPacket->SeqNum,PrevSeqNum);
            pExtPacket->pPacket=pRedundancyPacket;
            // put the extended packet into the buffer
            PutExtPacket(pExtPacket,1);
            // if this piece of data allows other data to be recovered,
            // this whill be done in this routine
            Recover();
        }

        // free the memory occupied by InPacket
        free(pInPacket->pData);
        delete pInPacket;

        // keep the buffer structure from eating up all the memory
        AdjustTheMemory();
    }
    /*****************************************************************************************/
```

```
/* RS_DECODER.H */

#ifndef RS_DECODER
#define RS_DECODER

#include "types.h"

void rs_decoder(MESSAGEBOX* pInBox, MESSAGEBOX* pOutBox);

void init_rs_decoder(unsigned char n, unsigned char k, int BufferLenght);

#endif


/* RTOS.CPP */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "RTOS.h"

void* GetMessageFromBox( MESSAGEBOX* pMessageBox )
{
    MESSAGE*    pMessage;
    void*       pData;

    if( !pMessageBox )
        return NULL;

    if( !pMessageBox->pFirstMessage )
        return NULL;

    pMessage = pMessageBox->pFirstMessage;
    pMessageBox->pFirstMessage = pMessageBox->pFirstMessage->pNextMessage;

    if( pMessageBox->pFirstMessage == NULL )
        pMessageBox->pLastMessage = NULL;

    pData = pMessage->pData;
    free( pMessage );
```

```
    return pData;
};

int AddMessageToBox( MESSAGEBOX* pMessageBox, void* pData )
{
    MESSAGE* pNewMessage;

    if( !pMessageBox )
        return 0;

    pNewMessage = (MESSAGE*)malloc( sizeof(MESSAGE) );

    if( !pNewMessage )
        return 0;

    pNewMessage->pData = pData;
    pNewMessage->ucFlags = 0;
    pNewMessage->pNextMessage = NULL;

    if( pMessageBox->pLastMessage )
    {
        /* not the first message in this message box */
        pMessageBox->pLastMessage->pNextMessage = pNewMessage;
        pMessageBox->pLastMessage = pNewMessage;
    }

    else
    {
        /* first message in this message box */
        pMessageBox->pFirstMessage = pNewMessage;
        pMessageBox->pLastMessage = pNewMessage;
    }

    return 1;
}

int IsMessageInBox( MESSAGEBOX* pMessageBox )
{
    if( !pMessageBox )
        return 0;
```

```
    if( !pMessageBox->pFirstMessage )
        return 0;

    return 1;
}

MESSAGEBOX* CreateEmptyMessageBox()
{
    MESSAGEBOX* pNewMessageBox;

    pNewMessageBox = (MESSAGEBOX*)malloc( sizeof(MESSAGEBOX) );

    if( pNewMessageBox )
    {
        pNewMessageBox->pFirstMessage   = NULL;
        pNewMessageBox->pLastMessage    = NULL;
    }

    return pNewMessageBox;
}

int DestroyMessageBox( MESSAGEBOX* pMessageBox )
{
    MESSAGE* pCurrentMessage;
    MESSAGE* pNextMessage;

    if( !pMessageBox )
        return 0;

    pCurrentMessage = pMessageBox->pFirstMessage;

    while( pCurrentMessage )
    {
        free( pCurrentMessage->pData );
        pNextMessage = pCurrentMessage->pNextMessage;
        free( pCurrentMessage );
        pCurrentMessage = pNextMessage;
    }

    return 1;
}
```

```
/* RTOS.H */

#ifndef RTOS
#define RTOS

struct MESSAGEstruct
{
    void*                   pData;
    unsigned char           ucFlags;
    struct MESSAGEstruct*   pNextMessage;
};
typedef struct MESSAGEstruct MESSAGE;

struct MESSAGEBOXstruct
{
    struct MESSAGEstruct*   pFirstMessage;
    struct MESSAGEstruct*   pLastMessage;
};
typedef struct MESSAGEBOXstruct MESSAGEBOX;

void*   GetMessageFromBox( MESSAGEBOX* pMessageBox );
int     AddMessageToBox( MESSAGEBOX* pMessageBox, void* pData );
int     IsMessageInBox( MESSAGEBOX* pMessageBox );

MESSAGEBOX* CreateEmptyMessageBox();
int         DestroyMessageBox( MESSAGEBOX* pMessageBox );

#endif


/* TOOLS.CPP */

#include "tools.h"
#include "math.h"

unsigned __int32 ExtendSeqNum(unsigned char SeqNum, unsigned __int32 PrevSeqNum)
{
    // figure out the LL internal sequence number
    // its the 32 bit unsigned integer the closest to the
    // Extended SeqNum PrevSeqNum of the last packet
    unsigned __int32 x, r, FofR1, FofR0, FofR_1, ReturnSeqNum;
```

```
x=(PrevSeqNum/256);

r=x+1; FofR1 =abs((256*r+SeqNum)-(PrevSeqNum));
r=x;   FofR0 =abs((256*r+SeqNum)-(PrevSeqNum));
if (x>0) {
    r=x-1; FofR_1=abs((256*r+SeqNum)-(PrevSeqNum));}
else{
    FofR_1=256;}

if(FofR1<FofR0){
    if(FofR1<FofR_1){
        r=x+1;}
    else{
        r=x-1;}}
else{
    if(FofR0<FofR_1){
        r=x;}
    else{
        r=x-1;}}
ReturnSeqNum = 256*r+SeqNum;
    ReturnSeqNum = 256*r+SeqNum;
    return ReturnSeqNum;
}
```

```
/* TOOLS.H */

#ifndef TOOLS
#define TOOLS

unsigned __int32 ExtendSeqNum(unsigned char SeqNum, unsigned __int32 PrevSeqNum);

#endif
```

```
/* TYPES.CPP */

#include "types.h"
#include "stdio.h"

void PrintPacket(PKT* pPacket)
{
    printf("+++ Packet +++\n");
```

```
    if (pPacket==NULL)
    {
        printf("NULL pointer \n");
    }
    else
    {
        printf("SeqNum: %d\n",pPacket->SeqNum);
        printf("nLengthBytes: %d\n",pPacket->nLengthBytes);
        for(int i=0; i<pPacket->nLengthBytes;i++)
        {
            printf("%03i=%c, ",*(((unsigned char*)pPacket->pData)+i),
                *(((unsigned char*)pPacket->pData)+i));
        }
        printf("\n");
    }
}
```

/* TYPES.H */

```
#ifndef TYPES
#define TYPES

#include "windows.h"  //for the definition of WORD

struct PKTstruct
{
    void* pData; /*Bytes*/
    WORD nLengthBytes;
    unsigned char SeqNum;
};
typedef struct PKTstruct PKT;

// a simple print routine for the packet structure
// is in the "types.cpp" file
void PrintPacket(PKT* pPacket);

#endif
```

APPENDIX B

```
/* CODER.C */

/*****************************************************************************/
/*                                                                           */
/* coder.c                                                                   */
/*                                                                           */
/* 10/7/97                                                                   */
/*                                                                           */
/* Guido Schuster                                                            */
/* Advanced Technologies                                                     */
/* Carrier Systems Division                                                  */
/* 3COM                                                                      */
/* 1800 W. Central Rd.                                                       */
/* Mount Prospect, Il  60056                                                 */
/*                                                                           */
/* This is a test program for the forward error correction modules.  Twenty packets of      */
/* variable length are created.  These packets are then put into the channel coder.  Then   */
/* some of the output packets of the channel coder are  dropped and/or rearranged. This     */
/* results in a packet  stream with out of order packets and lost packets. This stream is   */
/* then fed into the channel decoder which attempts to recover the dropped packets. Finally */
/* the resulting packet stream is displayed                                                 */
/*****************************************************************************/


/******************************** The Include Files **************************************/
#include "stdlib.h" // the basic libraries
#include "malloc.h"
#include "string.h"
#include "windows.h"

#include "types.h"  // this is where the general types are defined
#include "RTOS.h"   // the real time operating system (rtos),
                    // basically the messageboxes

#include "rs_coder.h"       // the reed solomon channel coder
#include "rs_decoder.h"     // the reed solomon channel decoder
#include "xor_coder.h"      // the XOR based channel coder
#include "xor_decoder.h"    // the XOR based channel decoder
/*****************************************************************************/


/******************************** The Define Statements **********************************/
```

```
#define RS      0   // the reed solomon coder is used instead of the XOR coder
#define N_RS    6   // the block length for the FEC (forward error correction)
#define K_RS    4   // the number of information bytes
#define MEM_RS  10000   // the target number of bytes used for storing the packets

#define XOR     1   // the XOR coder is used instead of the RS coder
#define N_XOR   6   // the block length for the XOR
#define MEM_XOR 10000   // the target number of bytes used for storing the packets
/*****************************************************************************/


/****************************** The Main Program *****************************************/
/* No Inputs, the only output is what is printed to the screen, which is the packet stream  */
/* exiting the decoder                                                                      */
void main (void )
 {

    PKT *pCoderInPacket[20]; // used to store the packets
    PKT *pCoderOutPacket[20];
    PKT *pDeCoderInPacket[20];
    PKT *pDeCoderOutPacket[20];

    MESSAGEBOX *pCoderInBox = CreateEmptyMessageBox(); // used to pass the packets
    MESSAGEBOX *pCoderOutBox = CreateEmptyMessageBox();
    MESSAGEBOX *pDeCoderInBox = CreateEmptyMessageBox();
    MESSAGEBOX *pDeCoderOutBox = CreateEmptyMessageBox();

    // init the coder and the decoder
    if (XOR)
    {
        init_xor_coder(N_XOR);
        init_xor_decoder(N_XOR,MEM_XOR);
    }
    if (RS)
    {
        init_rs_coder(N_RS,K_RS);
        init_rs_decoder(N_RS,K_RS,MEM_RS);
    }

    // Initialize the test packets
    for(int i=0;i<20;i++)
    {
```

```
        pCoderInPacket[i] = new PKT;
        pCoderInPacket[i]->nLengthBytes=i;
        pCoderInPacket[i]->pData=malloc(pCoderInPacket[i]->nLengthBytes);
        for (int j=0;j<pCoderInPacket[i]->nLengthBytes;j++)
        {
            *(((unsigned char*)pCoderInPacket[i]->pData)+j)='0'+i%10;
        }
    }

    // Send the test packets into the coder and run the coder
    for(i=0;i<20;i++)
    {
        AddMessageToBox(pCoderInBox,(void*)pCoderInPacket[i]);
        if(XOR)
        {
            xor_coder(pCoderInBox,pCoderOutBox);
        }
        if(RS)
        {
            rs_coder(pCoderInBox,pCoderOutBox);
        }
    }

    // Simulate the Internet, i.e., packets can be lost or arrive out of sequence
    // first read the packets
    for(i=0;i<20;i++)
    {
        pCoderOutPacket[i]=(PKT*)GetMessageFromBox(pCoderOutBox);
    }

    // drop some packets
    for (i=0;i<20;i++)
    {
        if (i==1||i==7||i==13||i==14)
        {
            pCoderOutPacket[i]=NULL;
        }
    }

    // change the sequence
    int from[20]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19};
    int to[20]  ={1,2,0,3,4,5,9,6,7,8,10,11,12,13,14,18,15,16,17,19};
```

```
for(i=0;i<20;i++)
{
    pDeCoderInPacket[to[i]]=pCoderOutPacket[from[i]];
}

// pass the packets to the decoder
for(i=0;i<20;i++)
{
    if(pDeCoderInPacket[i])
    {
        AddMessageToBox(pDeCoderInBox,pDeCoderInPacket[i]);
        // run the decoder
        if(RS)
        {
            rs_decoder(pDeCoderInBox,pDeCoderOutBox);
        }
        if(XOR)
        {
            xor_decoder(pDeCoderInBox,pDeCoderOutBox);
        }
    }

    // display the result
    for(i=0;i<20;i++)
    {
        while(pDeCoderOutPacket[i]=(PKT*)GetMessageFromBox(pDeCoderOutBox))
        {
            PrintPacket(pDeCoderOutPacket[i]);
        }
    }

}
/*****************************************************************************************/


/* DLLIST.C */

#include <stdio.h>
#include <stdlib.h>
#include "dllist.h"
```

```
static int nNextDLLId = 0;

/*************************************************************************/
/**                                                                     **/
/**  Call this function to create a doubly linked list and to get a     **/
/**  pointer to the list.  This pointer is needed to do all operations  **/
/**  on the list.                                                       **/
/**                                                                     **/
/**  When a list is done being used (such as at the end of the          **/
/**  program), call DLLDestroy() with this pointer.                     **/
/**                                                                     **/
/**  Returns:  pointer to the linked list structure.                    **/
/**                                                                     **/
/*************************************************************************/
DLLType* DLLCreate()
{
    DLLType* pDLL;

    /* generate a new ID for the list */
    nNextDLLId++;

    pDLL = (DLLType*)malloc(sizeof(DLLType));
    if( pDLL == NULL )
    {
        /* error -- not enough memory */
        return (NULL);
    }

    /* fill in fields in DLL structure */
    pDLL->pHead   = NULL;
    pDLL->pTail   = NULL;
    pDLL->ulCount = 0;
    pDLL->nID     = nNextDLLId;
    pDLL->pPtr    = NULL;
    return (pDLL);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to destroy a doubly linked list that was        **/
/**  created with the DLLCreate() function.  The pointer obtained from  **/
/**  DLLCreate() must be supplied to this function.                     **/
```

```
/**                                                                    **/
/**  Note that only the nodes of the linked list and the data          **/
/**  structure holding information about the linked list is destroyed. **/
/**  If any nodes are in the list when this function is called, the    **/
/**  data pointed to by the nodes is NOT cleared.  The clearing of     **/
/**  that data is the responsibility of the program that allocated     **/
/**  that memory.                                                      **/
/**                                                                    **/
/************************************************************************/
void DLLDestroy( DLLType* pDLL )
{
    DLLNodeType* pDLLNode;
    DLLNodeType* pDLLNextNode;

    if( DLLVerifyIntegrity(pDLL) )
    {
        /* DLL integrity not violated.  */
        /* Erase all nodes      */
        pDLLNode = pDLL->pHead;
        while( pDLLNode )
        {
            pDLLNextNode = pDLLNode->pNextNode;
            free( pDLLNode );
            pDLLNode = pDLLNextNode;
        }
    }

    free( pDLL );
}

/************************************************************************/
/**                                                                    **/
/**  Call this function to insert a node into the linked list at the   **/
/**  head of the list.  This node will point to the data stored at     **/
/**  pData.                                                            **/
/**                                                                    **/
/**  Returns:  0 if operation failed (out of memory)                   **/
/**            1 if operation successful                               **/
/**                                                                    **/
/************************************************************************/
int DLLInsertAtHead( DLLType* pDLL, void* pData )
{
```

```
    DLLNodeType* pDLLNode;

    pDLLNode = (DLLNodeType*)malloc(sizeof(DLLNodeType));
    if( pDLLNode == NULL )
    {
        /* error -- not enough memory */
        return (0);
    }

    /* update all pointers */
    if( pDLL->pHead == NULL )
    {
        /* list empty */
        pDLL->pHead = pDLLNode;
        pDLL->pTail = pDLLNode;
        pDLLNode->pNextNode = NULL;
        pDLLNode->pPrevNode = NULL;
    }

    else
    {
        /* list not empty */
        pDLL->pHead->pPrevNode = pDLLNode;
        pDLLNode->pNextNode = pDLL->pHead;
        pDLLNode->pPrevNode = NULL;
        pDLL->pHead = pDLLNode;
    }

    /* change pointer to data */
    pDLLNode->pData = pData;

    /* update counter information */
    pDLL->ulCount++;

    return (1);
}

/*****************************************************************************/
/**                                                                         **/
/**  Call this function to insert a node into the linked list at the        **/
/**  tail of the list.  This node will point to the data stored at          **/
/**  pData.                                                                 **/
```

```
/**                                                                    **/
/**  Returns:  0 if operation failed (out of memory)                   **/
/**            1 if operation successful                               **/
/**                                                                    **/
/************************************************************************/
int DLLInsertAtTail( DLLType* pDLL, void* pData )
{
    DLLNodeType* pDLLNode;

    pDLLNode = (DLLNodeType*)malloc(sizeof(DLLNodeType));
    if( pDLLNode == NULL )
    {
        /* error -- not enough memory */
        return (0);
    }

    /* update all pointers */
    if( pDLL->pTail == NULL )
    {
        /* list empty */
        pDLL->pTail = pDLLNode;
        pDLL->pHead = pDLLNode;
        pDLLNode->pNextNode = NULL;
        pDLLNode->pPrevNode = NULL;
    }

    else
    {
        /* list not empty */
        pDLL->pTail->pNextNode = pDLLNode;
        pDLLNode->pPrevNode = pDLL->pTail;
        pDLLNode->pNextNode = NULL;
        pDLL->pTail = pDLLNode;
    }

    /* change pointer to data */
    pDLLNode->pData = pData;

    /* update counter information */
    pDLL->ulCount++;

    return (1);
```

```
}

/*****************************************************************************/
/**                                                                         **/
/**  Call this function to remove a node from the linked list at the        **/
/**  head of the list.  A pointer to the data pointed to by that node       **/
/**  is returned.                                                           **/
/**                                                                         **/
/**  Returns:  NULL            if operation failed (list empty)             **/
/**            pointer to data if operation successful                      **/
/**                                                                         **/
/*****************************************************************************/
void* DLLRemoveFromHead( DLLType* pDLL )
{
    void* pData;
    DLLNodeType* pNewHead;

    /* check for empty list */
    if( pDLL->pHead == NULL )
    {
        /* list empty   */
        return (NULL);
    }

    /* remember where data is */
    pData = pDLL->pHead->pData;

    /* check for future validity of DLL Pointer */
    if( pDLL->pPtr == pDLL->pHead )
    {
        /* DLL Pointer points to head node  */
        /* DLL Pointer becomes invalid now  */
        pDLL->pPtr = NULL;
    }

    /* update all pointers */
    if( pDLL->pHead == pDLL->pTail )
    {
        /* only one element in list */
        free( pDLL->pHead );
        pDLL->pHead = NULL;
        pDLL->pTail = NULL;
```

```
    }

    else
    {
        /* not the only element in list */
        pNewHead = pDLL->pHead->pNextNode;
        pNewHead->pPrevNode = NULL;
        free( pDLL->pHead );
        pDLL->pHead = pNewHead;
    }

    /* update counter information */
    pDLL->ulCount--;

    return (pData);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to remove a node from the linked list at the    **/
/**  tail of the list.  A pointer to the data pointed to by that node   **/
/**  is returned.                                                       **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLRemoveFromTail( DLLType* pDLL )
{
    void* pData;
    DLLNodeType* pNewTail;

    /* check for empty list */
    if( pDLL->pTail == NULL )
    {
        /* list empty   */
        return (NULL);
    }

    /* remember where data is */
    pData = pDLL->pTail->pData;
```

```
    /* check for future validity of DLL Pointer */
    if( pDLL->pPtr == pDLL->pTail )
    {
        /* DLL Pointer points to head node  */
        /* DLL Pointer becomes invalid now  */
        pDLL->pPtr = NULL;
    }

    /* update all pointers */
    if( pDLL->pHead == pDLL->pTail )
    {
        /* only one element in list */
        free( pDLL->pTail );
        pDLL->pHead = NULL;
        pDLL->pTail = NULL;
    }

    else
    {
        /* not the only element in list */
        pNewTail = pDLL->pTail->pPrevNode;
        pNewTail->pNextNode = NULL;
        free( pDLL->pTail );
        pDLL->pTail = pNewTail;
    }

    /* update counter information */
    pDLL->ulCount--;

    return (pData);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to remove a node from the linked list at the    **/
/**  specified node.                                                    **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLRemove( DLLType* pDLL, DLLNodeType* pDLLNode )
```

```
{
    void* pData;

    if( pDLLNode == NULL )
    {
        /* error -- supplied pointer invalid */
        return (NULL);
    }

    if( pDLLNode == pDLL->pHead )
    {
        /* removing from head */
        return (DLLRemoveFromHead(pDLL));
    }

    if( pDLLNode == pDLL->pTail )
    {
        /* removing from tail */
        return (DLLRemoveFromTail(pDLL));
    }

    /* removing one in middle */
    if( pDLLNode == pDLL->pPtr )
    {
        /* remove node pointed to by DLL Pointer */
        /* this will invalidate that pointer     */
        pDLL->pPtr = NULL;
    }

    if( pDLLNode->pPrevNode == NULL )
    {
        /* error -- supplied pointer not    */
        /* part of this DLL         */
        return (NULL);
    }

    if( pDLLNode->pNextNode == NULL )
    {
        /* error -- supplied pointer not    */
        /* part of this DLL         */
        return (NULL);
    }
```

```
    pDLLNode->pPrevNode->pNextNode = pDLLNode->pNextNode;
    pDLLNode->pNextNode->pPrevNode = pDLLNode->pPrevNode;

    /* remember where data is */
    pData = pDLLNode->pData;

    /* free node */
    free( pDLLNode );

    /* update counter information */
    pDLL->ulCount--;

    return (pData);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to peek at the head node in the linked list.    **/
/**  The node is not removed from the list, the pointer to the data     **/
/**  is simply returned.                                                **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLPeekAtHead( DLLType* pDLL )
{
    return( DLLPeek(pDLL, pDLL->pHead) );
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to peek at the tail node in the linked list.    **/
/**  The node is not removed from the list, the pointer to the data     **/
/**  is simply returned.                                                **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLPeekAtTail( DLLType* pDLL )
```

```
{
    return( DLLPeek(pDLL, pDLL->pTail) );
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to peek at the specified node in the linked     **/
/**  list.  The node is not removed from the list, the pointer to the   **/
/**  data is simply returned.                                           **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLPeek( DLLType* pDLL, DLLNodeType* pDLLNode )
{
    /* prevent compiler complaining */
    pDLL->ulCount += 0;

    if( pDLLNode == NULL )
    {
        /* error -- supplied pointer invalid */
        return (NULL);
    }

    return (pDLLNode->pData);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to insert a node pointing to the specified      **/
/**  data before the specified node in the list.                        **/
/**                                                                     **/
/**  Returns:  0 if operation failed (out of memory, etc)               **/
/**            1 if operation successful                                **/
/**                                                                     **/
/*************************************************************************/
int DLLInsertBefore( DLLType* pDLL, DLLNodeType* pDLLNode, void* pData )
{
    DLLNodeType* pDLLNewNode;

    if( pDLLNode == NULL )
```

```
    {
        /* means put at end */
        return (DLLInsertAtTail(pDLL,pData));
    }

    if( pDLL->pHead == pDLLNode )
    {
        /* inserting at beginning */
        return (DLLInsertAtHead(pDLL,pData));
    }

    if( pDLLNode->pPrevNode == NULL )
    {
        /* error -- supplied pointer is not */
        /* head of list, but previous node  */
        /* is NULL -- perhaps node not part */
        /* of this linked list              */
        return (0);
    }

    pDLLNewNode = (DLLNodeType*)malloc(sizeof(DLLNodeType));
    if( pDLLNewNode == NULL )
    {
        /* error -- not enough memory */
        return (0);
    }

    /* inserting in middle */
    pDLLNewNode->pNextNode = pDLLNode;
    pDLLNewNode->pPrevNode = pDLLNode->pPrevNode;
    pDLLNewNode->pPrevNode->pNextNode = pDLLNewNode;
    pDLLNewNode->pNextNode->pPrevNode = pDLLNewNode;

    /* change pointer to data */
    pDLLNewNode->pData = pData;

    /* update counter information */
    pDLL->ulCount++;

    return (1);
}
```

```
/*************************************************************************/
/**                                                                     **/
/**  Call this function to insert a node pointing to the specified      **/
/**  data after the specified node in the list.                         **/
/**                                                                     **/
/**  Returns:  0 if operation failed (out of memory, etc)               **/
/**            1 if operation successful                                **/
/**                                                                     **/
/*************************************************************************/
int DLLInsertAfter( DLLType* pDLL, DLLNodeType* pDLLNode, void* pData )
{
    DLLNodeType* pDLLNewNode;

    if( pDLLNode == NULL )
    {
        /* means put at beginning */
        return (DLLInsertAtHead(pDLL,pData));
    }

    if( pDLL->pTail == pDLLNode )
    {
        /* inserting at end */
        return (DLLInsertAtTail(pDLL,pData));
    }

    if( pDLLNode->pNextNode == NULL )
    {
        /* error -- supplied pointer is not */
        /* tail of list, but next node is   */
        /* NULL -- perhaps node not part of */
        /* this linked list         */
        return (0);
    }

    pDLLNewNode = (DLLNodeType*)malloc(sizeof(DLLNodeType));
    if( pDLLNewNode == NULL )
    {
        /* error -- not enough memory */
        return (0);
    }

    /* inserting in middle */
```

```
    pDLLNewNode->pPrevNode = pDLLNode;
    pDLLNewNode->pNextNode = pDLLNode->pNextNode;
    pDLLNewNode->pNextNode->pPrevNode = pDLLNewNode;
    pDLLNewNode->pPrevNode->pNextNode = pDLLNewNode;

    /* change pointer to data */
    pDLLNewNode->pData = pData;

    /* update counter information */
    pDLL->ulCount++;

    return (1);
}

/*****************************************************************************/
/**                                                                         **/
/**  Call this function to move the DLL Pointer to the head of the          **/
/**  linked list.                                                           **/
/**                                                                         **/
/**  Returns:  no return value                                              **/
/**                                                                         **/
/*****************************************************************************/
void DLLMovePtrToHead( DLLType* pDLL )
{
    pDLL->pPtr = pDLL->pHead;
}

/*****************************************************************************/
/**                                                                         **/
/**  Call this function to move the DLL Pointer to the tail of the          **/
/**  linked list.                                                           **/
/**                                                                         **/
/**  Returns:  no return value                                              **/
/**                                                                         **/
/*****************************************************************************/
void DLLMovePtrToTail( DLLType* pDLL )
{
    pDLL->pPtr = pDLL->pTail;
}

/*****************************************************************************/
/**                                                                         **/
```

```
/**  Call this function to move the DLL Pointer to the specified node   **/
/**  of the linked list.                                                **/
/**                                                                     **/
/**  Returns:  no return value                                          **/
/**                                                                     **/
/*************************************************************************/
void DLLSetPtrToPtr( DLLType* pDLL, DLLNodeType* pDLLNode )
{
    pDLL->pPtr = pDLLNode;
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to remove the node pointed to by the DLL        **/
/**  Pointer.                                                           **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLRemoveAtPtr( DLLType* pDLL )
{
    return DLLRemove( pDLL, pDLL->pPtr );
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to peek at the node pointed to by the DLL       **/
/**  Pointer.  The node is not removed from the list, the pointer to    **/
/**  the data is simply returned.                                       **/
/**                                                                     **/
/**  Returns:  NULL            if operation failed (list empty)         **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLPeekAtPtr( DLLType* pDLL )
{
    return DLLPeek( pDLL, pDLL->pPtr );
}

/*************************************************************************/
/**                                                                     **/
```

```
/**  Call this function to peek at the node pointed to by the DLL       **/
/**  Pointer, and then to advance the DLL Pointer to the next node.     **/
/**  If the DLL Pointer is NULL (such as is the case when the previous  **/
/**  call to this function read in the last node), then this function   **/
/**  returns NULL.                                                      **/
/**                                                                     **/
/**  Returns:  NULL            if list empty or if already read the     **/
/**                               last node previously                  **/
/**            pointer to data if operation successful                  **/
/**                                                                     **/
/*************************************************************************/
void* DLLPeekNextPtr( DLLType* pDLL )
{
    void* pData;

    pData = DLLPeekAtPtr( pDLL );

    if( pDLL->pPtr != NULL )
    {
        pDLL->pPtr = pDLL->pPtr->pNextNode;
    }

    return (pData);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to advance the DLL Pointer to the next node     **/
/**  in the list (the node in the direction of the tail node).  If the  **/
/**  pointer currently points to the last node, the pointer does not    **/
/**  take on a NULL value; the pointer does not change, and a 0 is      **/
/**  returned.                                                          **/
/**                                                                     **/
/**  Returns:  0 if operation failed (list empty or already at last     **/
/**              node                                                   **/
/**            1 if operation successful                                **/
/**                                                                     **/
/*************************************************************************/
int DLLAdvancePtr( DLLType* pDLL )
{
    if( pDLL->pPtr == NULL )
    {
```

```
        pDLL->pPtr = pDLL->pHead;
        return (1);
    }

    else if( pDLL->pPtr == pDLL->pTail )
    {
        /* already at end */
        return (0);
    }

    else
    {
        pDLL->pPtr = pDLL->pPtr->pNextNode;
        return (1);
    }
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to retreat the DLL Pointer to the previous node **/
/**  in the list (the node in the direction of the head node).  If the  **/
/**  pointer currently points to the first node, the pointer does not   **/
/**  take on a NULL value; the pointer does not change, and a 0 is      **/
/**  returned.                                                          **/
/**                                                                     **/
/**  Returns:  0 if operation failed (list empty or already at first    **/
/**              node                                                   **/
/**            1 if operation successful                                **/
/**                                                                     **/
/*************************************************************************/
int DLLRetreatPtr( DLLType* pDLL )
{
    if( pDLL->pPtr == NULL )
    {
        pDLL->pPtr = pDLL->pTail;
        return (1);
    }

    else if( pDLL->pPtr == pDLL->pHead )
    {
        /* already at end */
        return (0);
```

```
    }

    else
    {
        pDLL->pPtr = pDLL->pPtr->pPrevNode;
        return (1);
    }
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to insert a node pointing to the specified      **/
/**  data before the node pointed to by the DLL Pointer.                **/
/**                                                                     **/
/**  Returns:  0 if operation failed (out of memory, etc)               **/
/**            1 if operation successful                                **/
/**                                                                     **/
/*************************************************************************/
int DLLInsertBeforePtr( DLLType* pDLL, void* pData )
{
    return DLLInsertBefore( pDLL, pDLL->pPtr, pData );
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to insert a node pointing to the specified      **/
/**  data after the node pointed to by the DLL Pointer.                 **/
/**                                                                     **/
/**  Returns:  0 if operation failed (out of memory, etc)               **/
/**            1 if operation successful                                **/
/**                                                                     **/
/*************************************************************************/
int DLLInsertAfterPtr( DLLType* pDLL, void* pData )
{
    return DLLInsertAfter( pDLL, pDLL->pPtr, pData );
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to get a pointer to the node pointed to by the  **/
/**  DLL Pointer.                                                       **/
/**                                                                     **/
```

```
/**  Returns:  NULL            if pointer not valid                    **/
/**        pointer to node if pointer valid                            **/
/**                                                                    **/
/************************************************************************/
DLLNodeType* DLLGetPtrToPtr( DLLType* pDLL )
{
    return pDLL->pPtr;
}

/************************************************************************/
/**                                                                    **/
/**  Call this function to determine whether the specified node is     **/
/**  contained in the specified linked list.                           **/
/**                                                                    **/
/**  Returns:  0 if no node specified or if node not in linked list    **/
/**        1 if node in linked list                                    **/
/**                                                                    **/
/************************************************************************/
int DLLIsNodeInDLL( DLLType* pDLL, DLLNodeType* pDLLNode )
{
    unsigned long ulNumNodes;
    unsigned long ulCounter;
    DLLNodeType* pCurrentNode;

    if( pDLLNode == NULL )
    {
        /* error -- supplied pointer invalid */
        return (0);
    }

    ulNumNodes = pDLL->ulCount;

    ulCounter = 0;
    pCurrentNode = pDLL->pHead;

    while( (ulCounter<ulNumNodes)   &&
           (pCurrentNode!=NULL)     &&
           (pCurrentNode!=pDLLNode)    )
    {
        pCurrentNode = pCurrentNode->pNextNode;
        ulCounter++;
    }
```

```
    if( pCurrentNode == pDLLNode )
        return (1);
    else
        return (0);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to determine the number of nodes in a linked    **/
/**  list.                                                              **/
/**                                                                     **/
/**  Returns:  the number of nodes in linked list                       **/
/**                                                                     **/
/*************************************************************************/
unsigned long DLLCount( DLLType* pDLL )
{
    return (pDLL->ulCount);
}

/*************************************************************************/
/**                                                                     **/
/**  Call this function to check the integrity of a linked list.  This  **/
/**  function checks certain criteria about the linked list to make     **/
/**  certain the structure remains valid.  If one suspects that a       **/
/**  linked list has become corrupt, this function should be used.      **/
/**                                                                     **/
/**  Returns:  0 if linked list corrupt                                 **/
/**            1 if linked list okay                                    **/
/**                                                                     **/
/*************************************************************************/
int DLLVerifyIntegrity( DLLType* pDLL )
{
    /* get number of nodes                                          */
    /* start from head and go until                                 */
    /*  (1) null, or                                                */
    /*  (2) check "number of nodes"                                 */
    /*  --- if reached end too soon or didn't reach end             */
    /*          after checking "number of nodes", violation         */
    /*      --- if last node != tail, violation                      */
    /* start from tail and go until                                 */
    /*  (1) null, or                                                */
```

```
/*  (2) check "number of nodes"                                    */
/*  --- if reached end too soon or didn't reach end                */
/*          after checking "number of nodes", violation            */
/*      --- if last node != head, violation                        */

unsigned long ulNumNodes;
unsigned long ulCounter;
DLLNodeType*  pCurrentNode;
DLLNodeType*  pLastNode;
int       Violation1 = 0;
int       Violation2 = 0;
int       Violation3 = 0;
int       Violation4 = 0;
int       Violation5 = 0;
int       Violation6 = 0;

ulNumNodes = pDLL->ulCount;

ulCounter = 0;
pCurrentNode = pDLL->pHead;
pLastNode = NULL;
while( (ulCounter<ulNumNodes) &&
       (pCurrentNode!=NULL)       )
{
    ulCounter++;
    pLastNode = pCurrentNode;
    pCurrentNode = pCurrentNode->pNextNode;
}

if( ulCounter != pDLL->ulCount )
    Violation1 = 1;

if( pLastNode != pDLL->pTail )
    Violation2 = 1;

if( pCurrentNode != NULL )
    Violation3 = 1;

ulCounter = 0;
pCurrentNode = pDLL->pTail;
pLastNode = NULL;
while( (ulCounter<ulNumNodes) &&
```

```
        (pCurrentNode!=NULL)       )
    {
        ulCounter++;
        pLastNode = pCurrentNode;
        pCurrentNode = pCurrentNode->pPrevNode;
    }

    if( ulCounter != pDLL->ulCount )
        Violation4 = 1;

    if( pLastNode != pDLL->pHead )
        Violation5 = 1;

    if( pCurrentNode != NULL )
        Violation6 = 1;

    if( Violation1 ||
        Violation2 ||
        Violation3 ||
        Violation4 ||
        Violation5 ||
        Violation6    )
    {
        return (0);
    }

    else
    {
        return (1);
    }

}

/*****************************************************************************/
/**                                                                         **/
/**  Call this function to determine whether the DLL Pointer is valid       **/
/**  or not.  The DLL Pointer is invalid under two conditions:  (1) at      **/
/**  startup (because the pointer has not yet been defined), and (2)        **/
/**  when the node pointed to by the DLL Pointer is removed.                **/
/**                                                                         **/
/**  Returns:  0 if DLL Pointer invalid                                     **/
/**            1 if DLL Pointer valid                                       **/
```

```
/**                                                                    **/
/************************************************************************/
int DLLIsPtrValid( DLLType* pDLL )
{
    if( pDLL->pPtr == NULL )
        return (0);
    else
        return (1);
}

/************************************************************************/
/**                                                                    **/
/**  Call this function to perform a very simple dump of the linked    **/
/**  list.  The ASCII value of the first byte of the data pointed to   **/
/**  by each node is printed to the screen, starting from the head and **/
/**  ending at the tail.                                               **/
/**                                                                    **/
/**  Returns:  no return value                                         **/
/**                                                                    **/
/************************************************************************/
void DLLSimpleDump( FILE* pfileOut, DLLType* pDLL )
{
    void* pData;
    DLLNodeType* pOldPtr;

    /* remember where point used to be */
    pOldPtr = DLLGetPtrToPtr( pDLL );

    /* move pointer to head */
    DLLMovePtrToHead( pDLL );

    /* continue going through list until end */
    while( (pData=DLLPeekNextPtr(pDLL)) != NULL )
        fprintf( pfileOut, "%c ", *((unsigned char*)pData) );

    /* move pointer back to original spot */
    DLLSetPtrToPtr( pDLL, pOldPtr );
}


/* DLLIST.H */
```

```
#define _DLList_h_

struct structDLLNodeType
{
    void*                       pData;
    struct structDLLNodeType*   pNextNode;
    struct structDLLNodeType*   pPrevNode;
};
typedef struct structDLLNodeType DLLNodeType;

struct structDLLType
{
    DLLNodeType*  pHead;
    DLLNodeType*  pTail;
    unsigned long ulCount;
    int           nID;
    DLLNodeType*  pPtr;
};
typedef struct structDLLType DLLType;

DLLType* DLLCreate();
void DLLDestroy( DLLType* pDLL );

int DLLInsertAtHead( DLLType* pDLL, void* pData );
int DLLInsertAtTail( DLLType* pDLL, void* pData );
void* DLLRemoveFromHead( DLLType* pDLL );
void* DLLRemoveFromTail( DLLType* pDLL );
void* DLLRemove( DLLType* pDLL, DLLNodeType* pDLLNode );
void* DLLPeek( DLLType* pDLL, DLLNodeType* pDLLNode );
void* DLLPeekAtHead( DLLType* pDLL );
void* DLLPeekAtTail( DLLType* pDLL );
int DLLInsertBefore( DLLType* pDLL, DLLNodeType* pDLLNode, void* pData );
int DLLInsertAfter( DLLType* pDLL, DLLNodeType* pDLLNode, void* pData );

void DLLMovePtrToHead( DLLType* pDLL );
void DLLMovePtrToTail( DLLType* pDLL );
void DLLSetPtrToPtr( DLLType* pDLL, DLLNodeType* pDLLNode );
void* DLLRemoveAtPtr( DLLType* pDLL );
void* DLLPeekAtPtr( DLLType* pDLL );
void* DLLPeekNextPtr( DLLType* pDLL );
int DLLAdvancePtr( DLLType* pDLL );
int DLLRetreatPtr( DLLType* pDLL );
```

```
int DLLInsertBeforePtr( DLLType* pDLL, void* pData );
int DLLInsertAfterPtr( DLLType* pDLL, void* pData );
DLLNodeType* DLLGetPtrToPtr( DLLType* pDLL );

int DLLIsNodeInDLL( DLLType* pDLL, DLLNodeType* pDLLNode );
unsigned long DLLCount( DLLType* pDLL );
int DLLVerifyIntegrity( DLLType* pDLL );
int DLLIsPtrValid( DLLType* pDLL );

/*int DLLSwapNodes( DLLType* pDLL, DLLNodeType* pDLLNode1, DLLNodeType* pDLLNode2 );*/
/*void DLLSimpleDump( FILE* pfileOut, DLLType* pDLL );*/


/* RTOS.CCP */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "RTOS.h"

void* GetMessageFromBox( MESSAGEBOX* pMessageBox )
{
    MESSAGE*    pMessage;
    void*       pData;

    if( !pMessageBox )
        return NULL;

    if( !pMessageBox->pFirstMessage )
        return NULL;

    pMessage = pMessageBox->pFirstMessage;
    pMessageBox->pFirstMessage = pMessageBox->pFirstMessage->pNextMessage;

    if( pMessageBox->pFirstMessage == NULL )
        pMessageBox->pLastMessage = NULL;

    pData = pMessage->pData;
    free( pMessage );

    return pData;
};
```

```
int AddMessageToBox( MESSAGEBOX* pMessageBox, void* pData )
{
    MESSAGE* pNewMessage;

    if( !pMessageBox )
        return 0;

    pNewMessage = (MESSAGE*)malloc( sizeof(MESSAGE) );

    if( !pNewMessage )
        return 0;

    pNewMessage->pData = pData;
    pNewMessage->ucFlags = 0;
    pNewMessage->pNextMessage = NULL;

    if( pMessageBox->pLastMessage )
    {
        /* not the first message in this message box */
        pMessageBox->pLastMessage->pNextMessage = pNewMessage;
        pMessageBox->pLastMessage = pNewMessage;
    }

    else
    {
        /* first message in this message box */
        pMessageBox->pFirstMessage = pNewMessage;
        pMessageBox->pLastMessage = pNewMessage;
    }

    return 1;
}

int IsMessageInBox( MESSAGEBOX* pMessageBox )
{
    if( !pMessageBox )
        return 0;

    if( !pMessageBox->pFirstMessage )
        return 0;
```

```
    return 1;
}

MESSAGEBOX* CreateEmptyMessageBox()
{
    MESSAGEBOX* pNewMessageBox;

    pNewMessageBox = (MESSAGEBOX*)malloc( sizeof(MESSAGEBOX) );

    if( pNewMessageBox )
    {
        pNewMessageBox->pFirstMessage   = NULL;
        pNewMessageBox->pLastMessage    = NULL;
    }

    return pNewMessageBox;
}

int DestroyMessageBox( MESSAGEBOX* pMessageBox )
{
    MESSAGE* pCurrentMessage;
    MESSAGE* pNextMessage;

    if( !pMessageBox )
        return 0;

    pCurrentMessage = pMessageBox->pFirstMessage;

    while( pCurrentMessage )
    {
        free( pCurrentMessage->pData );
        pNextMessage = pCurrentMessage->pNextMessage;
        free( pCurrentMessage );
        pCurrentMessage = pNextMessage;
    }

    return 1;
}

/* RTOS.H */
```

```
#ifndef RTOS
#define RTOS

struct MESSAGEstruct
{
    void*                   pData;
    unsigned char           ucFlags;
    struct MESSAGEstruct*   pNextMessage;
};
typedef struct MESSAGEstruct MESSAGE;

struct MESSAGEBOXstruct
{
    struct MESSAGEstruct*   pFirstMessage;
    struct MESSAGEstruct*   pLastMessage;
};
typedef struct MESSAGEBOXstruct MESSAGEBOX;

void*   GetMessageFromBox( MESSAGEBOX* pMessageBox );
int     AddMessageToBox( MESSAGEBOX* pMessageBox, void* pData );
int     IsMessageInBox( MESSAGEBOX* pMessageBox );

MESSAGEBOX* CreateEmptyMessageBox();
int         DestroyMessageBox( MESSAGEBOX* pMessageBox );

#endif


/* TOOLS.CCP */

#include "tools.h"
#include "math.h"

unsigned __int32 ExtendSeqNum(unsigned char SeqNum, unsigned __int32 PrevSeqNum)
{
    // figure out the LL internal sequence number
    // its the 32 bit unsigned integer the closest to the
    // Extended SeqNum PrevSeqNum of the last packet
    unsigned __int32 x, r, FofR1, FofR0, FofR_1, ReturnSeqNum;
    x=(PrevSeqNum/256);

    r=x+1; FofR1 =abs((256*r+SeqNum)-(PrevSeqNum));
```

```
        r=x;    FofR0 =abs((256*r+SeqNum)-(PrevSeqNum));
        if (x>0) {
            r=x-1; FofR_1=abs((256*r+SeqNum)-(PrevSeqNum));}
        else{
            FofR_1=256;}

        if(FofR1<FofR0){
            if(FofR1<FofR_1){
                r=x+1;}
            else{
                r=x-1;}}
        else{
            if(FofR0<FofR_1){
                r=x;}
            else{
                r=x-1;}}
        ReturnSeqNum = 256*r+SeqNum;
        return ReturnSeqNum;
    }


    /* TOOLS.H */

    #ifndef TOOLS
    #define TOOLS

    unsigned __int32 ExtendSeqNum(unsigned char SeqNum, unsigned __int32 PrevSeqNum);

    #endif


    /* TYPES.CCP */

    #include "types.h"
    #include "stdio.h"

    void PrintPacket(PKT* pPacket)
    {
        printf("+++ Packet +++\n");
        if (pPacket==NULL)
        {
            printf("NULL pointer \n");
```

```
    }
    else
    {
        printf("SeqNum: %d\n",pPacket->SeqNum);
        printf("nLengthBytes: %d\n",pPacket->nLengthBytes);
        for(int i=0; i<pPacket->nLengthBytes;i++)
        {
            printf("%03i=%c, ",*(((unsigned char*)pPacket->pData)+i),
                *(((unsigned char*)pPacket->pData)+i));
        }
        printf("\n");
    }
}
```

/* TYPES.H */

```
#ifndef TYPES
#define TYPES

#include "windows.h"  //for the definition of WORD

struct PKTstruct
{
    void* pData; /*Bytes*/
    WORD nLengthBytes;
    unsigned char SeqNum;
};
typedef struct PKTstruct PKT;

// a simple print routine for the packet structure
// is in the "types.cpp" file
void PrintPacket(PKT* pPacket);

#endif
```

/* XOR_CODER.CPP */

```
/*****************************************************************************************/
/*                                                                                       */
/* xor_coder.c                                                                           */
```

```
/*                                                                              */
/* 10/15/97                                                                     */
/*                                                                              */
/* Guido Schuster                                                               */
/* Advanced Technologies                                                        */
/* Carrier Systems Division                                                     */
/* 3COM                                                                         */
/* 1800 W. Central Rd.                                                          */
/* Mount Prospect, Il  60056                                                    */
/*                                                                              */
/* This file is the implementation of a xor based channel coder for packets. The basic idea */
/* behind this scheme is that we calculate a redundancy (parity) packet, using k previous   */
/* packets. This redundancy packet is then appended to the current packet and               */
/* shipped off. Since this is done for every packet and not, as it is commonly done for     */
/* block codes, every block, this results in an overall scheme able to recover up to k      */
/* packet losses in a row. Clearly the bit rate roughly doubles, but the packet rate stays  */
/* constant since the additional information is appended. Note that the xor_coder.c is      */
/* meant to work hand in hand with the xor_decoder.c                                        */
/********************************************************************************/


/*********************************** The Include Files ***************************************/
#include "stdlib.h" // the standard ones
#include "malloc.h"
#include "string.h"
#include "windows.h"

#include "types.h"  // commonly used types
#include "RTOS.h"   // definition of the message boxes
#include "xor_coder.h" // the header file

extern "C"
{
#include "dllist.h" // the doubly linked list
}
/********************************************************************************/



/***************************** Global static variables ***********************************/
static unsigned int n_old=0; // the old block lenght, used instead of n
static unsigned int k_old=0; // the old data lenght, for the xor coder =n-1
```

```
/*****************************************************************************/


/************************************ The Functions ************************************/

/********************************** The Initialization Routine *********************************/
/* This routine must be called before the encoder is called.  It sets the block size n and  */
/* generates k (=n-1) previous empty packets, which are sent to the encoder inbox and the   */
/* the resulting packets are removed from the outbox.  The same procedure is done at the    */
/* decoder and hence encoder and decoder have the same history after the initialization.    */
/* By doing this, the first k real packets received do not have to be treated differently   */
/* than any other packets.                                                                  */
void init_xor_coder(unsigned char n)
{
    // the message boxes used to pass the data around
    MESSAGEBOX* pInBox=CreateEmptyMessageBox();
    MESSAGEBOX* pOutBox=CreateEmptyMessageBox();

    // the data packets used to put into the InBox and take out of the OutBox
    PKT* pOutPacket;
    PKT* pInPacket;

    // Does the xor coder need to be initilized?
    if (n!=n_old)
    {
        n_old=n;
        k_old=n-1;
    }

    // create the k_old packets which are used for the initial state
    for(unsigned char i=0; i<k_old;i++)
    {
        // create empty packet
        pInPacket=new PKT;
        pInPacket->nLengthBytes=0;
        pInPacket->pData=NULL;

        AddMessageToBox(pInBox,(void*)pInPacket);
        xor_coder(pInBox,pOutBox);
        pOutPacket=(PKT*)GetMessageFromBox(pOutBox);
```

```
        free(pOutPacket->pData);
        delete pOutPacket;
    }

    DestroyMessageBox(pInBox);
    DestroyMessageBox(pOutBox);
}
/*****************************************************************************************/



/********************************** The Xor Encoder ***************************************/
/* This is the main routine of this file. It receives data packets in its inbox and puts the*/
/* modified packets in its outbox.  The modified packets have a sequence number, the       */
/* selected n and k (since k=n-1, k could be neglected) and the redundancy added. The basic */
/* flow of the algorithm is as follows. We get the data packet and add the sequence number. */
/* then n and k are added and two bytes to indicated the length of the original data. Then  */
/* the original data follows. If there is redundancy, then two bytes are added to indicate  */
/* the length of the redundancy (if there is no redundacy, then the two bytes are set to    */
/* zero) and the redundancy is added.  The packet is put into the outbox. It is also added  */
/* into a doubly linked list which holds the last k packets. Then the next redundancy packet*/
/* is calculated and stored for appending it to the next data packet and the last entry in  */
/* the linked list is erased since it is no longer required.                                */
void xor_coder(MESSAGEBOX* pInBox, MESSAGEBOX* pOutBox)
{
    static unsigned char i_p=0; //counting the parity packet, its 0 or 1
    static unsigned char SeqNum=0; // counter of the Sequence Numbers
    PKT *pDataListEntry; // pointer to an element in the LL
    PKT *pOutPacket;  // Packet we return
    PKT *pInPacket;   // packet we receive
    static PKT *pParityPacket; // used to store the parity data
    int nByte; // a counter which is used to traverse the data
    static DLLType *pDataList=DLLCreate(); // the DLL used to store the last k data packets

    //get the packet
    pInPacket=(PKT*) GetMessageFromBox(pInBox);
    if(pInPacket==NULL) return; // no packet, no service

    // create the outgoing data packet
    pOutPacket = new PKT;
```

```
// the format is as follows: Byte 1: SeqNum, Byte 2: n_old, Byte 3: k_old
// Byte 4,5 Data Lenght, Byte 6..6+DataLength-1: Data, next two bytes: Redundancy length
// after that its all redundancy
if(i_p) // parity bytes available
{   // yes
    pOutPacket->nLengthBytes=5+pInPacket->nLengthBytes+2+pParityPacket->nLengthBytes;
}
else
{   // no
    pOutPacket->nLengthBytes=5+pInPacket->nLengthBytes+2;
}
pOutPacket->pData=malloc(pOutPacket->nLengthBytes * sizeof(unsigned char));
// write SeqNum
*((unsigned char*)pOutPacket->pData)=SeqNum;
// also add the SeqNum to the out gong packet
pOutPacket->SeqNum=SeqNum;
// increase the counter
SeqNum++;
// write n_old and k which is n-1
*(((unsigned char*)pOutPacket->pData)+1)=n_old;
*(((unsigned char*)pOutPacket->pData)+2)=k_old;
// write the Data Length
*((WORD*)(((unsigned char*)pOutPacket->pData)+3))=(WORD) pInPacket->nLengthBytes;
// copy the incomming data to the outgoing packet
memmove(((unsigned char*)pOutPacket->pData)+5, pInPacket->pData,pInPacket->nLengthBytes );
// if the parity data is available, add that to the outgoing packet
if(i_p)
{
    // write the length of the parity data
    *((WORD*)(((unsigned char*)pOutPacket->pData)+5+pInPacket->nLengthBytes))=
        (WORD) pParityPacket->nLengthBytes;
    // copy parity
    memmove((((unsigned char*)pOutPacket->pData)+5+pInPacket->nLengthBytes+2),
        pParityPacket->pData,pParityPacket->nLengthBytes);
    // the  parity has been read, we free the memory
    free(pParityPacket->pData);
    delete pParityPacket;
    i_p--;
}
else
{
```

```
    // write the length of the parity data, which is zero
    *((WORD*)(((unsigned char*)pOutPacket->pData)+5+pInPacket->nLengthBytes))=
        (WORD)0;
}

// we can now put the data into the out box
AddMessageToBox(pOutBox,(void*) pOutPacket);

// put packets into the LL
DLLInsertAtTail(pDataList,pInPacket);

// do we have k (=n-1) data packets to run the xor coder?
if (DLLCount(pDataList)==k_old)
{   // allocate the space for the parity and set it to zero
    // The space needed for the parity is the maximum of the
    // length of the data lenght's
    pParityPacket=new PKT;
    pParityPacket->nLengthBytes=0;
    DLLMovePtrToHead(pDataList);
    while(pDataListEntry=(PKT*)DLLPeekNextPtr(pDataList))
    {
        if(pDataListEntry->nLengthBytes>pParityPacket->nLengthBytes)
        {
            pParityPacket->nLengthBytes=pDataListEntry->nLengthBytes;
        }
    }
    // add the 2 data length Bytes
    pParityPacket->nLengthBytes+=2;
    pParityPacket->pData=calloc(pParityPacket->nLengthBytes,1);
    i_p++;

    // the outer loop going through the packets
    DLLMovePtrToHead(pDataList);
    while(pDataListEntry=(PKT*)DLLPeekNextPtr(pDataList))
    {   // first do the 2 Data Length Bytes
        for (nByte=0; nByte<2; nByte++)
        {
            *(((unsigned char*)(pParityPacket->pData))+nByte)^=
                *(((unsigned char*)(&(pDataListEntry->nLengthBytes)))+nByte);
        }

        // Now do the loop going through the Data bytes
```

```
            for(nByte=0; nByte<pParityPacket->nLengthBytes-2; nByte++)
            {
                // is zero padding required? Then we don't include it in the XOR
                if (nByte<pDataListEntry->nLengthBytes)
                {
                    *(((unsigned char*)(pParityPacket->pData))+nByte+2)^=
                        *((unsigned char*)(pDataListEntry->pData)+nByte);
                }
            }
        }

        // Remove the Head
        pDataListEntry=(PKT*)DLLRemoveFromHead(pDataList);
        free(pDataListEntry->pData);
        delete pDataListEntry;
    }
}
/*****************************************************************************************/
```



```
/* XOR_CODER.H */

#ifndef XOR_CODER
#define XOR_CODER

#include "types.h"


// the init function needs to be called before the
// encoder can be used.  The parameter n is the
//block length.  I.e., the XOR over n-1 (=k)
// packets is sent with the n-th packet
void init_xor_coder(unsigned char n);

// the coder takes a pointer to PKT from its
// inbox, processes it an puts a pointer to the
// output packet into the outbox.
// The output packet is the original data,
// and the appended redundancy
// Note that some additional info is also
// put out, such as sequence number and lenght info
void xor_coder(MESSAGEBOX* pInBox, MESSAGEBOX* pOutBox);
```

```
#endif

/* XOR_DECODER.CPP */

/*****************************************************************************/
/*                                                                           */
/* xor_decoder.c                                                             */
/*                                                                           */
/* 10/15/97                                                                  */
/*                                                                           */
/* Guido Schuster                                                            */
/* Advanced Technologies                                                     */
/* Carrier Systems Division                                                  */
/* 3COM                                                                      */
/* 1800 W. Central Rd.                                                       */
/* Mount Prospect, Il  60056                                                 */
/*                                                                           */
/* This file is the implementation of a xor based channel decoder for packets. The idea     */
/* behind this scheme is that we calculate a redundancy (parity) packet, using k previous   */
/* packets. This redundancy packet is then appended to the current packet and               */
/* shipped off. Since this is done for every packet and not, as it is commonly done for     */
/* block codes, every block, this results in an overall scheme able to recover up to k      */
/* packet losses in a row. Clearly the bit rate roughly doubles, but the packet rate stays  */
/* constant since the additional information is appended.                                   */
/* The basic algorithm for the efficient decoding of the xor encoded packet stream works as */
/* follows. After the arrival of a packet, we check if we can use the additional information*/
/* to recover a missing data packet. Each packet has a sphere of influence, in the sense    */
/* that it carries the parity frame of the previous k frames and is part of the calculation */
/* for the redundancy of the next k frames.  In other words, having knowledge about this    */
/* frame might enable us to recover a missing packet only within this sphere.  If we can    */
/* recover a packet, then this new information broadens the sphere of influence.  This is   */
/* done until all packets which we can recover are recovered. Note that the xor_decoder.c is*/
/* meant to work hand in hand with the xor_coder.c                                          */
/*****************************************************************************/


/********************************** The Include Files ************************************/
#include "stdlib.h" // the standard include files
#include "malloc.h"
```

```
#include "string.h"
#include "windows.h"
#include "math.h"

#include "types.h"  // the commonly used types
#include "RTOS.h"   // implementation of the mailbox principle
#include "tools.h"  // functions used in more than one file
#include "xor_decoder.h" // the obvious

extern "C"
{
#include "dllist.h" // the doubly linked list
}
/*****************************************************************************/


/********************************** The Global Variables *********************************/
static unsigned int n_old=0, k_old=0; // store the block size n and the message length k=n-1
static int BufferLength_old=0; // target size of buffer in bytes
static unsigned __int32 PrevSeqNum=0; // extended sequence number of previously arrived packet
static DLLType *pList=DLLCreate(); // the doubly linked list storing the data and redundancy
/*****************************************************************************/


/********************************** The Types ********************************************/
struct NODEstruct // node of the dll
{
    PKT* pDataPacket; // store the data
    PKT* pRedundancyPacket; // store the redundancy
    unsigned __int32 SeqNum; // the extended sequence number
};
typedef struct NODEstruct Node_t;
/*****************************************************************************/



/************************************ The Functions ************************************/

/************************************ Put Node *****************************************/
/* The input to the PutNode function is a pointer to a data packet and a pointer to a parity*/
/* packet.  Both packets have the same sequence number and the extended sequence number is  */
/* is the third parameter.  PutNode creates a dll node and puts it into the buffer according*/
```

```
/* to the extended sequence number.  It also returns a pointer to the DLLNode which was     */
/* inserted in the dll to accomodate the new packets.  This pointer is later used as a      */
/* starting point for the packet recover routine.                                           */
DLLNodeType* PutNode(PKT* pDataPacket, PKT* pRedundancyPacket, unsigned __int32 ExtSeqNum)
{
    unsigned __int32 i;  // all purpose counter
    Node_t *pTempNode;  // a temporary node used to count through the dll
    DLLNodeType *pReturn;  // this is where we store the pointer which we return at the end

    // function prototyp, doing the obvious
    void KillNode(Node_t *p);

    // create the node
    Node_t *pNewNode=new Node_t;
    pNewNode->pDataPacket=pDataPacket;
    pNewNode->pRedundancyPacket=pRedundancyPacket;
    pNewNode->SeqNum=ExtSeqNum;

    // check if the DLL exists
    if (!DLLCount(pList))
    {   // empty DLL
        DLLInsertAtHead(pList,pNewNode);
        DLLMovePtrToHead(pList);
        pReturn=DLLGetPtrToPtr(pList);
    }
    else
    {   // DLL exists
        pTempNode=(Node_t*)DLLPeekAtHead(pList);
        unsigned __int32 HeadSeqNum = pTempNode->SeqNum;
        if (HeadSeqNum> pNewNode->SeqNum)
        {   // new node to the right of head, create the empty nodes
            for (i=HeadSeqNum-1; i>pNewNode->SeqNum; i--)
            {   // go through the SeqNum's of the missing nodes
                pTempNode=new Node_t;
                pTempNode->SeqNum=i;
                pTempNode->pDataPacket=NULL;
                pTempNode->pRedundancyPacket=NULL;
                DLLInsertAtHead(pList,pTempNode);
            }
            // Now insert the new node
            DLLInsertAtHead(pList,pNewNode);
            DLLMovePtrToHead(pList);
```

```
    pReturn=DLLGetPtrToPtr(pList);
}
else
{   // node not to the right of head
    pTempNode=(Node_t*)DLLPeekAtTail(pList);
    unsigned __int32 TailSeqNum = pTempNode->SeqNum;
    if (pNewNode->SeqNum > TailSeqNum)
    {   // node to the left of tail, create the empty nodes
        for (i=TailSeqNum+1; i<pNewNode->SeqNum; i++)
        {   // go through the SeqNum's of the missing nodes
            pTempNode=new Node_t;
            pTempNode->SeqNum=i;
            pTempNode->pDataPacket=NULL;
            pTempNode->pRedundancyPacket=NULL;
            DLLInsertAtTail(pList,pTempNode);
        }
        // Now insert the new node
        DLLInsertAtTail(pList,pNewNode);
        DLLMovePtrToTail(pList);
        pReturn=DLLGetPtrToPtr(pList);
    }
    else
    {   // Node between head and tail, find the corresponding node
        DLLMovePtrToHead(pList);
        pTempNode=(Node_t*)DLLPeekAtPtr(pList);
        while(pTempNode->SeqNum!=pNewNode->SeqNum)
        {
            DLLAdvancePtr(pList);
            pTempNode=(Node_t*)DLLPeekAtPtr(pList);
        }
        // the intenal DLL ptr points to the node we are looking for
        // we now exchange that node with the new node
        DLLInsertBeforePtr(pList,pNewNode);
        DLLRetreatPtr(pList);
        pReturn=DLLGetPtrToPtr(pList);
        DLLAdvancePtr(pList);
        pTempNode=(Node_t*)DLLRemoveAtPtr(pList);
        KillNode(pTempNode);
    }
  }
}
return pReturn;
```

```
}
/*****************************************************************************************/



/********************************* Initialization Routine ********************************/
/* This routine must be called before the xor_decoder is envoked, since the block size n and*/
/* the message lenght k are set in here.  Furthermore the desired bufferdepth is also set   */
/* using the BufferLength parameter which is the number of bytes which should be used for   */
/* storing the data and the redundancy packets.  In addition, k empty packets are sent into */
/* the decoder, which is also done at the encoder and hence encoder and decoder are in the  */
/* same state.  This is done so that the first k real packets do not have to be treated     */
/* differently form the other packets.                                                      */
void init_xor_decoder(unsigned char n, int BufferLength)
{
    MESSAGEBOX* pInBox=CreateEmptyMessageBox();
    MESSAGEBOX* pOutBox=CreateEmptyMessageBox();

    PKT* pInPacket;
    PKT* pOutPacket;

    // Does the xor decoder need to be initilized?
    if (n!=n_old)
    {
        n_old=n;
        k_old=n-1;
    }

    // Adjust Buffer Size
    if (BufferLength_old!=BufferLength)
    {
        BufferLength_old=BufferLength;
    }


    // create the k empty packets
    for(unsigned char i=0; i<k_old;i++)
    {
        pInPacket=new PKT;
        pInPacket->nLengthBytes=7;
        pInPacket->pData=malloc(pInPacket->nLengthBytes*sizeof(unsigned char));
        *((unsigned char*)pInPacket->pData)=i;
```

```
            *((unsigned char*)pInPacket->pData+1)=(unsigned char)n_old;
            *((unsigned char*)pInPacket->pData+2)=(unsigned char)(k_old);
            *((WORD*)((unsigned char*)pInPacket->pData+3))=(WORD)0;
            *((WORD*)((unsigned char*)pInPacket->pData+5))=(WORD)0;

            // pass the packet to the xor_decoder
            AddMessageToBox(pInBox,(void*)pInPacket);
            xor_decoder(pInBox,pOutBox);
            pOutPacket=(PKT*)GetMessageFromBox(pOutBox);

            free(pOutPacket->pData);
            delete pOutPacket;
        }

        DestroyMessageBox(pInBox);
        DestroyMessageBox(pOutBox);
    }
    /*****************************************************************************************/
```



```
    /************************************ My_malloc ***************************************/
    /* This routine, together with "AdjustMemory", is used to control the amount of memory    */
    /* used for storing past data packets and past redundancy.  During initialization, the    */
    /* desired number of bytes is stored in the global variable BufferLength_old.Whenever memory*/
    /* is allocated to store the data or the redundancy, my_malloc instead of malloc is called. */
    /* My_malloc behaves like malloc and in addition, it subtracts the used bytes from the    */
    /* available bytes.                                                                       */
    static void *my_malloc(size_t size)
    {
        BufferLength_old-=size;
        return malloc(size);
    }
    /*****************************************************************************************/


    /********************************** Adjust Memory *****************************************/
    /* This function is called after all the processing for a newly arrived packet has been done*/
    /* It simply goes through the doubly linked list, erasing nodes at the head (past) until the*/
```

```
/* size of the buffer (pList) is just below the set target BufferLength_old                */
void AdjustMemory(void)
{
    void KillNode(Node_t *p); // function prototyp
    Node_t *p;
    while(BufferLength_old<=0)
    {
        p=(Node_t*) DLLRemoveFromHead(pList);
        KillNode(p);
    }
}
/*****************************************************************************************/



/********************************* Kill Node *********************************************/
/* a node of the dll is erased,i.e., the memory it occupies is given back                  */
void KillNode(Node_t *p)
{
    if(p!=NULL) // no node, no kill
    {   // kill!
        if(p->pDataPacket!=NULL) // no data, no kill
        {   // kill and keep book
            BufferLength_old+=p->pDataPacket->nLengthBytes;
            free (p->pDataPacket->pData);
            delete p->pDataPacket;
        }
        if(p->pRedundancyPacket!=NULL) // no redundancy, no kill
        {   // kill and keep book
            BufferLength_old+=p->pRedundancyPacket->nLengthBytes;
            free (p->pRedundancyPacket->pData);
            delete p->pRedundancyPacket;
        }
        delete p;
    }
}
/*****************************************************************************************/



/********************************** XOR decoder ******************************************/
```

```
/* This is the main routine of this file.  It is meant to work hand in hand with the        */
/* xor_encoder, using the same block lenght n.  It receives the packets through a inbox     */
/* and returns the data packets and the recovered packets to an outbox.                     */
/* The basic algorithm is like this: The packet is taken from the inbox and the data is     */
/* quickly forwarded to the outbox.  Then the data and the redundancy are entered in the    */
/* buffer structure.  Then the recovery algorithm is run starting at the position where the */
/* data was entered.  If data can be recovered, it is also put into the outbox until no more*/
/* data can be recovered. Note that the sequence number field of the out stream is used     */
void xor_decoder(MESSAGEBOX* pInBox, MESSAGEBOX* pOutBox)
{
    PKT *pInPacket;
    PKT *pOutDataPacket=new PKT;
    PKT *pRedundancyPacket;
    PKT *pDataPacket=new PKT;
    PKT *pParityPacket;

    // First thing we do is to get the data (i.e., the redundancy is not copied)
    // and put it into the OutBox
    pInPacket = (PKT*) GetMessageFromBox(pInBox);
    if (!pInPacket) return; // no packet, no service
    // explicitly add the sequence number
    pOutDataPacket->SeqNum=*((unsigned char*)(pInPacket->pData));
    // figure out the extended sequence number and store it as the previous one
    __int32 ExtSeqNum=ExtendSeqNum(pOutDataPacket->SeqNum,PrevSeqNum);
    PrevSeqNum=ExtSeqNum;
    // read the (n,k)
    n_old =*(((unsigned char*)(pInPacket->pData))+1);
    k_old =*(((unsigned char*)(pInPacket->pData))+2);
    // read the  data length
    pOutDataPacket->nLengthBytes=
        *((WORD*)(((unsigned char*)(pInPacket->pData))+3));
    // Read the length of the redundancy
    int RedundancyLength=
        *((WORD*)(((unsigned char*)(pInPacket->pData))+5+pOutDataPacket->nLengthBytes));
    if (RedundancyLength==0)
    { // No redundancy sent
        pRedundancyPacket=NULL;
    }
    else
    {
    // copy the redundancy
        pRedundancyPacket = new PKT;
```

```
    pRedundancyPacket->nLengthBytes=RedundancyLength;
    pRedundancyPacket->SeqNum=pOutDataPacket->SeqNum;
    pRedundancyPacket->pData=my_malloc(pRedundancyPacket->nLengthBytes * sizeof(unsigned char));
    memmove(pRedundancyPacket->pData,(((unsigned char*)(pInPacket->pData))+5+
    pOutDataPacket->nLengthBytes+2),pRedundancyPacket->nLengthBytes);
}
// copy the data to the out packet
pOutDataPacket->pData=malloc(pOutDataPacket->nLengthBytes * sizeof(unsigned char));
memmove(pOutDataPacket->pData,(((unsigned char*)(pInPacket->pData))+5),pOutDataPacket->nLengthBytes);

// Before we place the data into the outbox, we need a local
// copy, which includes the two data length bytes
pDataPacket->nLengthBytes=pOutDataPacket->nLengthBytes+2;
pDataPacket->SeqNum = pOutDataPacket->SeqNum;
pDataPacket->pData=my_malloc(pDataPacket->nLengthBytes * sizeof(unsigned char));
memmove(pDataPacket->pData,(((unsigned char*)(pInPacket->pData))+3),pDataPacket->nLengthBytes);

// put the data into the OutBox
AddMessageToBox(pOutBox, (void*) pOutDataPacket);

// put the copy of the data and the redundancy into the LL
DLLNodeType *pI = PutNode(pDataPacket, pRedundancyPacket,ExtSeqNum);

// free the memory occupied by InPacket
free(pInPacket->pData);
delete pInPacket;

// recover lost packets
Node_t *pMissingData; // pointer to the missing data
Node_t *pINode; // pointer for the first loop
Node_t *pIINode; // pointer for the second loop
unsigned int i, NumOfMissingData=0;
unsigned __int32 LL; // the lower limit SeqNum for the recovery

// initialize
Node_t *pHead=(Node_t*)DLLPeekAtHead(pList);
DLLSetPtrToPtr(pList,pI);
pINode=(Node_t*)DLLPeekAtPtr(pList);
LL = pINode->SeqNum-k_old<pHead->SeqNum ? pHead->SeqNum : pINode->SeqNum-k_old;

// the main recovery loop
while(pI && pINode->SeqNum>=LL)
```

```
{
    // first we find out how many packets are missing
    NumOfMissingData=0;
    DLLSetPtrToPtr(pList,pI);
    for(i=0; i<k_old && (pIINode=(Node_t*)DLLPeekNextPtr(pList)); i++)
    {
        if(!pIINode->pDataPacket) NumOfMissingData++;
    }

    if(i==k_old && NumOfMissingData==1 &&
        (pIINode=(Node_t*)DLLPeekNextPtr(pList)) && pIINode->pRedundancyPacket)
    {
        // we have the Redundancy and only one packet is lost out of k_old
        // hence we can recover that packet

        // allocate the space for the parity and set it to zero
        // we need as many bytes as the redundancy is long
        pParityPacket=new PKT;
        pParityPacket->nLengthBytes=pIINode->pRedundancyPacket->nLengthBytes;
        pParityPacket->pData=calloc(pParityPacket->nLengthBytes,1);

        DLLSetPtrToPtr(pList,pI);
        for(i=0;i<k_old;i++)  // go through the k last data packets
        {
            pIINode=(Node_t*)DLLPeekAtPtr(pList);
            if(!pIINode->pDataPacket) // no data
            {   // store the pointer to the missing data
                pMissingData=pIINode;
                // reset the pointers since we will recover this packet
                pI=DLLGetPtrToPtr(pList);
                // recalc the lower limit
                LL= LL<=pIINode->SeqNum-k_old ? LL : pIINode->SeqNum-k_old;
                // check the limit
                LL= LL<pHead->SeqNum ? pHead->SeqNum : LL;
            }
            else
            {
                // do the XOR operation for every Byte
                for (int Byte=0; Byte<pIINode->pDataPacket->nLengthBytes;Byte++)
                {
                    *(((unsigned char*)(pParityPacket->pData))+Byte)^=
                        *(((unsigned char*)pIINode->pDataPacket->pData)+Byte);
```

```
        }
    }
    DLLAdvancePtr(pList);
}
// Finally, we need to do the XOR with the redundancy
// do the XOR operation for every Byte
pIINode=(Node_t*)DLLPeekAtPtr(pList);
for (int Byte=0; Byte<pIINode->pRedundancyPacket->nLengthBytes;Byte++)
{
    *(((unsigned char*)(pParityPacket->pData))+Byte)^=
        *(((unsigned char*)pIINode->pRedundancyPacket->pData)+Byte);
}


// build a packet for the recovered data
pMissingData->pDataPacket=new PKT;
// set the correct SeqNum
pMissingData->pDataPacket->SeqNum=
(unsigned char) (pMissingData->SeqNum % 256);
// write the correct length for the data into the structure
// Note that 2 Length bytes are also in the structure
// this adjustment is done later on
pMissingData->pDataPacket->nLengthBytes=*((WORD*)pParityPacket->pData);
// correct for the additional 2 data length bytes
pMissingData->pDataPacket->nLengthBytes+=2;
// copy the data
pMissingData->pDataPacket->pData=
    my_malloc(pMissingData->pDataPacket->nLengthBytes*sizeof(unsigned char));
memmove(pMissingData->pDataPacket->pData,
    pParityPacket->pData,pMissingData->pDataPacket->nLengthBytes);
// free the temp data storage
free(pParityPacket->pData);
delete pParityPacket;

// create a packet to be put into the outbox
pOutDataPacket=new PKT;
// explicitly add the sequence number
pOutDataPacket->SeqNum= (unsigned char) (pMissingData->SeqNum % 256);
// add the length, subtract the 2 length bytes
pOutDataPacket->nLengthBytes=pMissingData->pDataPacket->nLengthBytes-2;
// copy the data, not including the two data length bytes
pOutDataPacket->pData=malloc(pOutDataPacket->nLengthBytes * sizeof(unsigned char));
```

```
            memmove(pOutDataPacket->pData,(((unsigned char*)(pMissingData->pDataPacket->pData))+2),
            pOutDataPacket->nLengthBytes);
            // put the data into the OutBox
            AddMessageToBox(pOutBox, (void*) pOutDataPacket);

        }
        else
        {
            // we cannot recover any data, move the block backwards
            DLLSetPtrToPtr(pList,pI);
            if(DLLRetreatPtr(pList))
            {
                pI=DLLGetPtrToPtr(pList);
                pINode=(Node_t*)DLLPeekAtPtr(pList);
            }
            else
            {
                pI=NULL;
            }
        }

    }

    AdjustMemory();
}
/*****************************************************************************************/


/* XOR_DECODER.H */

#ifndef XOR_DECODER
#define XOR_DECODER

#include "types.h"

void xor_decoder(MESSAGEBOX* pInBox, MESSAGEBOX* pOutBox);

void init_xor_decoder(unsigned char n, int BufferLenght);

#endif
```

CLAIMS

We claim:

1. In a packet switched telecommunications network in which routers route packets based on information contained in the packet headers, a method of encoding packets from a sequence of payload blocks to enable recovery of lost packets, said sequence of payload blocks including a sequential group of *k* payload blocks and a subsequent sequential group of payload blocks, said method characterized by:

deriving *p* redundancy blocks from said sequential group of *k* payload blocks; and

generating *p* packets by a process comprising combining each of said *p* redundancy blocks, respectively, with a payload block in said subsequent sequential group of payload blocks, whereby each one of said *p* packets includes a payload block and one of said *p* redundancy blocks.

2. A method as claims in claim 1, wherein deriving *p* redundancy blocks comprises employing a Reed-Solomon block coder.

3. A method as claimed in claim 1, wherein said subsequent sequential group of payload blocks are the *k* payload blocks next in sequence after said sequential group of *k* payload blocks.

4. A method as claimed in claim 1, wherein said payload blocks cooperatively represent a real-time media signal selected from the group consisting of audio and video.

5. A method as claimed in claim 1, wherein $p=1$

6. A method as claimed in claim 5, wherein deriving a redundancy block from a sequential group of *k* payload blocks comprises computing an XOR sum of said *k* payload blocks.

7. A method as claimed in claim 6, wherein said subsequent sequential group of payload blocks are the *p* payload blocks next in sequence after said sequential group of *k* payload blocks.

8. A method as claimed in claim 1, further characterized by:

transmitting said *p* packets into said network, said *p* packets corresponding in sequence to said sequence of said payload blocks.

9. In a packet switched telecommunications network in which routers route packets based on information contained in the packet headers, an apparatus for encoding packets from a sequence of payload blocks to enable recovery of lost packets, said sequence of payload blocks including a sequential group of *k* payload blocks and a subsequent sequential group of payload blocks, said apparatus having a computer processor and a memory, said apparatus characterized by:

a first set of machine language instructions stored in said memory and executed by said processor for deriving *p* redundancy blocks from said sequential group of *k* payload blocks; and

a second set of machine language instructions stored in said memory and executed by said processor for generating *p* packets by a process comprising combining each of said *p* redundancy blocks, respectively, with a payload block in said subsequent sequential group of payload blocks, whereby each one of said *p* packets includes a payload block and one of said *p* redundancy blocks.

10. An apparatus as claimed in claim 9, wherein said first set of machine language instructions comprises a Reed-Solomon block coder.

11. An apparatus as claimed in claim 9, wherein said subsequent sequential group of payload blocks are the *k* payload blocks next in sequence after said sequential group of *k* payload blocks.

76909-146

12. An apparatus as claimed in claim 9, wherein said payload blocks cooperatively represent a real-time media signal selected from the group consisting of audio and video.

13. An apparatus as claimed in claim 9, wherein p=1.

14. An apparatus as claimed in claim 13, wherein said first set of machine language instructions derives a redundancy block from a sequential group of k payload blocks by computing an XOR sum of said k payload blocks.

15. A method as claimed in claim 14, wherein said subsequent sequential group of payload blocks are the p payload blocks next in sequence after said sequential group of k payload blocks.

16. A method as claimed in claim 9, further characterized by:

a third set of machine language instructions stored in said memory and executed by said processor for transmitting said p packets into said network, said p packets corresponding in sequence to said sequence of said payload blocks.

17. In a packet switched telecommunications network in which routers route packets based on information contained in the packet headers, a method for communicating a payload, said payload being divided into a sequence of payload blocks, PL[k-w], ..., PL[k-2], PL[k-1], PL[k], PL[k+1], …, PL[k+u], said method characterized by:

for each payload block PL[i] in said sequence of payload blocks, generating a forward error correction code FEC[i] comprising an XOR sum of a predetermined number of

preceding payload blocks and appending said forward error correction code FEC[i] to said payload block PL[i], said predetermined number being greater than one;

transmitting a sequence of packets P[k], P[k+1], ..., P[k+u] from a first device in said network for receipt by a second device in said network, wherein each packet

P[i] in said sequence of packets includes payload block PL[i] and forward error code FEC[i].

18. A method as claimed in claim 17, wherein if a packet P[j] is lost in transmission, said second device recreates payload block PL[j] by a process comprising extracting information from one or more packets that follow packet P[j].

19. A method as claimed in claim 18, wherein said predetermined number defines a length of burst error from which said second device may recover said payload.

20. A method as claimed in claim 19, wherein said predetermined number is 2.

21. A method as claimed in claim 19, wherein said predetermined number is 3.

22. A method as claimed in claim 17, wherein said payload represents a real-time media signal.

23. A method as claimed in claim 22, wherein said payload represents a voice signal.

24. A method as claimed in claim 17, further characterized by, in order:

padding a plurality of said payload blocks, to create a plurality of padded payload blocks equal in length to each other; and

generating said forward error correction codes for said padded payload blocks.

25. A method as claimed in claim 17, further characterized by:

applying a sliding window along said sequence of payload blocks and, within said sliding window, generating and appending said forward error correction codes.

76909-146

26. In a packet switched telecommunications network in which routers route packets based on information contained in the packet headers, a method of recovering a lost packet in a sequence of packets transmitted in said network, each packet in said sequence defining a sequence number and carrying a payload block and a redundancy block, said method characterized by:

(a) receiving an incoming packet of said sequence of packets;

(b) establishing a window of analysis beginning with said incoming packet and extending for a predetermined number of packets of said sequence following said incoming packet; and

(c) if only one payload block in said window of analysis has not yet been received, recovering said one payload block by taking and XOR sum of a plurality of payload blocks within said window of analysis.

27. A method as claimed in claim 26, further characterized by:

if all the of the payload blocks in said window of analysis have been received, or if more than one of said payload blocks in said window of analysis have not been received, moving said window of analysis back by one packet so that the window of analysis begins with one packet of said sequence prior to said incoming packet and extends for said predetermined number of packets of said packet following said one packet; and

repeating step (c) of claim 26.

28. In a packet switched telecommunications network in which routers route packets based on information contained

in the packet headers, an apparatus for communicating a payload, said payload being divided into a sequence of payload blocks, PL[k-w], …, PL[k-2], PL[k-1] PL[k], PL[k+1], …,PL[k+u], said apparatus having a computer processor and a memory, said apparatus characterized by:

a first segment for generating, for each payload block PL[i] in said sequence of payload blocks, a forward error correction code FEC[i] comprising an XOR sum of a predetermined number of preceding payload blocks and appending said forward error correction code FEC[i] to said payload block PL[i], said predetermined number being greater than one; and

a second segment for transmitting a sequence of packets P[k], P[k+1], ..., P[k+u] from a first device in said network for receipt by a second device in said network, wherein each packet P[i] in said sequence of packets includes payload block PL[i] and forward error code FEC[i].

29. An apparatus as claimed in claim 28, wherein said first segment comprises a set of machine language instructions stored in said memory and executable by said computer processor.

30. An apparatus as claimed in claim 28, further characterized by:

a third segment operated by said second device for recreating a lost payload block PL[j] by a process comprising extracting information from one or more other packets in said sequence of packets.

31. An apparatus as claimed in claim 30, wherein said third segment comprises a set of machine language instructions stored in said memory and executable by said computer processor.

32. An apparatus as claimed in claim 28, wherein said predetermined number defines a maximum number of lost payload blocks in a row that may be recovered.

33. An apparatus as claimed in claim 28, wherein said payload represents a real-time media signal.

34. An apparatus as claimed in claim 33, wherein said payload represents a voice signal.



Fig. 1

Fig. 2

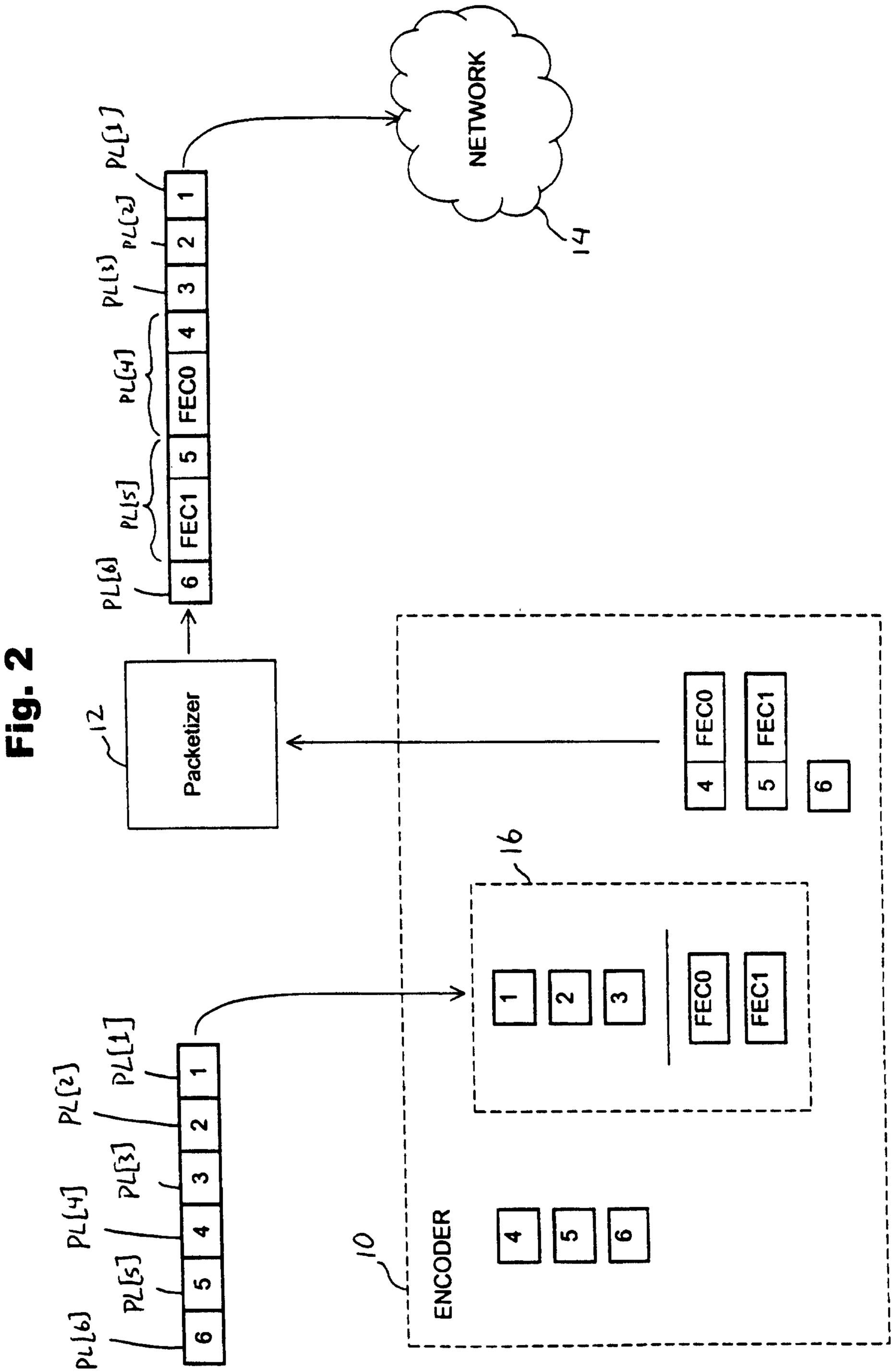

ENCODER
10
1
2
3
FEC0
FEC1
16
4
5
6
Packetizer
12
PL[1]
PL[2]
PL[3]
PL[4]
PL[5]
PL[6]
NETWORK
14

Fig. 3

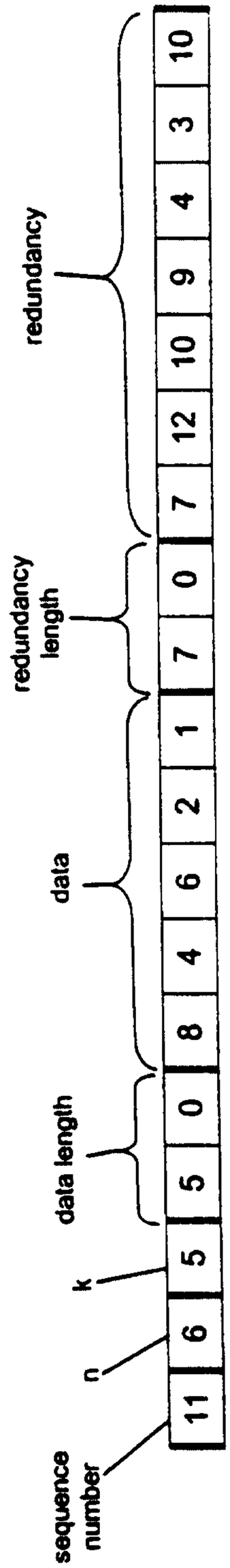
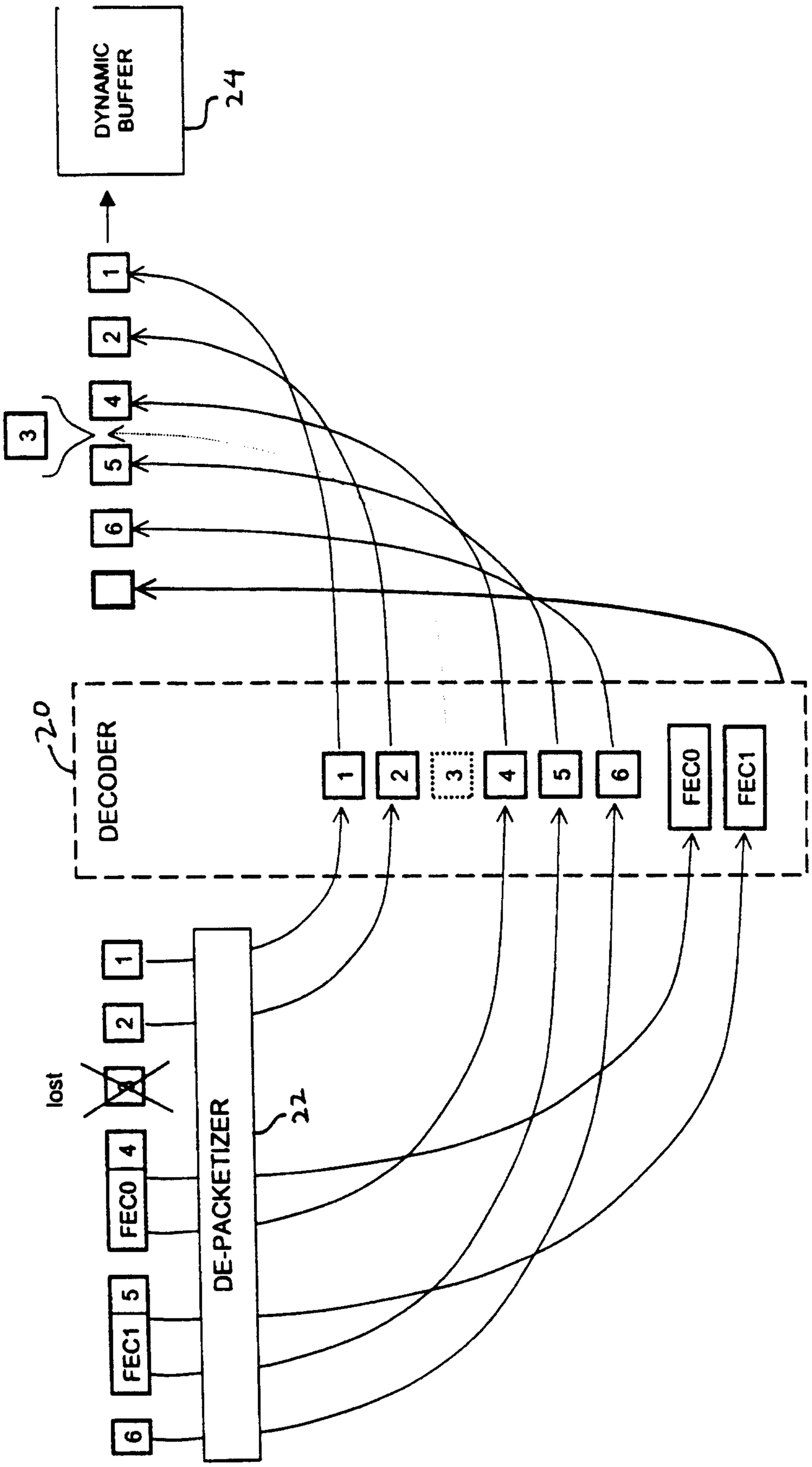

redundancy
redundancy length
data
data length
k
n
sequence number
1
7
0
7
12
10
9
4
3
10
2
6
4
8
0
5
5
6
11

Fig. 4


DYNAMIC BUFFER
24
1
2
3
4
5
6
20
DECODER
1
2
3
4
5
6
FEC0
FEC1
lost
1
2
4
FEC0
FEC1
5
6
22
DE-PACKETIZER

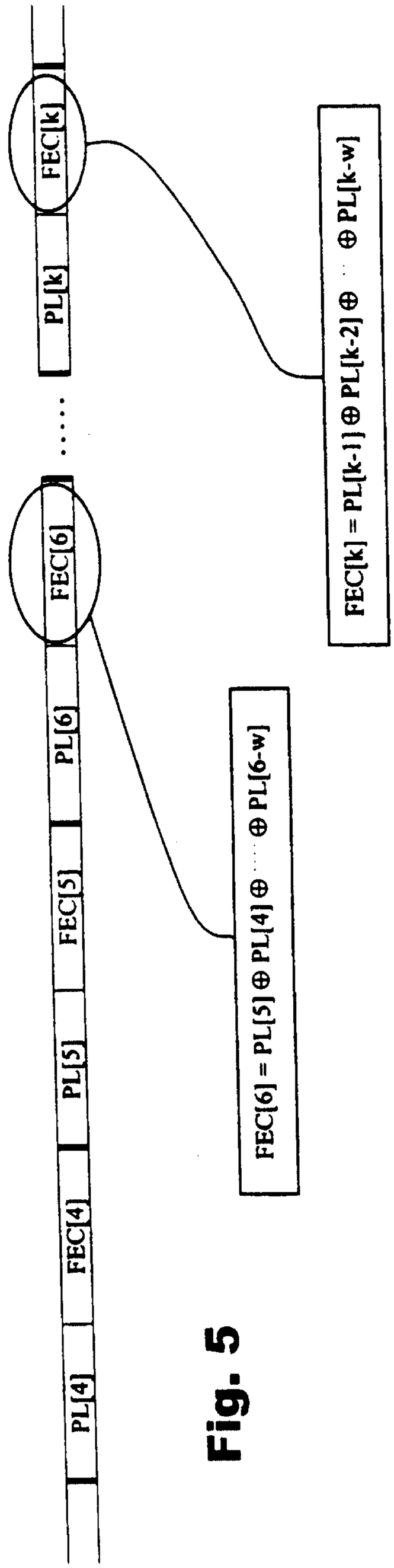

FEC[k]
PL[k]
FEC[6]
PL[6]
FEC[5]
PL[5]
FEC[4]
PL[4]
⊕ PL[k-w]
FEC[k] = PL[k-1] ⊕ PL[k-2] ⊕ ⊕
FEC[6] = PL[5] ⊕ PL[4] ⊕ ⊕ PL[6-w]

Fig. 5

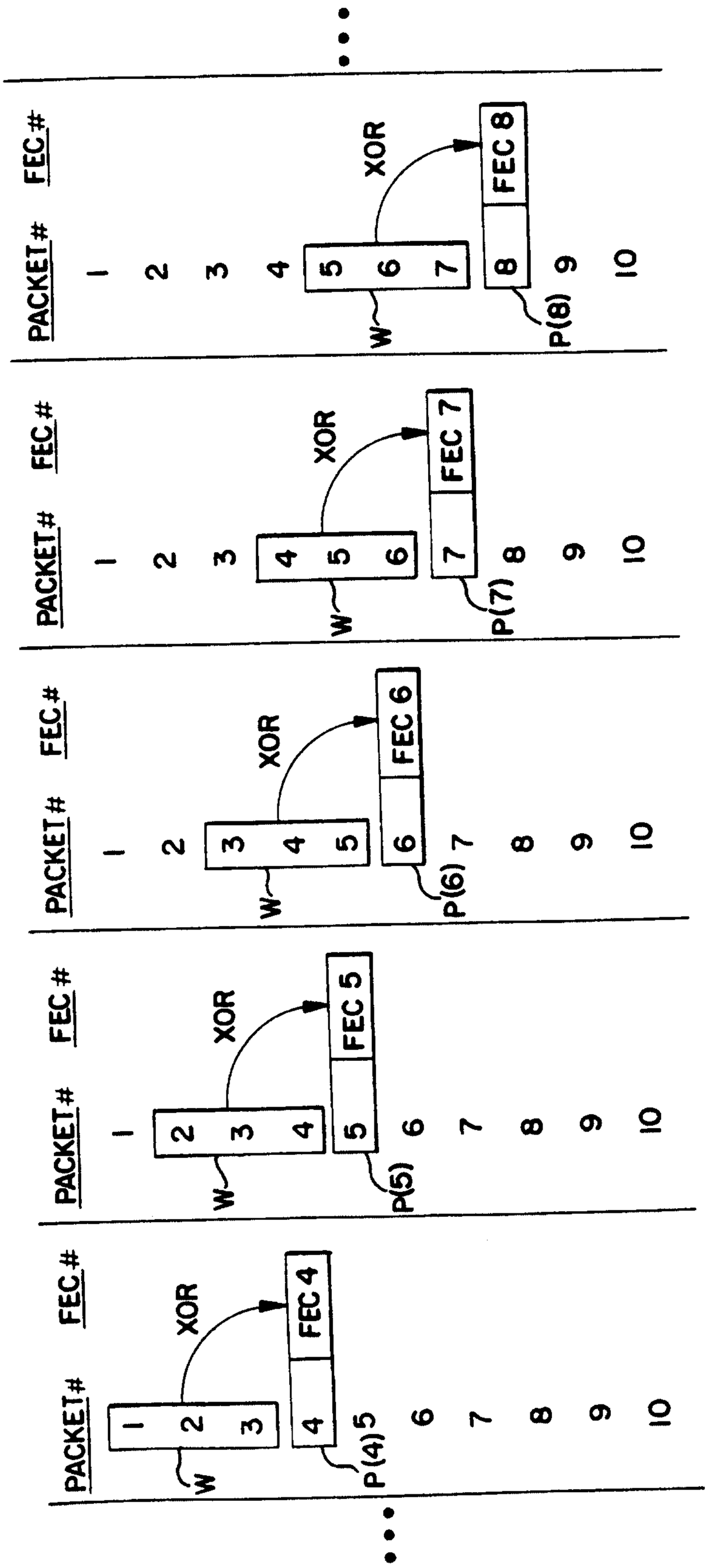

PACKET#
FEC#
1
2
3
4
5
6
7
8
9
10
W
XOR
FEC 8
P(8)
FEC 7
P(7)
FEC 6
P(6)
FEC 5
P(5)
FEC 4
P(4)

Fig. 6

Fig. 7

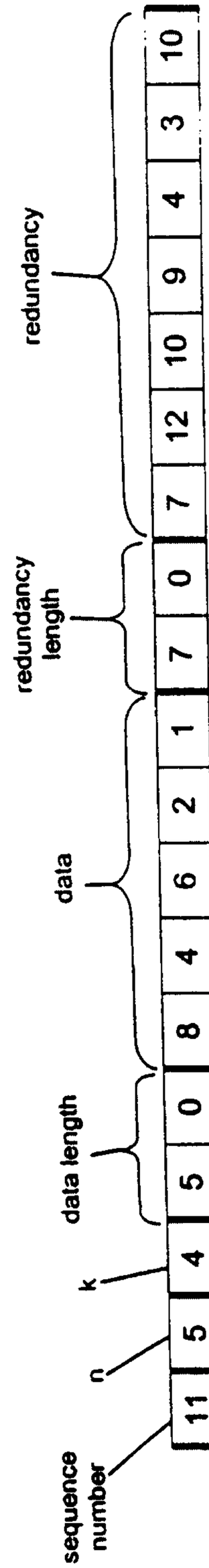

redundancy
redundancy length
data
data length
k
n
sequence number
1
7
0
7
12
10
9
4
3
10
2
6
4
8
0
5
4
5
11

Fig. 8

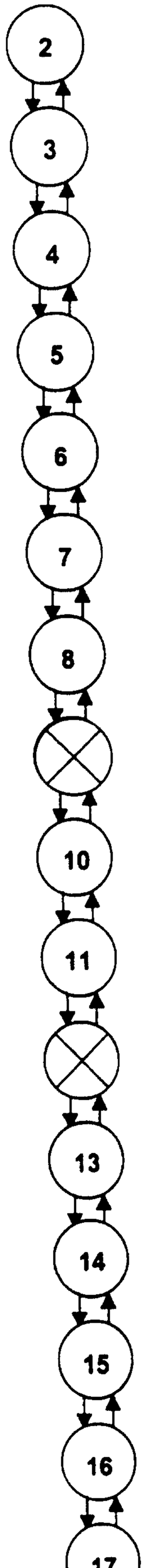


Fig. 9

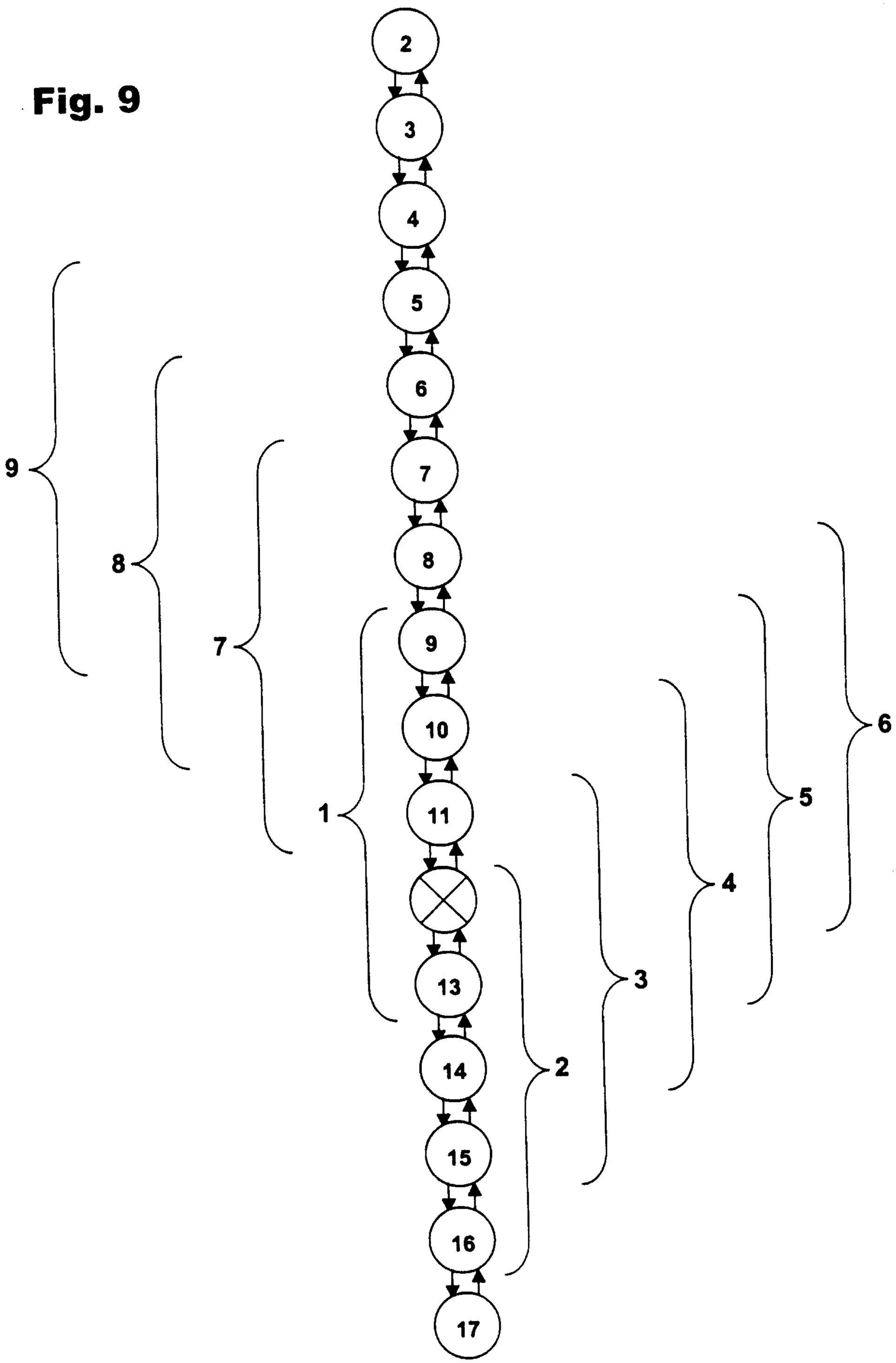
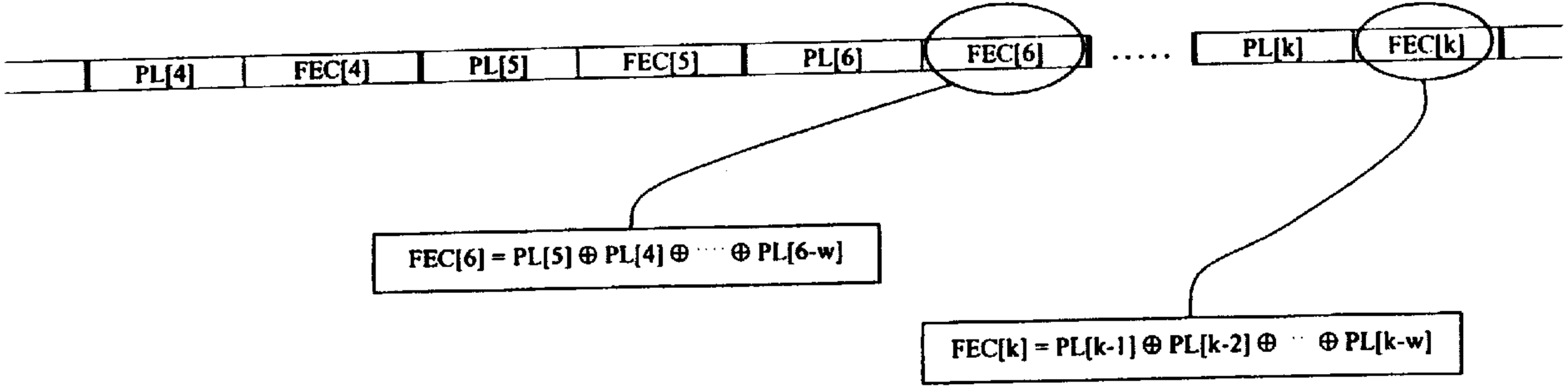


Fig. 10

Sequence of
Windows of Analysis

Step	Action	Window of Analysis
1	Recover lost data block 12	9 to 13
2	Move window to bubbles 12-16	12 to 16
3	No recovery	11 to 15
4	No recovery	10 to 14
5	No recovery	9 to 13
6	No recovery; Complete "inner loop"	8 to 12
7	No recovery	7 to 11
8	No recovery	6 to 10
9	No recovery; Complete "outer loop"	5 to 9


PL[4]
FEC[4]
PL[5]
FEC[5]
PL[6]
FEC[6]
.....
PL[k]
FEC[k]
FEC[6] = PL[5] ⊕ PL[4] ⊕ ⋯ ⊕ PL[6-w]
FEC[k] = PL[k-1] ⊕ PL[k-2] ⊕ ⋯ ⊕ PL[k-w]