AMENDMENT

ETS 300 075

A1

November 1995

Source: ETSI TC-TE

Reference: RE/TE-01058

ICS: 33.040

Key words: processable data, file transfer

This amendment A1 modifies
the European Telecommunication Standard ETS 300 075 (1994) - Second Edition

Terminal Equipment (TE);
Processable data;
File transfer

ETSI

European Telecommunications Standards Institute

ETSI Secretariat

Postal address: F-06921 Sophia Antipolis CEDEX - FRANCE
Office address: 650 Route des Lucioles - Sophia Antipolis - Valbonne - FRANCE
X.400: c=fr, a=atlas, p=etsi, s=secretariat - **Internet:** secretariat@etsi.fr

Tel.: +33 92 94 42 00 - Fax: +33 93 65 47 16

Copyright Notification: No part may be reproduced except as authorized by written permission. The copyright and the foregoing restriction extend to reproduction in all media.

© European Telecommunications Standards Institute 1995. All rights reserved.

New presentation - see History box

Whilst every care has been taken in the preparation and publication of this document, errors in content, typographical or otherwise, may occur. If you have comments concerning its accuracy, please write to "ETSI Editing and Committee Support Dept." at the address shown on the title page.

Foreword

This amendment to ETS 300 075 (1994), second edition, has been produced by the Terminal Equipment (TE) Technical Committee of the European Telecommunications Standards Institute (ETSI).

Transposition dates	
Date of adoption of this ETS:	13 October 1995
Date of latest announcement of this ETS (doa):	29 February 1996
Date of latest publication of new National Standard or endorsement of this ETS (dop/e):	31 August 1996
Date of withdrawal of any conflicting National Standard (dow):	31 August 1996

Amendments

The amendments are as follows:

Page 15, clause 2, normative references. Add CCITT Recommendation T.51 (1992) as reference [8].

[8] CCITT Recommendation T.51 (1992): "Latin based coded character sets for telematic services".

Page 54, subclause 5.5.1.28. For applications that display the progress of the file transfer and calculate the remaining time, it would be helpful to know the exact amount of physical data to transfer. This can be achieved by an optional attribute in the file header. Subclause 5.5.1.28 is to be renumbered as subclause 5.5.1.29 with the following being inserted:

5.5.1.28 Length of file during transmission

This attribute indicates the size in bytes of the file content in compressed form. If the file is transferred in uncompressed form, this parameter, when present, shall be set equal to the file length parameter.

Modify table 2 in the re-numbered subclause 5.5.1.29 (status of file attributes) as follows:

Table 2

File Attributes	Application		Default Value
	Telesoftware	Printer	
File type	O	O	Text file
Execution order	O	-	Don't care
Transfer name	O	O	No
Filename	O	O	No
Date of last modification	O	O	No
File length	O	O	No
Destination code	O	-	Don't care
File coding	O	O	Depends on file type
Destination name	O	-	No
Cost	O	O	No
User field/Application reference	O	O	No
Load address (abs.)	O	-	No
Execute address (abs.)	O	-	No
Execute address (rel.)	O	-	No
Compression mode	O	O	No compression
Device	O	O	Don't care
File checksum	O	O	No
Author name	O	O	No
Future file length	O	O	No
Permitted actions	O	O	No
Legal qualification	O	O	No
Creation	O	O	No
Last read access	O	O	No
Identity of the last modifier	O	O	No
Identity of the last reader	O	O	No
Recipient	O	O	No
Telematic file transfer version	O	O	No
Length of file during transmission	O	O	No
O: Optional -: Irrelevant NOTE: Other attributes may be added to take into account the auxiliary device application requirements.			

Page 54, subclause 5.6.1, third and fifth paragraphs, replace the reference to subclause 5.5.1.17 by subclause 5.5.1.29 to read as follows:

All the attributes listed in subclause 5.5.1.29 may be used to characterize a file involved in a telesoftware application.

The attributes listed in subclause 5.5.1.29 may be used to characterize a file involved in a printer device application.

Page 66, subclause 6.2.5.1, tenth line, delete the bullet item "user abort of the access regime;" to read as shown below.

6.2.5.1 Content of the T-End-Access TDU and the associated response

T-End-Access

Reason
User data

T-Response-positive

User data

The use of the parameters is described in the service definition.

The Reason parameter in T-End-Access takes one of the following values:

- termination of the access regime requested by the service user:
  - reason not specified (default value);
  - insufficient primitives handled;
  - other reason (this last value may be followed by a string of no more than 62 displayable characters).

Page 78, subclause 7.1.1.1, amend the sentence to replace "table 5" by "table 4":

The first byte of a TDU identifies the TDU according to the coding in table 4.

Page 78, subclause 7.1.1.3, amend the first bullet point to replace "table 6" by "table 5":

- Parameter Identifier field (PI): a single byte identifying the parameter. The coding of PIs is specified in table 5;

Page 80, subclause 7.1.2.1.2, amend the final sentence on page 81 to read:

A T-associate response coded on one byte indicates that only the Basic kernel is supported by the receiver.

Page 94, table 6, subclause 7.3.2, modify the table to read as follows:

Table 6: Coding of file attributes

Attribute	PI
File type	2/0
Execution order	2/1
Transfer name	2/2
File name	2/3
Date	2/4
File length	2/5
Destination code	2/6
File coding	2/7
Destination name	2/8
Cost	2/9
User field/Application reference	2/10
Load address	2/11
Execute address (absolute)	2/12
Execute address (relative)	2/13
Compression mode	2/14
Device	2/15
File checksum	3/0
Author name	3/1
Future file length	3/2
Permitted actions	3/3
Legal qualification	3/4
Creation	3/5
Last read access	3/6
Identity of the last modifier	3/7
Identity of the last reader	3/8
Recipient	3/9
Telematic file transfer version	3/10
Length of file during transmission	3/11

Page 96, subclause 7.3.2.8. The possible use of the file transfer protocol outside the Videotex area requires the identification of new contents and their encodings. One of them is the facsimile encoding according to ITU-T Recommendations T.4 and T.6.

For this purpose, extend the text file list in subclause 7.3.2.8 as follows:

Text file:	5/0	other code,
	5/1	As for 3/1 above (default value),
	5/2	Videotex code profile 2 (as defined by ETS 300 072 [2]),
	5/3	Videotex code profile 1 (as defined by ETS 300 072 [2]),
	5/4	Videotex code profile 3 (as defined by ETS 300 072 [2]),
	5/7	geometric,
	5/6	photographic,
	5/7	sound.
facsimile coded file:	6/0	ITU-T Recommendation T.4 encoding schemes,
	6/1	ITU-T Recommendation T.6 encoding schemes.

Subsequent bytes, if present, for "6/0" and "6/1" values are coded as follows:

The following byte identifies the compression mode:

2/0 no compression,
2/1 monodimensional compression -T.4 (default),
2/2 bidimensional compression with K=2 -T.4,
2/3 bidimensional compression with K=4 -T.4,
2/4 bidimensional compression with K=infinite -T.6.

The following byte identifies the paper length:

2/10 ISO A4 (default),
2/11 ISO B4,
2/12 ISO A3,
2/13 unlimited.

The following byte identifies the horizontal resolution:

3/0 R8 = 1 728 pels/215 mm for ISO A4 (default),
3/1 R8 = 2 048 pels/255 mm for ISO B4,
3/2 R8 = 2 432 pels/303 mm for ISO A3,
3/3 R16 = 3 456 pels/215 mm for ISO A4 (default),
3/4 R16 = 4 096 pels/255 mm for ISO B4,
3/5 R16 = 4 864 pels/303 mm for ISO A3.

The following byte identifies the vertical resolution:

4/0 3,85 lines/mm,
4/1 7,7 lines/mm (default),
4/2 15,4 lines/mm.

The following byte identifies inch-based resolutions:

5/0 200*200 pels/25,4 mm,
5/1 240*240 pels/25,4 mm,
5/2 300*300 pels/25,4 mm,
5/1 400*400 pels/25,4 mm.

Page 101, insert the following new subclause 7.3.2.28:

7.3.2.28 Length of the file during transmission

PI compressed file length	= 3/11
LI	= n < 9
PV	= n bytes

The value is a variable length parameter in which the actual number of bytes in the compressed file (the file length after performing the compression algorithm) is coded in an absolute binary form using all eight bits of each byte. If the parameter is coded in more than one byte, the first byte to be transmitted contains the most significant bits.

Page 106, final paragraph of subclause 7.4.3, modify the text to read as follows:

An elementary word is a byte sequence of no more than 12 bytes. Each byte can take any value according to CCITT Recommendation T.51 [7] in the range 2/1 and 7/14 excepted 2/8, 2/9, 2/11 and 2/15, moreover this word cannot contain more than one byte 2/10 (displayed "*").

Page 106, subclause 7.4.4, re-name the title as follows:

7.4.4 Designation in T-load, T-Save, T-Rename, T-Delete and NewName in T-Rename

Page 126, annex A, delete the current text and replace with the following:

Annex A (informative): A compression algorithm

This compression is based on two compression algorithms named LZSS and adaptive Huffman compression. The LZSS is a dictionary based algorithm, and Huffman reduces the most frequent characters.

The following set of files is an example

- Lzhuf.c , compression algorithm
- Huftbl.c, huffman Tables
- Crc32.c, CRC calculation
- all.h, general.h, ansi.h, include files

```
/*==========================================================================*
 * lzhuf.c                                                                  *
 *      LZSS and adaptive Huffman data compression                          *
 *==========================================================================*/
/*
 * written by Haruyasu Yoshizaki 11/20/1988
 * some minor changes 4/6/1989
 * comments translated by Haruhiko Okumura 4/7/1989
 * modified for Btx-FIF by InfoTeSys GmhH 1991-11-04
 *     - use malloc() instead of huge static areas
 *     - files have to be opened outside of this module
 *     - FCS-check included
 *     - length field "long(Filesize)" has been removed from file
 * modified by Orsenna 15/06/1994
 *     - comments added
 *     - some minor changes
 */
/*==============*
 * HEADER FILES *
 *==============*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "all.h"

/*===========*
 * CONSTANTS *
 *===========*/
#ifndef EXIT_FAILURE
#define EXIT_FAILURE    0
#define EXIT_SUCCESS    1
#endif
#ifndef SEEK_SET
#define SEEK_SET    0
#define SEEK_END    2
#endif
/*
 * DIC_BIT_COUNT is the number of bits to encode a position in the dictionary
 */
#define DIC_BIT_COUNT   12
/*
 * The buffer size is equal to 1 << 12 = 4096
 */
#define WINDOW_SIZE     (1 << DIC_BIT_COUNT)
```

```
/*
 * Lookahead buffer size (needs 6 bits) ; used for length encoding so
 * LENGTH_BIT_COUNT = 6
 */
#define LOOK_AHEAD_SIZE 60
/*
 * Threshold is a number to see from which length it's better to encode
 * with an index/length (12+6=18bits) or only a separate character (8 bits).
 * THRESHOLD = ((1+DIC_BIT_COUNT+LENGTH_BIT_COUNT)/9)
 * The length must be > to THRESHOLD to encode with an index/length.
 */
#define THRESHOLD       2
/*
 * NIL is used by the leafs of the tree.
 */
#define NIL             WINDOW_SIZE
#define TEXT_BUF_LEN    (WINDOW_SIZE + LOOK_AHEAD_SIZE - 1)
/*
 * kinds of characters (character code = 0..N_CHAR-1) :
 * # 256 'standard' codes (0..255)
 * # LOOK_AHEAD_SIZE is used for length codes
 * # - THRESHOLD because we start encoding length codes from THRESHOLD
 */
#define N_CHAR          (256 - THRESHOLD + LOOK_AHEAD_SIZE)
/*
 * Size of table
 */
#define TABLE_SIZE      (N_CHAR * 2 - 1)
/*
 * Position of root
 */
#define ROOT_NODE       (TABLE_SIZE - 1)
/*
 * MAX_FREQ is used to update tree when the root frequency comes to
 * this value.
 */
#define MAX_FREQ        0x8000
/*==================*
 * STATIC VARIABLES *
 *==================*/
static FILE  *inFile, *outFile;
static int32  inpSize, outSize;
static char   writeError[] = "Can't write.";
static uint8  *textBuffer;
static uint16  matchPosition;      /* best position */
static int     matchLength;        /* best length   */
static int16  *leftSon;            /* [WINDOW_SIZE + 1]; */
/*
 * [WINDOW_SIZE+1..WINDOW_SIZE + 257 - 1] is used to conserve roots
 */
static int16  *rightSon;           /* [WINDOW_SIZE + 257]; */
static int16  *dad;                /* [WINDOW_SIZE + 1];   */
/*======================*
 * STRUCTURE DEFINITIONS *
 *======================*/
/*
 * You can change the input and output buffer size, it's faster with big
 * buffer size.
 */
typedef struct LzhEnvStruct {
   uint8    inputBuffer[2048];              /* input buffer : 2Ko */
   int      inputPosition;
   int      inputMax;
```

```
    uint8    outputBuffer[1024];              /* output buffer : 1Ko */
    int      outputPosition;
    int      outputMax;
    uint16   freq[TABLE_SIZE + 1];            /* frequency table */
   /*
    * pointers to parent nodes, except for the
    * elements [TABLE_SIZE..TABLE_SIZE + N_CHAR - 1] which are used to get
    * the positions of leaves corresponding to the codes.
    */
    int      parent[TABLE_SIZE + N_CHAR];
    /*
     * pointers to child nodes (son[], son[] + 1)
     */
    int      son[TABLE_SIZE];
} LzhEnv;
static LzhEnv *lzhEnv;
/*==============================*
 * INTERNAL PROCEDURE prototypes *
 *==============================*/
static void     Error _ANSI_ARGS_((char *message));
static int16    InputByte _ANSI_ARGS_((void));
static int      OutputByte _ANSI_ARGS_((int16 byte_to_output));
static void     InitTree _ANSI_ARGS_((void));
static void     InsertNode _ANSI_ARGS_((int16 new_node));
static void     DeleteNode _ANSI_ARGS_((int p));
static int16    GetBit _ANSI_ARGS_((void));
static int16    GetByte _ANSI_ARGS_((void));
static void     PutCode _ANSI_ARGS_((int16 l, uint16 c));
static void     StartHuffman _ANSI_ARGS_((void));
static void     Reconst _ANSI_ARGS_((void));
static void     UpDate _ANSI_ARGS_((int c));
static void     EncodeChar _ANSI_ARGS_((unsigned c));
static void     EncodePosition _ANSI_ARGS_((unsigned position));
static void     EncodeEnd _ANSI_ARGS_((void));
static int      DecodeChar _ANSI_ARGS_((void));
static int      DecodePosition _ANSI_ARGS_((void));
static void     LzhEncode _ANSI_ARGS_((uint32 *encode_crc_32));
static void     LzhDecode _ANSI_ARGS_((long filesize, uint32 *decode_crc_32));
/*=============================================================================*
 * Error
 *      An error handler.
 * RETURNS:     nothing
 *=============================================================================*/
static void
Error(message)
    char *message;
{
    printf("\r\n%s\n", message);
    exit(EXIT_FAILURE);
}
/*=============================================================================*
 * InputByte
 *      Input a byte from a file.
 * RETURNS:     a byte
 *=============================================================================*/
static int16
InputByte()
{
    if (lzhEnv->inputPosition >= lzhEnv->inputMax) {
        lzhEnv->inputMax = fread((uint8 *) lzhEnv->inputBuffer, 1,
                                 lzhEnv->inputMax, inFile);
        if (lzhEnv->inputMax == 0) {
            return EOF;
```

```
        }
        lzhEnv->inputPosition = 0;
        inpSize += lzhEnv->inputMax;
        printf("\r %6ld\r", inpSize);
    }
    return(lzhEnv->inputBuffer[lzhEnv->inputPosition++]);
}
/*============================================================================*
 * OutputByte
 *      Output a byte to a file.
 * RETURNS:     1 if no error
 *============================================================================*/
static int
OutputByte(byte_to_output)
    int16 byte_to_output;
{
    if (lzhEnv->outputPosition == lzhEnv->outputMax) {
        lzhEnv->outputPosition = fwrite((uint8 *) lzhEnv->outputBuffer,
                                1, lzhEnv->outputPosition, outFile);
        outSize += lzhEnv->outputPosition;
        lzhEnv->outputPosition = 0;
        printf("\r\t\t\t\t\t  %6ld\r", outSize);
    }
    lzhEnv->outputBuffer[lzhEnv->outputPosition++] = (uint8) byte_to_output;
    return 1;
}
/*============================================================================*
 * InitTree
 *      Initialize LZSS tree.
 * RETURNS:     nothing
*============================================================================*/
static void
InitTree()
{
    register int i;
    /*
     * [WINDOW_SIZE+1..WINDOW_SIZE + 256] is used to conserve roots.
     */
    for (i = WINDOW_SIZE + 1; i <= WINDOW_SIZE + 256; i++) {
        rightSon[i] = NIL;
    }
    for (i = 0; i < WINDOW_SIZE; i++) {
        dad[i] = NIL;               /* node */
    }
}
/*============================================================================*
 * InsertNode
 *      Insert a node to tree; find the best match.
 * RETURNS:     nothing
 *============================================================================*/
static void
InsertNode(new_node)
    register int16 new_node;
{
    register int16  p;
    int             i, cmp;
    uint8           *key;
    unsigned        c;
    cmp  = 1;
    /*
     * Key is a pointer to the current string.
     */
    key  = &textBuffer[new_node];
```

```
/*
 * To start, p is a root used to find where this character (key[0])
 * already appared in the dictionary. We use WINDOW_SIZE + 1 + key[0]
 * because rightSon from WINDOW_SIZE + 1 is used to conserve roots.
 */
p    = WINDOW_SIZE + 1 + (int16) key[0];
rightSon[new_node] = NIL;
leftSon[new_node]  = NIL;
matchLength = 0;
for ( ; ; ) {
    /*
     * If the difference between the current string and the best string
     * found (for the moment) is greater than 0, we seek for another
     * string in checking rightSon.
     */
    if (cmp >= 0) {
        if (rightSon[p] != NIL) {
            p = rightSon[p];
        } else {                    /* add a node */
            /*
             *      p (dad[new_node])
             *      |
             *      ---- new_node (rightSon[p])
             */
            rightSon[p] = new_node;
            dad[new_node]  = p;
            return;
        }
    } else {
        if (leftSon[p] != NIL) {
            p = leftSon[p];
        } else {
            leftSon[p] = new_node;
            dad[new_node]  = p;
            return;
        }
    }
    for (i = 1; i < LOOK_AHEAD_SIZE; i++) {
        /*
         * Compare the current string with the position where this
         * string already appared.
         * key is the current string, p is the better position.
         */
        if ((cmp = key[i] - textBuffer[p + i]) != 0) {
            break;    /* Exit for(...) if not equal char */
        }
    }
    /*
     * i is the corresponding length find from the current string
     */
    if (i > THRESHOLD) {
        if (i > matchLength) {   /* i is > to the current best length ? */
            /*
             * matchPosition is the position difference from the current
             * character.
             */
            matchPosition = (uint16) (((new_node - p) &
                                (WINDOW_SIZE - 1)) - 1);
            if ((matchLength = i) >= LOOK_AHEAD_SIZE) {
                break;
            }
        }
        if (i == matchLength) {  /* i is = to the current best length */
```

```
            if ((c = ((new_node - p) & (WINDOW_SIZE - 1)) - 1)
                    < matchPosition) {
                matchPosition = (uint16) c;
            }
        }
    }
    }
    /*
     * Set the new node equal to the node of the best position found.
     */
    i = dad[p];
    dad[new_node]      = (uint16) i;
    leftSon[new_node]  = leftSon[p];
    rightSon[new_node] = rightSon[p];
    dad[leftSon[p]]    = new_node;
    dad[rightSon[p]]   = new_node;
    if (rightSon[i] == p) {     /* if p is on the right */
        rightSon[i] = new_node;
    } else {                    /* p is on the left */
        leftSon[i]  = new_node;
    }
    /*
     * remove the old node p (best position found)
     */
    dad[p] = NIL;
}
/*=============================================================================*
 * DeleteNode
 *      Remove a node from tree.
 * RETURNS:    nothing
 *=============================================================================*/
static void
DeleteNode(p)
    register int p;
{
    register int replacement;
    if (dad[p] == NIL) {
        return;                 /* not registered ! */
    }
    if (rightSon[p] == NIL) {
        /*
         * Right link is null so we'll pull the non-null (left) link up one
         * to replace the existing link.
         */
        replacement = leftSon[p];
    } else {
        if (leftSon[p] == NIL) {
            /*
             * Left link is null so we'll pull the non-null (right) link
             * up one to replace the existing link.
             */
            replacement = rightSon[p];
        } else {
            /*
             * Both links exist, we instead delete the next link in order,
             * which is guaranteed to have a null link, then replace the node
             * to be deleted with the next link.
             */
            replacement = leftSon[p];
            if (rightSon[replacement] != NIL) {
                do {
                    replacement = rightSon[replacement];
                } while (rightSon[replacement] != NIL);
```

```
            rightSon[dad[replacement]] = leftSon[replacement];
            dad[leftSon[replacement]]  = dad[replacement];
            leftSon[replacement]       = leftSon[p];
            dad[leftSon[p]]            = (int16) replacement;
         }
         rightSon[replacement]      = rightSon[p];
         dad[rightSon[p]]           = (int16) replacement;
      }
   }
   dad[replacement] = dad[p];
   if (rightSon[dad[p]] == p) {    /* p is a right child */
      rightSon[dad[p]] = (int16) replacement;
   } else {                        /* p is a left child */
      leftSon[dad[p]]  = (int16) replacement;
   }
   dad[p] = NIL;
}
static uint16 getbuf = 0;
static int16  getlen = 0;
/*============================================================================*
 * GetBit
 *      Get one bit.
 * RETURNS:     the bit needed !
 *============================================================================*/
static int16
GetBit()
{
   register int16 i, j;
   j = getlen;
   while (j <= 8) {
      if ((i = InputByte()) == EOF) {
         i = 0;
      }
      getbuf |= (i << (8 - j));
      j += 8;
   }
   i = getbuf;
   getbuf <<= 1;
   getlen = --j;
   return((int16) (i < 0));
}
/*============================================================================*
 * GetByte
 *      Get one byte.
 * RETURNS:     a byte
 *============================================================================*/
static int16
GetByte()
{
   register uint16 i;
   register int16  j;
   j = getlen;
   while (j <= 8) {
      if ((i = InputByte()) == (unsigned) EOF) {
         i = 0;
      }
      getbuf |= i << (8 - j);
      j += 8;
   }
   i = getbuf;
   getbuf <<= 8;
   getlen = (int16) (j - 8);
   return(i >> 8);
```

```
}
static uint16 putbuf = 0;
static int16  putlen = 0;
/*============================================================================*
 * PutCode
 *      Output a code.
 * RETURNS:     nothing
 *============================================================================*/
static void
PutCode(l, c)
    int16 l;                /* length */
    register uint16 c;      /* code   */
{
    register int16 j;
    j = putlen;
    putbuf |= c >> j;
    if ((j += l) >= 8) {
        if (! OutputByte((int16) (putbuf >> 8))) {   /* HIBYTE putbuf */
            Error(writeError);
        }
        if ((j -= 8) >= 8) {
            if (! OutputByte((int16) putbuf)) {      /* LOBYTE putbuf */
                Error(writeError);
            }
            j       -= 8;
            putbuf    = (uint16) (c << (l - j));
        } else {
            putbuf <<= 8;
        }
    }
    putlen = j;
}
/*============================================================================*
 * StartHuffman
 *      Initialization of Huffman's tree.
 * RETURNS:     nothing
 *============================================================================*/
static void
StartHuffman()
{
    register int i, j;
    leftSon     = calloc(1, (WINDOW_SIZE +   1) * sizeof(int16));
    rightSon    = calloc(1, (WINDOW_SIZE + 257) * sizeof(int16));
    dad         = calloc(1, (WINDOW_SIZE +   1) * sizeof(int16));
    textBuffer  = calloc(1, TEXT_BUF_LEN);
    memset(textBuffer, ' ', (WINDOW_SIZE - LOOK_AHEAD_SIZE));
    lzhEnv      = calloc(1, sizeof(struct LzhEnvStruct));
    lzhEnv->inputMax   = lzhEnv->inputPosition = sizeof(lzhEnv->inputBuffer);
    lzhEnv->outputMax  = sizeof(lzhEnv->outputBuffer);
    getbuf = getlen = putbuf = putlen = 0;
    matchLength    = 0;
    matchPosition  = 0;
    inpSize = outSize = 0;
    for (i = 0; i < N_CHAR; i++) {
        lzhEnv->freq[i]               = 1;
        lzhEnv->son[i]                = i + TABLE_SIZE;
        lzhEnv->parent[i + TABLE_SIZE] = i;
    }
    i = 0;
    j = N_CHAR;
    while (j <= ROOT_NODE) {
        lzhEnv->freq[j] = lzhEnv->freq[i] + lzhEnv->freq[i + 1];
        lzhEnv->son[j]  = i;
```

```
        lzhEnv->parent[i] = lzhEnv->parent[i + 1] = j;
        i += 2;
        j++;
    }
    lzhEnv->freq[TABLE_SIZE]  = 0xffff;
    lzhEnv->parent[ROOT_NODE] = 0;
}
/*==========================================================================*
 * Reconst
 *      Reconstruction of tree (frequencies values are too high).
 * RETURNS:     nothing
 *==========================================================================*/
static void
Reconst()
{
    register int i, k, j;
    uint16 f, l;
    /*
     * Collect leaf nodes in the first half of the table and replace
     * the freq by (freq + 1) / 2.
     */
    for (i = k = 0; i < TABLE_SIZE; ++i) {
        if (lzhEnv->son[i] >= TABLE_SIZE) {
            lzhEnv->freq[k] = (lzhEnv->freq[i] + 1) / 2;
            lzhEnv->son[k]  = lzhEnv->son[i];
            k++;
        }
    }
    /*
     * Begin constructing tree by connecting sons.
     */
    for (i = 0, j = N_CHAR; j < TABLE_SIZE; i += 2, j++) {
        /*
         * Build new node by adding freqs
         */
        k = i + 1;
        f = lzhEnv->freq[j] = lzhEnv->freq[i] + lzhEnv->freq[k];
        /*
         * Find the position in node list.
         */
        for (k = j - 1; f < lzhEnv->freq[k]; k--);
        k++;
        l = (uint16) ((j - k) * 2);
        /* With some UNIX systems, replace memmove() by memcpy() */
        memmove(&lzhEnv->freq[k + 1], &lzhEnv->freq[k], l);
        lzhEnv->freq[k] = f;
        memmove(&lzhEnv->son[k + 1], &lzhEnv->son[k], l);
        lzhEnv->son[k] = i;
    }
    /*
     * Connect parent.
     */
    for (i = 0; i < TABLE_SIZE; i++) {
        if ((k = lzhEnv->son[i]) >= TABLE_SIZE) {
            lzhEnv->parent[k] = i;
        } else {
            lzhEnv->parent[k] = lzhEnv->parent[k + 1] = i;
        }
    }
}
```

```
/*==========================================================================*
 * UpDate
 *      Increment frequency of given code by one, and update tree.
 * RETURNS:     nothing
 *==========================================================================*/
static void
UpDate(c)
    register int c;
{
    register int l;
    int      i, j;
    uint16   newFreq;
    /*
     * If the root frequency is maximum then rebuild tree
     */
    if (lzhEnv->freq[ROOT_NODE] == MAX_FREQ) {
        Reconst();
    }
    /*
     * Get the position of leaf corresponding to the code.
     */
    c = lzhEnv->parent[c + TABLE_SIZE];
    do {
        /*
         * Increment frequency of given code by one.
         */
        newFreq = ++lzhEnv->freq[c];
        /*
         * If the order is disturbed, exchange nodes.
         */
        if (newFreq > lzhEnv->freq[l = c + 1]) {
            /*
             * Freq must be less than right brother !
             * Seek for a less or equal right brother.
             */
            while (newFreq > lzhEnv->freq[++l]);
            l--;
            /*
             * Reverse freqs
             */
            lzhEnv->freq[c] = lzhEnv->freq[l];
            lzhEnv->freq[l] = newFreq;
            i = lzhEnv->son[c];
            lzhEnv->parent[i] = l;
            if (i < TABLE_SIZE) {
                lzhEnv->parent[i + 1] = l;
            }
            j = lzhEnv->son[l];
            lzhEnv->son[l] = i;
            lzhEnv->parent[j] = c;
            if (j < TABLE_SIZE) {
                lzhEnv->parent[j + 1] = c;
            }
            lzhEnv->son[c] = j;
            c = l;
        }
    } while ((c = lzhEnv->parent[c]) != 0);   /* Repeat up to root */
}
```

```
/*=========================================================================*
 * EncodeChar
 *      Encode a symbol travelling in Huffman tree and call UpDate to
 *      update the model with the new character.
 * RETURNS:     nothing
 *=========================================================================*/
static void
EncodeChar(c)
    unsigned c;
{
    register uint16 code;
    register int    currentNode;
    register int16  codeLength;
    code = codeLength = 0;
    /*
     * Get the position of leaf corresponding to the code.
     */
    currentNode = lzhEnv->parent[c + TABLE_SIZE];
    /*
     * Travel from leaf to root.
     */
    do {
        /*
         * Because we work from leaves to root, we get bits in inverse
         * order. So we have to shift-right to obtain correct order.
         */
        code >>= 1;
        /*
         * If node's address is odd-numbered, choose bigger brother node.
         * Add 0x8000 because of right-shifting.
         */
        if (currentNode & 1) {
            code += 0x8000;  /* 1000 0000 0000 0000b */
        }
        codeLength++;
    } while ((currentNode = lzhEnv->parent[currentNode]) != ROOT_NODE);
    PutCode(codeLength, code);
    UpDate(c);
}
/*=========================================================================*
 * EncodePosition
 *      Encode a position using position tables. Don't use Huffman tree.
 * RETURNS:     nothing
 *=========================================================================*/
static void
EncodePosition(position)
    register unsigned position;
{
    register unsigned tblPos;
    /*
     * Output upper 6 bits by table lookup (position is 12 bits length).
     * Erase lower 6 bits and only takes upper 6 bits by right-shifting.
     */
    tblPos = position >> 6;
    PutCode(p_len[tblPos], (uint16) p_code[tblPos] << 8);
    /*
     * Output lower 6 bits verbatim : 0x3f = 111111b
     */
    PutCode(6, (uint16) ((position & 0x3f) << 10));
    /*
     * Finaly, we have :
     *    3 to 8 bits(see huftbl.c : use upper 6 bits and p_len) of p_code
     *     + 6   bits lower bits of position
```

```
    */
}
/*============================================================================*
 * EncodeEnd
 *      End of encoding.
 * RETURNS:     nothing
 *============================================================================*/
static void
EncodeEnd()
{
    if (putlen) {
        if (! OutputByte((int16) (putbuf >> 8))) {      /* HIBYTE putbuf */
            Error(writeError);
        }
    }
}
/*============================================================================*
 * DecodeChar
 *      Decode a char travelling in Huffman's tree and update model.
 * RETURNS:     the decoded char
 *============================================================================*/
static int
DecodeChar(void)
{
    register unsigned c;
    c = lzhEnv->son[ROOT_NODE];
    /*
     * travel from root to leaf, choosing the smaller child node (son[])
     * if the read bit is 0, the bigger (son[]+1) if 1
     */
    while (c < TABLE_SIZE) {
        c += GetBit();        /* 0 or 1 */
        c = lzhEnv->son[c];
    }
    c -= TABLE_SIZE;
    UpDate(c);
    return c;
}
/*============================================================================*
 * DecodePosition
 *      Decode a position using Huffman's position table.
 * RETURNS:
 *============================================================================*/
static int
DecodePosition()
{
    register uint16 readByte, j, c;
    /*
     * Recover upper 6 bits from table
     */
    readByte = GetByte();
    c = ((uint16) d_code[readByte] << 6);
    j = d_len[readByte];
    /*
     * Read lower 6 bits verbatim
     */
    j -= 2;  /* Number of bits to read : 2 bits (8 - 6 = 2) already readen */
    while (j--) {
        readByte = (readByte << 1) + GetBit();
    }
    /*
     * i & 0x3f : 0x3f = 111111b -> only takes the lower 6 bits of i
     */
```

```
    return(c | (readByte & 0x3f));
}
/*============================================================================*
 * LzhEncode
 *      Main compression routine.
 * RETURNS:     nothing
 *============================================================================*/
static void
LzhEncode(encode_crc_32)
    uint32 *encode_crc_32;
{
    register int16 r, s;
    int           i, c, len, lastMatchLength;
    uint32        inputCrc32 = 0xffffffffL;
    StartHuffman();
    InitTree();
    /*
     * Load up the look ahead buffer.
     */
    r = WINDOW_SIZE - LOOK_AHEAD_SIZE;
    for (s = 0; s < LOOK_AHEAD_SIZE && (c = InputByte()) != EOF; ++s) {
        inputCrc32 = UpdateCharacterCrc32((uint8) c, inputCrc32);
        textBuffer[r + s] = (uint8) c;
    }
    len = s;
    for (s = 1; s <= LOOK_AHEAD_SIZE; ++s) {
        InsertNode(r - s);
    }
    InsertNode(r);
    /*
     * Main compression loop.
     */
    s = 0;
    do {
        if (matchLength > len) {
            matchLength = len;
        }
        if (matchLength <= THRESHOLD) {
            /*
             * Output a single character.
             */
            matchLength = 1;
            EncodeChar(textBuffer[r]);
        } else {
            /*
             * Output an index/length token that defines a phrase.
             *
             * The length:
             *   Emit (255 - THRESHOLD + matchLength) :
             *     255 to indicate it's a special code (in fact 255 + 1...)
             *     - THRESHOLD because we encode a length from THRESHOLD (2)
             *       so we can encode it as if it was 0 !
             *      examples: - if matchLength is 3; encode 256
             *                - if matchLength is 4, encode 257
             *
             * The position is encoded from the current character so the
             * real position is (pos-matchPosition)
             *
             */
            EncodeChar(255 - THRESHOLD + matchLength);
            EncodePosition(matchPosition);
        }
        /*
```

```
         * Reads in new characters and deletes the strings that are
         * overwritten by the new character.
         */
        lastMatchLength = matchLength;
        for (i = 0; i < lastMatchLength && (c = InputByte()) != EOF; ++i) {
            inputCrc32 = UpdateCharacterCrc32((uint8) c, inputCrc32);
            DeleteNode(s);
            textBuffer[s] = (uint8) c;
            if (s < LOOK_AHEAD_SIZE - 1) {
                textBuffer[s + WINDOW_SIZE] = (uint8) c;
            }
            s = (int16) ((s + 1) & (WINDOW_SIZE - 1));
            r = (int16) ((r + 1) & (WINDOW_SIZE - 1));
            InsertNode(r);
        }
        while (i++ < lastMatchLength) {   /* End of file */
            DeleteNode(s);
            s = (int16) ((s + 1) & (WINDOW_SIZE - 1));
            r = (int16) ((r + 1) & (WINDOW_SIZE - 1));
            if (--len) {
                InsertNode(r);
            }
        }
    } while (len > 0);
    EncodeEnd();
    if (lzhEnv->outputPosition) {
        fwrite(lzhEnv->outputBuffer, 1, lzhEnv->outputPosition, outFile);
        outSize += lzhEnv->outputPosition;
    }
    printf("\nIn : %ld bytes\n", inpSize);
    printf("Out: %ld bytes\n", outSize);
    printf("Out/In: %.3f\n", (double) inpSize / outSize);
    free(lzhEnv);
    free(textBuffer);
    free(dad);
    free(leftSon);
    free(rightSon);
    *encode_crc_32  = inputCrc32;
}
/*============================================================================*
 * LzhDecode
 *      The decoding function.
 * RETURNS:     nothing
 *============================================================================*/
static void
LzhDecode(filesize, decode_crc_32)
    long    filesize;
    uint32 *decode_crc_32;
{
    register int  i, j, k, r, c;
    uint32 outputCrc32 = 0xffffffffL;
    StartHuffman();
    r = WINDOW_SIZE - LOOK_AHEAD_SIZE;    /* r = 4050 */
    while (inpSize < filesize + lzhEnv->inputMax) {
        if ((c = DecodeChar()) < 256) {   /* c is a standard code */
            if (! OutputByte((short) c)) {
                Error(writeError);
            }
            outputCrc32 = UpdateCharacterCrc32((uint8) c, outputCrc32);
            textBuffer[r++] = (uint8) c;
            r &= (WINDOW_SIZE - 1);
        } else {
            /*
```

```
            * c is a length code : decode the position
            */
           i = (r - DecodePosition() - 1) & (WINDOW_SIZE - 1);
           j = c - 255 + THRESHOLD;  /* j is the length */
           for (k = 0; k < j; ++k) {
              c = textBuffer[(i + k) & (WINDOW_SIZE - 1)];
              if (! OutputByte((short) c)) {
                 Error(writeError);
              }
              outputCrc32 = UpdateCharacterCrc32((uint8) c, outputCrc32);
              textBuffer[r++] = (uint8) c;
              r &= (WINDOW_SIZE - 1);
           }
        }
     }
     if (lzhEnv->outputPosition) {
        fwrite(lzhEnv->outputBuffer, 1, lzhEnv->outputPosition, outFile);
     }
     free(lzhEnv);
     free(textBuffer);
     free(dad);
     free(leftSon);
     free(rightSon);
     *decode_crc_32  = outputCrc32;
}
/*=============================================================================*
 * main
 *      Beginning of the program.
 * RETURNS:     an integer
 *=============================================================================*/
int
main(argc, argv)
     int   argc;
     char *argv[];
{
     uint32 crc_32;
     fprintf(stderr, "LZHUF - LZSS and adaptive Huffman data compression\n");
     if (argc!=4 || (argv[1][0] != 'C' && argv[1][0] != 'U')) {
        fprintf(stderr, "Usage: LZHUF <C,U> infile.xxx outfile.yyy\n\n");
        fprintf(stderr, " - C is for Compress\n");
        fprintf(stderr, " - U is for Uncompress\n");
        fprintf(stderr, " - infile.xxx is your input file\n");
        fprintf(stderr, " - outfile.yyy is your output file\n\n");
        fprintf(stderr, "Examples :\n");
        fprintf(stderr, "      LZHUF C myfile.doc myfile.lzh\n");
        fprintf(stderr, "      LZHUF U myfile.lzh myfile.dcp\n");
        return 1;
     }
     if (argv[1][0] == 'C') {
        inFile  = fopen(argv[2], "rb");
        if (inFile == NULL) {
           fprintf(stderr, "Can't open file %s\n", argv[2]);
           return 1;
        }
        outFile = fopen(argv[3], "wb");
        LzhEncode(&crc_32);
        printf("\nEncoding CRC 32 bits : %lu (decoding must be equal)\n",
           crc_32);
        fclose(inFile);
        fclose(outFile);
     }
     if (argv[1][0] == 'U') {
        long textsize;
```

```
        inFile   = fopen(argv[2], "rb");
        if (inFile == NULL) {
            fprintf(stderr, "Can't open file %s\n", argv[2]);
            return 1;
        }
        outFile  = fopen(argv[3], "wb");
        fseek(inFile, 0L, SEEK_END);   /* determine filelength */
        textsize = ftell(inFile);
        fseek(inFile, 0L, SEEK_SET);
        LzhDecode(textsize, &crc_32);
        printf("\nDecoding CRC 32 bits : %lu (encoding must be equal)\n",
            crc_32);
        fclose(inFile);
        fclose(outFile);
    }
    return 0;
}
/*============================================================================*
 * huftbl.c                                                   *
 *      Tables for encoding and decoding position                         *
 *============================================================================*/
/*
 * HEADER FILES
 */
#include "all.h"
/*
 * EXPORTED VARIABLES
 */
/*
 * Tables for encoding and decoding the upper 6 bits of position :
 * 6 bits is 64 possibles value.
 */
/*
 * encoding : code's length
 * p_len is used with the upper 6 bits of a position
 */
uint8 p_len[64] = {
/* code 0 is 3 bits length */
0x03,
/* codes 1..3  are 4 bits length  */
0x04, 0x04, 0x04,
/* codes 4..11 are 5 bits length  */
0x05, 0x05, 0x05, 0x05,
0x05, 0x05, 0x05, 0x05,
/* codes 12..23 are 6 bits length */
0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
/* codes 24..47 are 7 bits length */
0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
/* codes 48..63 are 8 bits length */
0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08
};
/*
 * encoding: code
 * p_code is also used with the 6 upper bits of a position
 */
uint8 p_code[64] =
{
/* 000b */
```

```
0x00,
/* 0010b, 0011b, 0100b */
0x20, 0x30, 0x40,
/* 01010b, 01011b, 01100b, 01101b */
/* 01110b, 01111b, 10000b, 10001b */
0x50, 0x58, 0x60, 0x68,
0x70, 0x78, 0x80, 0x88,
/* 100100b, 100101b, 100110b, 100111b, 101000b, 101001b */
/* 101010b, 101011b, 101100b, 101101b, 101110b, 101111b */
0x90, 0x94, 0x98, 0x9C, 0xA0, 0xA4,
0xA8, 0xAC, 0xB0, 0xB4, 0xB8, 0xBC,
/* 1100000b, 1100001b, 1100010b, 1100011b, 1100100b, 1100101b, ... */
0xC0, 0xC2, 0xC4, 0xC6, 0xC8, 0xCA, 0xCC, 0xCE, 0xD0, 0xD2, 0xD4, 0xD6,
0xD8, 0xDA, 0xDC, 0xDE, 0xE0, 0xE2, 0xE4, 0xE6, 0xE8, 0xEA, 0xEC, 0xEE,
/* 11110000b, 11110001b, ... */
0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7,
0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, 0xFF
};
/*
 * decoding : codes
 */
uint8 d_code[256] =
{
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02,
0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02,
0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,
0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,
0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A,
0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B, 0x0B,
0x0C, 0x0C, 0x0C, 0x0C,
0x0D, 0x0D, 0x0D, 0x0D,
0x0E, 0x0E, 0x0E, 0x0E,
0x0F, 0x0F, 0x0F, 0x0F,
0x10, 0x10, 0x10, 0x10,
0x11, 0x11, 0x11, 0x11,
0x12, 0x12, 0x12, 0x12,
0x13, 0x13, 0x13, 0x13,
0x14, 0x14, 0x14, 0x14,
0x15, 0x15, 0x15, 0x15,
0x16, 0x16, 0x16, 0x16,
0x17, 0x17, 0x17, 0x17,
0x18, 0x18,
0x19, 0x19,
0x1A, 0x1A,
0x1B, 0x1B,
0x1C, 0x1C,
0x1D, 0x1D,
0x1E, 0x1E,
0x1F, 0x1F,
0x20, 0x20,
0x21, 0x21,
```

```
0x22, 0x22,
0x23, 0x23,
0x24, 0x24,
0x25, 0x25,
0x26, 0x26,
0x27, 0x27,
0x28, 0x28,
0x29, 0x29,
0x2A, 0x2A,
0x2B, 0x2B,
0x2C, 0x2C,
0x2D, 0x2D,
0x2E, 0x2E,
0x2F, 0x2F,
0x30,
0x31,
0x32,
0x33,
0x34,
0x35,
0x36,
0x37,
0x38,
0x39,
0x3A,
0x3B,
0x3C,
0x3D,
0x3E,
0x3F
};
/*
 * decoding : code's length; use a byte
 */
uint8 d_len[256] =
{
/*
 * The first code (0) is 3 bits length, so in a byte, there are (8-3=) 5 bits
 * variables, so there are (2^5 = 2*2*2*2*2) 32 possibles values for length 3
 */
0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
/* 8 - 4 = 4 bits; 2^4 = 16 possible values for length 4 */
0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
/* idem : there are 3 codes for length 4 ! */
0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
/* 16 possibles values */
0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
/* 8 - 5 = 3 bits; 2^3 = 8 possible values for length 5 */
/* and there are 8 codes ... so 8 * 8 = 64 values */
0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05, 0x05,
```

```
/* 8 - 6 = 2 bits; 2^2 = 4 possible values for length 6 */
/* and there are 12 codes ... so 12 * 4 = 48 values */
0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06, 0x06,
/* 8 - 7 = 1 bits; 2^1 = 2 possible values for length 7 */
/* and there are 24 codes ... so 24 * 2 = 48 values */
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
0x07, 0x07,
/* 8 - 8 = 0 bits; 2^0 = 1 possible value for length 8 */
/* and there are 16 codes ... so 16 * 1 = 16 values */
0x08,
0x08,
0x08,
0x08,
0x08,
0x08,
0x08,
0x08,
0x08,
0x08,
0x08,
0x08,
0x08,
0x08,
0x08,
0x08
};
```

```
/*=============================================================================*
 * crc32.c                                                                     *
 *      Calculation of the CRC checksum                                        *
 * EXPORTED PROCEDURES:                                                        *
 *      UpdateCharacterCrc32                                                   *
 *=============================================================================*/
/*==============*
 * HEADER FILES *
 *==============*/
#include "all.h"
/*==================*
 * STATIC VARIABLES *
 *==================*/
/*
 * This is the lookup table used when performing the CRC calculation.
 */
static uint32 ccitt32Table[256] = {
0x00000000L, 0x77073096L, 0xee0e612cL, 0x990951baL, 0x076dc419L, 0x706af48fL,
0xe963a535L, 0x9e6495a3L, 0x0edb8832L, 0x79dcb8a4L, 0xe0d5e91eL, 0x97d2d988L,
0x09b64c2bL, 0x7eb17cbdL, 0xe7b82d07L, 0x90bf1d91L, 0x1db71064L, 0x6ab020f2L,
0xf3b97148L, 0x84be41deL, 0x1adad47dL, 0x6ddde4ebL, 0xf4d4b551L, 0x83d385c7L,
0x136c9856L, 0x646ba8c0L, 0xfd62f97aL, 0x8a65c9ecL, 0x14015c4fL, 0x63066cd9L,
0xfa0f3d63L, 0x8d080df5L, 0x3b6e20c8L, 0x4c69105eL, 0xd56041e4L, 0xa2677172L,
0x3c03e4d1L, 0x4b04d447L, 0xd20d85fdL, 0xa50ab56bL, 0x35b5a8faL, 0x42b2986cL,
0xdbbbc9d6L, 0xacbcf940L, 0x32d86ce3L, 0x45df5c75L, 0xdcd60dcfL, 0xabd13d59L,
0x26d930acL, 0x51de003aL, 0xc8d75180L, 0xbfd06116L, 0x21b4f4b5L, 0x56b3c423L,
0xcfba9599L, 0xb8bda50fL, 0x2802b89eL, 0x5f058808L, 0xc60cd9b2L, 0xb10be924L,
0x2f6f7c87L, 0x58684c11L, 0xc1611dabL, 0xb6662d3dL, 0x76dc4190L, 0x01db7106L,
0x98d220bcL, 0xefd5102aL, 0x71b18589L, 0x06b6b51fL, 0x9fbfe4a5L, 0xe8b8d433L,
0x7807c9a2L, 0x0f00f934L, 0x9609a88eL, 0xe10e9818L, 0x7f6a0dbbL, 0x086d3d2dL,
0x91646c97L, 0xe6635c01L, 0x6b6b51f4L, 0x1c6c6162L, 0x856530d8L, 0xf262004eL,
0x6c0695edL, 0x1b01a57bL, 0x8208f4c1L, 0xf50fc457L, 0x65b0d9c6L, 0x12b7e950L,
0x8bbeb8eaL, 0xfcb9887cL, 0x62dd1ddfL, 0x15da2d49L, 0x8cd37cf3L, 0xfbd44c65L,
0x4db26158L, 0x3ab551ceL, 0xa3bc0074L, 0xd4bb30e2L, 0x4adfa541L, 0x3dd895d7L,
0xa4d1c46dL, 0xd3d6f4fbL, 0x4369e96aL, 0x346ed9fcL, 0xad678846L, 0xda60b8d0L,
0x44042d73L, 0x33031de5L, 0xaa0a4c5fL, 0xdd0d7cc9L, 0x5005713cL, 0x270241aaL,
0xbe0b1010L, 0xc90c2086L, 0x5768b525L, 0x206f85b3L, 0xb966d409L, 0xce61e49fL,
0x5edef90eL, 0x29d9c998L, 0xb0d09822L, 0xc7d7a8b4L, 0x59b33d17L, 0x2eb40d81L,
0xb7bd5c3bL, 0xc0ba6cadL, 0xedb88320L, 0x9abfb3b6L, 0x03b6e20cL, 0x74b1d29aL,
0xead54739L, 0x9dd277afL, 0x04db2615L, 0x73dc1683L, 0xe3630b12L, 0x94643b84L,
0x0d6d6a3eL, 0x7a6a5aa8L, 0xe40ecf0bL, 0x9309ff9dL, 0x0a00ae27L, 0x7d079eb1L,
0xf00f9344L, 0x8708a3d2L, 0x1e01f268L, 0x6906c2feL, 0xf762575dL, 0x806567cbL,
0x196c3671L, 0x6e6b06e7L, 0xfed41b76L, 0x89d32be0L, 0x10da7a5aL, 0x67dd4accL,
0xf9b9df6fL, 0x8ebeeff9L, 0x17b7be43L, 0x60b08ed5L, 0xd6d6a3e8L, 0xa1d1937eL,
0x38d8c2c4L, 0x4fdff252L, 0xd1bb67f1L, 0xa6bc5767L, 0x3fb506ddL, 0x48b2364bL,
0xd80d2bdaL, 0xaf0a1b4cL, 0x36034af6L, 0x41047a60L, 0xdf60efc3L, 0xa867df55L,
0x316e8eefL, 0x4669be79L, 0xcb61b38cL, 0xbc66831aL, 0x256fd2a0L, 0x5268e236L,
0xcc0c7795L, 0xbb0b4703L, 0x220216b9L, 0x5505262fL, 0xc5ba3bbeL, 0xb2bd0b28L,
0x2bb45a92L, 0x5cb36a04L, 0xc2d7ffa7L, 0xb5d0cf31L, 0x2cd99e8bL, 0x5bdeae1dL,
0x9b64c2b0L, 0xec63f226L, 0x756aa39cL, 0x026d930aL, 0x9c0906a9L, 0xeb0e363fL,
0x72076785L, 0x05005713L, 0x95bf4a82L, 0xe2b87a14L, 0x7bb12baeL, 0x0cb61b38L,
0x92d28e9bL, 0xe5d5be0dL, 0x7cdcefb7L, 0x0bdbdf21L, 0x86d3d2d4L, 0xf1d4e242L,
0x68ddb3f8L, 0x1fda836eL, 0x81be16cdL, 0xf6b9265bL, 0x6fb077e1L, 0x18b74777L,
0x88085ae6L, 0xff0f6a70L, 0x66063bcaL, 0x11010b5cL, 0x8f659effL, 0xf862ae69L,
0x616bffd3L, 0x166ccf45L, 0xa00ae278L, 0xd70dd2eeL, 0x4e048354L, 0x3903b3c2L,
0xa7672661L, 0xd06016f7L, 0x4969474dL, 0x3e6e77dbL, 0xaed16a4aL, 0xd9d65adcL,
0x40df0b66L, 0x37d83bf0L, 0xa9bcae53L, 0xdebb9ec5L, 0x47b2cf7fL, 0x30b5ffe9L,
0xbdbdf21cL, 0xcabac28aL, 0x53b39330L, 0x24b4a3a6L, 0xbad03605L, 0xcdd70693L,
0x54de5729L, 0x23d967bfL, 0xb3667a2eL, 0xc4614ab8L, 0x5d681b02L, 0x2a6f2b94L,
0xb40bbe37L, 0xc30c8ea1L, 0x5a05df1bL, 0x2d02ef8dL
};
```

```
/*====================*
 * EXPORTED PROCEDURE *
 *====================*/
/*=============================================================================*
 * UpdateCharacterCrc32
 *      This function computes a byte's CRC, taking a CRC value in entry to
 *      perform on the fly crc calculations.
 * RETURNS:     a 32 bits crc
 *=============================================================================*/
uint32
UpdateCharacterCrc32(c, crc)
    uint8  c;
    uint32 crc;
{
    uint32 temp1;
    uint32 temp2;
    temp1 = (crc >> 8) & 0x00FFFFFFL;
    temp2 = ccitt32Table[((int) crc ^ c) & 0xff];
    return(temp1 ^ temp2);
}
/*=============================================================================*
 * all.h                                                       *
 *       stuff included from ALL source files                          *
 *=============================================================================*/
/*
 * HEADER FILES
 */
#include "ansi.h"
#include "general.h"
/*
 * GLOBAL VARIABLES
 */
/*
 * Defined in huftbl.c
 */
extern uint8 p_len[64];
extern uint8 p_code[64];
extern uint8 d_code[256];
extern uint8 d_len[256];
/*
 * Defined in crc32.c
 */
uint32 UpdateCharacterCrc32 _ANSI_ARGS_((uint8 c, uint32 crc));
/*=============================================================================*
 * ansi.h                                                      *
 *      macro for non-ansi compilers                                   *
 *=============================================================================*/
/*
 * Define NON_ANSI_COMPILER in the MakeFile if K&R compiler
 */
#ifdef NON_ANSI_COMPILER
#   define _ANSI_ARGS_(x)    ()
#else
#   define _ANSI_ARGS_(x)    x
#endif
```

```
/*============================================================================*
 * general.h                                                                  *
 *      general stuff                                                         *
 *============================================================================*/
/*=================*
 * TYPE DEFINITIONS *
 *=================*/
typedef unsigned char  uint8;    /* must hold values 0..255 */
typedef unsigned int   uint;     /* 16 bits */
typedef         short  int16;   /* signed 16 bits */
typedef unsigned short uint16;   /* must hold values 0..65535 */
typedef         long   int32;   /* signed 32 bits */
typedef unsigned long  uint32;   /* unsigned 32 bits */
```

Insert the following as a new annex to this ETS:

Annex B (normative): T.51String

B.1 Scope

This annex describes the rules to be applied when CCITT Recommendation T.51 (1992) [8] is referenced for the encoding of fields. The T.51String defined by this annex serves as a reference model by restricting CCITT Recommendation T.51 (1992) [8] to those elements of the code extension mechanisms of the character sets and repertoire which are necessary to ease the implementations.

This definition is not specific to an individual telematic service, but it can be referenced by telematic application standards.

B.2 Graphic character set

The primary set of graphic characters (see figure B.1) is identical with the set of graphic characters of the International Reference Version (IRV) of the 7-bit coded character set of CCITT Recommendation T.50 (1992).

					b7	0	0	0	0	1	1	1	1
					b6	0	0	1	1	0	0	1	1
					b5	0	1	0	1	0	1	0	1
b4	b3	b2	b1			**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	**0**					**O**	@	**P**	`	**p**
0	0	0	1	**1**				**!**	**1**	**A**	**Q**	**a**	**q**
0	0	1	0	**2**				**"**	**2**	**B**	**R**	**b**	**r**
0	0	1	1	**3**				**#**	**3**	**C**	**S**	**c**	**s**
0	1	0	0	**4**				**$**	**4**	**D**	**T**	**d**	**t**
0	1	0	1	**5**				**%**	**5**	**E**	**U**	**e**	**u**
0	1	1	0	**6**				**&**	**6**	**F**	**V**	**f**	**v**
0	1	1	1	**7**				**'**	**7**	**G**	**W**	**g**	**w**
1	0	0	0	**8**				**(**	**8**	**H**	**X**	**h**	**x**
1	0	0	1	**9**				**)**	**9**	**I**	**Y**	**i**	**y**
1	0	1	0	**10**				*****	**:**	**J**	**Z**	**j**	**z**
1	0	1	1	**11**				**+**	**;**	**K**	**[**	**k**	**{**
1	1	0	0	**12**				**,**	**<**	**L**	****	**l**	**|**
1	1	0	1	**13**				**-**	**=**	**M**	**]**	**m**	**}**
1	1	1	0	**14**				**.**	**>**	**N**	**^**	**n**	**~**
1	1	1	1	**15**				***/***	**?**	**O**	**_**	**o**	

Figure B.1: Primary set of graphic characters for T.51String

The supplementary set of graphic characters is specified in figure B.2.

b4 b3 b2 b1		b7	0	0	0	0	1	1	1	1
		b6	0	0	1	1	0	0	1	1
		b5	0	1	0	1	0	1	0	1
			0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0 0 0 0	**0**				NBSP			—	Ω	K
0 0 0 1	**1**				¡	±	`	¹	Æ	æ
0 0 1 0	**2**				¢	²	´	®	Ð	
0 0 1 1	**3**				£	³	^	©	ª	ð
0 1 0 0	**4**					×	~	™		
0 1 0 1	**5**				¥	µ	¯			
0 1 1 0	**6**					¶	∨	¬		
0 1 1 1	**7**				§	·		¦		
1 0 0 0	**8**				¤	÷	¨			
1 0 0 1	**9**				‘	’			∅	∅
1 0 1 0	**10**				“	”	°		Œ	œ
1 0 1 1	**11**				«	»	¸		º	ß
1 1 0 0	**12**				←	¼			Þ	þ
1 1 0 1	**13**				↑	½	"			
1 1 1 0	**14**				→	¾	¿			ŋ
1 1 1 1	**15**				↓	¿	˅			

Figure B.2 : Supplementary set of graphic characters for T.51String

For the T.51String the following applies:

1. The primary set is designated as the G0 set and invoked in column 2 to 7 of the code table.
2. The supplementary set is designated as a G2 set. In the 8-bit environment the set is invoced in column 10 to 15.
3. In the T.51String no designation and invocation sequences are allowed.
4. All characters in column 4 of the supplementary set are non-spacing characters (diacritical marks).
5. Unallocated code positions are reserved and shall not be used.
6. The compatibility provisions described in CCITT Recommendation T.51 (1992) [8], § 2.2.4. Notes 3, 4 and 6 shall not be applied.

B.3 Code extension technique

In the 8-bit environment no code extension sequence is allowed. In the 7-bit environment the single shift function SS2 (code 1/9) is used to invoke one character from the G2 set. All other shift functions are not allowed.

B.4 Repertoire of the Latin Based Character Set

The T.51String repertoire is identical with the superset of the repertoire of the latin based character set specified in CCITT Recommendation T.51 (1992) [8], annex A. All combinations of diacritical marks with basic letters as specified in Figure A-2/T.51 [8] shall be supported.

B.5 Control functions

Invocation and designation sequences for control functions shall not be allowed. From columns 0 and 1 of the code table in use only the control character CR, Carriage Return (0/13) and LF, Line Feed (0/10) can be used. They are specified in CCITT Recommendation T.50 (1992).

History

Document history	
November 1990	First Edition
November 1995	Amendment 1 to Second Edition
February 1996	Converted into Adobe Acrobat Portable Document Format (PDF)
NOTE:	The references to the changed pages refer to an old presentation. The clause numbering has not changed

ISBN 2-7437-0370-9 - Amendement 1
Dépôt légal : Novembre 1995