```
//*****************************************************************************
**
// FILE:                Hoffman.c
// PROJECT:             zarLib  - ver 1.00 -
// COPYRIGHTS:  The famous group 4 of "Roskilde Universites Center"
//*****************************************************************************
***/
// Headers:
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <search.h>
#include <string.h>
#include "zarLib.h"
#include "main.h"


//*****************************************************************************
**
// Constants:

#define END_OF_STREAM   599


//*****************************************************************************
**
// Typedefs:

typedef struct NODE
{
        _dword  dwCount;
        int             nChar;
        _dword  dwBitCode;
        _dword  dwNumBits;
        struct  NODE*   pNext;
        struct  NODE*   pChild0;
        struct  NODE*   pChild1;
} NODE;


//*****************************************************************************
**
// Function prototypes:

static NODE* buildHuffmanTree(NODE* nodes, NODE* pStart);
static NODE* initHuffmanCompression(ZAR_FILE* pzSrc, NODE* nodes);
static NODE* initHuffmanExtraction(ZAR_FILE* pzSrc, NODE* nodes);
static int   loadHeader(ZAR_CHAR_FREQ* pzCharFreq, ZAR_FILE* pzFile);
static int   saveHeader(const NODE* nodes, ZAR_FILE* pzFile);
static void  assignCodes(NODE* pParent);
static void  sortNodes(NODE* nodes);
static int   nodesCompare(const void* pA, const void* pB);


//*****************************************************************************
**
// HuffmanExtract()

int HuffmanExtract(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        int             i, j=0;
        NODE*   root, *pStart, *pNode;
        NODE    nodes[600];

        // Print info:
        printf("--- Starting Huffman extraction ---\n");
        printf("Progress: ");

        pStart = initHuffmanExtraction(pzSrc, nodes);
```



```
        root = buildHuffmanTree(nodes, pStart);

        // build bit-codes:
        assignCodes(root);

        pNode = root;

        while (1)
        {
                // Get next bit:
                i = zarGetBit(pzSrc);

                if (i)
                // Go left:
                        pNode = pNode->pChild1;
                else
                // Go right:
                        pNode = pNode->pChild0;

                // Are we finished?
                if (pNode->pChild0 == NULL)
                {
                        // Have we reached end of stream?
                        if (pNode == pStart)
                                break;

                        zarSetByte(pzDst, pNode->nChar);
                        j++;

                        // Reset position:
                        pNode = root;
                        continue;
                }
        }

        return 1;
}

//*****************************************************************************
**
// HuffmanCompress()

int HuffmanCompress(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        int             i;
        NODE*   root, *pStart, *pCodes;
        NODE    nodes[600];

        // Print info:
        printf("--- Starting Huffman compression ---\n");
        printf("Progress: ");

        pStart = initHuffmanCompression(pzSrc, nodes);

        root = buildHuffmanTree(nodes, pStart);

        // build bit-codes:
        assignCodes(root);

        // Remove end code:
        nodes[END_OF_STREAM].dwBitCode = pStart->dwBitCode;
        nodes[END_OF_STREAM].dwNumBits = pStart->dwNumBits;
        pStart->dwNumBits = 0;
        pStart->dwCount   = 0;

        // Sort the nodes after code:
        pCodes = nodes +1;
        sortNodes(pCodes);

        // Write header:
```

```
        if (!saveHeader(pCodes, pzDst))
                return zarError(ERR_OUTPUT_FILE_NOT_WRITEABLE, pzDst);

        // Write the compressed output:
        while (EOF != (i = zarGetByte(pzSrc)))
                zarSetBits(pzDst, pCodes[i].dwBitCode, pCodes[i].dwNumBits);

        // Write end of stream bits:
        zarSetBits(pzDst, nodes[END_OF_STREAM].dwBitCode, nodes[END_OF_STREAM].d
wNumBits);
        zarSetBits(pzDst, 0, 8);        // Flush the last bits:

        return 1;
}


//***************************************************************************
**
// initHuffmanCompression()

static NODE* initHuffmanCompression(ZAR_FILE* pzSrc, NODE* nodes)
{
        int                             i, nLowest;
        ZAR_FREQ*               pzFreq;
        ZAR_CHAR_FREQ*  pzCharFreq;

        // Allocate memory:
        pzCharFreq = (ZAR_CHAR_FREQ*) malloc(sizeof(ZAR_CHAR_FREQ));

        if (pzCharFreq == NULL)
        {
                zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);
                return NULL;
        }

        // Count the frequence of characters in the file:
        zarInitCharFreq(pzCharFreq);
        zarCountCharFreq(pzSrc, pzCharFreq);
        zarSortCharFreq(pzCharFreq);    // faster than qsort!

        pzFreq = pzCharFreq->freq;

        // Remove freqs with a zero-count:
        for (i=0; pzFreq[i].cxChar != 0 && i<256; i++);         // Don´t need an
y body!

        nLowest = 257 -i -1;

        // Convert to nodes and setup list:
        memset(nodes, 0, 600 *sizeof(NODE));

        for (i=1; i<257; i++)
        {
                // Add node:
                nodes[i].dwCount = pzCharFreq->freq[256 -i].cxChar;
                nodes[i].nChar  = pzCharFreq->freq[256 -i].ubChar;

                // Add to list:
                nodes[i].pNext = nodes +i +1;
        }

        // Add end of stream code:
        nodes[nLowest].pNext   = nodes +nLowest +1;
        nodes[nLowest].dwCount = 1;

        // Clean up:
        free(pzCharFreq);
        nodes[256].pNext = NULL;        // Last node doesn´t have a next node!

        // Return pointer to lowest node:
```



```
        return (nodes +nLowest);
}


//***************************************************************************
**
// initHuffmanExtraction()

static NODE* initHuffmanExtraction(ZAR_FILE* pzSrc, NODE* nodes)
{
        int                             i, nLowest;
        ZAR_FREQ*               pzFreq;
        ZAR_CHAR_FREQ*  pzCharFreq;

        // Allocate memory:
        pzCharFreq = (ZAR_CHAR_FREQ*) malloc(sizeof(ZAR_CHAR_FREQ));

        if (pzCharFreq == NULL)
        {
                zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);
                return NULL;
        }

        // Get the frequence of characters from the file header:
        zarInitCharFreq(pzCharFreq);
        loadHeader(pzCharFreq, pzSrc);
        zarSortCharFreq(pzCharFreq);

        pzFreq = pzCharFreq->freq;

        // Remove freqs with a zero-count:
        for (i=0; pzFreq[i].cxChar != 0 && i<256; i++);         // Don´t need an
y body!

        nLowest = 257 -i -1;

        // Convert to nodes and setup list:
        memset(nodes, 0, 600 *sizeof(NODE));

        for (i=0; i<257; i++)
        {
                // Add node:
                nodes[i].dwCount = pzCharFreq->freq[256 -i].cxChar;
                nodes[i].nChar  = pzCharFreq->freq[256 -i].ubChar;

                // Add to list:
                nodes[i].pNext = nodes +i +1;
        }

        // Add end of stream code:
        nodes[nLowest].pNext   = nodes +nLowest +1;
        nodes[nLowest].dwCount = 1;


        // Clean up:
        free(pzCharFreq);
        nodes[256].pNext = NULL;        // Last node doesn´t have a next node!

        // Return pointer to lowest node:
        return (nodes +nLowest);
}



//***************************************************************************
**
// buildHuffmanTree()

static NODE* buildHuffmanTree(NODE* nodes, NODE* pStart)
{
```

```
        int     nNumNodes = 257;
        NODE*   low0 = pStart;
        NODE*   low1 = pStart->pNext;
        NODE*   tmp;
        NODE*   root = NULL;

        while (1)
        {
                // Add the lowest elements to node:
                nodes[nNumNodes].pChild0 = low0;
                nodes[nNumNodes].pChild1 = low1;
                nodes[nNumNodes].dwCount = low0->dwCount + low1->dwCount;

                // Remove the added nodes form list:
                low0 = low1->pNext;
                low1 = low0;

                // Test for end condition:
                if (low1 == NULL)
                {
                        root = nodes +nNumNodes;
                        break;
                }

                // Should we add the node to first element in list?
                if (low0->dwCount > nodes[nNumNodes].dwCount)
                {
                        low0 = nodes +nNumNodes;
                        low0->pNext = low1;
                }
                else
                {
                        // Find spot where to insert the created node in the lis
t:
                        do
                        {
                                tmp  = low1;
                                low1 = low1->pNext;
                        }
                        while (low1 && low1->dwCount < nodes[nNumNodes].dwCount)
;

                        // Insert new node:
                        (tmp->pNext = nodes +nNumNodes)->pNext = low1;
// same as tmp->pNext->pNext = low1, but faster on MS VC++ 6, because low1->pNex
t already load into register!
                }

                low1 = low0->pNext;

                nNumNodes++;
        }

        return root;
}


//****************************************************************************
**
// assignCodes()

static void assignCodes(NODE* pParent)
{
        if (pParent->pChild0 != NULL)
        {
                static NODE* pChild;

                // Add zero code to child0:
                pChild = pParent->pChild0;
                pChild->dwBitCode = pParent->dwBitCode <<1;
```



```
                pChild->dwNumBits = pParent->dwNumBits +1;

                // Move further down 0 brance:
                assignCodes(pChild);

                // Add one code to child1:
                pChild = pParent->pChild1;
                pChild->dwBitCode = (pParent->dwBitCode <<1) +1;
                pChild->dwNumBits = pParent->dwNumBits +1;

                // Move further down 1 brance:
                assignCodes(pChild);
        }
}


//****************************************************************************
**
// sortNodes()

static void sortNodes(NODE* nodes)
{
        qsort(nodes, 256, sizeof(NODE), nodesCompare);
}


//****************************************************************************
**
// nodesCompare()

static int nodesCompare(const void* pA, const void* pB)
{
        return (((const NODE*) pA)->nChar -((const NODE*) pB)->nChar);
}


//****************************************************************************
**
// saveHeader()

static int saveHeader(const NODE* nodes, ZAR_FILE* pzFile)
{
        int             j, i=0, nOk=1;

        // Start to write the file header:
        while (nOk)
        {
                // Find first entry to write::
                while (nodes[i].dwCount == 0)   // Find non-zero!
                {
                        if (++i == 255)
                        {
                                i = 255;
                                break;
                        }
                }

                // Write the index of the found entry:
                fputc(i, pzFile->pFile);

                // Find the end of the entry (3 zeros):
                j = i +1;

                do
                {
                        if (++j >= 256)
                        {
                                j   = 256;
                                nOk = 0;        // end loop
                                break;
```

```
                        }
                }
                while (nodes[j].dwCount != 0 || nodes[j +1].dwCount != 0 || node
s[j +2].dwCount != 0);


                // Store the ending index:
                fputc(j -1, pzFile->pFile);

                // Save the values between the indexs:
                if (i != 255)
                {
                        for (; i<j; i++)
                                fputc(nodes[i].dwCount, pzFile->pFile);
                }
        }

        return 1;
}

//*************************************************************************
**
// loadHeader()

static int loadHeader(ZAR_CHAR_FREQ* pzCharFreq, ZAR_FILE* pzFile)
{
        int i, from, to;
        ZAR_FREQ* pFreq = pzCharFreq->freq;

        // Extract the frequence from the header:
        while (1)
        {
                from = fgetc(pzFile->pFile);
                to   = fgetc(pzFile->pFile);

                if (from == 255)
                        break;

                // Fill the range up to i with zeros:
                for (i=from; i<=to; i++)
                        pFreq[i].cxChar = fgetc(pzFile->pFile);

                if (to == 255)
                        break;
        }

        // Read next byte:
        pzFile->nCurByte = zarGetByte(pzFile);
        pzFile->nBitMask = 0x80;                // Make sure not to read next by
te!

        return 1;
}
```