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


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

#define ZAR_SORT_CUTOFF         8


//*****************************************************************************
**
// Globals:

static clock_t g_clock;


//*****************************************************************************
**
// Inline functions or macros:

#if _MSC_VER && _MSC_EXTENSIONS

__inline int comp(ZAR_FREQ* pA, ZAR_FREQ* pB)
{
        return (pB->cxChar -pA->cxChar);
}

__inline void swap(ZAR_FREQ* pA, ZAR_FREQ* pB)
{
    ZAR_FREQ tmp;

    tmp = *pA;
    *pA = *pB;
    *pB = tmp;
}

#else   // Use macros to emulate inline functions ): ):

// Global needed for macro:
static ZAR_FREQ g_tmp;

// Macros:
#define swap(pA, pB)    \
        g_tmp = *pA;            \
        *pA   = *pB;            \
        *pB   = g_tmp;

#define comp(pA, pB)    \
        (int)(pB->cxChar -pA->cxChar)

#endif  // _MSC_VER && _MSC_EXTENSIONS


//*****************************************************************************
**
// Function prototypes:
static int  charFreqCompare(const void* pA, const void* pB);
```



```
static void shortSort(ZAR_FREQ* lo, ZAR_FREQ* hi);


//*****************************************************************************
**
// zarInitTime()

void zarInitTime()
{
        g_clock = clock();
}

//*****************************************************************************
**
// zarDumpTime()

long zarDumpTime()
{
        return (clock() -g_clock);
}


//*****************************************************************************
**
// zarGetNumSwitches()

int zarGetNumSwitches(_dword dwSwitches)
{
        int             nFound = 0;
        _dword  dwMask = 0x10000000;

        while (dwMask)
        {
                if (dwSwitches & dwMask)
                        nFound++;

                dwMask >>= 1;
        }

        return nFound;
}



//*****************************************************************************
**
// zarInitCharFreq()

void zarInitCharFreq(ZAR_CHAR_FREQ* pzCharFreq)
{
        int i;
        ZAR_FREQ* pFreq = pzCharFreq->freq;

        // Zero counts:
        memset(pzCharFreq, 0, sizeof(ZAR_CHAR_FREQ));

        // Fill in the charecters:
        for (i=0; i<256; i++)
                pFreq[i].ubChar = (_ubyte) i;
}


//*****************************************************************************
**
// zarCountCharFreq()

void zarCountCharFreq(const ZAR_FILE* pzFile, ZAR_CHAR_FREQ* pzCharFreq)
{
        int             nByte, i;
        long      lPos;
```

```
        _dword    dwMax, dwScale;
        FILE*     pFile = pzFile->pFile;        // Could be faster on some stupi
d compilers!
        ZAR_FREQ* pFreq = pzCharFreq->freq;     // Could be faster on some stupi
d compilers!

        lPos = ftell(pFile);

        while (EOF != (nByte = fgetc(pFile)))
                pFreq[nByte].cxChar++;


        // Get the maximum count, so we know if scaling is required!
        dwMax = pFreq->cxChar;

        for (i=1; i<256; i++)
        {
                if (pFreq[i].cxChar > dwMax)
                        dwMax = pFreq[i].cxChar;
        }

        if (dwMax > 256)
        {
                // The scale factor:
                dwScale = 1 + dwMax /256;

                // Scale all values:
                for (i=0; i<256; i++)
                {
                        if (pFreq->cxChar != 0) // Don´t scale zero values:
                        {
                                pFreq->cxChar /= dwScale;

                                if (pFreq->cxChar == 0)
                                        pFreq->cxChar = 1;
                        }

                        pFreq++;
                }
        }

        // Seek to old pos of file:
        fseek(pFile, lPos, SEEK_SET);
}


//***************************************************************************
**
// zarSortCharFreq()

void zarSortCharFreq(ZAR_CHAR_FREQ* pzCharFreq)
{
        ZAR_FREQ        *lo, *hi;                       /* ends of sub-array curr
ently sorting */
    ZAR_FREQ    *mid;                   /* points to middle of subarray */
    ZAR_FREQ    *loguy, *higuy;         /* traveling pointers for partition step
*/
    _uint               size;
        ZAR_FREQ        *lostk[30], *histk[30];
    int stkptr;                                 /* stack for saving sub-array to
be processed */

    stkptr = 0;                                             /* initialize stack *
/

    lo = pzCharFreq->freq;
    hi = pzCharFreq->freq +255;

lbl_recurse:
```



```
    size = (hi -lo) +1;        /* number of el's to sort */

    /* below a certain size, it is faster to use a O(n^2) sorting method */
    if (size <= ZAR_SORT_CUTOFF)
                shortSort(lo, hi);
    else
        {
        mid = lo +(size /2);       /* find middle element */
        swap(mid, lo);             /* swap it to beginning of array */

        loguy = lo;
        higuy = hi +1;

        for (;;)
                {
                        do  {
                    loguy++;
                } while (loguy <= hi && comp(loguy, lo) <= 0);

                do  {
                    higuy--;
                } while (higuy > lo && comp(higuy, lo) >= 0);

                if (higuy < loguy)
                    break;

                swap(loguy, higuy);
                }

        swap(lo, higuy);     /* put partition element in place */

        if (higuy - 1 - lo >= hi - loguy )
                {
                if (lo +1 < higuy)
                        {
                    lostk[stkptr] = lo;
                    histk[stkptr] = higuy -1;
                    ++stkptr;
                }                           /* save big recursion for later */

                if (loguy < hi)
                        {
                    lo = loguy;
                    goto lbl_recurse;           /* do small recursion */
                }
        }
        else
                {
                if (loguy < hi)
                        {
                    lostk[stkptr] = loguy;
                    histk[stkptr] = hi;
                    ++stkptr;               /* save big recursion for later */
                }

                if (lo +1 < higuy)
                        {
                    hi = higuy -1;
                    goto lbl_recurse;           /* do small recursion */
                }
        }
    }

    /* We have sorted the array, except for any pending sorts on the stack.
       Check if there are any, and do them. */

    --stkptr;
    if (stkptr >= 0)
        {
        lo = lostk[stkptr];
```

```
        hi = histk[stkptr];
        goto lbl_recurse;           /* pop subarray from stack */
    }
}


//*************************************************************************
**
// shortSort()

static void shortSort(ZAR_FREQ* lo, ZAR_FREQ* hi)
{
    ZAR_FREQ    *p;
        ZAR_FREQ        *max;

    while (hi > lo)
        {
        max = lo;

                for (p = lo +1; p <= hi; p++)
                {
                        if (comp(p, max) > 0)
                max = p;
        }

        swap(max, hi);
        hi--;
    }
}
```