31 Creating GIF files

"Love is patient and kind."

GIF stands for Graphics Interchange Format. GIF is a good file format introduced by CompuServe Incorporated. GIF files can be classified into (i) Ordinary GIF files (ii) Animated GIF files. GIF files are widely used in Internet. GIF took its popularity by the capacity to get animated and by using the very efficient "one-pass" LZW compression algorithm.

31.1 Important Notice

The Graphics Interchange Format © is the Copyright property of CompuServe Incorporated. GIF ™ is a Service Mark property of CompuServe Incorporated.

Once Unisys was a well-known computer company. Unisys was awarded the patent in 1985 for the very famous compression algorithm namely Unisys Lempel Zev Welch (LZW). As I said earlier, GIF uses the LZW compression algorithm. GIF became popular through the drastic development of internet. When Unisys learned that the LZW method was incorporated in the GIF specification, it immediately began negotiating with CompuServe in January of 1993. They reached an agreement with CompuServe on licensing the technology in June 1994, which calls for CompuServe to pay Unisys a royalty of 1% of the average selling price it charges for its software.

Unisys demands that the web sites that use GIF should pay them $5000 or more to use GIF graphics if the software originally used to create the GIFs was not covered by an appropriate Unisys license. Thus freebased people or open-based people are highly against Unisys and GIF, because other, much better, methods of data compression are not covered by any patent. They say that the flaw is in US patent system which makes even pencil-and-paper calculations patentable. One may easily violate some US patents by solving a problem found on Mathematics book! *Indians* might aware of the patent of *Basmati rice*!!!

People who are against to such silly patent, merely substitute PNG files, MNG files and shock waves (Flash) for GIF in their web pages. Open-based people are the one for open languages. Open language never claims royalties, etc. C, C++, Java, Linux are open. On the other side you've got proprietary language that claims royalties etc and it is closed. C# is one of proprietary languages. Microsoft often produces proprietary languages and so it has got so many opponents!

Note

Good discussion about "GIF politics" can be found on www.BurnAllGifs.org

31.2 GIFSAVE

GIFSAVE was developed by **Sverre H. Huseby**. It is a function to save the image in GIF format. **Sverre H. Huseby** says that GIFSAVE is little bit slow and the reason is Borland's `getpixel( )`function and not the GIFSAVE functions.

GIFSAVE consists of four functions, all declared in GIFSAVE.H:

1. `GIF_Create()` creates new GIF-files. It takes parameters specifying the filename, screen size, number of colors, and color resolution.
2. `GIF_SetColor()` sets up the red, green and blue color components. It should be called once for each possible color.
3. `GIF_CompressImage()` performs the compression of the image. It accepts parameters describing the position and size of the image on screen, and a user defined callback function that is supposed to fetch the pixel values.
4. `GIF_Close()` terminates and closes the file.

The functions should be called in the listed order for each GIF-file. One file must be closed before a new one is created.

31.3 Gifsave.h

```
#ifndef GIFSAVE_H
#define GIFSAVE_H

enum GIF_Code {
    GIF_OK,
    GIF_ERRCREATE,
    GIF_ERRWRITE,
    GIF_OUTMEM
};

int  GIF_Create(
        char *filename,
        int width, int height,
        int numcolors, int colorres
    );

void GIF_SetColor(
        int colornum,
        int red, int green, int blue
    );

int  GIF_CompressImage(
        int left, int top,
        int width, int height,
         int (*getpixel)(int x, int y)
     );
int  GIF_Close(void);

#endif
```

31.4 Gifsave.c

```
/**********************************************************************
 *  FILE:           GIFSAVE.C
 *
 *  MODULE OF:      GIFSAVE
 *
 *  DESCRIPTION:    Routines to create a GIF-file.
 **********************************************************************
*/

#include <stdlib.h>
#include <stdio.h>
#include "gifsave.h"

/**********************************************************************
 *                        P R I V A T E    D A T A
 **********************************************************************
*/

typedef unsigned Word;          /* At least two bytes (16 bits) */
typedef unsigned char Byte;     /* Exactly one byte (8 bits) */

/*=====================================================================
 *                              I/O Routines
 *=====================================================================
*/

static FILE *OutFile;           /* File to write to */

/*=====================================================================
 *                       Routines to write a bit-file
 *=====================================================================
*/

static Byte Buffer[256];        /* There must be one to much !!! */

static int  Index,              /* Current byte in buffer */
            BitsLeft;           /* Bits left to fill in current byte.
                                /* These are right-justified */
```

```
/*=====================================================================
 *                Routines to maintain an LZW-string table
 *=====================================================================
*/

#define RES_CODES 2

#define HASH_FREE 0xFFFF
#define NEXT_FIRST 0xFFFF

#define MAXBITS 12
#define MAXSTR (1 << MAXBITS)

#define HASHSIZE 9973
#define HASHSTEP 2039

#define HASH(index, lastbyte) (((lastbyte << 8) ^ index) % HASHSIZE)

static Byte *StrChr = NULL;
static Word *StrNxt = NULL,
            *StrHsh = NULL,
            NumStrings;

/*======================================================================
 *                              Main routines
 *======================================================================
*/

typedef struct {
    Word LocalScreenWidth,
         LocalScreenHeight;
    Byte GlobalColorTableSize : 3,
         SortFlag             : 1,
         ColorResolution      : 3,
         GlobalColorTableFlag : 1;
    Byte BackgroundColorIndex;
    Byte PixelAspectRatio;
} ScreenDescriptor;

typedef struct {
    Byte Separator;
    Word LeftPosition,
         TopPosition;
    Word Width,
         Height;
    Byte LocalColorTableSize : 3,
         Reserved            : 2,
         SortFlag           : 1,
         InterlaceFlag      : 1,
         LocalColorTableFlag : 1;
} ImageDescriptor;

static int  BitsPrPrimColor,    /* Bits pr primary color */
            NumColors;          /* Number of colors in color table */
static Byte *ColorTable = NULL;
static Word ScreenHeight,
            ScreenWidth,
            ImageHeight,
            ImageWidth,
            ImageLeft,
            ImageTop,
            RelPixX, RelPixY;         /* Used by InputByte() -function
*/
static int  (*GetPixel)(int x, int y);

/**********************************************************************
 *                   P R I V A T E    F U N C T I O N S
 **********************************************************************
*/

/*===================================================================
 *                         Routines to do file IO
 *===================================================================
*/

/*-------------------------------------------------------------------
 *  NAME:           Create()
 *
 *  DESCRIPTION:    Creates a new file, and enables referencing using
 *                  the global variable OutFile. This variable is only
 *                  used by these IO-functions, making it relatively
 *                  simple to rewrite file IO.
 *
 *  PARAMETERS:     filename - Name of file to create
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error opening the file
 *
 */

static int Create(char *filename)
{
    if ((OutFile = fopen(filename, "wb")) == NULL)
        return GIF_ERRCREATE;
    return GIF_OK;
}

/*-----------------------------------------------------------------------
 *  NAME:           Write()
 *
 *  DESCRIPTION:    Output bytes to the current OutFile.
 *
 *  PARAMETERS:     buf - Pointer to buffer to write
 *                  len - Number of bytes to write
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error writing to the file
 */

static int Write(void *buf, unsigned len)
{
    if (fwrite(buf, sizeof(Byte), len, OutFile) < len)
        return GIF_ERRWRITE;

    return GIF_OK;
}

/*-----------------------------------------------------------------------
 *  NAME:           WriteByte()
 *
 *  DESCRIPTION:    Output one byte to the current OutFile.
 *
 *  PARAMETERS:     b - Byte to write
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error writing to the file
 */

static int WriteByte(Byte b)
{
    if (putc(b, OutFile) == EOF)
        return GIF_ERRWRITE;

    return GIF_OK;
}

/*----------------------------------------------------------------------
 *  NAME:           WriteWord()
 *
 *  DESCRIPTION:    Output one word (2 bytes with byte-swapping, like on
 *                  the IBM PC) to the current OutFile.
 *
 *  PARAMETERS:     w - Word to write
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error writing to the file
 */

static int WriteWord(Word w)
{
    if (putc(w & 0xFF, OutFile) == EOF)
        return GIF_ERRWRITE;

    if (putc((w >> 8), OutFile) == EOF)
        return GIF_ERRWRITE;

    return GIF_OK;
}

/*-------------------------------------------------------------------
 *  NAME:           Close()
 *
 *  DESCRIPTION:    Close current OutFile.
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        Nothing
 */

static void Close(void)
{
    fclose(OutFile);
}

/*===================================================================
 *                      Routines to write a bit-file
 *===================================================================
*/

/*-------------------------------------------------------------------
 *  NAME:           InitBitFile()
 *
 *  DESCRIPTION:    Initiate for using a bitfile. All output is sent to
 *                  the current OutFile using the I/O-routines above.
 *
 *  PARAMETERS:     None
 *  RETURNS:        Nothing
 */
```

```
static void InitBitFile(void)
{
    Buffer[Index = 0] = 0;
    BitsLeft = 8;
}

/*-------------------------------------------------------------------
 *  NAME:           ResetOutBitFile()
 *
 *  DESCRIPTION:    Tidy up after using a bitfile
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        0 - OK, -1 - error
 */

static int ResetOutBitFile(void)
{
    Byte numbytes;
    /*
     *  Find out how much is in the buffer
     */
    numbytes = Index + (BitsLeft == 8 ? 0 : 1);

    /*
     *  Write whatever is in the buffer to the file
     */
    if (numbytes) {
        if (WriteByte(numbytes) != GIF_OK)
            return -1;

        if (Write(Buffer, numbytes) != GIF_OK)
            return -1;

        Buffer[Index = 0] = 0;
        BitsLeft = 8;
    }

    return 0;
}
/*--------------------------------------------------------------------
 *  NAME:           WriteBits()
 *
 *  DESCRIPTION:    Put the given number of bits to the outfile.
 *
 *  PARAMETERS:     bits    - bits to write from (right justified)
 *                  numbits - number of bits to write
 *
 *  RETURNS:        bits written, or -1 on error.
 */

static int WriteBits(int bits, int numbits)
{
    int  bitswritten = 0;
    Byte numbytes = 255;

    do {
        /*
         *  If the buffer is full, write it.
         */
        if ((Index == 254 && !BitsLeft) || Index > 254) {
            if (WriteByte(numbytes) != GIF_OK)
                return -1;

            if (Write(Buffer, numbytes) != GIF_OK)
                return -1;

            Buffer[Index = 0] = 0;
            BitsLeft = 8;
        }

        /*
         *  Now take care of the two specialcases
         */
        if (numbits <= BitsLeft) {
            Buffer[Index] |= (bits & ((1 << numbits) - 1)) << (8 -
BitsLeft);
            bitswritten += numbits;
            BitsLeft -= numbits;
            numbits = 0;
        } else {
            Buffer[Index] |= (bits & ((1 << BitsLeft) - 1)) << (8 -
BitsLeft);
            bitswritten += BitsLeft;
            bits >>= BitsLeft;
            numbits -= BitsLeft;

            Buffer[++Index] = 0;
            BitsLeft = 8;
        }
    } while (numbits);

    return bitswritten;
}
```

```
/*====================================================================
 *                 Routines to maintain an LZW-string table
 *====================================================================
*/

/*--------------------------------------------------------------------
 *  NAME:           FreeStrtab()
 *
 *  DESCRIPTION:    Free arrays used in string table routines
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        Nothing
 */

static void FreeStrtab(void)
{
    if (StrHsh) {
        free(StrHsh);
        StrHsh = NULL;
    }

    if (StrNxt) {
        free(StrNxt);
        StrNxt = NULL;
    }

    if (StrChr) {
        free(StrChr);
        StrChr = NULL;
    }
}

/*-------------------------------------------------------------------
 *  NAME:           AllocStrtab()
 *
 *  DESCRIPTION:    Allocate arrays used in string table routines
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        GIF_OK     - OK
 *                  GIF_OUTMEM - Out of memory
 */

static int AllocStrtab(void)
{
    /* Just in case . . .  */
    FreeStrtab();

    if ((StrChr = (Byte *) malloc(MAXSTR * sizeof(Byte))) == 0) {
        FreeStrtab();
        return GIF_OUTMEM;
    }

    if ((StrNxt = (Word *) malloc(MAXSTR * sizeof(Word))) == 0) {
        FreeStrtab();
        return GIF_OUTMEM;
    }

    if ((StrHsh = (Word *) malloc(HASHSIZE * sizeof(Word))) == 0) {
        FreeStrtab();
        return GIF_OUTMEM;
    }

    return GIF_OK;
}

/*-----------------------------------------------------------------------
 *  NAME:           AddCharString()
 *
 *  DESCRIPTION:    Add a string consisting of the string of index plus
 *                  the byte b.
 *
 *                  If a string of length 1 is wanted, the index should
 *                  be 0xFFFF.
 *
 *  PARAMETERS:     index - Index to first part of string, or 0xFFFF is
 *                          only 1 byte is wanted
 *                  b     - Last byte in new string
 *
 *  RETURNS:        Index to new string, or 0xFFFF if no more room
 *
 */
static Word AddCharString(Word index, Byte b)
{
    Word hshidx;

    /*
     *  Check if there is more room
     */
    if (NumStrings >= MAXSTR)
        return 0xFFFF;
    /*
     *  Search the string table until a free position is found
     */
    hshidx = HASH(index, b);
    while (StrHsh[hshidx] != 0xFFFF)
        hshidx = (hshidx + HASHSTEP) % HASHSIZE;

    /*
     *  Insert new string
     */
    StrHsh[hshidx] = NumStrings;
    StrChr[NumStrings] = b;
    StrNxt[NumStrings] = (index != 0xFFFF) ? index : NEXT_FIRST;

    return NumStrings++;
}

/*----------------------------------------------------------------------
 *  NAME:           FindCharString()
 *
 *  DESCRIPTION:    Find index of string consisting of the string of
 *                  index plus the byte b.
 *
 *                  If a string of length 1 is wanted, the index should
 *                  be 0xFFFF.
 *
 *  PARAMETERS:     index - Index to first part of string, or 0xFFFF is
 *                          only 1 byte is wanted
 *                  b     - Last byte in string
 *
 *  RETURNS:        Index to string, or 0xFFFF if not found
 */

static Word FindCharString(Word index, Byte b)
{
    Word hshidx, nxtidx;

    /*
     *  Check if index is 0xFFFF. In that case we need only
     *  return b, since all one-character strings has their
     *  bytevalue as their index
     */
    if (index == 0xFFFF)
        return b;

    /*
     *  Search the string table until the string is found, or
     *  we find HASH_FREE. In that case the string does not
     *  exist.
     */
    hshidx = HASH(index, b);
    while ((nxtidx = StrHsh[hshidx]) != 0xFFFF) {
        if (StrNxt[nxtidx] == index && StrChr[nxtidx] == b)
            return nxtidx;
        hshidx = (hshidx + HASHSTEP) % HASHSIZE;
    }

    /*
     *  No match is found
     */
    return 0xFFFF;
}

/*-------------------------------------------------------------------
 *  NAME:           ClearStrtab()
 *
 *  DESCRIPTION:    Mark the entire table as free, enter the 2**codesize
 *                  one-byte strings, and reserve the RES_CODES reserved
 *                  codes.
 *
 *  PARAMETERS:     codesize - Number of bits to encode one pixel
 *
 *  RETURNS:        Nothing
 */

static void ClearStrtab(int codesize)
{
    int q, w;
    Word *wp;

    /*
     *  No strings currently in the table
     */
    NumStrings = 0;

    /*
     *  Mark entire hashtable as free
     */
    wp = StrHsh;
    for (q = 0; q < HASHSIZE; q++)
        *wp++ = HASH_FREE;
    /*
     *  Insert 2**codesize one-character strings, and reserved codes
     */
    w = (1 << codesize) + RES_CODES;
    for (q = 0; q < w; q++)
        AddCharString(0xFFFF, q);
}

/*=====================================================================
 *                          LZW compression routine
 *=====================================================================
*/

/*---------------------------------------------------------------------
 *  NAME:           LZW_Compress()
 *
 *  DESCRIPTION:    Perform LZW compression as specified in the
 *                  GIF-standard.
 *
 *  PARAMETERS:     codesize  - Number of bits needed to represent
 *                              one pixelvalue.
 *                  inputbyte - Function that fetches each byte to
 *                              compress.
 *                              Must return -1 when no more bytes.
 *
 *  RETURNS:        GIF_OK     - OK
 *                  GIF_OUTMEM - Out of memory
 */

static int LZW_Compress(int codesize, int (*inputbyte)(void))
{
    register int c;
    register Word index;
    int  clearcode, endofinfo, numbits, limit, errcode;
    Word prefix = 0xFFFF;

    /* Set up the given outfile */
    InitBitFile();

    /*
     *  Set up variables and tables
     */
    clearcode = 1 << codesize;
    endofinfo = clearcode + 1;

    numbits = codesize + 1;
    limit = (1 << numbits) - 1;

    if ((errcode = AllocStrtab()) != GIF_OK)
        return errcode;
```

```
ClearStrtab(codesize);

/*
 *  First send a code telling the unpacker to clear the stringtable.
 */
WriteBits(clearcode, numbits);

/*
 *  Pack image
 */
while ((c = inputbyte()) != -1) {
    /*
     *  Now perform the packing.
     *  Check if the prefix + the new character is a string that
     *  exists in the table
     */
    if ((index = FindCharString(prefix, c)) != 0xFFFF) {
        /*
         *  The string exists in the table.
         *  Make this string the new prefix.
         */
        prefix = index;

    } else {
        /*
         *  The string does not exist in the table.
         *  First write code of the old prefix to the file.
         */
        WriteBits(prefix, numbits);

        /*
         *  Add the new string (the prefix + the new character)
         *  to the stringtable.
         */
        if (AddCharString(prefix, c) > limit) {
            if (++numbits > 12) {
                WriteBits(clearcode, numbits - 1);
                ClearStrtab(codesize);
                numbits = codesize + 1;
            }
            limit = (1 << numbits) - 1;
        }
        /*
         *  Set prefix to a string containing only the character
         *  read. Since all possible one-character strings exists
         *  int the table, there's no need to check if it is found.
         */
```

```
            prefix = c;
        }
    }

    /*
     *  End of info is reached. Write last prefix.
     */
    if (prefix != 0xFFFF)
        WriteBits(prefix, numbits);

    /*
     *  Write end of info -mark.
     */
    WriteBits(endofinfo, numbits);

    /*
     *  Flush the buffer
     */
    ResetOutBitFile();

    /*
     *  Tidy up
     */
    FreeStrtab();

    return GIF_OK;
}

/*======================================================================
 *                            Other routines
 *======================================================================
*/

/*----------------------------------------------------------------------
 *  NAME:           BitsNeeded()
 *
 *  DESCRIPTION:    Calculates number of bits needed to store numbers
 *                  between 0 and n - 1
 *
 *  PARAMETERS:     n - Number of numbers to store (0 to n - 1)
 *
 *  RETURNS:        Number of bits needed
 */

static int BitsNeeded(Word n)
{
    int ret = 1;
    if (!n--)
        return 0;

    while (n >>= 1)
        ++ret;

    return ret;
}

/*-------------------------------------------------------------------
 *  NAME:           InputByte()
 *
 *  DESCRIPTION:    Get next pixel from image. Called by the
 *                  LZW_Compress()-function
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        Next pixelvalue, or -1 if no more pixels
 */

static int InputByte(void)
{
    int ret;

    if (RelPixY >= ImageHeight)
        return -1;

    ret = GetPixel(ImageLeft + RelPixX, ImageTop + RelPixY);

    if (++RelPixX >= ImageWidth) {
        RelPixX = 0;
        ++RelPixY;
    }

    return ret;
}

/*-----------------------------------------------------------------
 *  NAME:           WriteScreenDescriptor()
 *
 *  DESCRIPTION:    Output a screen descriptor to the current GIF-file
 *
 *  PARAMETERS:     sd - Pointer to screen descriptor to output
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error writing to the file
 */
```

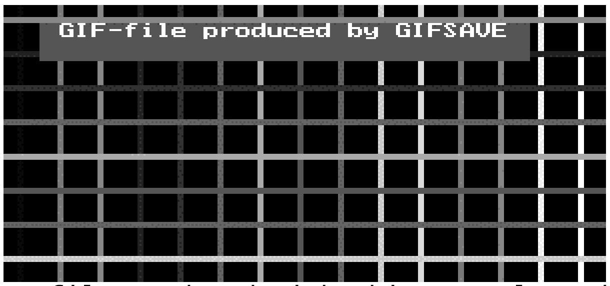
```
static int WriteScreenDescriptor(ScreenDescriptor *sd)
{
    Byte tmp;

    if (WriteWord(sd->LocalScreenWidth) != GIF_OK)
        return GIF_ERRWRITE;
    if (WriteWord(sd->LocalScreenHeight) != GIF_OK)
        return GIF_ERRWRITE;
    tmp = (sd->GlobalColorTableFlag << 7)
          | (sd->ColorResolution << 4)
          | (sd->SortFlag << 3)
          | sd->GlobalColorTableSize;
    if (WriteByte(tmp) != GIF_OK)
        return GIF_ERRWRITE;
    if (WriteByte(sd->BackgroundColorIndex) != GIF_OK)
        return GIF_ERRWRITE;
    if (WriteByte(sd->PixelAspectRatio) != GIF_OK)
        return GIF_ERRWRITE;

    return GIF_OK;
}

/*-------------------------------------------------------------------
 *  NAME:           WriteImageDescriptor()
 *
 *  DESCRIPTION:    Output an image descriptor to the current GIF-file
 *
 *  PARAMETERS:     id - Pointer to image descriptor to output
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error writing to the file
 */

static int WriteImageDescriptor(ImageDescriptor *id)
{
    Byte tmp;

    if (WriteByte(id->Separator) != GIF_OK)
        return GIF_ERRWRITE;
    if (WriteWord(id->LeftPosition) != GIF_OK)
        return GIF_ERRWRITE;
    if (WriteWord(id->TopPosition) != GIF_OK)
        return GIF_ERRWRITE;
    if (WriteWord(id->Width) != GIF_OK)
        return GIF_ERRWRITE;
    if (WriteWord(id->Height) != GIF_OK)
        return GIF_ERRWRITE;
    tmp = (id->LocalColorTableFlag << 7)
          | (id->InterlaceFlag << 6)
          | (id->SortFlag << 5)
          | (id->Reserved << 3)
          | id->LocalColorTableSize;
    if (WriteByte(tmp) != GIF_OK)
        return GIF_ERRWRITE;
    return GIF_OK;
}

/********************************************************************
 *                    P U B L I C    F U N C T I O N S
 ********************************************************************/

/*-------------------------------------------------------------------
 *  NAME:           GIF_Create()
 *
 *  DESCRIPTION:    Create a GIF-file, and write headers for both screen
 *                  and image.
 *
 *  PARAMETERS:     filename  - Name of file to create (including
 *                              extension)
 *                  width     - Number of horisontal pixels on screen
 *                  height    - Number of vertical pixels on screen
 *                  numcolors - Number of colors in the colormaps
 *                  colorres  - Color resolution. Number of bits for
 *                              each primary color
 *
 *  RETURNS:        GIF_OK        - OK
 *                  GIF_ERRCREATE - Couldn't create file
 *                  GIF_ERRWRITE  - Error writing to the file
 *                  GIF_OUTMEM    - Out of memory allocating color table
 */
int GIF_Create(char *filename, int width, int height,
               int numcolors, int colorres)
{
    int q, tabsize;
    Byte *bp;
    ScreenDescriptor SD;

    /*
     *  Initiate variables for new GIF-file
     */
    NumColors = numcolors ? (1 << BitsNeeded(numcolors)) : 0;
    BitsPrPrimColor = colorres;
    ScreenHeight = height;
    ScreenWidth = width;
    /*
     *  Create file specified
     */
    if (Create(filename) != GIF_OK)
        return GIF_ERRCREATE;

    /*
     *  Write GIF signature
     */
    if ((Write("GIF87a", 6)) != GIF_OK)
        return GIF_ERRWRITE;

    /*
     *  Initiate and write screen descriptor
     */
    SD.LocalScreenWidth = width;
    SD.LocalScreenHeight = height;
    if (NumColors) {
        SD.GlobalColorTableSize = BitsNeeded(NumColors) - 1;
        SD.GlobalColorTableFlag = 1;
    } else {
        SD.GlobalColorTableSize = 0;
        SD.GlobalColorTableFlag = 0;
    }
    SD.SortFlag = 0;
    SD.ColorResolution = colorres - 1;
    SD.BackgroundColorIndex = 0;
    SD.PixelAspectRatio = 0;
    if (WriteScreenDescriptor(&SD) != GIF_OK)
        return GIF_ERRWRITE;

    /*
     *  Allocate color table
     */
    if (ColorTable) {
        free(ColorTable);
        ColorTable = NULL;
    }
    if (NumColors) {
        tabsize = NumColors * 3;

        if ((ColorTable = (Byte *) malloc(tabsize * sizeof(Byte))) ==
NULL)
            return GIF_OUTMEM;

        else {
            bp = ColorTable;
            for (q = 0; q < tabsize; q++)
                *bp++ = 0;
        }
    }
    return 0;
}

/*-----------------------------------------------------------------------
 *  NAME:           GIF_SetColor()
 *
 *  DESCRIPTION:    Set red, green and blue components of one of the
 *                  colors. The color components are all in the range
 *                  [0, (1 << BitsPrPrimColor) - 1]
 *
 *  PARAMETERS:     colornum - Color number to set. [0, NumColors - 1]
 *                  red      - Red component of color
 *                  green    - Green component of color
 *                  blue     - Blue component of color
 *
 *  RETURNS:        Nothing
 */

void GIF_SetColor(int colornum, int red, int green, int blue)
{
    long maxcolor;
    Byte *p;

    maxcolor = (1L << BitsPrPrimColor) - 1L;
    p = ColorTable + colornum * 3;
    *p++ = (Byte) ((red * 255L) / maxcolor);
    *p++ = (Byte) ((green * 255L) / maxcolor);
    *p++ = (Byte) ((blue * 255L) / maxcolor);
}
/*----------------------------------------------------------------------
 *  NAME:           GIF_CompressImage()
 *
 *  DESCRIPTION:    Compress an image into the GIF-file previousely
 *                  created using GIF_Create(). All color values should
 *                  have been specified before this function is called.
 *
 *                  The pixels are retrieved using a user defined
 *                  callback function. This function should accept two
 *                  parameters, x and y, specifying which pixel to
 *                  retrieve. The pixel values sent to this function are
 *                  as follows:
 *
 *                    x : [ImageLeft, ImageLeft + ImageWidth - 1]
 *                     y : [ImageTop, ImageTop + ImageHeight - 1]
 *
 *                   The function should return the pixel value for the
 *                   point given, in the interval [0, NumColors - 1]
 *
 *  PARAMETERS:      left     - Screen-relative leftmost pixel
 *                              x-coordinate of the image
 *                   top      - Screen-relative uppermost pixel
 *                              y-coordinate of the image
 *                   width    - Width of the image, or -1 if as wide as
 *                              the screen
 *                   height   - Height of the image, or -1 if as high as
 *                              the screen
 *                   getpixel - Address of user defined callback
 *                              function.
 *                              (See above)
 *
 *  RETURNS:         GIF_OK       - OK
 *                   GIF_OUTMEM   - Out of memory
 *                   GIF_ERRWRITE - Error writing to the file
 */

int GIF_CompressImage(int left, int top, int width, int height,
                      int (*getpixel)(int x, int y))
{
    int codesize, errcode;
    ImageDescriptor ID;

    if (width < 0) {
        width = ScreenWidth;
        left = 0;
    }

    if (height < 0) {
        height = ScreenHeight;
        top = 0;
    }
    if (left < 0)
        left = 0;
    if (top < 0)
        top = 0;

    /*
     *  Write global colortable if any
     */
    if (NumColors)
        if ((Write(ColorTable, NumColors * 3)) != GIF_OK)
```

```
            return GIF_ERRWRITE;

    /*
     *  Initiate and write image descriptor
     */
    ID.Separator = ',';
    ID.LeftPosition = ImageLeft = left;
    ID.TopPosition = ImageTop = top;
    ID.Width = ImageWidth = width;
    ID.Height = ImageHeight = height;
    ID.LocalColorTableSize = 0;
    ID.Reserved = 0;
    ID.SortFlag = 0;
    ID.InterlaceFlag = 0;
    ID.LocalColorTableFlag = 0;

    if (WriteImageDescriptor(&ID) != GIF_OK)
        return GIF_ERRWRITE;

    /*
     *  Write code size
     */
    codesize = BitsNeeded(NumColors);
    if (codesize == 1)
        ++codesize;
    if (WriteByte(codesize) != GIF_OK)
        return GIF_ERRWRITE;

    /*
     *  Perform compression
     */
    RelPixX = RelPixY = 0;
    GetPixel = getpixel;
    if ((errcode = LZW_Compress(codesize, InputByte)) != GIF_OK)
        return errcode;

    /*
     *  Write terminating 0-byte
     */
    if (WriteByte(0) != GIF_OK)
        return GIF_ERRWRITE;

    return GIF_OK;
}

/*-------------------------------------------------------------------
 *  NAME:           GIF_Close()
 *  DESCRIPTION:    Close the GIF-file
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error writing to file
 */
int GIF_Close(void)
{
    ImageDescriptor ID;

    /*
     *  Initiate and write ending image descriptor
     */
    ID.Separator = ';';
    if (WriteImageDescriptor(&ID) != GIF_OK)
        return GIF_ERRWRITE;
    /*
     *  Close file
     */
    Close();
    /*
     *  Release color table
     */
    if (ColorTable) {
        free(ColorTable);
        ColorTable = NULL;
    }
    return GIF_OK;
}
```

Compile the above Gifsave.c file to create the Gifsave.lib file. Using Gifsave.lib & Gifsave.h files we can create GIF files quickly.

31.5 Example usage of GIFSAVE

Following example code shows how to use the GIFSAVE library in our program to create a GIF file.



GIF file produced with this Example code

```
/**************************************************************************
 *  FILE:           EXAMPLE.C
 *
 *  MODULE OF:      EXAMPLE
 *
 *  DESCRIPTION:    Example program using GIFSAVE.
 *
 *                  Produces output to an EGA-screen, then dumps it to
 *                  a GIF-file.
 *************************************************************************
*/

#ifndef __TURBOC__
  #error This program must be compiled using a Borland C compiler
#endif

#include <stdlib.h>
#include <stdio.h>
#include <graphics.h>

#include "gifsave.h"

/************************************************************************
 *                   P R I V A T E   F U N C T I O N S
 ************************************************************************
*/

/*-----------------------------------------------------------------------
 *  NAME:           DrawScreen()
 *
 *  DESCRIPTION:    Produces some output on the graphic screen.
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        Nothing
 */
static void DrawScreen(void)
{
    int  color = 1, x, y;
    char *text = "GIF-file produced by GIFSAVE";

    /*
     *  Output some lines
     */
    setlinestyle(SOLID_LINE, 0, 3);
    for (x = 10; x < getmaxx(); x += 20) {
        setcolor(color);
```

```
            line(x, 0, x, getmaxy());
            if (++color > getmaxcolor())
                color = 1;
        }
        for (y = 8; y < getmaxy(); y += 17) {
            setcolor(color);
            line(0, y, getmaxx(), y);
            if (++color > getmaxcolor())
                color = 1;
        }

        /*
         *  And then some text
         */
        setfillstyle(SOLID_FILL, DARKGRAY);
        settextstyle(TRIPLEX_FONT, HORIZ_DIR, 4);
        bar(20, 10, textwidth(text) + 40, textheight(text) + 20);
        setcolor(WHITE);
        outtextxy(30, 10, text);
    }

    /*------------------------------------------------------------------
     *  NAME:           gpixel()
     *
     *  DESCRIPTION:    Callback function. Near version of getpixel()
     *
     *                  If this program is compiled with a model using
     *                  far code, Borland's getpixel() can be used
     *                  directly.
     *
     *  PARAMETERS:     As for getpixel()
     *
     *  RETURNS:        As for getpixel()
     */

    static int gpixel(int x, int y)
    {
        return getpixel(x, y);
    }

    /*------------------------------------------------------------------
     *  NAME:           GIF_DumpEga10()
     *
     *  DESCRIPTION:    Outputs a graphics screen to a GIF-file. The screen
     *                  must be in the mode 0x10, EGA 640x350, 16 colors.
     *
     *                  No error checking is done! Probably not a very good
```

```
 *                  example, then . . . :-)
 *
 *  PARAMETERS:     filename - Name of GIF-file
 *
 *  RETURNS:        Nothing
 */

static void GIF_DumpEga10(char *filename)
{
  #define WIDTH             640  /* 640 pixels across screen */
  #define HEIGHT            350  /* 350 pixels down screen */
  #define NUMCOLORS          16  /* Number of different colors */
  #define BITS_PR_PRIM_COLOR 2  /* Two bits pr primary color */

    int q,                       /* Counter */
        color,                   /* Temporary color value */
        red[NUMCOLORS],          /* Red component for each color */
        green[NUMCOLORS],        /* Green component for each color */
        blue[NUMCOLORS];         /* Blue component for each color */
    struct palettetype pal;

    /*
     *  Get the color palette, and extract the red, green and blue
     *  components for each color. In the EGA palette, colors are
     *  stored as bits in bytes:
     *
     *      00rgbRGB
     *
     *  where r is low intensity red, R is high intensity red, etc.
     *  We shift the bits in place like
     *
     *      000000Rr
     *
     *  for each component
     */
    getpalette(&pal);
    for (q = 0; q < NUMCOLORS; q++) {
        color = pal.colors[q];
        red[q]   = ((color & 4) >> 1) | ((color & 32) >> 5);
        green[q] = ((color & 2) >> 0) | ((color & 16) >> 4);
        blue[q]  = ((color & 1) << 1) | ((color & 8) >> 3);
    }

    /*
     *  Create and set up the GIF-file
     */
    GIF_Create(filename, WIDTH, HEIGHT, NUMCOLORS, BITS_PR_PRIM_COLOR);
    /*
     *  Set each color according to the values extracted from
     *  the palette
     */
    for (q = 0; q < NUMCOLORS; q++)
        GIF_SetColor(q, red[q], green[q], blue[q]);

    /*
     *  Store the entire screen as an image using the user defined
     *  callback function gpixel() to get pixel values from the screen
     */
    GIF_CompressImage(0, 0, -1, -1, gpixel);

    /*
     *  Finish it all and close the file
     */
    GIF_Close();
}

/**********************************************************************
 *                     P U B L I C    F U N C T I O N S
 **********************************************************************
*/
int main(void)
{
    int gdr, gmd, errcode;

    /* Initiate graphics screen for EGA mode 0x10, 640x350x16 */

    gdr = EGA;
    gmd = EGAHI;
    initgraph(&gdr, &gmd, "");
    if ((errcode = graphresult()) != grOk) {
        printf("Graphics error: %s\n", grapherrormsg(errcode));
        exit(-1);
    }
    /* Put something on the screen    */
    DrawScreen();

    /* Dump the screen to a GIF-file  */
    GIF_DumpEga10("EXAMPLE.GIF");

    /* Return to text mode            */
    closegraph();
    return 0;
}
```