08-6465-SIS-ZCH66

MARCH 9, 2006

2.1

Application Programmers Interface for GIF Decoder

ABSTRACT:

Application Programmers Interface for GIF Decoder

KEYWORDS:

Multimedia codecs, Image, GIF

APPROVED:

Wang Zening

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.1	24-Jun-2004	Shailesh R	Created and reviewed at MIEL
1.0	11-Nov-2004	Sameer Rapate, Shailesh R	Updated with new API for animated GIF
1.1	29-Dec--2004	Sameer Rapate Shailesh R	Updated for Release 1.0
1.2	04-Apr-2005	Sameer Rapate Shailesh R	Added description for RGB formats
1.3	20-Jan-2006	Atul Duggal	Edited variable names for output formats and scaling modes.
1.4	25-Jan-2006	Kunal Goel	Updated with review comments
2.0	06-Feb-2006	Lauren Post	Using new format
2.1	19-Nov-08	Eagle Zhou	Add BGR output format and api version

Table of Contents

Introduction ... **5**
1.1 Purpose ... 5
1.2 Scope ... 5
1.3 Audience Description ... 5
1.4 References ... 5
1.4.1 References ... 5
1.4.2 Freescale Multimedia References ... 5
1.5 Definitions, Acronyms, and Abbreviations ... 6
1.6 Document Location ... 6
2 API Description ... **7**
2.1 Frame by Frame Decoding ... 7
3 GIF Decoder – Data Structures ... **8**
3.1 Basic Data Types ... 8
3.2 GIF_DECODER_OBJECT ... 8
3.3 GIF_MEM_ALLOC_INFO ... 9
3.4 GIF_DECODER_PARAMS ... 10
3.5 GIF_DECODER_INFO_INIT ... 11
4 GIF Decoder - Interface ... **14**
4.1 Memory Query ... 14
4.2 Initialization ... 15
4.3 Decoding ... 16
4.4 API Version ... 16
4.5 Function implemented by application ... 17
4.6 Suspension ... 17
4.7 Overview of API Usage ... 18
Appendix A RGB/BGR output formats supported ... **19**
A-1 RGB888 FORMAT ... 19
A-1-1 Unwrapped format ... 19
A-1-2 Wrapped format ... 19
A-2 RGB565 FORMAT ... 20
A-2-1 Unwrapped format ... 20
A-2-2 Wrapped format ... 20
A-3 RGB555 FORMAT ... 21
A-3-1 Unwrapped format ... 21
A-3-2 Wrapped format ... 21
A-4 RGB666 FORMAT ... 22
A-4-1 Unwrapped format ... 22
A-4-2 Wrapped format ... 22
A-5 BGR FORMAT ... 22
Appendix B Suspension and Resumption Mechanism ... **23**

Appendix C Debug and Log Support..25

Introduction

1.1 Purpose

This document gives the application programmer's interface for the GIF Decoder. The purpose of this document is to specify the functional interface of the GIF decoder.

1.2 Scope

This document describes only the functional interface of the GIF decoder. It does not describe the internal design of the decoder. Specifically, it describes only those functions needed by a software module to use the decoder.

The GIF decoder decodes GIF formats for image storage, with the following features:

- The GIF decoder supports GIF files containing more than one image with 1 to 8 bits per pixel (GIF 87a and 89a)
- Supports LZW compression method to compress image data.
- Supports interlacing in the image data
- Supports transparency in the images
- Supports animation of images
- The output formats supported are 24 bit RGB (RGB888), 16 bit RGB565, 15 bit RGB555, 18 bit RGB666 and corresponding BGR format.

1.3 Audience Description

The reader is expected to have basic understanding of GIF decoding. The intended audience for this document is the development community who wish to use the GIF decoder in their systems.

1.4 References

1.4.1 References

- Compressed Image File formats by John Miano, ACM Press/Addison Wesley Longman.

1.4.2 Freescale Multimedia References

- GIF Decoder Application Programming Interface - gif_dec_api.doc
- GIF Decoder Requirements Book - gif_dec_reqb.doc
- GIF Decoder Test Plan - gif_dec_test_plan.doc
- GIF Decoder Release notes - gif_dec_release_notes.doc
- GIF Decoder Test Results – gif_dec_test_results.doc
- GIF Decoder Performance Results – gif_dec_perf_results.doc
- GIF Decoder Interface Header – gif_def_interface.h
- GIF Decoder Application Code – gif_test.c

1.5 Definitions, Acronyms, and Abbreviations

TERM/ACRONYM	DEFINITION
API	Application Programming Interface
ARM	Advanced RISC Machine
BMP	Bitmap
FSL	Freescale
IEC	International Electro-technical Commission
ISO	International Organization for Standardization
OS	Operating System
RGB	Raw pixel data organized in the order of Red, green and blue components. RGB888 denotes 8 bits per pixel each for R, G and B components
BGR	Raw pixel data organized in the order of Blue, green and red components. BGR888 denotes 8 bits per pixel each for B, G and R components
RVDS	ARM RealView Development Suite
TBD	To Be Determined
UNIX	Linux PC x/86 C-reference binaries

1.6 Document Location

docs/gif_dec

2 API Description

The external software interface to the GIF Decoder consists of the following functions:

GIF_query_dec_mem	:	Memory query
GIF_decoder_init	:	Initialization
GIF_query_dec_mem_frame	:	Memory query for a frame
GIF_decoder_init_frame	:	Initialization for a frame
GIF_decode	:	Decoding and post processing

The GIF decoder is provided as a library that contains the relevant routines including GIF_query_dec_mem, GIF_query_dec_mem_frame, GIF_decoder_init, GIF_decoder_init_frame and GIF_decode.

Function for reading the data from input stream needs to be implemented by the calling application. The GIF decoder API uses function pointers to invoke this function.

GIF_get_new_data :

Function pointer to the function that reads data from the input stream (the function needs to be implemented by calling application)

2.1 Frame by Frame Decoding

In the GIF file format, the file header includes a "screen size" expressed in pixels. Each frame within the file has a “frame size” which is the actual data area of the frame, plus an x,y offset that allows the frame to be positioned within the screen area defined
by the file header.

For a multi-frame (animated) GIF, the decoder library would parse the GIF global data, provide the necessary information to the application. The application needs to then allocate memory for each frame, initialize the GIF decoder with frame information by calling GIF_decoder_init_frame(), allocate the output buffer for each frame. For each frame, the calling function calls needs to call the GIF_decode().After decoding of each frame the output buffer contains decoded data of that frame. The application is expected to maintain a screen area (buffer) equal to the screen size declared in the header and inserts the successive frames at the locations indicated (by the decoder library) within this area.

3 GIF Decoder – Data Structures

3.1 Basic Data Types

```
typedef                 int                     GIF_INT32;
typedef                 unsigned int            GIF_UINT32;
typedef                 char                    GIF_INT8;
typedef                 unsigned char           GIF_UINT8;
typedef                 short                   GIF_INT16;
typedef                 unsigned short          GIF_UINT16;
```

3.2 GIF_DECODER_OBJECT

In order to call any GIF decode function, the application that calls the GIF decoder needs to create a new instance of the decoder object. The calling application maintains a list of pointers to all currently active instances of the object, and manages them. The caller should also ensure that there is sufficient memory available to run the instance that is being created. All data structures used by the GIF functions need to be allocated by the caller on a per instance basis, and hence are part of GIF Decoder Object instance structure. Input data that is required for this particular instance of the decoder should be filled into the instance structure by the calling function. After completion of the intended functions, the caller needs to delete the instance and free all memory associated with it.

```
typedef struct
{
      GIF_Mem_Alloc_Info       mem_info;
      GIF_Decoder_Params       dec_param;
      GIF_Decoder_Info_Init    dec_info_init;
      GIF_Decoder_Info_Dec     dec_info_dec;
      GIFD_RET_TYPE   (*GIF_get_new_data)(GIF_UINT8**,GIF_UINT32   *,struct
      GIF_Decoder_Object *);
      void                     *vptr;
      GIF_INT32                      number_of_frames;
      GIF_INT32                      bytes_read_in_a_frame;
} GIF_Decoder_Object;
```

Element	Description
GIF_Mem_Alloc_Info mem_info	Filled by decoder in GIF_query_dec_mem function
GIF_Decoder_Params dec_param	Caller needs to fill this structure before calling the decoder functions
GIF_Decoder_Info_Init dec_info_init	GIF_decoder_init fills this structure up, which can be used by the

	caller
GIFD_RET_TYPE (*GIF_get_new_data)(GIF_UINT8**,GIF_UINT32 *,struct GIF_Decoder_Object *);	Function pointer to the function to read new data
void *vptr	Codec specific structure elements not needed by caller
number_of_frames	Count of the number of frames
bytes_read_in_a_frame	Number of bytes read in a frame

3.3 GIF_MEM_ALLOC_INFO

GIF_Mem_Alloc_Info is filled by the decoder in GIF_query_dec_mem function, which specifies the number of memory requests and each request has size, alignment, and type (Fast or Slow) of the memory need to be allocated. After querying for memory, application has to allocate the required memory and assign pointers for all requests.

```
typedef struct
{
      GIF_INT32                   num_reqs;
      GIF_Mem_Alloc_Info_Sub  mem_info_sub[MAX_NUM_MEM_REQS];
}GIF_Mem_Alloc_Info;
```

Element	Description
num_reqs	Number of valid memory requests
GIF_Mem_Alloc_Info_Sub mem_info_sub[MAX_NUM_MEM_REQS]	Pointer to structure containing memory size, type, alignment and ptr.
MAX_NUM_MEM_REQS	Currently 10

```
typedef struct
{
      GIF_INT32         size;        /* Size in bytes */
      GIF_Mem_type      type;        /* Memory type Fast or Slow */
      GIF_INT32         align;       /* Alignment of memory in bytes */
      void              *ptr;        /* Pointer to the memory */
}GIF_Mem_Alloc_Info_Sub;
```

Element	Description
Size	Memory size
GIF Mem type type	Memory type –fast or slow
Align	Alignment of memory in bytes
void *ptr	Pointer to memory

```
typedef   enum
{
      E_FAST_MEMORY,
      E_SLOW_MEMORY
}GIF_Mem_type;
```

3.4 GIF_DECODER_PARAMS

GIF_Decoder_Params needs to be filled by the application calling the GIF decoder before it calls the Decoder functions. The calling application needs to indicate the desired output format. In case the calling application needs the GIF decoder to also rescale the decoded output, it needs to set the sw_scaling_set structure member to 1. In such a case, the calling application also provides information on the width and height of output to be displayed. It should be noted that it is the responsibility of the calling application to ensure that all the structure members of GIF_Decoder_Params are initialized to the correct values.

If the scaling feature is turned on, the calling application provides the desired width and height that the decoded image (specifically the width and height of 'logical screen' as defined by the GIF spec) must be scaled down to. The scaling factor will be an integer and the aspect ratio of the image (i.e. the 'logical screen' as defined by the GIF spec) will be preserved.

```
typedef struct
{
    gif_output_format   outformat;
    gif_scaling_mode    scale_mode;
    GIF_UINT16          output_width;
    GIF_UINT16          output_height;
} GIF_Decoder_Params;
```

Element	Description
gif output format outformat	Enum for output formats supported
gif scaling mode scale mode	Enum for scaling mode
Output width	Width of output to be displayed
Output height	Height of output to be displayed

```
typedef enum
{
      E_GIF_OUTPUTFORMAT_RGB888,
      E_GIF_OUTPUTFORMAT_RGB565,
      E_GIF_OUTPUTFORMAT_RGB555,
      E_GIF_OUTPUTFORMAT_RGB666,
      E_GIF_OUTPUTFORMAT_BGR888,
      E_GIF_OUTPUTFORMAT_BGR565,
      E_GIF_OUTPUTFORMAT_BGR555,
      E_GIF_OUTPUTFORMAT_BGR666,
     E_GIF_LAST_OUTPUT_FORMAT
}gif_output_format;
```

For more details on these formats refer to Appendix B [**Error! Reference source not found.**]

This enum for the output format indexes into an array of function pointers – the functions are responsible for rendering the output in the required format.

```
typedef enum
{
      E_GIF_NO_SCALE,                    /* No software scaling */
      E_GIF_INT_SCALE_PRESERVE_AR,      /* Software scaling using
                                         integer scaling factor
```

```
                                                   preserving pixel aspect ratio
                                                   */
      E_GIF_LAST_SCALE_MODE
} gif_scaling_mode;
```

3.5 GIF_DECODER_INFO_INIT

GIF_Decoder_Info_Init is filled by the decoder whenever the application invoking the GIF decoder calls the GIF decoder initialization function GIF_decoder_init.

The information that is available after the initialization includes the width, height, number of bits per pixel (1 to 8 bits per pixel) in the global header, flags, background color and pixel aspect ratio of the global screen; the width, height, position of the individual images to be displayed in the global screen,rendering width and height of each image,local color table size,flag for interlaced images and flag indicating local color table.

```
typedef struct
{
      /*Global Fields*/
      GIF_INT16 globwidth;          /*Width of the global screen*/
      GIF_INT16 globheight;         /*Height of the global screen*/
      GIF_INT16 glob_out_width;     /*Width of the global screen*/
      GIF_INT16 glob_out_height;    /*Height of the global screen*/

      GIF_UINT8 globpixbits;        /*Number of bits per pixel in global
                                    table*/
      GIF_UINT8 globbc;             /*Back ground color*/
      GIF_UINT8 globaspect;         /*Pixel aspect ratio*/
      GIF_UINT8 glob_color_tbl_size;/*2 power N+1 gives entries in color
                                    table*/
      GIF_UINT8 glob_color_tbl_sort_flag;/*Color table Sort Flag*/
      GIF_UINT8 glob_bpp;                /*Bits per pixel minus 1*/
      GIF_UINT8 glob_color_tbl_flag;/*Set if Global color table is
                                    present*/

      /*Local Fields*/
      GIF_INT16 image_left;/*Left offset of Image within logical screen*/
      GIF_INT16 image_top;/*Top offset of Image within logical screen*/
      GIF_INT16 scaled_image_left; /*Scaled left offset of Image within
                                    logical screen*/
      GIF_INT16 scaled_image_top; /*Scaled top offset of Image within
                                    logical screen*/
      GIF_INT16 image_width;        /*Input Image width*/
      GIF_INT16 image_height;       /*Input Image height*/
      GIF_INT16 out_image_width;    /*Output Image width*/
      GIF_INT16 out_image_height;   /*Output Image height*/
      GIF_UINT8 image_pixbits;      /*Local color table size*/
      GIF_UINT8 interlace;    /*No Interlace - 0 and Interlaced - 1*/
      GIF_UINT8 local_color_table_flag;/*Indicator for local color table
                                          flag presence*/

      GIF_UINT8 trans_color_flag;   /*Flag to indicate the  usage  of
```

```
                                 transparency color index*/
      GIF_UINT8 user_input_flag;      /*User input flag*/
      GIF_UINT8 disposal_method;      /*Disposal Method*/
      GIF_UINT16 delay_time;          /*Delay Time*/
      GIF_UINT16 trans_color_index;   /*Transparency Color index*/
      GIF_INT16  loop_count;          /*Number of times animation should
      repeat. Present in application extension block*/
      GIF_INT32 pass;                 /*Pass*/
      GIF_UINT32 pix_count;           /*Pixel count*/
} GIF_Decoder_Info_Init;
```

Element	Description
globwidth	Width of the global screen
globheight	Height of the global screen
Glob_out_width	Ouptut width of the global screen
Glob_out_height	Output height of the global screen
Glob_pixbits	Number of bits per pixel in global header (Value 1 to 8)
Globbc	Background color index (into the global color table)
globaspect	Global Aspect Ratio
glob_color_tbl_size	2 power N+1 gives entries in global color table
glob_color_tbl_sort_flag	Color table Sort Flag
glob_bpp	Bits per pixel minus 1
glob_color_tbl_flag	Set if Global color table is present
image_left	Left offset of Image within logical screen
image_top	Top offset of Image within logical screen
Scaled_image_left	Scaled left offset of Image within logical screen
Scaled_image_top	Scaled top offset of Image within logical screen
image width	Frame width
image height	Frame height
out image width	Rendered frame width
out image height	Rendered frame height
image_pixbits	2 power(image_pixbits+1) is the number of entries in the local color table.(Range 0 to 7)
interlace	Interlace Flag Non Interlaced-0 Interlaced - 1
local_color_table_flag	Valid values 0 & 1.If set image uses a local color table
trans_color_flag	Valid values 0 & 1.Set when the transparent color index is used.
user_input_flag	Valid values 0 &1.When set ,the application should wait for the user input before displaying the next image.
disposal_method	Specifies what the decoder is to do after image is displayed. 0 No action 1 Leave the image in place 2 Restore the bkgd color

	3 Restore what was in place beforethe image was drawn
delay_time	Amount of time the decoder should wait before continuing to process the stream in 1/100th of a second
trans_color_index	If transparent color flag is set,pixels withis color value are not written to the display
loop_count	Number of times animation should repeat. Present in application extension block
pix count	Pixel Counrt
Pass	Number of passes (for interlaced images)

```
/*Bitcount in enumerated data types*/
typedef enum
{
    E_BIT_COUNT_1  = 1,
    E_BIT_COUNT_2  = 2,
    E_BIT_COUNT_3  = 3,
    E_BIT_COUNT_4  = 4,
    E_BIT_COUNT_5  = 5,
    E_BIT_COUNT_6  = 6,
    E_BIT_COUNT_7  = 7,
    E_BIT_COUNT_8 =  8
}bit_count;
```

4 GIF Decoder - Interface

This section describes the interfaces of the GIF Decoder.

4.1 Memory Query

The GIF decoder does not perform any dynamic memory allocation. However, the decoder memory requirements may depend on the type of GIF bit stream. The application has to allocate memory as required by the decoder. Querying for memory requirements is divided into two parts.

- **Memory requirement for global data of a GIF input stream**

Application first needs to query for memory by calling the function *GIF_query_dec_mem*. This function must be called before all other decoder functions are invoked. This function parses the global information (global header and global color table) from the bitstream and fills the memory information structure array. The application will then allocate memory and gives the memory pointers to the decoder by calling the initialization function (*GIF_decoder_init*). During the memory query, this function pointed by function pointer GIF_get_new_data to provide input bit stream required for the memory query. This routine needs to be called at the beginning of every new file/stream.

- **Memory requirement for individual frames of a GIF input stream**

GIF_query_dec_mem_frame needs to be called for every frame.This function is invoked after the GIF_query_dec_mem and GIF_decoder_init functions are called. This function parses the information related to each frame from the bit stream and fills the memory information structure array. The application will then allocate memory and gives the memory pointers to the decoder by calling the frame initialization function(GIF_dec_init). During the memory query, this function pointed by function pointer GIF_get_new_data to provide input bit stream required for the memory query. This routine needs to be called at the beginning of every new frame.

C prototype:

```
GIFD_RET_TYPE GIF_query_dec_mem (GIF_Decoder_Object *);
```

Arguments:

Decoder Object pointer.

Return value:

- GIFD_OK - Memory query successful.
- Other code - Error

C prototype:
```
GIFD_RET_TYPE GIF_query_dec_mem_frame (GIF_Decoder_Object *);
```

Arguments:
Decoder Object pointer.

Return value:
- GIFD_OK - Memory query for a frame successful.
- Other codes - Error

4.2 Initialization

All initializations required for the decoder is done in the initialization routines. Initialization is also divided into two parts.

- **Initialization for the global data**

GIF_decoder_init() initializes the global data required for decoding the GIF input stream. This routine must be invoked after *GIF_query_dec_mem* is called. It calls GIF_get_new_data to provide input bits required for initialization. The application need to allocate the memory needed by the decoder and fill the pointers of the *GIF_Mem_Alloc_Info* structure before calling the function.The function also initializes the members of the *Gif_Decoder_Info_Init* structure (members pertaining global information).The initialization routine needs to be called at the beginning of every new file/stream.

- **Initialization for each frame of a GIF input stream**

GIF_dec_init_frame() initializes the data required for decoding each frame of a GIF input stream. This routine must be invoked after *GIF_query_dec_mem_frame* is called. It calls *GIF_get_new_data* to provide input bits required for initialization. The application needs to allocate the memory needed by the decoder and fill the pointers of the *GIF_Mem_Alloc_Info* structure before calling the function. The function also initializes the members of the Gif_Decoder_Info_Init structure (members pertaining frame information).This initialization routine needs to be called at the beginning of every new frame.

C prototype:
```
GIFD_RET_TYPE GIF_decoder_init (GIF_Decoder_Object *);
```

Arguments:
- Decoder Object pointer.

Return value:
- GIFD_OK - Initialization successful.
- Other codes - Initialization Error

C prototype:
*GIFD_RET_TYPE GIF_decoder_init_frame(GIF_Decoder_Object *gif_dec_obj);*

Arguments:

- Decoder Object pointer.

Return value:

- GIFD_OK - Initialization for the frame successful.
- Other codes - Initialization Error

4.3 Decoding

The main decoder function is *GIF_decode()*. This function decodes one frame from the GIF bit stream to generate the decoded image pixels in RGB format for that frame. The decoder should be initialized with global and frame information before this function is called. During the process of decoding, the function *GIF_get_new_data()* gets called whenever the decoder runs out of input. The calling application needs to provide a new buffer filled with input data when *GIF_get_new_data* is called. The decoder returns the used up buffer to the calling application. The calling application can fill up fresh data in the returned buffer and keep it ready for use in the next *GIF_get_new_data* call.

The output buffer is filled for each frame with RGB pixels of the required output format and intended size for display.

If errors are encountered in the bit stream, the decoder handles these errors internally[1].

C prototype:

```
GIFD_RET_TYPE GIF_decode (GIF_Decoder_Object *dec_obj,
                          GIF_UINT8 *output_buf)
```

Arguments:

dec_obj	Decoder Object pointer
output_buf	Output buffer pointer

Return value:

GIFD_OK	-	indicates decoding for frame was successful.
Others codes	-	indicates error

4.4 API Version

This is the decoder function to get the API version information.

C prototype:

```
const char * GIFD_CodecVersionInfo(void)
```

[1] Example error handling framework listed in .h file in Appendix

Arguments:
None

Return value:

const char *	The pointer to the constant char string of the version information string

4.5 Function implemented by application

The GIF decoder requires functions to read data from input stream which needs to be implemented by the calling application. The GIF decoder API uses function pointers to invoke these functions.

Function pointed by this function pointer is called by the decoder library whenever it runs out of the input data. It returns the used up buffer to the calling application. The calling application fills up new data in the returned buffer and makes it available for use in the next call to GIF_get_new_data.The amount of data read from the input stream is updated in the buffer length field.The variable 'dec_obj' is a pointer to decoder object.This is particulary useful when the application needs to suspend the decoder.

C prototype:
```
GIFD_RET_TYPE GIF_get_new_data (
            GIF_UINT8 **new_buf_ptr,
            GIF_INT32 *new_buf_len,
            GIF_Decoder_Object *gif_dec_obj);
```

Arguments:

new_buf_ptr	Pointer to pointer to new buffer data
new_buf_len	Length of the new buffer data
gif_dec_obj	Pointer to GIF decoder object

Return value:

GIFD_OK	-	indicates fetching of data was successful.
GIFD_SUSPEND	-	Suspend the decoder
Others codes	-	indicates error

4.6 Suspension

There are two ways the application can suspend the GIF decoder. The first method is by the use of *GIF_decode()* after which control is returned to the calling application. The second method is by the use of *GIF_get_new_data()*.

Suspension using the second method takes place as follows:

1. The flag TEST_SUSPENSION is defined in the test application

2. A static variable is declared in *GIF_get_new_data()* function and is incremented each time the function is called.
3. After some calls to the function, *GIF_get_new_data()* returns the code GIFD _SUSPEND.
4. The library comes out of the decoding function with return code as GIFD _SUSPEND. The decoder library also updates a state variable, which will tell the application how many bytes of data have been read in the current frame. This will help for the application to seek back that many bytes in the current frame so that the decoding of the frame can be started from the beginning of the frame when the data is ready.
5. The application sets the state of the decoder as suspended.
6. When the data is ready, the application sets the input pointer to the start of the current frame .The application then resumes with the decoding of the frame that was being decoded before the suspension took place. The application needs to call *GIF_query_dec_mem_frame()*, *GIF_decoder_init_frame* and *GIF_decode()* sequentially for that particular frame, irrespective of the routine it was suspended from, whether *GIF_query_dec_mem_frame(), GIF_decoder_init_f*rame or *GIF_decode()*.

4.7 Overview of API Usage

- Query for memory using *GIF_query_dec_mem()*. GIF Decoder returns memory required
- Calling function (i.e. the application that uses the GIF decoder) allocates memory for global data and fills up GIF_Decoder_Object.mem_info.mem_info_sub[i].ptr
- Calling function fills up the decoder parameters.
- The calling function initializes the GIF decoder with global information by calling *GIF_decoder_init()*
- Calling function allocates memory for frame data and fills up GIF_Decoder_Object.mem_info.mem_info_sub[i].ptr by calling *GIF_query_dec_mem_frame()*
- The calling function initializes the GIF decoder with frame information by calling *GIF_decoder_init_frame()*
- The calling function sets the required output format to be displayed, say RGB888 and allocates the output buffer for each frame.
- For each frame, the calling function calls the GIF decoder, i.e. *GIF_decod*e() that is required to decode and post process the decoded output.After decoding of each frame the output buffer contains decoded data of that frame.

The *GIF_query_dec_mem(),GIF_decoder_init*, *GIF_query_dec_mem_frame*(), *GIF_decoder_init_frame* and *GIF_decode*() internally call the function pointed by the function pointer *GIF_get_new_data* when they run out of the input bits. This function returns the used input buffer and accepts the new input buffer.

Appendix A RGB/BGR output formats supported

A-1 RGB888 FORMAT

A-1-1 Unwrapped format

In the RGB888 image data format, each pixel requires 3 bytes. The image data is organized as follows.

Unwrapped RGB888 Image data format

DATA (MSB -> LSB)
$R_7\ R_6\ R_5\ R_4\ R_3\ R_2\ R_1\ R_0\ G_7 G_6\ G_5\ G_4\ G_3\ G_2\ G_1\ G_0\ B_7\ B_6\ B_5\ B_4\ B_3\ B_2\ B_1\ B_0$

The library provides data in the aforementioned unwrapped format.

A-1-2 Wrapped format

In order to facilitate easy viewing of the raw RGB888 data, the sample test wrapper prepends headers to make it compatible with the Portable Bit-Map formats, i.e. PGM (Portable GrayMap) in case of grayscale data or PPM (Portable PixelMap) in case of colour data.

Wrapped RGB888 Image Fields

HEADER	DATA (MSB -> LSB)
	$R_7\ R_6\ R_5\ R_4\ R_3\ R_2\ R_1\ R_0\ G_7 G_6\ G_5\ G_4\ G_3\ G_2\ G_1\ G_0\ B_7\ B_6\ B_5\ B_4\ B_3\ B_2\ B_1\ B_0$

Please refer to http://netpbm.sourceforge.net/doc/ppm.html for details on PPM header and http://netpbm.sourceforge.net/doc/pgm.html for details on PGM header format.

A-2 RGB565 FORMAT

A-2-1 Unwrapped format

In the RGB565 image data format, each pixel requires 2 bytes. Consider the RGB888 data depicted in the previous section. The derived RGB 565 data would be as follows.

Unwrapped RGB565 Image data format

DATA (MSB -> LSB)
R_7 R_6 R_5 R_4 R_3 G_7G_6 G_5 G_4 G_3 G_2 B_7 B_6 B_5 B_4 B_3

The library provides data in the aforementioned unwrapped format. Note that this data can be organized in the little endian or big endian format, depending on the endianness of the target of execution.

A-2-2 Wrapped format

In order to be consistent with the wrapped format for RGB888, the sample test wrapper prepends headers to make it compatible with the Portable Bit-Map formats, i.e. PGM (Portable GrayMap) in case of grayscale data or PPM (Portable PixelMap) in case of colour data.

Wrapped RGB565 Image Fields

HEADER	DATA (MSB -> LSB)
	R_7 R_6 R_5 R_4 R_3 G_7G_6 G_5 G_4 G_3 G_2 B_7 B_6 B_5 B_4 B_3

Please refer to http://netpbm.sourceforge.net/doc/ppm.html for details on PPM header and http://netpbm.sourceforge.net/doc/pgm.html for details on PGM header format.

A-3 RGB555 FORMAT

A-3-1 Unwrapped format

In the RGB555 image data format, each pixel requires 2 bytes. Consider the RGB888 data depicted in the previous section. The derived RGB 555 data would be as follows

Unwrapped RGB555 Image data format

DATA (MSB -> LSB)
0 R_7 R_6 R_5 R_4 R_3 G_7G_6 G_5 G_4 G_3 B_7 B_6 B_5 B_4 B_3

Among the 16 bits, the most significant bit is set to zero.

The library provides data in the aforementioned unwrapped format. Note that this data can be organized in the little endian or big endian format, depending on the endianness of the target of execution.

A-3-2 Wrapped format

In order to be consistent with the wrapped format for RGB888, the sample test wrapper prepends headers to make it compatible with the Portable Bit-Map formats, i.e. PGM (Portable GrayMap) in case of grayscale data or PPM (Portable PixelMap) in case of colour data.

Wrapped RGB555 Image Fields

HEADER	DATA (MSB -> LSB) **0 R_7 R_6 R_5 R_4 R_3 G_7G_6 G_5 G_4 G_3 B_7 B_6 B_5 B_4 B_3**

Please refer to http://netpbm.sourceforge.net/doc/ppm.html for details on PPM header and http://netpbm.sourceforge.net/doc/pgm.html for details on PGM header format.

A-4 RGB666 FORMAT

A-4-1 Unwrapped format

In the RGB666 image data format, each pixel requires 3 bytes. Consider the RGB888 data depicted in the previous section. The derived RGB 666 data would be as follows

Unwrapped RGB666 Image data format

DATA (MSB -> LSB)
$R_7\ R_6\ R_5\ R_4\ R_3\ R_2$ 0 0 $G_7\ G_6\ G_5\ G_4\ G_3\ G_2$ 0 0 $B_7\ B_6\ B_5\ B_4\ B_3\ B_2$ 0 0

Within each byte, the two least significant bits are set to zero. This choice of padding zeros towards the LSB lends itself to easy viewing of the rendered RGB666 data.

The library provides data in the aforementioned unwrapped format.

A-4-2 Wrapped format

In order to facilitate easy viewing of the raw RGB666 data, the sample test wrapper prepends headers to make it compatible with the Portable Bit-Map formats, i.e. PGM (Portable GrayMap) in case of grayscale data or PPM (Portable PixelMap) in case of colour data.

Wrapped RGB555 Image Fields

HEADER	DATA (MSB -> LSB)
	$R_7\ R_6\ R_5\ R_4\ R_3\ R_2$ 0 0 $G_7\ G_6\ G_5\ G_4\ G_3\ G_2$ 0 0 $B_7\ B_6\ B_5\ B_4\ B_3\ B_2$ 0 0

Please refer to http://netpbm.sourceforge.net/doc/ppm.html for details on PPM header and http://netpbm.sourceforge.net/doc/pgm.html for details on PGM header format.

A-5 BGR FORMAT

In BGR format, R component and B component are exchanged in store order according to corresponding RGB format addressed above.

Appendix B Suspension and Resumption Mechanism

To test the suspension mechanism, two compile time flags ENABLE_SUSPENSION and TEST_SUSPENSION have been provided.

ENABLE_SUSPENSION – This flag is defined in the file /ARM11/src/image/gif_dec/library/debug.h. It is used to enable/disable the suspension-resumption mechanism in the library.

TEST_SUSPENSION – This flag is defined in the file /ARM11/src/image/gif_dec/test/c_source/gif_test.c. When this flag is set, the sample application provided (gif_test.c) enables the code that specifically tests the suspension-resumption feature provided by the library. A prerequisite for TEST_SUSPENSION to be set is that the ENABLE_SUSPENSION needs to be set.

Note that by default (as in the sample library provided), both flags have been disabled. The user can set these as per need[2].

To simulate this suspension mechanism following concept is implemented in the application code.

- The flag TEST_SUSPENSION is defined in the test application
- A static variable is declared in GIF_get_new_data() function and is incremented each time the function is called.
- After some calls to the function, GIF_get_new_data() returns the code GIFD _SUSPEND.
- The library comes out of the decoding function with return code as GIFD _SUSPEND. The decoder library also updates a state variable (gif_dec_obj. bytes_read_in_a_frame), which indicates to the application how many bytes of data have been read in the current frame. This application needs to use this variable to seek back that many bytes in the current frame so that the decoding of the frame can be started from the beginning of the frame when the data is ready.
- The application sets the state of the decoder as suspended.
- When the data is ready, the application sets the input pointer to the start of the current frame .The application then resumes with the decoding of the frame that was being decoded before the suspension took place. The application needs to call GIF_query_dec_mem_frame(), GIF_decoder_init_frame and GIF_decode() sequentially for

[2] Specifically, the libraries (.a files) present in the folder/library have been built with ENABLE_SUSPENSION flag disabled. So, to test the suspension mechanism library must be rebuilt with the procedure mentioned earlier with ENABLE_SUSPENSION flag enabled. The executable may then be generated by enabling the flag TEST_SUSPENSION in gif_test.c file.

that particular frame, irrespective of the routine it was suspended from, whether GIF_query_dec_mem_frame(),GIF_decoder_init_frame or GIF_decode().

- The output generated was found to be bit matching with the reference output.

Appendix C Debug and Log Support

To test the debug and log support, the calling application needs to enable/disable certain compile time flags in the debug.h file provided in /ARM11/src/image/gif_dec/library/include/ directory.

Following is the list of the compile time flags.

- DEBUG_LEVEL_0
- DEBUG_LEVEL_1
- DEBUG_LEVEL_2
- ENTRY_EXIT
- DECODER_STATE
- OTHER_INFO
- READ_HDR_DATA_IN_INIT
- GLOBAL_HEADER_DATA
- GLOBAL_COLOR_TABLE
- FRAME_HEADER_DATA
- FRAME_COLOR_TABLE
- FRAME_NUMBER

GIF decoder uses three levels of debug flags DEBUG_LEVEL_0,DEBUG_LEVEL_1 and DEBUG_LEVEL_3.Other flags are nested in these 3 levels and are enabled/ disabled depending upon the contents to be logged. Sample debug.h file is provided below .The comments following the definition of the flags give detailed information about them.

```
//4 bit representing the various components
//0x1 means level 0  (Function Entry-Exit/General Info)
//0x2 means level 1  (Global GIF data)
//0x3 means 0 & 1    (Global GIF data + Fn Entry exit/General Info)
//0x4 mean 0,1 & 2   (Global GIF data + Frame data +Fn Entry exit/General
Info)
//If this flag is enabled then a flag, TEST_SUSPENSION should
//also be enabled in the application code
//#define ENABLE_SUSPENSION
#define debug_level 0x7
/*On enabling debug level 0 we get messages regarding
   a.Function Entry Exit
   b.State of the decoder
*/
#define DEBUG_LEVEL_0 ((debug_level >> 0 ) & 0x1)

/*On enabling debug level 1 we get global data in the
  input GIF stream
*/
#define DEBUG_LEVEL_1 ((debug_level >> 1 ) & 0x1)
/*On enabling debug level 2 we get frame data in the
  input GIF stream
*/
```

```
#define DEBUG_LEVEL_2 ((debug_level >> 2 ) & 0x1)
/*Nested flags in debug levels*/

#if DEBUG_LEVEL_0

  #define ENTRY_EXIT   1  /*Get function entry and exit point messages*/
  #define DECODER_STATE 1 /*Get info regarding the state of decoder.
                                       For e.g. Querying for Mem
Req,Initializing etc*/
  #define OTHER_INFO 1    /* Get the encoding mode info*/
  #define READ_HDR_DATA_IN_INIT 1/*If we want to read header data in init
once again*/
#endif

#if DEBUG_LEVEL_1
   #define GLOBAL_HEADER_DATA 1/*Get global header data*/
   #define GLOBAL_COLOR_TABLE 1/*Get global color table*/

#endif

#if DEBUG_LEVEL_2
   #define FRAME_HEADER_DATA 1/*Get frame header data*/
   #define FRAME_COLOR_TABLE 1/*Get frame color table*/
   #define FRAME_NUMBER  1   /*"Get the decoding frame number*/
#endif
```

The sample debug.h file when used with decoder library outputs all the possible messages and data in the log file