08-6464-SIS-ZCH66
MARCH 9, 2006
2.1

Application Programmers Interface for BMP Decoder

ABSTRACT:

Application Programmers Interface for BMP Decoder

KEYWORDS:

Multimedia codecs, Image, Bitmap

APPROVED:

Wang Zening

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
1.0	03-Mar-2004	Shailesh R	Initial Draft
1.1	10-Mar-2004	Shailesh R	Updated for (software) scaling feature capable of being turned off or on
1.2	15-Mar-2004	Shailesh	Changed scaling to enum
1.3	30-Nov-2004	Sameer Rapate Shailesh R	Added support for new PCS requirements
1.4	20-Apr-2005	Sameer Rapate Shailesh R	Added description for RGB formats
2.0	06-Feb-2006	Lauren Post	Draft version using new format
2.1	29-Mar-2006	Sanjeev Kumar	Document review

Table of Contents

Introduction

1.1 Purpose

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

1.2 Scope

This document describes only the functional interface of the BMP 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 BMP decoder decodes BMP formats[1] as defined in Windows SDK (version 3 onward), with the following features

- The BMP decoder supports BMP files containing one image with 1, 4, 8 16 and 24 bits per pixel
- Supports simple run length compression for 4 and 8 bits per pixel
- The output formats supported are RGB555, 16 bit RGB (RGB565), RGB666 and 24 bit RGB (RGB888)

1.3 Audience Description

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

1.4 References

1.4.1 Standards

- Windows SDK (version 3 onward) – BMP Format Documentation

1.4.2 References

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

[1] Over the years, there have been several different and incompatible versions of BMP format. In addition to the supported features of BMP listed in this section, the following restrictions of the BMP decoder in the scope of this project should clearly be understood:

- BMP formats in use since Windows 3 are supported - OS/2 format for BMP not supported
- 32 bits per pixel format (rare) not supported

1.4.3 Freescale Multimedia References

- BMP Decoder Application Programming Interface - bmp_dec_api.doc
- BMP Decoder Requirements Book - bmp_dec_reqb.doc
- BMP Decoder Test Plan - bmp_dec_test_plan.doc
- BMP Decoder Release notes - bmp_dec_release_notes.doc
- BMP Decoder Test Results – bmp_dec_test_results.doc
- BMP Decoder Performance Results - bmp_dec_perf.doc
- BMP Decoder Interface Header – bmp_interface.h
- BMP Decoder Application Code – test_bmp.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
RVDS	ARM RealView Development Suite
TBD	To Be Determined
UNIX	Linux PC x/86 C-reference binaries

1.6 Document Location

docs/bmp_dec

2 API Description

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

BMP_query_dec_mem
: Memory query

BMP_decoder_init
: Initialization

BMP_decode_row_pp
: Decoding and postprocessing (i.e. palletization and rescaling)

BMP_get_new_data
: API to be implemented by the application calling the BMO decoder, to enable the BMP decoder to fetch new data.

BMP_seek_file
: Seeks the specified number of bytes from the current position or from the start position. This API needs to be implemented by the application calling the BMP decoder, to enable the BMP decoder to reposition the stream pointer (which points to the input)

The BMP decoder is provided as a library that contains the relevant routines including BMP_query_dec_mem, BMP_decoder_init, BMP_decode_row_pp. The application calling the BMP decoder needs to implement the callback routine BMP_get_new_data and BMP_seek_file.

3 BMP Decoder – Data Structures

3.1 Basic Data Types

```
typedef          unsigned long          BMP_UINT32;
typedef          long                   BMP_INT32;
typedef          unsigned short         BMP_UINT16;
typedef          short                  BMP_INT16;
typedef          unsigned char          BMP_UINT8;
typedef          char                   BMP_INT8;
```

3.2 BMP_Decoder_Object

In order to call any BMP decode function, the application that calls the BMP 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 BMP functions need to be allocated by the caller on a per instance basis, and hence are part of BMP_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 BMP_Decoder_Object {
      BMP_Mem_Alloc_Info      mem_info;
      BMP_Decoder_Params      dec_param;
      BMP_Decoder_Info_Init   dec_info_init;
      BMP_UINT32              rows_decoded;
      BMP_INT32               num_byte_read_in_row;
      BMP_error_type          (*BMP_get_new_data)
                  (BMP_UINT8 **new_buf_ptr, BMP_UINT32 *new_buf_len,
                  struct      BMP_Decoder_Object *dec_object);
      BMP_error_type          (*BMP_seek_file)
                  (struct BMP_Decoder_Object *dec_object, BMP_INT32
                  num_bytes, BMP_Seek_File_Position start_or_current);
      void                    *vptr;
} BMP_Decoder_Object;
```

Element	Description
BMP_Mem_Alloc_Info mem_info	Filled by decoder in BMP_query_dec_mem function.
BMP_Decoder_Params dec_param	Caller needs to fill in this structure before calling the decoder functions
BMP_Decoder_Info_Init	BMP decoder init fills this structure

dec_info_init	up, which can be used by the caller
rows_decoded	Rows decoded- BMP decoder operates on a row by row basis
num_byte_read_in_row	Number of bytes read in current row
(*BMP_get_new_data)(BMP_UINT8 **new_buf_ptr, BMP_UINT32 *new_buf_len, struct BMP_Decoder_Object *dec_object);	Pointer to the callback routine BMP_get_new_data
(*BMP_seek_file) (struct BMP_Decoder_Object *dec_object, BMP_INT32 num_bytes, BMP_Seek_File_Position start_or_current)	Pointer to the callback routine BMP_seek_file
Void *vptr	Other codec specific structure elements not needed by caller

3.3 BMP_Mem_Alloc_Info

BMP_Mem_Alloc_Info is filled by the decoder in BMP_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
      {
      BMP_INT32                   num_reqs;
      BMP_Mem_Alloc_Info_Sub      mem_info_sub[MAX_NUM_MEM_REQS[2]];
       } BMP_Mem_Alloc_Info;
```

Element	Description
num_reqs	Number of valid memory requests
BMP_Mem_Alloc_Info_Sub mem_info_sub	Array of structure

```
typedef struct {
      BMP_INT32                   size;   /* Size in bytes */
      BMP_Mem_type                type;   /*Memory type Fast or Slow */
      BMP_INT32                   align; /* Alignment of memory in
                                            bytes */
      void                        *ptr;  /* Pointer to the memory */
} BMP_Mem_Alloc_Info_Sub;
```

Element	Description
Size	Mem Size
BMP_Mem_type type	Memory type - fast or slow
Align	Alignment of mem in bytes

[2] MAX_NUM_MEM_REQS defined in .h file

Ptr	Pointer to memory

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

3.4 BMP_Decoder_Params

BMP_Decoder_Params needs to be filled by the application calling the BMP decoder before it calls the Decoder functions. The calling application needs to indicate the desired output format (RGB555, RGB666, RGB565 or RGB888). In case the calling application needs the BMP 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.[3] It should be noted that it is the responsibility of the calling application to ensure that all the structure members of BMP_Decoder_Params are initialized to the correct values.

```
typedef struct
{
    output_format           outformat;
    scaling_mode            scale_mode;
    BMP_UINT16              output_width;
    BMP_UINT16              output_height;
} BMP_Decoder_Params;
```

Element	Description
output_format outformat	Enum for output formats supported
scaling_mode scale_mode	Enum for scaling mode
output_width	Width of output to be displayed (if sw_scaling_set is set to 1)
output_height	Height of output to be displayed (if sw_scaling_set is set to 1)

```
typedef enum  {
 E_OUPUTFORMAT_RGB888,
 E_OUTPUTFORMAT_RGB565,
 E_OUTPUTFORMAT_RGB555,
 E_OUTPUTFORMAT_RGB666,
 E_LAST_OUTPUT_FORMAT
} output_format;
```

For more details on these formats refer to Appendix A

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  {
```

[3] Note that only scaling down is supported – if the output dimensions configured are greater than the BMP image size as it occurs in the header, the BMP image is left unscaled.

```
 E_NO_SCALE,                /* No software scaling */
 E_INT_SCALE_PRESERVE_AR,   /* Software scaling using integer scaling
                               factor preserving pixel aspect ratio */
 E_LAST_SCALE_MODE
} scaling_mode;
```

3.5 BMP_Decoder_Info_Init

BMP_Decoder_Info_Init is filled by the decoder whenever the application invoking the BMP decoder calls the BMP decoder initialization function *BMP_decoder_init*.

The information that is available after the initialization includes the image width, image height, the output width and height (which corresponds to the rendered output displayed), the number of bits per pixel used by BMP (supported: 1, 4, 8, 16 or 24 bits per pixel), compression type (Run length encoding RLE 4 or RLE 8, or RGB), compressed file size, number of components in BMP file and rendered output.

```
typedef struct
{
    BMP_UINT16              image_width;    /* Input Image width */
    BMP_UINT16              image_height;   /* Input Image height */
    BMP_UINT16              output_width;   /* width of rendered output
                                               [4] */
    BMP_UINT16              output_height;  /* height of rendered
                                               output */
    bit_count               bit_cnt;       /* Bits per pixel - 1, 4, 8,
                                               16, or 24 */
    compression_type        cmpr_type;    /* RGB, RLE4, RLE8 etc */
    BMP_UINT32              file_size;   /* BMP file size in bytes */
   BMP_UINT16               BMP_components; /* Number of components in
                                               the BMP */
  BMP_UINT16                output_components;/* Number of components
                                                 rendered output */
  } BMP_Decoder_Info_Init;
```

Element	Description
image_width	Input Image width
image_height	Input Image height
output_height	Height of output image to be rendered
output_width	Width of output image to be rendered
bit_count bpp	Bits per pixel - 1, 4, 8, 16, or 24
compression_type comp	RGB, RLE4, RLE8 - type of compression
File_size	BMP file size in bytes
BMP_components	Number of components in the BMP
output_components	Number of components rendered output

```
typedef enum  {
    E_BIT_COUNT_1   = 1,
    E_BIT_COUNT_4   = 4,
```

[4] The rendered ouput size may not exactly match the display size configured in BMP_Decoder_Params since the decoded output is scaled down by an integral multiple with aspect ratio preserved, to yield the rendered output.

```
    E_BIT_COUNT_8  = 8,
    E_BIT_COUNT_16 = 16,
    E_BIT_COUNT_24 = 24
} bit_count;

typedef enum  {
    E_RGB   = 0,
    E_RLE8  = 1,
    E_RLE4  = 2
} compression_type ;
```

4 BMP Decoder - Interface description

4.1 Memory Query

The BMP decoder does not perform any dynamic memory allocation. However, the decoder memory requirements may depend on the type BMP bitstream (due to image size, mode etc.). The application has to allocate memory as required by the decoder. So the application first needs to query for memory by calling the function BMP_query_dec_mem. This function must be called before all other decoder functions are invoked. This function parses the required decoder information 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, which is given in the next section. During the memory query, this function calls BMP_get_new_data to provide input bitstream required for the memory query. This routine needs to be called at the beginning of every new file/stream.

Important Note: The application should provide the bitstream from the beginning of the BMP stream when *BMP_query_dec_mem()* calls the *BMP_get_new_data()* function.

C prototype:
*BMP_error_type BMP_query_dec_mem (BMP_Decoder_Object *);*

Arguments:
Decoder Object pointer.

Return value:

- BMP_ERR_NO_ERROR - Memory query successful.
- Other codes - Error

4.2 Initialization

All initializations required for the decoder are done in *BMP_decoder_init()*. This function must be called after *BMP_query_dec_mem* is called. The initialization routine calls *BMP_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*BMP_Mem_Alloc_Info* structure before calling the function. The initialization routine needs to be called at the beginning of every new file/stream.

C prototype:

```
BMP_error_type BMP_decoder_init (BMP_Decoder_Object *);
```

Arguments:

Decoder Object pointer.

Return value:

BMP_ERR_NO_ERROR	-	Initialization successful.
Other codes	-	Initialization Error

4.3 Decoding and POST-PROCESSING

The main decoder function is *BMP_decode_row_pp()*. This function decodes the BMP bit stream row by row to generate the image pixels in RGB format. The decoder should be initialized before this function is called. **During the process of decoding, the function *BMP_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 *BMP_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 *BMP_get_new_data* call.**

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

The decoding and post processing are carried out row by row. If errors are encountered in the bitstream, the decoder handles these errors internally.

C prototype:

```
BMP_error_type BMP_decode_row_pp (BMP_Decoder_Object *dec_obj,
                                  BMP_UINT8 *output_buf)
```

Arguments:

dec_obj	Decoder Object pointer
output_buf	Output buffer pointer

Return value:

BMP_ERR_NO_ERROR	-	indicates decoding was successful.
Others	-	indicates error

4.4 Suspension

There are two ways the application can suspend the BMP decoder. The first method is with the use of BMP_decode_row_pp() after which control is returned to the calling application. The second method is by the use of BMP_get_new_data().

Suspension using the second method takes place as follows:

- A flag TEST_SUSPENSION is defined in the application code
- A static variable is declared in BMP_get_new_data() function and is incremented each time the function is called.
- After a few calls to the function, get_new_data() returns the code BMP_ERR_SUSPEND.
- Library comes out of the decoding function with return code as BMP_ERR_SUSPEND. Decoder also updates a state variable, which will tell the application how many bytes of data have been read in the current row. This will help for the application to seek back that many bytes of data so that the row can be started from the beginning 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 row and the decoding proceeds

5 Overview of API usage

- Query for memory using BMP_query_dec_mem(). BMP Decoder returns memory required
- Calling function (i.e. the application that uses the BMP decoder) allocates memory and fills up BMP_Decoder_Object.mem_info.mem_info_sub[i].ptr
- Calling function fills up the decoder parameters.
- The calling function initializes the BMP decoder by calling BMP_decoder_init()
- The calling function sets the required output format to be displayed, say RGB565 and allocates the output buffer.
- For each row, the calling function calls the BMP decoder, i.e. BMP_decode_row_pp that is required to decode and postprocess the decoded output.

The *BMP_decoder_init and BMP_decode_row_pp* internally call *BMP_get_new_data* when they run out of the input bits. The *BMP_get_new_data* function returns the used input buffer and accepts the new input buffer.

/* This function is implemented by the application */

```
BMP_error_type  BMP_get_new_data (BMP_UINT8 ** new_buf_ptr,
                          BMP_UINT32 * new_buf_len,
                          BMP_Decoder_Object *dec_object)
{
      BMP_UINT8 *  ptr;

      /* Read *new_buf_ptr and free that memory */
      ptr = *new_buf_ptr;
      free(ptr);

      /* Obtain the input BMP stream */
      *new_buf_ptr = Obtained from some source known to application,
                  corresponding to decoder object;
      *new_buf_len = Obtained from some source known to application,
                  corresponding to decoder object;

      Return 0 to indicate that new buffer
      has been filled.
      OR
      Return 1  if the application
      encountered an error in passing the buffer

}
```

The *BMP_decoder_init and BMP_decode_row_pp* also call *BMP_seek_file* when they want to seek in the file from start or current position . Sample *BMP_seek_file* implementation can be found below.

```
BMP_error_type BMP_seek_file(BMP_Decoder_Object *dec_object, BMP_INT32
num_bytes, BMP_Seek_File_Position start_or_current)
{
    if(start_or_current == BMP_SEEK_FILE_CURR_POSITION)
```

```
    {
        if(fseek(fp_input_image, num_bytes, SEEK_CUR) != 0)
        {
            return BMP_ERR_ERROR;
        }

    }
    else if (start_or_current == BMP_SEEK_FILE_START)
    {
        if(fseek(fp_input_image, num_bytes, SEEK_SET) != 0)
        {
            return BMP_ERR_ERROR;
        }
    }
    return BMP_ERR_NO_ERROR;
}
```

Appendix A RGB Output Formats Supported

1. RGB888 FORMAT

- **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.

- **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.

2. RGB565 FORMAT

- **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_7$$G_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.

- **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_7$$G_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.

3. RGB555 FORMAT

- **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_7$$G_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.

- **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_7$$G_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.

4. RGB666 FORMAT

- **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.

- **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.

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 library/include/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/bmp_dec/test/c_source/test_bmp.c. When this flag is set, the sample application provided (test_bmp.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[5].

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

1. A static variable is declared in BMP_get_new_data() function and is incremented each time the function is called.
2. After four calls to the function, BMP_get_new_data() returns the code BMP_ERR_SUSPEND.
3. Library comes out of the decoding function with return code as BMP_ERR_SUSPEND. Decoder also updates a state variable (bmp_dec_obj. bytes_read_in_a_row), which indicates to the application how many bytes of data have been read in the current row. This application needs to use this variable to seek back that many bytes of data so that the row can be started from the beginning when the data is ready
4. The application sets the state of the decoder as suspended.
5. When the data is ready, the application sets the input pointer to the start of the current row and the decoding proceeds.

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

[5] Specifically, the libraries (.a files) present in the folder /ARM11/src/image/bmp_dec/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 test_bmp.c file.

Appendix C Debug and Logging 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 library/include/ directory. This header when used with decoder library outputs all the possible messages and data in the log file

Following is the list of the compile time flags.

- DEBUG_LEVEL_0
- DEBUG_LEVEL_1
- ENTRY_EXIT
- DECODER_STATE
- OTHER_INFO
- READ_HDR_DATA_IN_INIT
- HEADER_DATA
- COLOR_TABLE
- RGB_BIT_MASKS
- ROW_NUMBER

BMP decoder uses two levels of debug flags DEBUG_LEVEL_0 and DEBUG_LEVEL_1.Other flags are nested in these 2 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.

```
/*Flag to enable suspension code in the library*/
//#define ENABLE_SUSPENSION

//4 bit representing the various components
//0x1 means level 0 (Function Entry-Exit/General Info)
//0x2 means level 1  (input stream data)
//0x3 means 0 & 1   (input stream data + Fn Entry exit/General Info)

#define debug_level 0x0
/*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 data in the
  input BMP stream
*/
#define DEBUG_LEVEL_1 ((debug_level >> 1 ) & 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 HEADER_DATA 1/*Get global header data*/
   #define COLOR_TABLE 1/*Get global color table*/
      #define RGB_BIT_MASKS 1 /*Get bit masks.This applies for 16
bit BMP*/
   #define ROW_NUMBER  1   /*"Get the decoding row number*/
#endif
```