

# DirectFB Extensions RLE Image Provider Specifications & Report

**STLinux**

**Author:** Alain Clement

**Project:** STLINUX / M1

**Document ID:** DirectFB\_Extensions\_RLE\_Image\_Provider\_Report

**Version:** 1.1.2

**Status:** Draft

**Date:** 11/04/07

**Table 1: Document History**

VERSION	DATE	UPDATE	STATUS	AUTHOR
1.0.0	15/02/07	Initial release – RLE8 Image Provider	Draft	A. Clement
1.0.1	23/02/07	RLE8 Section 6.1 & 6.2 : Sample Code	Draft	A. Clement
1.0.2	03/12/07	RLE8 Section 2.4 : Supported output surfaces	Draft	A. Clement
1.1.0	02/04/07	BD-RLE8 compression extension	Draft	A. Clement
1.1.1	11/04/07	BD-RLE8 & Unit-Test changes	Draft	A. Clement
1.1.2	11/04/07	BD-RLE8 format amendments	Review	A. Clement

**Table 2: Document Review**


ROLE	NAME	REVIEW	DATE
Author	Alain Clement		
Project Planning	Wynne Crompton		
Distribution Integrator	Brendan O'Connor		
Department Manager	Robert Landau		

# Table of Contents

1	Description.....	6
1.1	Abstract.....	6
1.2	Scope.....	7
1.3	Requirements.....	7
1.4	Receivables.....	8
1.5	Deliverables.....	8
2	Package Status.....	9
2.1	RLE Image Provider Receivables Integration Status.....	9
2.2	RLE Image Provider Deliverables Status.....	9
2.3	RLE Image Provider Package ChangeLog.....	10
3	General Specifications.....	11
3.1	DirectFB-extensions Package.....	11
3.2	RLE Image Provider.....	11
3.3	Source Buffer Structure.....	12
3.4	Supported Output Surfaces.....	12
3.5	Image-Provider Modes.....	13
3.6	Supported DirectFB Releases.....	13
3.7	Dependencies.....	13
3.8	Packaging And Configuration.....	14
3.9	Unit-Test Applications.....	14
3.10	Sample Compressed Files.....	15
3.11	Standards Compliance.....	15
4	RLE-8 Code Implementation.....	16
4.1	Standard RLE-8 Payload.....	16
4.1.1	RLE-8 Payload Structure.....	16
4.1.2	RLE-8 Coding Example.....	16
4.2	BD-Specific RLE-8 Payload.....	16
4.2.1	BD-RLE-8 Payload Structure.....	17
4.2.2	BD-RLE-8 Line Coding Details.....	18
5	Packet Header Layout .....	20
5.1	Packet And File Format .....	20
6	Header Fields Details .....	22
6.1	Semantics.....	22
7	Application Code Example .....	26
7.1	DirectFB RLE Image Provider Unit-Test And Sample Code.....	26
7.2	DirectFB RLE Image Provider Packet Builder.....	32
7.3	DirectFB RLE Image Independent Image Loader.....	36

# Glossary

**Table 3: Glossary**

<i><b>Term</b></i>	<i><b>Definition</b></i>
	<i>DirectFB is a thin library that provides developers with hardware graphics acceleration, input device handling and abstraction, an integrated windowing system with support for translucent windows and multiple display layers on top of the Linux frame buffer device. It is a complete hardware abstraction layer with software fall backs for every graphics operation that is not supported by the underlying hardware.</i>

## ***References***

DirectFB Ref	DirectFB Reference Manual – Release 1.0.0 (html core package documentation)
DirectFB Intro	DirectFB Overview -- Version 0.2 (February 19, 2004, based on DirectFB 0.9.21)
Blu-Ray Core	"Blu-Ray Disk , Part 3-1 : Core Specifications" document, Section <u><b>9.14.4.2.2.2.1.1</b></u>

# 1 Description

## 1.1 Abstract

The DirectFB extensions package is meant to gather various STMicroelectronics Media Provider modules within a single package. The package relies on GNU “autotools” for integration, configuration and build and has been tailored to match DirectFB interfaces integration requirements through “autoconf” and “automake” macros.

The modules being fully compliant with standard Direct FB interfaces definition and framework, there is no need to specify any new API nor generate API documentation. Such extension modules include Image Providers, Video Providers and Font Providers and get installed on the target system as PlugIns in the form of dynamic shared libraries (.so files). DirectFB looks up and maps these PlugIns as needed thanks to a standard data header probing mechanism implemented by each module for a given recognised media type.

For example, an Image Provider collects a compressed Image (data packet) consisting of a header and its associated payload (e.g. JPEG), and decompresses it in an optimal way to a DirectFB RGB Surface. This abstraction layer together with the overall DirectFB framework allow for software support (fall-back) using the main processor (e.g. SH4) while keeping our options open for further seamless optimizations such as hardware acceleration through on-chip hardware blocks.

Image Providers can sink a media packet stored in a file or from a pre-set static memory buffer through a DirectFB Data Buffer instance (static unmanaged data buffer). Moreover, a streaming version of DirectFB Data Buffers allow for pushing data in, asynchronously through a multi threaded blocking mechanism (managed dynamic data buffer). The later case requires specific provisions from the Image Provider.

## 1.2 Scope

- Release of a RLE-compressed source media DirectFB Image Provider (shared library plug in).
- Compiled with and tested for DirectFB-1.0.0-rc1 (current STLinux release) .
- Compiled with and tested for latest DirectFB-1.0.0-rc3 CVS developer release.
- Initial support for unoptimized RLE-8 decoding (BD only – TS stream source).

## 1.3 Requirements

- BD – Transport Stream (TS) packets source – AS per “Blu-Ray Disk , Part 3-1 : Core Specifications” document, Section **9.14.4.2.2.1.1**
- Support for RLE-coded images originating from PES packets stored in individual memory buffers .
- Integration as a streamlined source software package into STLinux-M1 RPM-based distribution
- Provision to support other RLE compressed contents (HD-DVD – PS stream) in later release.

## 1.4 *Receivables*

- A)** Sonic RLE-8 cut down codec object code in the form of a set of 2 static object libraries  
Dependencies on standard C library only and compiled for:
  - Linux-Intel-32 ABI & binary code (for functional testing on PC/Fedora Core 6)
  - STLinux-SH4 ABI & binary code (for profiling and fine tuning on SH4/7100 platform)
- B)** Comprehensive header file for the aforementioned codec related decode function(s) and C structures describing input data packet header information (such as width, height, colormap, etc ...) and output pixels components layout.
- C)** Sonic RLE-8 image file (test vector) extracted from a BD/TS PES packet for implementation qualification and testing

## 1.5 *Deliverables*

- A)** STMicroelectronics RLE-8 Image Provider module (shared library plug in) for DirectFB-1.0.0 release
  - Linked against Sonic RLE-8 codec static object library
- B)** Qualification test to be performed against Sonic RLE-8 image file (test vector) based on Unit-Test code
- C)** Sample program source showing how to create an Image Provider instance from a static DirectFB Data Buffer holding a reference to the compressed data packet in a user-managed memory buffer.



## 2 Package Status

### 2.1 RLE Image Provider Receivables Integration Status

<b>A) Sonic RLE-8 codec object code .....</b>	<i>Not implemented</i>	<i>(N/A)</i>
<b>B) Sonic RLE-8 header files .....</b>	<i>Not used</i>	<i>(N/A)</i>
<b>C) Sonic RLE-8 image file (test vector) .....</b>	<i>Not tested</i>	<i>(N/A)</i>
<b>D) Sonic RLE-8 test image files (CLUT/Payload) ..</b>	<i>Visual Test</i>	<i>(Delivered)</i>

Note: N/A – Not Available

### 2.2 RLE Image Provider Deliverables Status

<b>A) ST-RLE-8 Image Provider module .....</b>	<u><i>Delivered</i></u>
- Embedded RLE-8 decoder substitute <i>(ST Code implemented from scratch)</i>	
- Standard media header format (54 bytes) <i>(Implemented according to BMP standard)</i>	
<b>B) ST-RLE-8 image file (test vector) .....</b>	<u><i>Pass (see notes below)</i></u>
- Internally generated RLE-8 file (GIMP tool) <i>(No BD-RLE qualification possible)</i>	
- Visual test of BD-RLE8 subtitle images (Sonic)	
<b>C) ST-Sample program and unit-test .....</b>	<u><i>Delivered</i></u>
- Helper library for header adaptation <i>(Example packet builder)</i>	

## 2.3 RLE Image Provider Package ChangeLog

2007-02-05 acst 15:16:00 Alain Clement <alain.clement@st.com>

Files: all

Initial version 1.0.0.

- Added RLE image provider and unit test / sample application code.
- Create DirectFB\_Extensions\_RLE\_Image\_Provider\_Specs\_and\_Report.pdf (1.0.0)

2007-04-02 acst 14:31:00 Alain Clement <alain.clement@st.com>

Files: all c & h but sample1.c

Bumped-up version 1.1.0.

- Added Blu-ray BD-RLE8 format support codec in image provider code.
- Update DirectFB\_Extensions\_RLE\_Image\_Provider\_Specs\_and\_Report.pdf (1.1.0)
- Fixup stm-target-directfb-extensions.spec RPM source file

2007-04-06 acst 14:31:00 Alain Clement <alain.clement@st.com>

Files: all c & h but sample1.c

Updated version 1.1.1.

- Fixed DB-RLE8 data files headers with [RLEIC\_BD\_RLE8] compression ID.
- Fixed unit-test\_rle.c program + handle multiple input files.
- Cleanup idirectfbimageprovider\_rle.c RLE8 & BD-RLE8 codecs source code.
- Added idirectfbimageprovider\_rle.h with specific types and macro defs.
- Update DirectFB\_Extensions\_RLE\_Image\_Provider\_Specs\_and\_Report.pdf (1.1.1)
- Hacked/fixed BD-RLE8 decoding algorithm based on BD sample images
- Integrated BD-RLE8 sample images for visual test (files include color-map)

## 3 General Specifications

### 3.1 DirectFB-extensions package

This package contains additional image/video/font providers and graphics/input drivers.

Directories contents:

- interfaces : DirectFB Media-Providers
- test : Unit Test programs
- samples : Sample applications
- doc : Documentation, Task Reports, etc ...
- data : Test Vectors, Image Samples, etc ...

### 3.2 RLE Image Provider

#### **CURRENT RLE IMAGE PROVIDER CODEC SPECIFICATIONS**

- Support for RLE Run-Length compressed image format.
- Supported payload formats:
  - Compressed: [RLEIC\_BD\_RLE8]
    - BD-RLE8 8 bits LUT-indexed / run-length-compressed / block mode  
(See “Blu-Ray Disk , Part 3-1 : Core Specifications” Section **9.14.4.2.2.2.1.1**)
  - Compressed: [RLEIC\_RLE8]
    - RLE8 8 bits LUT-indexed / run-length-compressed / block mode  
(Standard BMP format)
  - Uncompressed: [RLEIC\_NONE] / backward compatibility / testing
    - Uncompressed 1/2/4/8 bits LUT-indexed / row mode
    - Uncompressed 16/24 bits fixed RGB bit-field components / row mode

NB: Single plane picture only - Provisions for extensions / RLE formats variants

### 3.3 Source buffer structure

- Header : 54 bytes (unoptimized - byte-packed, BMP standard layout)
- Optional Palette: indexed RGB24 - (3x8 bits + 1x8 bits) x max\_colors (0-256)
- Payload: encapsulated within packet at given offset location

NB: Provisions for static memory/file access or streaming modes

### 3.4 Supported output surfaces

- Cached Internal Native ARGB 32 surface - Opaque alpha channel (0xff)
- Rendering to any DirectFB supported surface via core *"dfb\_copy\_buffer\_32()"*
- Automatic image scaling via core *"dfb\_scale\_linear\_32()"*
- Automatic color re indexing to closest match LUT-type destination surfaces colormap (hash coding)
- Indexed source with LUT-8 destination surface specific support (e.g. special rendering mode without input colormap attached or specified)

NOTE1: RLE Image Provider's `RenderTo()` accepts any destination surface (see note 2-a for exceptions). In the sample code provided (see section 6), we just want to be smart and match the native object format by getting this information from the Image Provider and then create a surface accordingly. This is not mandated though. Having no palette attached to the RLE payload won't allow for direct rendering (using `RenderTo()`) to anything else but the native format (LUT-8). You get an error in any other case (DFB\_INVARG). Likewise, a destination rectangle should not be specified either (see note 2-b).

NOTE2: Rendering to a non-LUT-8 surface (a) or/and scaling using a destination rectangle (b) without a palette attached to the input RLE packet (c) is not supported (`RenderTo()` would return DFB\_INVARG).

### **3.5 *Image-Provider modes***

- Direct memory static DirectFB Data Buffer (referencing user-managed packet buffer)
- Standard File loader (DirectFB managed buffer)
- No hardware acceleration
- Unoptimized software fall-back only

NB: Provisions for DFB streaming mode and hardware acceleration in later releases

### **3.6 *Supported DirectFB releases***

- DirectFB 1.0.0-rc1 (Current STLinux distribution)
- DirectFB 1.0.0-rc3 (Current DirectFB developer release)

### **3.7 *Dependencies***

- DirectFB 1.0.0 core package (and optional SDL library for functional testing on development host)

### 3.8 Packaging and configuration

- Standard GNU auto-tools package
- Maintainer mode source tree (autogen/CVS style)
  1. Run autogen.sh from the top source directory with any “configure” switch in order to create a standard distribution.
  2. The package may come already generated (e.g. has a valid configure script) , ready for target configuration and cross-build.
  3. The maintainer mode requires reasonably recent “autoconf” and “automake” tools on the development host machine (e.g. Fedora-Core 6)
- Specific “configure” options:
  1. **Media Trace “configure” switch** : allows for packet header data tracing (very verbose debug mode) in the Image Provider.  
  

```
--enable-media_trace      set media trace enable flag [default=no]
                           (USE_MEDIA_TRACE macro definition)
```
  2. **RC1 Compatibility “configure” switch** : allows for DirectFB-1.0.0-rc1 compatibility. Some API calls have changed in DirectFB-1.0.0-rc3 release.  
  

```
--enable-rc1_compat       set rc1 compatibility flag [default=yes]
                           (USE_RC1_COMPAT macro definition)
```
- Generates a standard mode distribution package (configure/make)
- Modeled as a DirectFB complementary package (DirectFB-extensions-1.0.0)
- Ships with sample(s) / unit-test code
  - Unit-test (DirectFB sample code) : Brief application source code showing how to use a DFB static memory buffer issued Image-Provider instance in order to render the RLE object stored in memory to some surface - Packet build helper function provided (see rle\_build\_packet.c).
  - General purpose DirectFB sample application showing some of the features included in the package. No Image Provider specifics yet. Rather comprehensive test for installed DirectFB core package application compile/link test.

NB:

1. Provisions for further STLinux DirectFB Media-Providers integration
2. SRPM spec file.

### 3.9 Unit-Test applications

The unit-test program "unit\_test\_rle" accepts encoded file names as parameters and renders each file in sequence on a DirectFB primary surface . Please check through this document for various RLE formats support information. Full source code in printed form is also provided at the end of this document. Sample compressed image files are provided with this distribution in directories "./data" and "./data/BD-RLE8". By default, unit\_test\_rle tries to load "[./data/directfb.rle](#)" (*RLEIC\_RLE8 format*) from the current working directory. The exact syntax is :     # unit\_test\_rle <options> [file [file ...]]

NOTE:            Use "--dfb:system=sdl" option switch in order to run any DirectFB application in SDL     mode.

### 3.10 Sample Compressed files

The "[./data](#)" directory contains various sample compressed image files to be used with unit\_test\_rle. The tar archive "BD-RLE8.tar.gz" contains specific DB-RLE8 format files including payload (with standard 54 bytes 'RL' tagged header) and integrated CLUT8 (palette) table.

NOTE:            Use " # tar -xf BD-RLE8.tar.gz" to install these files into "data/BD-RLE8" directory or any other subdirectory.

### 3.11 Standards Compliance

- Little Endian (byte-aligned packet header)
- BMP-Compliant – BMP packet/file header (see enclosed specs below)
- RLE-8 - Standard Run-Length RLE-8 decoder - Top or Bottom first (see enclosed specs)
- BD\_RLE-8 - as per "Blu-Ray Disk , Part 3-1 Specifications" document, Section 9.14.4.2.2.2.1.1
- GNU General Public License version 2

## 4 RLE-8 Code Implementation

### 4.1 Standard RLE-8 Payload

#### 4.1.1 RLE-8 Payload structure

RLE-8 bitmaps are compressed by using a run-length encoding (rle) format for an 8-bit bitmap. this format can be compressed in encoded or absolute modes. both modes can occur anywhere in the same bitmap. The ID for standard RLE-8 compression format is [RLEIC\_RLE8].

- Encoded mode consists of two bytes:

The first byte specifies the number of consecutive pixels to be drawn using the color index contained in the second byte. in addition, the first byte of the pair can be set to zero to indicate an escape that denotes an end of line, end of bitmap, or delta. the interpretation of the escape depends on the value of the second byte of the pair, which can be one of the following:

- 0 end of line.
- 1 end of bitmap.
- 2 delta. the two bytes following the escape contain unsigned values indicating the horizontal and vertical offsets of the next pixel from the current position.

- Absolute mode:

The first byte is zero and the second byte is a value in the range 03h through ffh. the second byte represents the number of bytes that follow, each of which contains the color index of a single pixel. when the second byte is 2 or less, the escape has the same meaning as in encoded mode. in absolute mode, each run must be aligned on a word boundary.

#### 4.1.2 RLE-8 Coding example

The following example shows the hexadecimal values of an 8-bit compressed bitmap:

```
03 04 05 06 00 03 45 56 67 00 02 78 00 02 05 01 02 78 00 00 09 1e 00 01
```

This bitmap would expand as follows (two-digit values represent a color index for a single pixel):

```
04 04 04
06 06 06 06 06
45 56 67
78 78
```

Move current position 5 right and 1 down  
78 78

End of line  
1e 1e 1e 1e 1e 1e 1e 1e 1e

End of rle bitmap

### 4.2 BD-Specific RLE-8 Payload

RLE-8-encoded BD streams (TS/PES source) have a simpler line-based format. The ID for Blu-Ray BD-specific RLE-8 compression format is [RLEIC\_BD\_RLE8]. The following sections summarize data contained within BD\_RLE-8 - as per "Blu-Ray Disk , Part 3-1 Specifications"



document, Section **9.14.4.2.2.1.1** - (*Version 2.01 DRAFT 4', January 2007 - System Description - Blu-ray Disc Read-Only Format - Part 3-1 Core Specifications - Chapter 9 Part 3: Audio Visual Basic Specifications, Pages 1-461 , 1-462 and 1-463*).

## 4.2.1 **BD-RLE-8 Payload structure**

### **9.14.4.2.2.1.1 object\_data() structure**

A graphical Object is represented by a rectangular pixel array of 8-bit indexed-color values which is coded into an `object_data()` structure. The `object_data()` structure is carried by one or more Object Definition Segments in the `object_data_fragment()` structure.

The syntax of the `object_data()` structure is defined as shown below:

Syntax	No. of bits	Mnemonics
<code>object_data() {</code>		
<code>object_data_length</code>	24	<code>uimsbf</code>
<code>object_width</code>	16	<code>uimsbf</code>
<code>object_height</code>	16	<code>uimsbf</code>
<code>while (processed_length &lt; object_data_length)</code>		
{ <code>rle_coded_line()</code> }		
<code>}</code>		

#### **9.14.4.2.2.1.1.1 Semantic definition of fields in object\_data()**

**object\_data\_length:** This field specifies the number of bytes contained in the `object_data()` structure immediately following the last byte of this field.

**object\_width:** This field specifies the width of this Object in pixels.

All Objects shall be equal to or greater than 8 pixels in width.

Objects in HDMV Presentation Graphics streams shall have a width equal to or less than 4096 pixels.

Objects in HDMV Interactive Graphics streams shall have a width equal to or less than `video_width`.

**object\_height:** This field specifies the height of this Object in pixels.

All Objects shall be equal to or greater than 8 pixels in height.

Objects in HDMV Presentation Graphics streams shall have a height equal to or less than 4096 pixels.

Objects in HDMV Interactive Graphics streams shall have a height equal to or less than `video_height`.

## 4.2.2 BD-RLE-8 Line Coding details

### 9.14.4.2.2.1.2 rle\_coded\_line() structure

The rectangular pixel array of 8-bit indexed-color values representing an Object is coded, using Run-Length Encoding (RLE), on a line-by-line basis where the first pixel on the first line is the top left pixel of the Object. The size of each line after coding shall not exceed the size of the line before coding (the “uncompressed” line) plus 16 (for coding overhead). Thus the coding of each line shall be governed by the following rule:

$$SIZE(rle\_coded\_line()) \leq SIZE(maximum\ uncompressed\_line) + 16$$

Where “SIZE” is a function that returns the size of the data, passed as a parameter, in bits. The “maximum uncompressed\_line” is the maximum size of object\_width, i.e.; Maximum uncompressed\_line for HDMV Presentation Graphics streams is 4096\*8 Maximum uncompressed\_line for HDMV Interactive Graphics streams is video\_width\*8

The syntax of rle\_coded\_line() structure is as follows:

Syntax	No. of bits	Mnemonics
rle_coded_line() {		
do {		
if (nextbits != '0000 0000b') {		
pixel_code	8	bslbf
} else {		
8-bit_zero	8	bslbf
switch_1	1	bslbf
switch_2	1	bslbf
if (switch_1 == '0b') {		
if (switch_2 == '0b') {		
if (nextbits != '00 0000b') {		
run_length_zero_1-63	6	bslbf
else		
end_of_line_signal	6	bslbf
} else {		
run_length_zero_64-16K	14	bslbf
}		
} else {		
if (switch_2 == '0b') {		
run_length_3-63	6	bslbf
pixel_code	8	bslbf
} else {		
run_length_64-16K	14	bslbf
pixel_code	8	bslbf
}		
}		
}		
} while (!end_of_line_signal)		
}		

**NOTE** : After testing real BD image patterns, it appears that the description above doesn't reflect accurately fields order, position or endianness conversion. Do not rely on bit stream logic for interpretation of this syntax ...

**9.14.4.2.2.1.2.1 Semantic definition of fields in rle coded line()****pixel\_code:**

An 8-bit code, specifying the pixel value as an entry number of a Palette with 256 entries.

**8-bit-zero:**

An 8-bit field filled with '0000 0000b'.

**switch\_1:**

A 1-bit switch that identifies the meaning of the fields that follow: if set to the value '0b', the field indicates a run-length for a pixel value of '0x00' or an end\_of\_line\_signal; if set to the value '1b', the field indicates a run-length for a pixel value that is not '0x00'.

**switch\_2:**

A 1-bit switch that identifies the meaning of the fields that follow: if set to the value '0b', the field indicates a small run-length or end\_of\_line\_signal; if set to the value '1b', the field indicates a long run-length.

**run\_length\_zero\_1-63:**

The number of pixels that shall be set to a value of '0x00'.

**end\_of\_line\_signal:**

A 6-bit field filled with '00 0000b'. The presence of this field signals the end of the coded line.

**run\_length\_3-63:**

The number of pixels that shall be set to the pixel value defined next. This field shall not have a value less than 3.

**run\_length\_zero\_64-16K:**

The number of pixels that shall be set to a value of '0x00'. This field shall not have a value less than 64.

**run\_length\_64-16K:**

The number of pixels that shall be set to the pixel value defined next. This field shall not have a value less than 64.

## 5 Packet Header Layout

### 5.1 Packet and File Format

- The header follows BMP specifications and layout except for the valid identifier values which adds the magic "RL" ID (to discriminate proprietary RLE semantics from original standard BMP and for eventual future expansion).

**Table 4: RLE Packet Header**

OFFSET	FIELD	SIZE	CONTENTS
0000h	identifier	2 bytes	<p>The signature identifying the bitmap. Possible entries : standard 'BM' (<b>BMP</b>) or proprietary 'RL' (pure <b>RLE</b>) mode.</p> <p><u>NOTE</u> :</p> <p>BM (BMP) mode does not recognize compression formats other than :</p> <ul style="list-style-type: none"> <li>0 - none (RLEIC_NONE)</li> <li>1 - RLE 8-bit (RLEIC_RLE8)</li> <li>2 - RLE 4-bit (RLEIC_RLE4) <b>UNSUPPORTED</b></li> <li>3 - Fields (RLEIC_BITFIELDS) <b>UNSUPPORTED</b></li> </ul> <p>RL (RLE) mode explicitly supports extensions such as (including the above) :</p> <ul style="list-style-type: none"> <li>4 - BD RLE 8-bit (RLEIC_BD_RLE8)</li> </ul>
0002h	file/packet size	1 dword	complete file/packet size in bytes.
0006h	reserved	1 dword	reserved for later use.
000ah	bitmap data offset	1 dword	offset from beginning of file/packet to the beginning of the bitmap data.
000eh	bitmap header size	1 dword	<p>length of the bitmap info header used to describe the bitmap colors, compression; the following sizes are possible:</p> <ul style="list-style-type: none"> <li>40 = 28h - Default</li> <li>12 = 0ch - <b>UNSUPPORTED</b></li> <li>240 = f0h - <b>UNSUPPORTED</b></li> </ul>
0012h	width	1 dword	horizontal width of bitmap in pixels.
0016h	height	1 dword	<p>vertical height of bitmap in pixels.</p> <p>Note : can be negative, denoting top-bottom scan</p>

OFFSET	FIELD	SIZE	CONTENTS
001ah	planes	1 word	number of planes in this bitmap. (Warning : SUPPORT FOR 1 PLANE ONLY ... )
001ch	bits per pixel	1 word	bits per pixel used to store palette entry information. this also identifies in an indirect way the number of possible colors. possible values are: 1 -- monochrome bitmap 4 -- 16 color bitmap 8 -- 256 color bitmap 16 -- 16 bit (high color) bitmap 24 -- 24 bit (true color) bitmap 32 -- 32bit (true color) bitmap
001eh	compression	1 dword	compression specifications. the following values are possible:  0 - none (RLEIC_NONE) 1 - RLE 8-bit (RLEIC_RLE8) 2 - RLE 4-bit (RLEIC_RLE4) <b>UNSUPPORTED</b> 3 - Fields (RLEIC_BITFIELDS) <b>UNSUPPORTED</b> 4 - BD RLE 8-bit (RLEIC_BD_RLE8)
0022h	bitmap data size	1 dword	size of the bitmap data in bytes. this number must be rounded to the next 4 byte boundary.
0026h	hresolution	1 dword	horizontal resolution expressed in pixel per meter. <b>UNUSED</b>
002ah	vresolution	1 dword	vertical resolution expressed in pixel per meter. <b>UNUSED</b>
002eh	colors	1 dword	number of colors used by this bitmap. for a 8-bit / pixel bitmap this will be 100h or 256.
0032h	important colors	1 dword	number of important colors. This number will be equal to the number of colors when any color is important. <b>UNUSED</b>
0036h	palette	n * 4 byte	colormap specification. for every entry in the palette four bytes are used to describe the rgb values of the color in the following byte order: 1 byte for blue component 1 byte for green component 1 byte for red component 1 byte filler set to 0
0436h	bitmap data	x bytes	depending on the compression specs, this field contains all the bitmap data bytes which represent indexes in the color palette for indexed formats.

## 6 Header Fields Details

### 6.1 Semantics

Some fields are left unused for now, and the RLE Image Provider currently support one plane only (the “planes” field at 001ah should be set to 1).

- **height field:** the height field identifies the height of the bitmap in pixels. in other words, it describes the number of scan lines of the bitmap. if this field is negative, indicating a top-down dib, the compression field must be either bi\_rgb or bi\_bitfields for “BM” identified headers. top-down dibs cannot be compressed in BMP mode. There is no such restriction in proprietary RLE mode (“RL” identified headers)
- **bits per pixel field:** the bits per pixel (bpp) field of the bitmap file determines the number of bits that define each pixel and the maximum number of colors in the bitmap.
  - **when this field equals 1.**

the bitmap is monochrome, and the palette contains two entries. each bit in the bitmap array represents a pixel. if the bit is clear, the pixel is displayed with the color of the first entry in the palette; if the bit is set, the pixel has the color of the second entry in the table.
  - **when this field equals 4.**

the bitmap has a maximum of 16 colors, and the palette contains up to 16 entries. each pixel in the bitmap is represented by a 4-bit index into the palette. for example, if the first byte in the bitmap is 1fh, the byte represents two pixels. the first pixel contains the color in the second palette entry, and the second pixel contains the color in the sixteenth palette entry.
  - **when this field equals 8.**

the bitmap has a maximum of 256 colors, and the palette contains up to 256 entries. in this case, each byte in the array represents a single pixel.

- **when this field equals 16.**

the bitmap has a maximum of  $2^{16}$  colors. if the compression field of the bitmap file is set to `bi_rgb`, the palette field does not contain any entries. each word in the bitmap array represents a single pixel. the relative intensities of red, green, and blue are represented with 5 bits for each color component. the value for blue is in the least significant 5 bits, followed by 5 bits each for green and red, respectively. the most significant bit is not used.

- **when this field equals 24.**

the bitmap has a maximum of  $2^{24}$  colors, and the palette field does not contain any entries. each 3-byte triplet in the bitmap array represents the relative intensities of blue, green, and red, respectively, for a pixel.

- **when this field equals 32.**

the bitmap has a maximum of  $2^{32}$  colors. if the compression field of the bitmap is set to `bi_rgb`, the palette field does not contain any entries. each dword in the bitmap array represents the relative intensities of blue, green, and red, respectively, for a pixel. the high byte in each dword is not used.

NOTE: if the compression field of the bitmap file is set to `bi_bitfields`, the palette section contains three dword color masks that specify the red, green, and blue components, respectively, of each pixel. each word in the bitmap array represents a single pixel.

- **compression field:** The compression field specifies the way the bitmap data is stored in the file/packet. this information together with the bits per pixel (bpp) field identifies the compression algorithm to follow.

The following values are possible in this field:

VALUE	MEANING
RLEIC_NONE (TEST MODE)	An uncompressed format for testing mainly.
RLEIC_RLE8	A run-length encoded (RLE) format for bitmaps with 8 bits per pixel. the compression format is a two-byte format consisting of a count byte followed by a byte containing a color index. for more information, see the following remarks section.
RLEIC_RLE4 (UNSUPPORTED)	A rle format for bitmaps with 4 bits per pixel. the compression format is a two-byte format consisting of a count byte followed by two word-length color indexes for more information, see the following remarks section.
RLEIC_BITFIELDS (UNSUPPORTED)	Specifies that the bitmap is not compressed and that the color table consists of three double word color masks that specify the red, green, and blue components, respectively, of each pixel. this is valid when used with 16- and 32- bits-per-pixel bitmaps.
RLEIC_BD_RLE8	A run-length encoded (RLE) format for bitmaps with 8 bits per pixel. The compression format differ from RLEIC_RLE8 defined above. Specifications according to "Blu-Ray Disk", Part 3-1 : Core Specifications" Section <b>9.14.4.2.2.2.1.1</b> .

- **colors field:** the colors field specifies the number of color indexes in the color table that are actually used by the bitmap. In BMP compatibility mode ("BM" identifier), if this value is zero, the bitmap uses the maximum number of colors corresponding to the value of the bpp field for the compression mode specified by the compression field.

- **if the colors field is non-zero and the bpp field less than 16**, the colors field specifies the actual number of colors the graphics engine or device driver accesses.
- **if the colors field is zero and the bpp field less than 16 in RLE proprietary mode** ("RL" identifier), the rendered image is not tied to any specific colormap and just only contains pure index values.
- **if the colors field is non-zero and the bpp field is 16 or greater**, then colors field specifies the size of the color table used to optimize performance of color palettes. **UNUSED**
- **if the colors field is non-zero and the bpp equals 16 or 32**, the optimal color palette starts immediately following the three double word masks with bitfields compression. **(UNSUPPORTED)**
- **if the bitmap is a packed bitmap** (a bitmap in which the bitmap array immediately follows the bitmap header and which is referenced by a single pointer), the colors field must be either 0 or the actual size of the color table.



- **Important colors field:** the important colors field specifies the number of color indexes that are considered important for displaying the bitmap. if this value is zero, all colors are important. (UNUSED)

## 7 Application Code Example

### 7.1 DirectFB RLE Image Provider Unit-Test and Sample code

UNIT TEST AND SAMPLE PROGRAM : `unit_test_rle.c`

```

/*
 * Copyright (C) 2006 ST-Microelectronics R&D <alain.clement@st.com>
 */

/*
 *
 *      RLE IMAGE PROVIDER UNIT TEST
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#include <directfb.h>
#include <direct/messages.h>

#include "unit_test_rle_helpers.h"
#include "rle_build_packet.h"

#define DFBFAIL(err) \
{ \
    fprintf( stderr, "%s <%d>:\n\t", __FILE__, __LINE__ ); \
    DirectFBErrorFatal( #err, err ); \
}

#define DFBCHECK(x) \
{ \
    DFBResult err = x; \
    \
    if (err != DFB_OK)        DFBFAIL(err); \
}

#define DISPLAY_INFO(x...) do { \
    if (!direct_config->quiet) \
        direct_messages_info( x ); \
} while (0)

// WE'RE USING A CLEVER APPROACH FOR THE RLE PACKET BUILDER NOW
#define CLEVER_APPROACH

// WE'RE USING THE RLE PACKET BUILDER ONLY FOR NOW (DIRECT FILE ACCESS)
#define USE_PACKET_BUILDER_ONLY

// WE WANT TO DISPLAY SUBTITLES
#define SUBTITLES_MODE

//-----
//      Pointer to user-managed buffer holding RLE packet data bytes
//-----
static void          *buffer          = NULL;

# ifdef CLEVER_APPROACH
    static DFBColor    *color_palette = NULL;

```

```

    static unsigned int    number_of_colors = 0;

#   endif

//-----
// Destroy DirectFB data buffer
//-----
static DFBResult
rle_destroy_databuffer (IDirectFBDataBuffer *databuffer)
{
    // Release the databuffer, we don't need it anymore.
    if (databuffer)    databuffer->Release (databuffer);
    // Release the static buffer, we don't need it anymore.
    if (buffer)        free(buffer);        buffer = NULL;

#   ifndef CLEVER_APPROACH

    // Release the static colormap, we don't need it anymore.
    if (color_palette) free(color_palette);    color_palette = NULL;
    number_of_colors = 0;

#   endif

    return DFB_OK;
}

//-----
// Create a DirectFB static data buffer holding our RLE data (read from a file)
//-----
static DFBResult
rle_build_databuffer (IDirectFB *dfb, char *filename,
                     IDirectFBDataBuffer **pdatabuffer)
{
    // An Image provider instance can also be created from a directfb buffer
    IDirectFBDataBuffer *databuffer = NULL;

    // Buffer description
    DFBDatabufferDescription    buffer_desc;

    // Image file elements and information
    unsigned long    width      = 0;
    unsigned long    height     = 0;
    unsigned long    depth      = 0;

    // Image file boolean attributes
    unsigned long    rawmode     = 0;
    unsigned long    bdmode      = 0;
    unsigned long    topfirst    = 0;

    // Size in bytes
    unsigned long    buffer_size = 0;

    unsigned long    payload_size = 0;
    unsigned long    colormap_size = 0;

    void *payload = NULL;
    void *colormap = NULL;

    int    success;

    // We actually read the file contents into memory and collect details
    // using a helper function (see "unit_test_rle_helpers.h").
    // NOTE: rle_load_image will allocate payload and colormap buffers
    // NULL buffer pointers means internal "allocation requested"
    // Those buffers must be released.
    success = rle_load_image (filename, &width, &height, &depth,
                             &rawmode, &bdmode, &topfirst, &buffer_size,
                             &payload, &payload_size,

```

```

        &colormap, &colormap_size);

if (!success)
{
    DISPLAY_INFO ("%s: Couldn't load image file %s\n", __FUNCTION__, filename);
    return DFB_UNSUPPORTED;
}

# ifdef CLEVER_APPROACH

//    Let's stash our colormap in global space first
{
    int    i;

    number_of_colors = colormap_size/4;

    color_palette = malloc (number_of_colors*4);
    if (!color_palette) return DFB_FAILURE;

    for (i = 0; i < number_of_colors; i++)
    {
        DFBColor c;

        c.a = 0xff;
        c.r = ((__u8*)colormap)[i*4+2];
        c.g = ((__u8*)colormap)[i*4+1];
        c.b = ((__u8*)colormap)[i*4+0];

        color_palette[i] = c;
    }
}

//    We won't attach the colormap to the RLE packet :
//    RLE header and payload only
buffer_size = RLE_HEADER_SIZE + payload_size;

//    Static memory data buffer (packet) allocation.
buffer = malloc (buffer_size);
if (!buffer) return DFB_FAILURE;

//    Let's paste our payload for encapsulation
//    (could have been read from the PES stream fragments as well).
memcpy (buffer+RLE_HEADER_SIZE, payload, payload_size);

//    We only want to setup the header, thus completing the encapsulation
//    We are using another helper function (see "rle_build_packet.h").
//    NULL payload and colormap means that we don't want the helper
//    function to copy anything. Zero "colormap_size" means that we don't
//    intend to attach any colormap to the packet.
//    "payload_size" must be set though" ...
DFB_CHECK ( rle_build_packet (buffer, buffer_size, width, height, depth,
                             rawmode, bdmode, topfirst,
                             NULL, 0,
                             NULL, payload_size));

//    NOTE: FEEL FREE TO TWEAK THE HELPER FUNCTION ABOVE IN ORDER TO
//    ACHIEVE TRUE ZERO-COPY ENCAPSULATION (PERHAPS USING THE
//    PAYLOAD OFFSET). YOU GET THE IDEA ...
//    THE RLE IMAGE PROVIDER DOESN'T HANDLE PAYLOAD FRAGMENTS,
//    SO THE PAYLOAD MUST USE A CONTIGUOUS SEGMENT IN THE RLE
//    PACKET.

# else

//    We attach the colormap to the RLE packet :
//    RLE header, colormap (if available) and payload together
buffer_size = RLE_HEADER_SIZE + colormap_size + payload_size;

//    Static memory data buffer allocation.
buffer = malloc (buffer_size);
if (!buffer) return DFB_FAILURE;

```

```
// We now rebuild the whole image buffer, setting up the header,
// copying the (eventual) collected colormap and payload.
// We are using another helper function (see "rle_build_packet.h").
DFB_CHECK ( rle_build_packet (buffer, buffer_size, width, height, depth,
rawmode,
bdmode, topfirst,
colormap, colormap_size,
payload, payload_size));

// Note: we avoided colormap & payload copies by adopting a
// more clever approach previously ...

# endif

// Now we can release colormap & payload buffers eventually allocated
// previously by "rle_load_image" helper function (since "buffer" has
// just been set by "rle_build_packet")
if (colormap) free(colormap); colormap = NULL; colormap_size = 0;
if (payload) free(payload); payload = NULL; payload_size = 0;

// An Image Provider can be obtained from a directfb data buffer that
// we specify and set up by ourselves using a static memory buffer
buffer_desc.flags = DBDESC_MEMORY;
buffer_desc.file = NULL;
buffer_desc.memory.data = buffer;
buffer_desc.memory.length = buffer_size;
DFB_CHECK (dfb->CreateDataBuffer (dfb, &buffer_desc, &databuffer));
// Note: if no description had been specified (NULL), a streamed data
// buffer would have been created instead.

// Return newly created databuffer
*p_databuffer = databuffer;

return DFB_OK;
}

//-----
// Unit Test main
//-----
int main (int argc, char **argv)
{
    int i, j;
    DFBResult rle_build_databuffer_err;

    // File name to load logo image from
    char *filename = NULL;

    // Basic directfb elements
    IDirectFB *dfb = NULL;
    IDirectFBSurface *primary = NULL;
    int screen_width = 0;
    int screen_height = 0;

    // The image is to be loaded into a surface that we can blit from.
    IDirectFBSurface *logo = NULL;

    // Loading an image is done with an Image Provider.
    IDirectFBImageProvider *provider = NULL;

    // An Image provider instance can also be created from a directfb buffer
    IDirectFBDataBuffer *databuffer = NULL;

    // Surface description
    DFBSurfaceDescription surface_desc;

    // Initialize directfb first
    DFB_CHECK (DirectFBInit (&argc, &argv));
    DFB_CHECK (DirectFBCreate (&dfb));
    DFB_CHECK (dfb->SetCooperativeLevel (dfb, DFSC_L_FULLSCREEN));
}
```

```

// Create primary surface
surface_dsc.flags = DSDESC_CAPS;
surface_dsc.caps = DSCAPS_PRIMARY | DSCAPS_FLIPPING;
DFB_CHECK (dfb->CreateSurface( dfb, &surface_dsc, &primary ));
DFB_CHECK (primary->GetSize (primary, &screen_width, &screen_height));

if (argc==1)
{
    argv[1] = "./data/directfb.rle";
    argc++;
}

DISPLAY_INFO ("Rendering %d files\n",argc-1);
for (j=1; j<argc; j++)
{
    //
    // --- WE CREATE OUR IMAGE PROVIDER INSTANCE HERE
    //
    filename = argv[j];
    DISPLAY_INFO ("Rendering : %s\n",filename);

    // We create a directfb data buffer holding RLE image contents that we
    // pick up from a file (could get the RLE contents from memory as well).
    // "rle_build_databuffer" details the process of dealing with a memory
    // RLE packet as a matter of fact.
    rle_build_databuffer_err = rle_build_databuffer (dfb, filename, &databuffer);
    if (rle_build_databuffer_err == DFB_OK) {
        // We want to create an Image Provider tied to a directfb data buffer.
        // DirectFB will find (or not) an Image Provider for the data type
        // depending on Image Providers probe method (sniffing data headers)
        DFB_CHECK (databuffer->CreateImageProvider (databuffer, &provider));
    }
    else {
        #ifdef USE_PACKET_BUILDER_ONLY
        DFB_FAIL(rle_build_databuffer_err);
        #else
        // We could also create an Image Provider by passing a filename.
        // DirectFB will find (or not) an Image Provider matching the file type.
        DFB_CHECK (dfb->CreateImageProvider (dfb, filename, &provider));
        #endif
    }

    // Get a surface description from the provider. It will contain the width,
    // height, bits per pixel and the flag for an alphachannel if the image
    // has one. If the image has no alphachannel the bits per pixel is set to
    // the bits per pixel of the primary layer to use simple blitting without
    // pixel format conversion.
    DFB_CHECK (provider->GetSurfaceDescription (provider, &surface_dsc));

    // Create a surface based on the description of the provider.
    DFB_CHECK (dfb->CreateSurface( dfb, &surface_dsc, &logo ));

    // Let the provider render to our surface. Image providers are supposed
    // to support every destination pixel format and size. If the size
    // differs the image will be scaled (bilinear). The last parameter allows
    // to specify an optional destination rectangle. We use NULL here so that
    // our image covers the whole logo surface.
    DFB_CHECK (provider->RenderTo (provider, logo, NULL));
    // Note: RLE Image Provider allows for direct non-scaled LUT-8 surface
    // rendering without any attached colormap.

    #ifndef CLEVER_APPROACH
    // Let's setup our logo surface palette outside of the RLE Image
    // Provider if we got a colormap from rle_build_databuffer ...
    if (color_palette)
    {
        IDirectFBPalette *palette;
        DFB_CHECK (logo->GetPalette (logo, &palette));
    }

```

```
    palette->SetEntries (palette, color_palette, number_of_colors, 0);
    palette->Release (palette);
}
#endif

//
// --- WE GET RID OF OUR IMAGE PROVIDER INSTANCE HERE
//
// Release the provider, we don't need it anymore.
provider->Release (provider);    provider    = NULL;
// Destroy the databuffer as well, we don't need it anymore.
rle_destroy_databuffer (databuffer);    databuffer = NULL;

# ifndef SUBTILES_MODE
// We want to let the logo slide in on the left and slide out on the
// right.
for (i = -surface_dsc.width; i < screen_width; i++)
# else
// We want to let the logo slide in on the right and slide out on the
// left.
for (i = screen_width-1; i >= -surface_dsc.width; i--)
# endif
{
    // Clear the screen.
    DFBCHECK (primary->FillRectangle (primary, 0, 0,
                                      screen_width, screen_height));

    // Blit the logo vertically centered with "i" as the X coordinate.
    // NULL means that we want to blit the whole surface.
    DFBCHECK (primary->Blit (primary, logo, NULL, i,
                           (screen_height - surface_dsc.height) / 2));

    // Flip the front and back buffer, but wait for the vertical
    // retrace to avoid tearing.
    DFBCHECK (primary->Flip (primary, NULL, DSFLIP_WAITFORSYNC));

    if (argc < 3)
    {
        usleep(1000*5);
    }
}

// Release the image.
if (logo)
{
    logo->Release (logo);
}

}

// Release everything else
primary->Release (primary);
dfb->Release (dfb);

return 0;
}
```

## 7.2 DirectFB RLE Image Provider Packet Builder

UNIT TEST AND SAMPLE CODE LIBRARY: rle\_build\_packet.c

```

/*
 * Copyright (C) 2006 ST-Microelectronics R&D <alain.clement@st.com>
 */

/*
 *
 *          RLE PACKET HELPER LIBRARY
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#include "rle_build_packet.h"

/*****
 *
 *   rle_build_packet:
 *
 *       DirectFB static memory buffer setup for RLE Image Provider
 *
 *   RLE Image Provider expects its DirectFB buffer holding header data+colormap
 *   (RLE_OVERHEAD) and the RLE payload. This helper function sets up such a
 *   buffer given a minimal set of parameters. The application must account for
 *   RLE_HEADER_SIZE bytes at the beginning, then colormap bytes (variable,
 *   dependent on depth) then the payload. The application must supply the
 *   colormap but has the flexibility of submitting the payload data for copy
 *   in the allocated I/O buffer (non NULL payload parameter), or collecting the
 *   data from source ahead of time into the I/O buffer, thus avoiding a copy
 *   step, as long as the payload sections make room for RLE_OVERHEAD(depth)
 *   reserved bytes at the beginning of "buffer".
 *
 *   Note 1: The colormap is not required to be supplied. In such a case (NULL
 *   pointer), the colormap section gets skipped (absent from header).
 *
 *   Note 2: BMP/DIB row addressing normally starts from the bottom. A negative
 *   height means images start from the top of the bitmap (weirdness ?)
 *   "topfirstmode" flag is provided here to get a start from the top of
 *   the bitmap. This behavior is normally applicable to non RLE
 *   compressed images, but extended to any format in our RLE Image
 *   Provider implementation.
 *
 *   -----
 *
 *               DirectFB buffer Mapping
 *
 *   -----
 *
 *   RLE_PREAMBLE
 *   (14)
 *
 *   -----
 *
 *   RLE_HEADER
 *   (54)
 *
 *   RLE_BIHEADER
 *   (40)
 *
 *   -----
 *
 *   RLE_OVERHEAD
 *   (54-310)
 *
 *   RLE_PALETTE
 *   (0-256)
 *
 *   -----
 *
 *   RLE_DFB_BUFFER
 *   (Packet Size)
 *
 *
 *
 *
 *               RLE_PAYLOAD
 *
 *
 *   rawmode = Raw-Pixels (1) or Run-Length coded (0)
 *   bdmode  = BD-RLE      (1) or Legacy-RLE coded (0)

```



```

*
* -----
*
*****/

DFBResult
rle_build_packet (
    __u8      *buffer,      /* I/O : Packet buffer */
    __u32     buffer_size, /* I   : Packet size */
    __u32     width,       /* I   : Image width */
    __u32     height,      /* I   : Image height */
    __u32     depth,       /* I   : Pixels depth */
    __u32     rawmode,     /* I   : Raw mode flag (0/1) */
    __u32     bdmode,      /* I   : BD-RLE mode flag (0/1) */
    __u32     topfirstmode, /* I   : Topfirst mode flag (0/1) */
    DFBColor  *colormap,   /* I   : Palette / NULL */
    __u32     colormap_size, /* I   : Palette size */
    void      *payload,    /* I   : Payload / NULL */
    __u32     payload_size /* I   : Payload size */
)
{
    /*
     * RLE Compression identifier (see BMP/RLE header - Offset:30 --- DWORD)
     */
    typedef enum {
        RLEIC_NONE      = 0, /* Implemented as of version 1.0.0 */
        RLEIC_RLE8      = 1, /* Implemented as of version 1.0.0 */
        RLEIC_RLE4      = 2, /* Unimplemented */
        RLEIC_BITFIELDS = 3, /* Unimplemented */
        RLEIC_BD_RLE8   = 4, /* Implemented as of version 1.1.0 */
    } RLEImageCompression;

    /* Little-endian portable data bytes order ... */
    #define RLE_HEADER_SET_MAGIC(ptr,l,h) { \
        ptr[0]=(__u8)l; \
        ptr[1]=(__u8)h; \
    }
    #define RLE_HEADER_WRITE_LE_16(ptr,data) { \
        *ptr++ = (data>>0 ) & 0xff; \
        *ptr++ = (data>>8 ) & 0xff; \
    }
    #define RLE_HEADER_WRITE_LE_32(ptr,data) { \
        *ptr++ = (data>>0 ) & 0xff; \
        *ptr++ = (data>>8 ) & 0xff; \
        *ptr++ = (data>>16) & 0xff; \
        *ptr++ = (data>>24) & 0xff; \
    }

    #define RLE_PALETTE_SIZE      (colormap_size & ~0x03)
    #define RLE_OVERHEAD          (RLE_HEADER_SIZE+RLE_PALETTE_SIZE)

    #define MIN(a,b)  (((a)<(b)) ? (a) : (b))

    DFBResult ret = DFB_OK;
    __u8      *hptr;
    __u32     bihsize      = RLE_BIHEADER_SIZE;
    __u32     num_colors   = RLE_PALETTE_SIZE/4;
    __u32     imp_colors   = RLE_PALETTE_SIZE/4;
    __u32     img_offset   = RLE_OVERHEAD;
    int      payload_max_size = MIN(buffer_size-RLE_OVERHEAD,payload_size);
    int      height_dib    = topfirstmode ? -height : height;

    RLEImageCompression compression;

    /* Check for any payload/buffer size discrepancy */
    if (payload_max_size<0) {
        return DFB_INVARG;
    }
}

```

```

/* Switch compression mode */
if (rawmode) {
    switch (depth) {
        case 16:
        case 24:
        case 32:
        case 8:
        case 4:
        case 1:
            compression = RLEIC_NONE;
            break;
        default:
            return DFB_INVARG;
    }
}
else {
    switch (depth) {
        case 8:
            if (!bdmode) {
                compression = RLEIC_RLE8;
                break;
            }
            else {
                compression = RLEIC_BD_RLE8;
                break;
            }
        case 4:
            if (!bdmode) {
                compression = RLEIC_RLE4;
                break;
            }
        case 1:
        case 16:
        case 24:
        case 32:
        default:
            return DFB_INVARG;
    }
}

/* Cleanup preamble, header and eventual palette area */
memset( buffer, 0, img_offset);

/* Pointer to buffer header start */
hptr = buffer;

/* Offset:00 --- 2 bytes: Magic - Zero for now */
RLE_HEADER_WRITE_LE_16 (hptr, 0);

/* Offset:02 --- 4 bytes: FileSize */
RLE_HEADER_WRITE_LE_32 (hptr, buffer_size);

/* Offset:06 --- 4 bytes: Reserved */
RLE_HEADER_WRITE_LE_32 (hptr, 0);

/* Offset:10 --- 4 bytes: DataOffset */
RLE_HEADER_WRITE_LE_32 (hptr, img_offset);

/* Offset:14 --- 4 bytes: HeaderSize */
RLE_HEADER_WRITE_LE_32 (hptr, bihsize);

/* Offset:18 --- 4 bytes: Width */
RLE_HEADER_WRITE_LE_32 (hptr, width);

/* Offset:22 --- 4 bytes: Height (negative value means top first mode) */
RLE_HEADER_WRITE_LE_32 (hptr, height_dib);

/* Offset:26 --- 2 bytes: Planes */
RLE_HEADER_WRITE_LE_16 (hptr, 1);

/* Offset:28 --- 2 bytes: Depth */
RLE_HEADER_WRITE_LE_16 (hptr, depth);

```

```
/* Offset:30 --- 4 bytes: Compression */
RLE_HEADER_WRITE_LE_32 (hptr, compression);

/* Offset:34 --- 4 bytes: CompressedSize */
RLE_HEADER_WRITE_LE_32 (hptr, payload_max_size);

/* Offset:38 --- 4 bytes: HorizontalResolution (don't care ...) */
RLE_HEADER_WRITE_LE_32 (hptr, 0);

/* Offset:42 --- 4 bytes: VerticalResolution (don't care ...) */
RLE_HEADER_WRITE_LE_32 (hptr, 0);

/* Offset:46 --- 4 bytes: UsedColors */
RLE_HEADER_WRITE_LE_32 (hptr, num_colors);

/* Offset:50 --- 4 bytes: ImportantColors (don't care ...) */
RLE_HEADER_WRITE_LE_32 (hptr, imp_colors);

/* Offset:54 --- 4 x num_colors bytes: Palette */
if (colormap_size>0 && colormap) {
    int i;
    for (i = 0; i < num_colors; i++) {
        if (i>=colormap_size/4){
            RLE_HEADER_WRITE_LE_32 (hptr, 0);
            continue;
        }
        ((__u8*)hptr)[i*4+3] = colormap[i].a;
        ((__u8*)hptr)[i*4+2] = colormap[i].r;
        ((__u8*)hptr)[i*4+1] = colormap[i].g;
        ((__u8*)hptr)[i*4+0] = colormap[i].b;
    }
}

/* Offset:54 + 4 x num_colors --- bytes: Payload */
if (payload_max_size>0 && payload) {
    memcpy(buffer+img_offset, payload, payload_max_size);
}

/* Set first 2 bytes: Magic - valid header now */
RLE_HEADER_SET_MAGIC(buffer, 'R', 'L');

return ret;
}
```

## 7.3 DirectFB RLE Image Independent Image Loader

UNIT TEST AND SAMPLE CODE LIBRARY:     **unit\_test\_rle\_helpers.c**

```

/*
 * Copyright (C) 2006 ST-Microelectronics R&D <alain.clement@st.com>
 */

/*
 *
 *      RLE/BMP FILE FORMAT HELPER LIBRARY
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#include "unit_test_rle_helpers.h"

typedef unsigned long DATA32;

/*****
 *
 *      RLE/BMP TYPE DEFINITIONS
 *
 *****/

typedef struct tagRGBQUAD {
    unsigned char    rgbBlue;
    unsigned char    rgbGreen;
    unsigned char    rgbRed;
    unsigned char    rgbReserved;
} RGBQUAD;

/*
 * RLE Compression identifier (see BMP/RLE header - Offset:30 --- DWORD)
 */
typedef enum {
    RLEIC_NONE       = 0,    /* Implemented      as of version 1.0.0 */
    RLEIC_RLE8       = 1,    /* Implemented      as of version 1.0.0 */
    RLEIC_RLE4       = 2,    /* Unimplemented */
    RLEIC_BITFIELDS  = 3,    /* Unimplemented */
    RLEIC_BD_RLE8    = 4,    /* Implemented      as of version 1.1.0 */
} RLEImageCompression;

/*****
 *
 *      RLE/BMP LITTLE-ENDIAN UTILITIES
 *
 *****/

static int ReadLeShort      (FILE * file, unsigned short *ret)
{
    unsigned char    b[2];

    if (fread(b, sizeof(unsigned char), 2, file) != 2) return 0;

    *ret = (b[1] << 8) | b[0];
    return 1;
}

static int ReadLeSignedShort (FILE * file, signed short *ret)
{
    unsigned char    b[2];

    if (fread(b, sizeof(unsigned char), 2, file) != 2) return 0;

```

```

    *ret = (b[1] << 8) | b[0];
    return 1;
}

static int    ReadleLong                (FILE * file, unsigned long *ret)
{
    unsigned char    b[4];

    if (fread(b, sizeof(unsigned char), 4, file) != 4) return 0;

    *ret = (b[3] << 24) | (b[2] << 16) | (b[1] << 8) | b[0];
    return 1;
}

static int    ReadleSignedLong          (FILE * file, signed long *ret)
{
    unsigned char    b[4];

    if (fread(b, sizeof(unsigned char), 4, file) != 4) return 0;

    *ret = (b[3] << 24) | (b[2] << 16) | (b[1] << 8) | b[0];
    return 1;
}

/*****
 *
 *                               RLE/BMP FILE FORMAT HELPER FUNCTION
 *
 *****/

int    rle_load_image                (char                *filename,
                                     unsigned long        *width_ptr,
                                     unsigned long        *height_ptr,
                                     unsigned long        *depth_ptr,
                                     unsigned long        *rawmode_ptr,
                                     unsigned long        *bdmode_ptr,
                                     unsigned long        *topfirst_ptr,
                                     unsigned long        *buffer_size_ptr,
                                     void                **payload_ptr,
                                     unsigned long        *payload_size_ptr,
                                     void                **colormap_ptr,
                                     unsigned long        *colormap_size_ptr)
{
    FILE                *f;
    char                type[2];
    unsigned long        size, offset, headSize, comp, imgsize;
    unsigned short        tmpShort, planes, bitcount, ncols;
    signed short        tmpSignedShort;
    signed long        tmpSignedLong;
    unsigned long        topfirst;
    unsigned long        i, w, h;
    RGBQUAD                rgbQuads[256];
    unsigned long        rmask = 0xff, gmask = 0xff, bmask = 0xff;
    unsigned long        rshift = 0, gshift = 0, bshift = 0;

    const    unsigned long    preambleSize = 14;

    f = fopen(filename, "rb");
    if (!f)
    {
        return 0;
    }

    /* HEADER */
    {
        struct stat statbuf;

        if (stat(filename, &statbuf) == -1)
        {
            fclose(f);
            return 0;
        }
        size = statbuf.st_size;

```

```

    if (fread(type, 1, 2, f) != 2)
    {
        fclose(f);
        return 0;
    }

#    define IS_MAGIC(ptr,l,h) ((ptr[0]==l) && (ptr[1]==h))
    if (!IS_MAGIC(type, 'R', 'L') && !IS_MAGIC(type, 'B', 'M'))
    {
        fclose(f);
        return 0;
    }

    fseek(f, 8, SEEK_CUR);
    ReadleLong(f, &offset);
    ReadleLong(f, &headSize);
    if (offset >= size || offset < headSize + preambleSize)
    {
        fclose(f);
        return 0;
    }

    /* HEADER DATA */
    switch (headSize)
    {
    case 40:
        ReadleLong(f, &w);
        tmpSignedLong = 0;
        ReadleSignedLong(f, &tmpSignedLong);
        topfirst = tmpSignedLong < 0 ? 1 : 0;
        h = topfirst ? -tmpSignedLong : tmpSignedLong;
        ReadleShort(f, &planes);
        ReadleShort(f, &bitcount);
        ReadleLong(f, &comp);
        ReadleLong(f, &imgsize);
        imgsize = size - offset;

        fseek(f, 16, SEEK_CUR);
        break;

    case 12:
        ReadleShort(f, &tmpShort);
        w = tmpShort;
        tmpSignedShort = 0;
        ReadleSignedShort(f, &tmpSignedShort);
        topfirst = tmpSignedShort < 0 ? 1 : 0;
        h = topfirst ? -tmpSignedShort : tmpSignedShort;
        ReadleShort(f, &planes);
        ReadleShort(f, &bitcount);
        imgsize = size - offset;
        comp = RLEIC_NONE;
        //break; /* UNSUPPORTED BACKWARD COMPAT. */

    default:
        fclose(f);
        return 0;
    }

    if (planes != 1)
    {
        fclose(f);
        return 0;
    }

    if ((w < 1) || (h < 1) || (w > 8192) || (h > 8192))
    {
        fclose(f);
        return 0;
    }

    ncols = 0;

```

```

    if (bitcount < 16)
    {
        /* COLORMAP DATA */
        switch (headSize)
        {
            case 40:
                ncols = (offset - headSize - preambleSize) / 4;
                if (ncols > 256)
                    ncols = 256;
                fread(rgbQuads, 4, ncols, f);
                break;

            case 12:
                ncols = (offset - headSize - preambleSize) / 3;
                if (ncols > 256)
                    ncols = 256;
                for (i = 0; i < ncols; i++)
                    fread(&rgbQuads[i], 3, 1, f);
                //break; /* UNSUPPORTED BACKWARD COMPAT. */
            default:
                fclose(f); /* UNSUPPORTED BACKWARD COMPAT. */
                return 0;
        }
    }
    else if (bitcount == 16 || bitcount == 32)
    {
        if (comp == RLEIC_BITFIELDS)
        {
            int bit;

            ReadleLong(f, &bmask);
            ReadleLong(f, &gmask);
            ReadleLong(f, &rmask);
            for (bit = bitcount - 1; bit >= 0; bit--)
            {
                if (bmask & (1 << bit))
                    bshift = bit;
                if (gmask & (1 << bit))
                    gshift = bit;
                if (rmask & (1 << bit))
                    rshift = bit;
            }
            fclose(f); /* UNSUPPORTED BACKWARD COMPAT. */
            return 0;
        }
        else if (bitcount == 16)
        {
            rmask = 0x7C00;
            gmask = 0x03E0;
            bmask = 0x001F;
            rshift = 10;
            gshift = 5;
            bshift = 0;
        }
        else if (bitcount == 32)
        {
            rmask = 0x00FF0000;
            gmask = 0x0000FF00;
            bmask = 0x000000FF;
            rshift = 16;
            gshift = 8;
            bshift = 0;
        }
    }
}

if (payload_ptr)
{
    fseek(f, offset, SEEK_SET);
    if (!*payload_ptr)
    {

```

```

        *payload_ptr = malloc(imgsize);
    }
    else if (!payload_size_ptr || *payload_size_ptr < imgsize)
    {
        return 0;
    }

    if (!*payload_ptr)
    {
        fclose(f);
        return 0;
    }

    fread(*payload_ptr, imgsize, 1, f);
}

fclose(f);

if (colormap_ptr && bitcount < 16)
{
    if (ncols==0)
    {
        char palname[512];
        sprintf (palname, "%s.%s", filename, "pal");
        f = fopen(palname, "rb");
        if (f)
        {
            fread(rgbQuads, 4, ncols=256, f);
            fclose(f);
        }

        if (ncols>0)
        {
            if (!*colormap_ptr)
            {
                *colormap_ptr = malloc(sizeof(RGBQUAD)*ncols);

                if (!*colormap_ptr)
                {
                    return 0;
                }
            }
            else if (!colormap_size_ptr
                || *colormap_size_ptr < sizeof(RGBQUAD)*ncols)
            {
                return 0;
            }

            memcpy(*colormap_ptr, rgbQuads, sizeof(RGBQUAD)*ncols);
        }
        else
        {
            *colormap_ptr = NULL;
        }
    }

    if (payload_size_ptr)
        *payload_size_ptr = imgsize;

    if (colormap_size_ptr)
        *colormap_size_ptr = sizeof(RGBQUAD)*ncols;

    if (width_ptr)
        *width_ptr = w;

    if (height_ptr)

```



```
        *height_ptr = h;
    if (depth_ptr)
        *depth_ptr = bitcount;
    if (rawmode_ptr)
        *rawmode_ptr = comp == RLEIC_NONE ? 1:0;
    if (bdmode_ptr)
        *bdmode_ptr = comp == RLEIC_BD_RLE8 ? 1:0;
    if (topfirst_ptr)
        *topfirst_ptr = comp == RLEIC_BD_RLE8 ? 1:topfirst;
    if (buffer_size_ptr)
        *buffer_size_ptr = preambleSize+headSize+imgsize+sizeof(RGBQUAD)*ncols;

    return 1;
}
```