User's Manual

RH850/D1x device family

Grape_app User Manual

User's Manual: Application

All information contained in these materials, including products and product specifications, represents information on the product at the time of publication and is subject to change by Renesas Electronics Corp. without notice. Please review the latest information published by Renesas Electronics Corp. through various means, including the Renesas Electronics Corp. website (http://www.renesas.com).

Renesas Electronics
www.renesas.com

Rev. 0.50 June 2017

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other disputes involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawing, chart, program, algorithm, application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics products.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
   Renesas Electronics products are neither intended nor authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems, surgical implantations etc.), or may cause serious property damages (space and undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for which the product is not intended by Renesas Electronics.
6. When using the Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat radiation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions or failure or accident arising out of the use of Renesas Electronics products beyond such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please ensure to implement safety measures to guard them against the possibility of bodily injury, injury or damage caused by fire, and social damage in the event of failure or malfunction of Renesas Electronics products, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures by your own responsibility as warranty for your products/system. Because the evaluation of microcomputer software alone is very difficult and not practical, please evaluate the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please investigate applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive carefully and sufficiently and use Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall not use Renesas Electronics products or technologies for (1) any purpose relating to the development, design, manufacture, use, stockpiling, etc., of weapons of mass destruction, such as nuclear weapons, chemical weapons, or biological weapons, or missiles (including unmanned aerial vehicles (UAVs)) for delivering such weapons, (2) any purpose relating to the development, design, manufacture, or use of conventional weapons, or (3) any other purpose of disturbing international peace and security, and you shall not sell, export, lease, transfer, or release Renesas Electronics products or technologies to any third party whether directly or indirectly with knowledge or reason to know that the third party or any other party will engage in the activities described above. When exporting, selling, transferring, etc., Renesas Electronics products or technologies, you shall comply with any applicable export control laws and regulations promulgated and administered by the governments of the countries asserting jurisdiction over the parties or transactions.
10. Please acknowledge and agree that you shall bear all the losses and damages which are incurred from the misuse or violation of the terms and conditions described in this document, including this notice, and hold Renesas Electronics harmless, if such misuse or violation results from your resale or making Renesas Electronics products available any third party.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.3.0-1 November 2016)

Trademark

- Trademarks and trademark symbols (® or ™) are omitted in the text of this manual.

How to Use This Manual

1. Purpose and Target Readers

This manual is designed to provide the user with an understanding of the application functions of the GRAPE (Graphical Application Environment) which uses the RGL.

This manual is written for application engineers who use the RGL.

The GRAPE application note uses the GRAPE framework providing the user an example of using a high-level API for the functions of the 2D/3D-Drawing engine, for the serial flash as well as for the Video Output and Video Input Interface of the RH850/D1L/D1M microcontrollers driver. Industry standard APIs (EGL, OpenVG) are not described in this manual.

The revision history summarizes the locations of revisions and additions. It does not list all revisions. Refer to the text of the manual for details.

Please refer to documents of drivers and hardware for a target system implementing RGL as necessary.

The following documents are related documents. Make sure to refer to the latest versions of these documents.

Document Type	Description	Document Title	Document No.
User's manual for Software	Description of RGL	Renesas Graphics Library User's Manual: Software	R01US0181ED0019
User's manual for Application	Description of GRAPE application note	Grape_app User Manual User's Manual: Software	This manual

2. Notation of Numbers and Symbols

This manual uses the following notation.

Binary	0bXXXXXXXX	(X=0 or 1)
Decimal	XXX	(X=0-9)
Hex	0xXXXXXXXX	(X=0-9,A-F)

3. List of Abbreviations and Acronyms

Abbreviation	Full Form
Capture device	A.k.a. video input device
Context	An internal state machine of the single framework
CPU	The microprocessor core of the MCU
Device	A SW abstraction of the HW or SW macro
Framebuffer	A region in the memory attached to a window that can be shown on the screen; a region in the memory holding the bitmap as the result of GPU rendering activities
GPU	The graphical processing unit HW macro of the MCU
HW	Hardware
Layer	A HW concept of the stackable visual area on the display
MCU	A RH850/D1x HW device
Pitch	(a.k.a. stride) Distance in pixels between two adjacent pixel rows of the framebuffer in the memory
RLE	TARGA run-length encoded image standard, for easy image compression, supported by the Sprite Engine macro of the D1x HW
Screen	A physical display surface; a SW abstraction of the attached physical display
Sprite	An image in the memory which can be registered with designated HW (or just SW if HW not present) for quicker access and manipulation
Surface	A concrete (i.e. physical) implementation of the window's area
SW	Software
Target	A platform (HW or SW) where the framework and application are intended to run
Texture	A binary image registered with the GPU driver that can be transformed and drawn to the framebuffer in a HW accelerated way
Unit	see *device*
VIN	Video input HW
VOUT	Video output HW
Window	A SW abstraction of the rectangular visual area that can be shown on the display

Table of Contents

1. Introduction

The grape_app provides the user a framework for quick and easy development of graphics software. It can be used for device promotion and proof of device features.

1.1 Framework Structure

The grape_app framework is divided into 2 parts (Figure 1-1):

- Grape_app's Demo Application part
- Grape_app's Generic Code part
- System Abstraction part

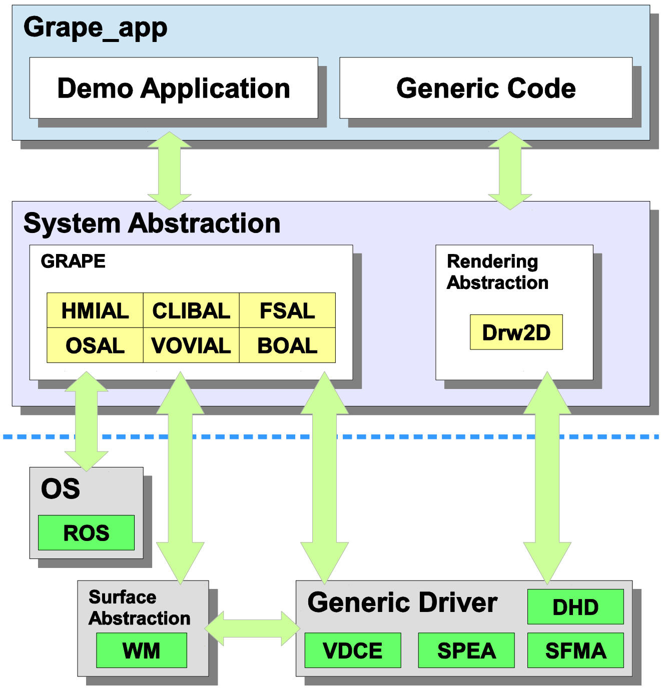

Figure 1-1: Grape_app framework structure

1.2 Demo application

All the application specific code is implemented at this layer. To keep the applications easy portable between different platforms, no generic, platform specific or device dependent functions are implemented in this part. The application can use the API functions in the system abstraction part as well as the API of the generic driver, which is not part of the grape_app framework.

1.2.1 Generic Code

In this part all the generic code for the grape_app are implemented, e.g. management of HMI elements, setup texture in the memory or multithreading. More details about the generic code are described in chapter 1.3.2.

1.2.2 System Abstraction

To better understand the system abstraction, it can be seen as consisting of two parts:

- The GRAPE API and
- the Rendering Abstraction

GRAPE API

There are 6 API layers in the Graphics Application Programming Environment (GRAPE) API part, which can be used by the application:

- **OSAL** (Operating System Abstraction Layer): this layer provides an adaption interface for different OS, the operating system specific functions are implemented at this layer, such as operations on thread and semaphore. In grape_app these functions are implemented by calling the API functions of ROS (Renesas Embedded Operating System). If the user is using his own operating system, all these functions have to be fitted at this layer accordingly.
- **VOVIAL** (Video Output/Video Input Abstraction Layer): this layer provides an interface between the video output driver, resp. the video input driver, and the applications which are using the video output and input macros. The implementation of this layer is also device dependent, since different D1x family devices have different video out- and inputs.
- **BOAL** (BOard setup Abstraction Layer): this layer provides an interface between the board support package’s initialization and de-initialization functions and the application using the board. The implementation of this layer is also device dependent, since different D1x family devices have different components that must be initialized.
- **CLIBAL** (CLIB Abstraction Layer): this layer provides an interface between some of the CLib functions (e.g. printf, sprintf) and the application. The underlying implementation is also OS and device dependent, since the required functions for realization can be implemented OS or board dependent.
- **FSAL** (File System Abstraction Layer): this layer provides an interface between an underlying file system and the applications, which access files.
- **HMIAL** (Human Machine Interface Abstraction Layer): this layer provides an interface between the connected device’s HMI elements (e.g. knob, buttons …) and an application relying on user interaction. The underlying implementation is also device dependent, since the availability of HMI elements depends on the actual device and its board support package.

Rendering Abstraction

In the rendering abstraction part lies the portable Drw2D rendering API. Drw2D makes use of the DHD driver.

1.3 Folder Structure

The grape_app folder has 3 sub folders:

- romfs
- src
- target

Figure 1-2 shows the folder structure of the grape_app in details.

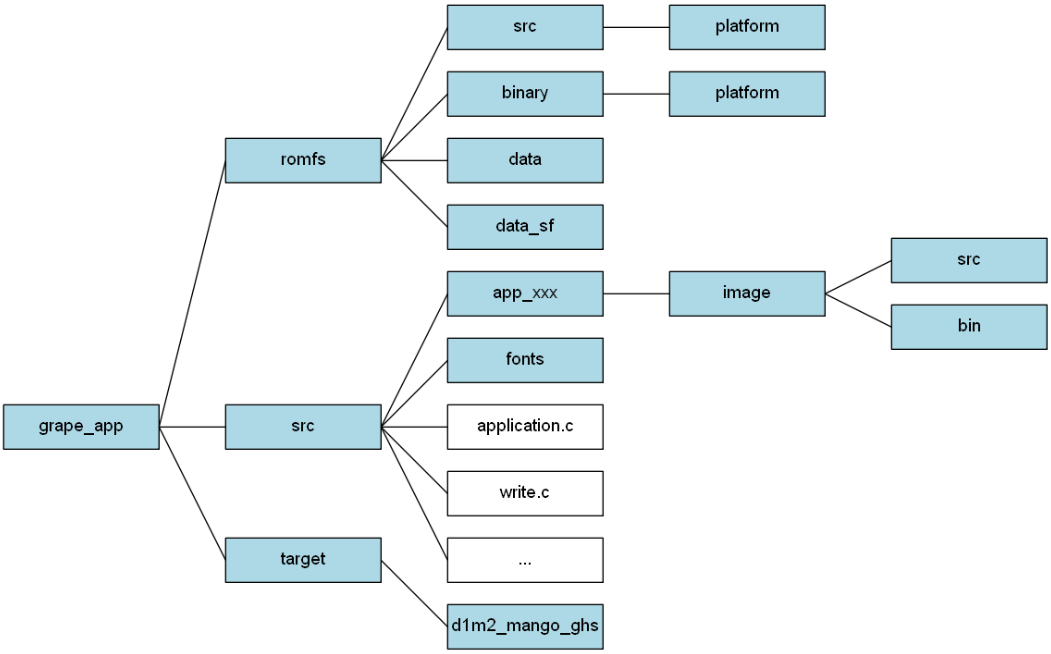

Figure 1-2: Grape_app folder structure

1.3.1 romfs folder

The images used in different applications are stored into 3 ROMFS files by executing the command: `make romfs`

Generally, there are two options to store the images used in the different applications:

- **iROM**: the image data is compiled as an C array, linked with grape_app and hereby placed in the internal ROM
- **Serial Flash**: the image data is written to a binary blob, which must later be written to the target device's flash memory manually

Whether a file will be added to the iROM or Serial Flash section of ROMFS depends on how it is referenced in "grape_app.mk". Files in a certain folder are added to the iROM section by adding the path to the `ROMFS_PATH` variable. To add the contents of a directory to the serial flash, its path must be added to the `ROMFS_SF_PATH` variable.

The images used in different applications are stored into 3 ROMFS files by executing the command: `make romfs`

- **data_flash_sf.srec**: the image data blob for the serial flash section in S-Record format.

- **data_flash_sf_to_0x***********: the image data blob for the serial flash section in raw format.

- **fs_data.c**: the image data for the iROM section and an array containing the start addresses in the target device's memory space.

The first two files are placed in the sub folder "romfs\binary\platform\[*platform name*]".

In the C file "fs_data.c" a constant data array of the images packed into the ROMFS files is defined. The name, size and start address of each image are stored in the array. The application will use this array to access the image data. The fs_data.c file is located in "romfs\src\platform\[*platform name*]"

The contents of the files irom.txt in the sub folder "romfs\data" and "flash.txt" in the sub folder "romfs\data_sf" are also stored in the ROMFS. The former one in the iROM section, the latter one in the serial flash section. At the start up of grape_app, the content of both files is read from the iROM and the flash memory to check if the file system is loaded successfully. For more details, see the locCheckRomFS and locCheckFlashFS function in the main.c file.

1.3.2 src folder

As described in chapter 1.1, the grape_app can be seen as divided into 3 parts, the source code in each part can be found in the corresponding sub folders as shown in Figure 1-3.

The generic function files are located directly under the src folder:

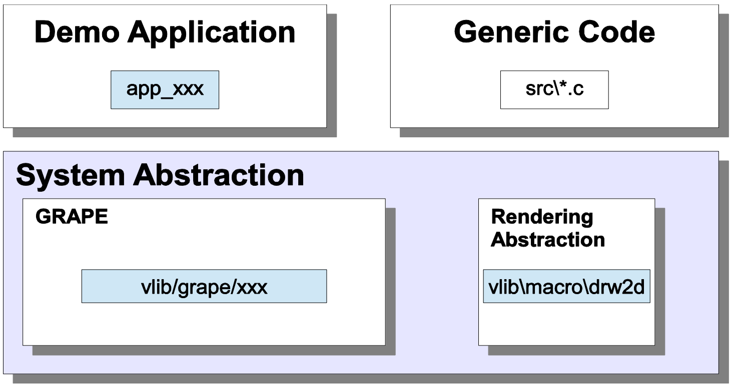

Figure 1-3: Grape_app framework and folder

- **application.c**: the application list is defined in this file, if the user wants to add a new application to the grape_app, the header file and a pointer to the standard structure of the new application have to be added into this file. More details will be introduced in chapter 2.
- **error.c**: contains the central error handling.
- **main.c**: contains the main function, from where the grape_app starts. It handles the command and background thread.
- **img.c**: the 2D texture utility functions are implemented in this file, e.g. prepare an image for drawing engine, remove an image from GPU memory.
- **img_drw2d.c**: the IMG format utility functions for Drw2D are implemented in this file, e.g. rendering an image in IMG to the frame buffer or generating an adequate R_DRW2D_Texture_t object.
- **write.c**: the write functions for bitmap fonts are implemented in this file, e.g. get the width and height of the font, write text at given position in the selected buffer.

1.3.3 target folder

In the target folder "target\[*platform name*]" the Makefile file is located, which should be used to build the project.

> *The folder "[platform name]" used in above chapters depends on the target platform the grape_app is running on.*

1.4 Work flow of grape_app

The grape_app starts with the initialization of the hardware, OS, application, etc. After the initialization the program runs into a main loop function, the application app_menu (see chapter 1.4.1) will be executed as the default application. Then, the program keeps checking if a command is sent by an application.

The flowchart of grape in Figure 1-4 shows the flowchart of the grape_app.

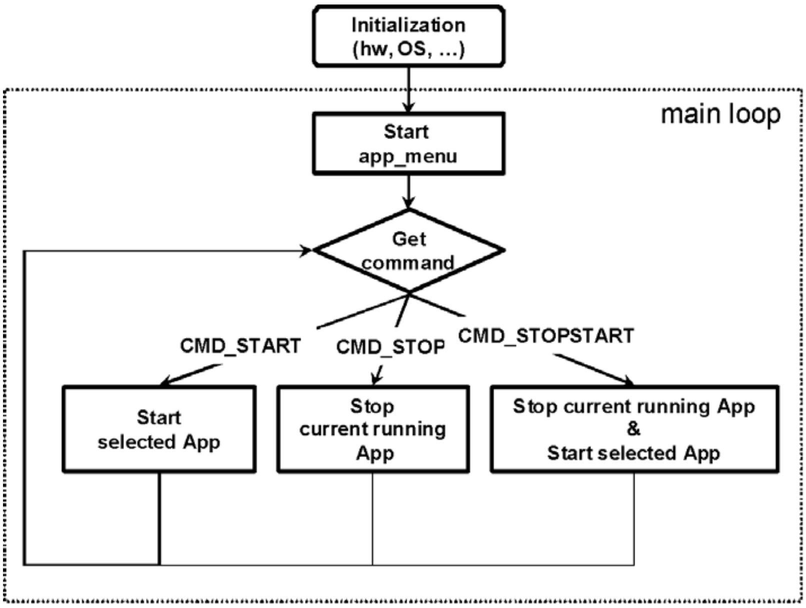

Figure 1-4: Flow chart of grape_app

1.4.1 The application app_menu

The application app_menu provides the user an interface to be able to select and run the available applications. After the app_menu has executed successfully there will be icons for each available application shown on the display. Using the knob on the application board the application can be selected and executed:

Rotate knob (left or right): the focus on the icon (the icon's size is pulsating) can be moved from one icon to another by rotating the knob.

Press knob: an icon having focus means that the corresponding application is selected, pressing the knob will start it and the app_menu will be terminated.

The started application is responsible for restarting the app_menu when it terminates.

2. How to build an application

This chapter describes creation and integration of a new application into the grape_app step by step. The topics covered are: where to start with a new application, what is necessary for an application and how it can be integrated into the current grape_app.

This chapter aims to give a practical example based on the demo application part (see chapter 1.2) to make the user able to build his own demo into grape_app. However, it does not offer a comprehensive description of all the source code of grape_app.

In this chapter an application named app_tutorial is built into grape_app step by step. The complete source code of the application can be found in the appendix of this document.

2.1 Preparation for a new application

In the new application app_tutorial we will draw two textures, and rotating the knob on the application board will change the color of one of the textures.

2.1.1 System environment

The required system environment is Windows including a Cygwin environment. Cygwin provides a Linux look and feel environment for Windows systems.

2.1.2 Makefiles customization

Makefile

The Makefile tells the compiler how the application will be compiled, e.g. use the pre-build library or not, define the ROMFS serial flash section start address in memory, some compiler option settings. The Makefile can be found in the folder:

"\grape_app\target\[*platform name*]"

There is also an application list in the Makefile, as shown below:

```
#
# Application
#

APP_MENU          = yes
APP_CLOCK         = yes
APP_SIMPLEMT      = yes
APP_TEST          = yes
APP_TUTORIAL      = yes
```

This list defines if an application demo will be included in the grape_app or not (yes: included, no: not included). The `APP_TUTORIAL = yes` is now added for the app_tutorial.

gfx_appnote.mk

In the application make file "grape_app.mk" (in folder "\grape_app") the source code search paths for all applications are defined, so that the compiler knows where to find the source code. This file is also included in the Makefile.

For app_tutorial the following lines are added in this file:

```
ifeq ($(APP_TUTORIAL), yes)
VLIB_CFLAGS += -DAPP_TUTORIAL
VLIB_INC+= $(VLIB_ROOT)/app/grape/grape_app/src/app_tutorial
VLIB_VPATH  += $(VLIB_ROOT)/app/grape/grape_app/src/app_tutorial
VLIB_VPATH  += $(VLIB_ROOT)/app/grape/grape_app/src/app_tutorial/image/src
ROMFS_PATH     += $(VLIB_ROOT)/app/grape/grape_app/src/app_tutorial/image/bin
endif
```

The app_tutorial and image folders defined above must also be created accordingly. "image\src" and "image\bin" will be created automatically, when converting the image data with the bitmap tool in the RGBench as described in chapter 2.1.3.

2.1.3 Creating a texture and an icon for the application

The freeware GIMP can be used for drawing the icon and the texture that is used by the app_tutorial.

The app_tutorial icon and texture

For each application in the grape_app an icon is needed, which will be shown on the screen by the app_menu. The icon is saved as a targa image file (.tga) without RLE-Compression, has the size 48x46 and the origin has to be set as top left.

The texture used in the app_tutorial has the size 120x19 and contains an alpha channel to show Drw2D alpha blending capabilities. Make sure to save the texture in a format that supports alpha channels (e.g. PNG, TGA).

The icon and the texture of the grape_app are shown in Figure 2-1 and Figure 2-2.

Figure 2-1: Icon for app_tutorial

Figure 2-2: Texture for app_tutorial

Creating binary and c file

The generated images have to be converted to binary and C files so that the application can use them. The bitmap tool in RGBench can go into action to convert the image. The “r_rgbench.exe” executable can be found in “vlib\app\util\d1mx_eco_system\p_rgbench\”.

In the RGBench main window, select bitmap tool to open the bitmap conversion dialog. The Bitmap tool dialog is shown in Figure 2-3.

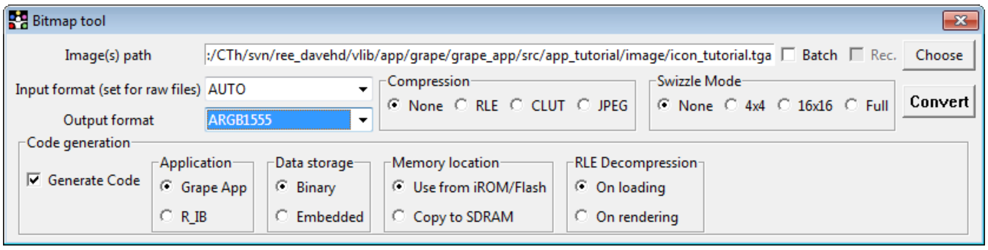

Figure 2-3: Bitmap Tool dialog

First, select the icon image that should be converted and integrated into grape_app. Click “Choose” to select the image file. The icon image (TGA file) is located in the folder “grape_app\src\app_test\image\”.

As “Output format”, select ARGB1555. One bit for the alpha channel is enough for the icon image, since there are only completely opaque and completely transparent pixels.

Since we also want to create code for the image’s integration into grape_app, make sure that “Generate code” is activated and “Grape App” is selected in the application frame.

Ensure that the configuration equates to the configuration shown in the image above and click “Convert”.

If successful, the Bitmap Tool will create two new folders in the “app_tutorial\image” folder:

- **bin**: Contains the binary file of the converted image data (BIN file).
- **src**: Contains a constant Img_t object describing the image (e.g. file name, resolution, pixel format, additional attributes …). A detailed description of Img_t can be found in the file “img_format.h”.
  The name of the constant is “Img_” concatenated with the converted image’s input filename (without suffix).

Now repeat the steps above for the texture image. But as “Output format”, select A8 this time. The Bitmap Tool will convert the image into a monochrome format with 8 bits per pixel.

2.2 Implementation of the application

After the preparation has been finished successfully, we can now start to implement the application.

2.2.1 Start point of the grape_app

The grape_app starts with the main function in the “main.c” file. This chapter shows what has to be initialized or configured before an application is being called.

main function

```
int main(void)
{
    FW_BOAL_LowLevelInit();
    FW_OSAL_InitOS();

    R_UTIL_DHD_Init(0);
    R_UTIL_DHD_Config((dhd_gpu_ptr_t)LOC_DHD_BASE, LOC_DHD_WORKING_BUFFER);

    FW_OSAL_ThreadCreate(locMainLoop, 0, locStackMainLoopThread,
                         LOC_MAINLOOP_STACKSIZE, LOC_MAINLOOP_PRIO);
    FW_OSAL_StartOS();
    FW_BOAL_LowLevelDeInit();  /* Depending on the OS, you may never get here */
    return 0;
}
```

First of all the `FW_BOAL_LowLevelInit` function is called, which initializes the BSP (Board Support Package) hardware, e.g. the CPU clock, clock setting for different domains on the target device, port setting for different macros.

Calls to `R_UTIL_DHD_Init` and `R_UTIL_DHD_Config` initialize DHD unit 0 and configure its base address and video memory size.

After the BSP and DHD have been initialized the OS is initialized by calling the function `FW_OSAL_InitOS`. A thread named `MainLoopThread` will then be created in which the following functions are called:

- **FW_OSAL_ThreadCreate**: Creates a background thread that measures the CPU load.
- **FW_OSAL_SemaCreate**: Creates a semaphore that is used to block the main thread until a HMI event occurs.
- **FW_CLIBAL_Init**: Initializes the CLib abstraction layer.
- **FW_BOAL_OsLevelInit**: Initializes the board for a running OS.
- **FW_HMIAL_Init**: Initializes the HMI control elements (e.g. setting up IRQs and IRQ handlers).
- **FW_VOVIAL_Init**: Initializes the video in/out layer for the display `LOC_DISPLAY_NAME`.
- **FW_FSAL_Init**: Initializes the ROM file system with the `R_ROMFS_Data_t` object from “fs_data.c”.
- **locCheck*FS**: Checks if the ROM and serial flash sections of the ROM file system are successfully loaded.
- **FW_HMIAL_SetControl**: Registers the HMI call backs of the app_menu.

Then, the `MainLoopThread` will enter the main loop, which processes the HMI user inputs (e.g. starting or stopping an app). If the main loop terminates, the used abstraction layers (VOVIAL, HMIAL, BOAL and CLIBAL) are all being de-initialized.

At the end of the `main` function, the OS will be started by calling the `FW_OSAL_StartOS` function.

After the OS has terminated (if at all) a call to `FW_BOAL_LowLevelDeInit` will de-initialize the board.

2.2.2 Initialization and Deinitialization

For the application app_tutorial there are 3 files to be created in the folder

“\grape_app\src\app_tutorial”:

- **app_tutorial_image.h**: In this file, the icon and the texture used by app_tutorial (`Img_icon_tutorial`, `Img_tutorial_text`) is declared as external.
- **app_tutorial.h**: This file is used as the application interface, so that the application can be included in the grape_app.

- **app_tutorial.c**: All the application dependent functions are implemented in this file.

app_tutorial_image.h

In this header file, the images created for the application are included:

```
extern const Img_t Img_icon_tutorial;
extern const Img_t Img_tutorial_text;
```

The details about `Img_icon_tutorial` and `Img_tutorial_text` can be found in “icon_tutorial.c” and “tutorial_text.c”.

> *The Bitmap Tool generates the global variable names of the images depending on their file name. Thus, it is a good manner to use preferably unique names, for example by attaching the applications name to the images' file names. This will help, preventing name conflicts.*

app_tutorial.h

A constant instance of the structure App_t (see file “application.h” for details) is declared in this header file:

```
extern const App_t AppTutorial;
```

Each application has to define such a constant, so that the application can be supported by the grape_app framework. This header file must be included in the “application.c” file:

```
#ifdef APP_TUTORIAL
#include "app_tutorial.h"
#endif
```

The flag APP_TUTORIAL is set in the “grape_app.mk” (refer chapter Makefiles customization2.1.2).

app_tutorial.c

All the app_tutorial dependent functions are implemented in this file. We will start with the application structure `App_t`, which is necessary for each application in grape_app. A constant instance of the structure is defined as below:

```
/*******************************************************
  Variable: AppTutorial
*/

const App_t AppTutorial = {
    &IconTutorial,
    "Tutorial Application",
    "Simple rendering",
    &locAppTutorialControl,
    0,
    locInit,
    locDeInit
};
```

This constant has also to be added into the application list, which is defined in the file “application.c”:

```
const App_t * const AppList[] = {
......
#ifdef APP_TUTORIAL
    &AppTutorial,
#endif
......
};
```

The first member of AppTutorial is the `Img_icon_tutorial`, which is a pointer to the application icon as described in chapter 2.1.3.

The 2nd and 3rd member of AppTutorial are the name and the description of the app_tutorial. The name is shown in the heading line of grape_app, when the app_tutorial icon is selected.

locAppTestControl

The member `locAppTutorialControl` defines a list of callback functions for the HMI control elements that are available on the application board:

```
/***********************************************************
  Variable: locAppTutorialControl

  Control function structure.

  List of callback functions for each control element.
*/
static const HmiControl_t locAppTutorialControl = {
    locKnobPress,    /* KNOB press callback */
    0,               /* KNOB release callback  */
    locKnobRotation, /* KNOB right callback */
    locKnobRotation, /* KNOB left callback  */
    0,               /* BUTTON up press callback */
    0,               /* BUTTON up release callback */
    0,               /* BUTTON down press callback  */
    0,               /* BUTTON down release callback */
    0,               /* BUTTON middle press callback */
    0,               /* BUTTON middle release callback */
    0,               /* BUTTON left press callback */
    0,               /* BUTTON left release callback */
    0,               /* BUTTON right press callback */
    0                /* BUTTON right release callback */
};
```

To control the application by using the HMI elements, the corresponding HMI callback function can be implemented as defined in `fw_hmial_Control_t` (refer to the file “fw_hmial_api.h” for more details).

For the app_tutorial there are 2 HMI callback functions implemented: `locKnobPress` and `locKnobRotation`, they will be introduced in chapter 2.2.4.

locInit

In the initialization function locInit, the following operations are performed:

- Create a surface for the application.
- Initialize the Drw2D API and a Drw2D unit including DHD graphics engine.
- Initialize the `Img_t` structure object for the `Img_tutorial_text` texture.
- Initialize a `r_drw2d_Texture_t` structure object for the texture.
- Create a semaphore that blocks the render thread until the VBlank interrupt occurs.

- Register the callback function for the VBlank interrupt.
- Create the app's main thread.

```
static void locInit(void) {
    r_drw2d_Error_t err;

    locSemaArm    = 0;
    locQuit       = 0;

    locSurface = FW_VOVIAL_CreateSurface(
            LOC_VOVIAL_UNIT,
            FW_VOVIAL_ARGB8888,
            0,
            0,
            0,
            LOC_DISPLAY_STRIDE,
            LOC_DISPLAY_WIDTH,
            LOC_DISPLAY_HEIGHT,
            0xff,
            2,
            0);
    if(locSurface == 0)
    {
        Error(ERR_VOVIAL);
    }

    err = R_DRW2D_Init();
    if(err != R_DRW2D_ERR_OK)
    {
        Error(ERR_DRW2D);
    }

    err = R_DRW2D_Open(0, R_DRW2D_UNIT_DHD0, &locDHDDev, &locDrw2dDev);
    if(err != R_DRW2D_ERR_OK)
    {
        Error(ERR_DRW2D);
    }

    locOldText = Init2DImg((Img_t*) &Img_tutorial_text);

    IMG_GetTexture(&Img_tutorial_text, &locTexture, R_DRW2D_TEX_NONE);

    FW_OSAL_SemaCreate(&locSema);

    FW_VOVIAL_RegisterVBlankIsr(locSurface, locVBlank);

    locAppTutorialThreadId = FW_OSAL_ThreadCreate(locAppTestThread, 0,
                                            locStackAppTutorialThread,
                                            LOC_APP_TUTORIAL_THREAD_STACKSIZE,
                                            LOC_APP_TUTORIAL_THREAD_PRIO);
}
```

locVBlank

The `locVBlank` function is the callback function of the VBlank interrupt. In this function, the render function will be triggered by means of a semaphore, each time the VBlank interrupt occurs. The render function `locRender` is described in chapter 2.2.3.

The semaphore is guarded by an if-statement to prevent an unlimited upcounting of the semaphore’s internal value. This behavior occurs when the execution of the rendering loop takes longer than the time slot between to VBlank IRQs and would lead to a loss of synchronization between the rendering loop and the VBlank.

Note that `locSemaArm` always has to be set to 1 before calling `FW_OSAL_SemaWait` on the semaphore `locSema` (see `locAppTestThread()`).

```
static void locVBlank()
{
    if(locSemaArm == 1)
    {
        FW_OSAL_SemaSignal(&locSema);
    }
}
```

locDeInit

The `locDeInit` function will be called each time when the application is terminated. The resources allocated for the application will be freed:

- Join the app's main thread.
- Clean up the `Img_t` structure object of the texture.
- Destroy the semaphore.
- Free the Drw2D unit and all global Drw2D resources.
- Disable the created surface.

```
static void locDeInit(void)
{
    /* Signal thread to quit */
    locQuit = 1;

    FW_OSAL_ThreadJoin(locAppTutorialThreadId, &locRetVal);

    Deinit2DImg((Img_t*) &Img_tutorial_text, locOldText);

    FW_OSAL_SemaDestroy(&locSema);

    R_DRW2D_Close(locDrw2dDev);
    R_DRW2D_Exit();

    FW_VOVIAL_DeleteSurface(locSurface);
}
```

2.2.3 Rendering functions

In the render function locRender, the texture is rendered two times. The first time by multiplicating the texture’s color channels with a blue color (`0xff0033ff`). The second time by multiplicating with the color given by `loc_Color`. Before rendering the texture, the background color is set to black.

locRender

```
static void locRender()
{
    r_drw2d_Point_t p;
    r_drw2d_FixedP_t rad;
    r_drw2d_Rect_t rect;
    r_drw2d_Point_t tri[3];
    uint8_t eFlags;

    /* Clear entire screen */
    R_DRW2D_FramebufferClear(locDrw2dDev);

    /* Setup a modulate effect for the texture's color channels */
    effect.Name = R_DRW2D_EFFECT_MODULATE;
    effect.Args[0].Source = R_DRW2D_EFFECT_SOURCE_TEXTURE_UNIT;
    effect.Args[0].Param.Color.Source.TextureUnit = 0;
    effect.Args[0].Param.Color.Operand = R_DRW2D_EFFECT_COLOR_OPERAND_RGBA;
    effect.Args[1].Source = R_DRW2D_EFFECT_SOURCE_CONSTANT_COLOR;
    effect.Args[1].Param.Color.Source.ConstantColor = 0xff0033ff;
    effect.Args[1].Param.Color.Operand = R_DRW2D_EFFECT_COLOR_OPERAND_RGBA;

    /* Setup the source rectangle for the whole texture */
    srcRect.Pos.X = R_DRW2D_2X(0);
    srcRect.Pos.Y = R_DRW2D_2X(0);
    srcRect.Size.Width = R_DRW2D_2X(Img_tutorial_text.Width);
    srcRect.Size.Height = R_DRW2D_2X(Img_tutorial_text.Height);

    /* Setup the destination rectangle */
    dstRect.Pos.X = R_DRW2D_2X(LOC_TEXT_POSX);
    dstRect.Pos.Y = R_DRW2D_2X(LOC_TEXT_POSY);
    dstRect.Size.Width = R_DRW2D_2X(Img_tutorial_text.Width);
    dstRect.Size.Height = R_DRW2D_2X(Img_tutorial_text.Height);

    /* Turn of transformation */
    R_DRW2D_CtxTransformMode(locDrw2dDev, R_DRW2D_TRANSFORM_NONE);

    /* Enable effect stage */
    R_DRW2D_CtxEffectsSet(locDrw2dDev, &effect, 1);

    /* Render the upper left texture */
    R_DRW2D_TextureBlit(locDrw2dDev, &srcRect, &dstRect);

    effect.Args[1].Param.Color.Source.ConstantColor = loc_Color;
    dstRect.Pos.X = R_DRW2D_2X(  LOC_DISPLAY_WIDTH
                               - LOC_TEXT_POSX
                               - Img_tutorial_text.Width);
    dstRect.Pos.Y = R_DRW2D_2X(  LOC_DISPLAY_HEIGHT
                               - LOC_TEXT_POSY
                               - Img_tutorial_text.Height);

    /* Update the effect stage */
    R_DRW2D_CtxEffectsUpdate(locDrw2dDev, R_DRW2D_EFFECT_MODULATE, 0, 2, effect.Args);

    /* Render the lower right texture*/
    R_DRW2D_TextureBlit(locDrw2dDev, &srcRect, &dstRect);

    /* Disable the effect stage */
    R_DRW2D_CtxEffectsDelete(locDrw2dDev);
```

```
    /* Turn on transformations */
    R_DRW2D_CtxTransformMode(locDrw2dDev, R_DRW2D_TRANSFORM_2D); }
```

2.2.4 HMI callback functions

As described in chapter 2.2.2, we are using two HMI callback functions to control the application:

- **locKnobPress**: this function is called, if the knob on the application board is pressed. We will use this function to stop the application and go back to the grape_app menu.
- **locKnobRotation**: this function is called, if the knob on the application board is rotated. The color stored in `loc_Color`, which is used for the modulate effect in the second render operation, will be modified.

The implementation of these HMI callback functions are shown below.

locKnobPress

```
/***********************************************************
  Function: locKnobPress

  KNOB pressed callback function.
  The application returns to the menu.

  Parameters:
  Void

  Returns:
  Void
*/
static void locKnobPress(void) {
    Cmd_t cmd;

    cmd.Cmd  = CMD_STOPSTART;
    cmd.Par1 = 0;
    cmd.Par2 = (AppNum - 1);
    CmdSend(&cmd);
}
```

The `Cmd_t` is the command type structure that is used to control the main loop of the program by different threads. The `Cmd_t` has three members:

- **Par1**: 1. command specific parameter. This parameter is reserved for other usage and set to 0 in all applications of grape_app.
- **Par2**: 2. command specific parameter. It is used to store the position in the application list AppList[] (defined in "application.c") of the application to be started .
- **Cmd**: Command identifier, that is sent to the main loop function.

The setting below

```
cmd.Cmd = CMD_STOPSTART;
cmd.Par1 = 0;
cmd.Par2 = (AppNum - 1);
```

allows that app_tutorial can be stopped and app_menu can be started by pressing the knob. `(AppNum - 1)` is the position of app_menu in the `AppList[]`, since app_menu is defined as the last one in the application list.

The `cmd.Cmd` is sent to the main loop by the function CmdSend.

> *Do not change the position of the app_menu in the AppList[].*

locKnobRotation

```
/*********************************************************
  Function: locKnobRotation

  KNOB right/left callback function.

  The color of forms changed.

  Parameters:
  void

  Returns:
  void
*/
static void locKnobRotation(void)
{
    loc_Color = (r_drw2d_Color_t) locRndColor();
}
```

The color used for the modulate effect in the second render operation is changed randomly, if the knob is rotated. For generating the random color, a pseudo random number generation function is implemented as below:

```
/**********************************************************
  Function: locRndColor

  Returns a pseudo random color for the forms.
*/
static uint32_t locRndColor(void) {
    uint32_t x;

    x = ((locRndSeed>>16) + 3715436908ul ) ^ 0x1fd8dae7;
    locRndSeed += x;

    return locRndSeed;
}
```

2.3 Application test

2.3.1 Packing the icon and texture images into ROMFS

The binary files of the icon and the texture have to be packed into the ROMFS file, which will be loaded into the flash memory on the target device at program start up.

In the file "grape_app.mk" for the app_tutorial the path "$(VLIB_ROOT)/app/grape/grape_app/src/app_tutorial/image/bin" is added to `ROMFS_PATH` (see chapter 2.1.2).

The `ROMFS_PATH` allows the ROMFS scripts to find the image data that ought to be packed into the ROMFS files. The following command can be called in a Cygwin console within the path "/grape_app/target/[*platform name*]" to build the ROMFS file:

```
make romfs
```

After executing make romfs, two ROMFS binary files have been created and the file fs_data.c located in \grape_app\romfs\src\platform\[platform name]\ has also been updated (see chapter 1.3.1).

Alternative: Using the serial flash

Alternatively, the icon image and the texture can be packed into the serial flash instead of the internal ROM to save memory. This is indispensable for huge mounds of data.

Therefore, the path “$(VLIB_ROOT)/app/grape/grape_app/src/app_tutorial/image/bin” has to be added to `ROMFS_SF_PATH` instead.

After executing make romfs, the images' binary data will be integrated in the files “data_flash_sf*” (see chapter 1.3.1 for more details). In order to make the binary data available at run time, it must be flashed to the target's serial flash memory. This can be done using the Renesas eFLASHLOAD tool, as depicted in chapter 6 of this document.

> *Note that serial flash is slower than the internal ROM. Rendering many textures directly from serial flash will have a bad impact on the performance.*

> *If you want to split your image files between serial flash and internal ROM, you can create two image folders where you convert the images with the Bitmap Tool. One for the serial flash images and one for the internal ROM images. Then, add the serial flash image path via `ROMFS_SF_PATH` and add the flash ROM image path via `ROMFS_PATH`.*

2.3.2 Compiling and running the grape_app

Grape_app can be compiled in a Cygwin console by calling the command `make` in the folder:

“\gfx_appnote\target\[*platform name*]”

After grape_app has been compiled successfully, an out file is generated: “grape_app.out”. This out file can be opened using the GHS MULTI Debugger.

After the program code and the ROMFS file is downloaded onto the device completely, grape_app will start. App_menu will run automatically first and the icons of the available applications will be shown on the screen. To run app_tutorial, rotate the knob to move the focus on the app_tutorial icon and press the knob.

3. D1Mx: Additional features

This chapter depicts the two applications that come with grape_app, that are available for the D1Mx platforms only. The application’s implementations are not covered completely in the following section. Instead, the particular features of the applications are shown, since all grape_app application are similar to the structure depicted in chapter 2.2.

3.1 App_clock

The app_clock application shows some more of the capabilities of Drw2D.

The application presents a clock with hands for seconds, minutes and hours as depicted in Figure 3-1. The time can be adjusted by using the board's knob. To move the hands by the second, the application makes use of the internal timer.

Figure 3-1: Screenshot of app_clock

3.1.1 Implementation

Background image

The application renders a background image that shows a clock without hands. To save memory, the background image is stored with RLE compression. This can be seen in the image's `Img_t` structure (the `IMG_ATTRIBUTE_RLE_COMPRESSED` flag is set), which can be found in "clocktrain.c".

However, the image is rendered by a call to IMG_Blit:

```
IMG_Blit(locDrw2dDev, &Img_clocktrain, 0, 0);
```

The compression issue is handled completely transparent on this level.

`IMG_Blit` internally uses Drw2D's capability to decompress RLE images on rendering by setting the `R_DRW2D_TEX_RLE` flag for the texture.

Updating the hands' positions

The clock hands' positions have to be updated for every frame, according to the time that actually passed by since the the last frame. The application measures the time difference between two frames in the main render thread as shown below:

```
static void *locRenderThread(void *Arg)
{
    uint32_t i=0;
    uint32_t time = FW_OSAL_TimeGet();

    while (!locQuit)
    {
        locSetRenderTarget(locSurface);

        locUpdateClock(FW_OSAL_TimeGet()-time);
        time = FW_OSAL_TimeGet();

        R_DRW2D_CtxBlendMode(locDrw2dDev, R_DRW2D_BLENDMODE_SRC);
        IMG_Blit(locDrw2dDev, &Img_clocktrain, 0, 0);

        locDrawNeedles(locDrw2dDev);

        R_DRW2D_GpuFinish(locDrw2dDev, R_DRW2D_FINISH_WAIT);
        FW_VOVIAL_SwapFrameBuffer(locSurface, i);
        ++i;
    }
    FW_OSAL_ThreadExit((void*)42);

    return 0;
}
```

The `FW_OSAL_TimeGet` function returns the value of an internal timer accurate to the millisecond. The difference between the current time and the time stored during the last frame is then used as a parameter for `locUpdateClock`. This function stores the time shown by the clock in a static variable, which is updated by the measured difference.

This way, the clock depends on this static variable instead of depending on the internal timer and the time can be set by the user.

Drawing the clock's hands

The clock's hands are drawn using Drw2D's line and ellipse drawing features in `locDrawNeedles` which is shown below:

```
static void locDrawNeedles(r_drw2d_Device_t  Dev)
{
    uint32_t i;

    R_DRW2D_ContextSelect(locDrw2dDev, locOutlineShadow_c);
    for (I = 0; I < _NEEDLE_NBR; i++)
    {
       locNeedleCalc(i, 4, 2);

       locNeedleLineStyle.Width = locNeedleWidthTop_s + R_DRW2D_2X(1);
       R_DRW2D_CtxLineStyle(locDrw2dDev, &locShadowLineStyle);
       locNeedleVectorDraw(locDrw2dDev, i);
    }

    R_DRW2D_ContextSelect(locDrw2dDev, locOutlineDefault_c);
    for (I = 0; I < _NEEDLE_NBR; i++)
    {
       locNeedleCalc(i, 0, 0);

       locNeedleLineStyle.Width = locNeedleWidthTop_s;
       R_DRW2D_CtxLineStyle(locDrw2dDev, &locNeedleLineStyle);
       R_DRW2D_CtxFgColor(Dev, 0xFF000000 | locNeedle[i].MainColor);
       locNeedleVectorDraw(locDrw2dDev, i);
    }

    /* drawing clock center */
    R_DRW2D_CtxFgColor(Dev, 0xFF000000);
    R_DRW2D_DrawEllipse(Dev, locNeedleCenterCoord_s, R_DRW2D_2X(2), R_DRW2D_2X(2));

    R_DRW2D_ContextSelect(locDrw2dDev, 0);
}
```

The hands are drawn in two steps: First, the shadow of the hands is drawn by selecting the `locOutlineShadow_c Drw2D` context. This will set the foreground color to a translucent grey.The hands are also moved a little bit to the lower right.

Afterwards, the hands are drawn in the `locOutlineDefault_c` context.

The function makes use of Drw2D's capability to draw lines and other shapes like ellipses. Drw2D allows to set various parameters for the drawing style. For example, the lines are drawn with a certain width, which is set by the calls in the following lines:

```
    locNeedleLineStyle.Width = locNeedleWidthTop_s + R_DRW2D_2X(1);
    R_DRW2D_CtxLineStyle(locDrw2dDev, &locShadowLineStyle);
```

3.2 App_simplemt

The app_simplemt shows Drw2D's, resp. the GPU's multithreading capabilities. A screenshot of the demo can be seen in Figure 3-2.

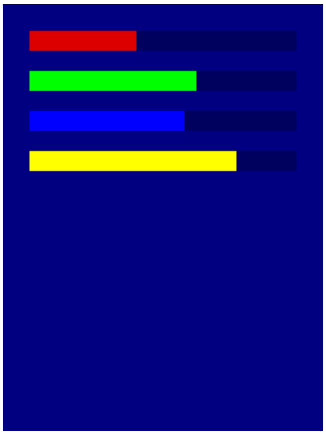

Figure 3-2: Screenshot of app_simplemt

To do so, the application creates four threads of which each draws a horizontal bar. The bar widens gradually with each pass of the thread's internal loop. Rotating the board's knob will change the threads' internal delays so that the drawing rate of the colored bar changes.

3.2.1 Implementation

Creating the threads

The rendering threads are initialized in `locThreadsInit`.

The threads will render to a common frame buffer that is created as below:

```
locDrawingSurface = FW_VOVIAL_CreateSurface(
            LOC_VOVIAL_UNIT,
            FW_VOVIAL_ARGB8888,
            20,
            20,
            1,
            LOC_DISPLAY_STRIDE,
            LOC_DISPLAY_WIDTH-40,
            LOC_DISPLAY_HEIGHT-40,
            0xff,
            1,
            0);
```

Note that the call creates a single-buffered surface. A double buffered surface and a vertical synchronization would synchronize the threads internal loops, which is not desired for this demo.

Before creating the threads, a `locThreadData_t` object has to be filled, containing the needed data for each thread. This way, every thread get a pointer to the frame buffer, a color for the bar to be drawn, an index number and a randomly choosen sleep time. The latter is needed to create random delays in the threads' internal rendering loop.

Furthermore, each thread needs a chunk of memory for its stack, which is allocated by a `R_CDI_ALLOC` call:

```
locThreadData[i].Stack = (void*) R_CDI_Alloc(&loc_lRAM_heap, LOC_STACK_SIZE);
```

The call allocates a memory chunk with a size of `LOC_STACK_SIZE` bytes in the local RAM.

Finally the threads are created by a call to the appropriate function of OSAL:

```
locThreadData[i].Thread = FW_OSAL_ThreadCreate(locDrawingThread,
                   &locThreadData[i], locThreadData[i].Stack, LOC_STACK_SIZE, 36);
```

`locDrawingThread` is the thread's entry point. The following parameters are the thread's working data, the pointer to its stack, the size of its stack and at last its priority.

The drawing thread

The drawing threads' functionality is implemented in locDrawingThread, which is also the thread's entry point for the `FW_OSAL_ThreadCreate` call.

Note that each thread creates its own Drw2D device handle by calling `R_DRW2D_Open` in the functon. A Drw2D device handle must not be used from different threads concurrently.

The drawing thread main loop is given below:

```
while(1)
    {
        /* Draw half transparent full bar */
        R_DRW2D_CtxFgColor(dev, 0x80808080);
        bar.Size.Width = R_DRW2D_2X(max_barwidth);
        R_DRW2D_DrawRect(dev, &bar);

        /* Draw growing bar */
        bar.Size.Width = R_DRW2D_2X(barwidth);
        R_DRW2D_CtxFgColor(dev, threadData->BarColor);
        R_DRW2D_DrawRect(dev, &bar);

        barwidth = (barwidth + 1) % max_barwidth;

        R_DRW2D_GpuFinish(dev, R_DRW2D_FINISH_NOWAIT);

        /* Thread goes to Sleep */
        if(threadData->Sleep)
        {
          FW_OSAL_ThreadSleep(threadData->Sleep);
        }

        R_DRW2D_GpuFinish(dev, R_DRW2D_FINISH_WAIT);

        /* deinit device and Thread if necessary */
        if(locEndThreads)
        {
            R_DRW2D_Close(dev);
            FW_OSAL_ThreadExit(0);
        }
    }
```

At the beginning of the loop, Drw2D is used to draw a half transparent bar with a greyish color. On top of that half transparent bar, the thread draws an opaque bar in its `BarColor` (which is a part of the `locThreadData_t` data). The width of the opaque bar starts with 1 and increases with every pass of the loop.

A call to `R_DRW2D_GpuFinish`, forces Drw2D and the GPU to finish all pending rendering operations. Note that the function is called using the `R_DRW2D_FINISH_NOWAIT` to enable a non-blocking behavior of the `R_DRW2D_GpuFinish` call. That means, the function will return immediately. This is reasonable here, because the thread will sleep afterwards by a call to `FW_OSAL_ThreadSleep`.

After the thread was woken up, another call to `R_DRW2D_GpuFinish` makes sure, that the pending rendering operations are finished by setting the `R_DRW2D_FINISH_WAIT` flag. This will cause the function to block until Drw2D and the GPU have finished the pending operations.

The threads will quit, when the global variable `locEndThreads` is set to 1. The Drw2D device handle will be closed and thread will exit by an OSAL call to `FW_OSAL_ThreadExit`.

3.3 App_drw2dcpu

This app is implemented using the Drw2D function subset that is also available in the Drw2D pure software implementation, which is available for the D1L2 board. The app shows the compatibility between the pure software implementation and the GPU based Drw2D implementation.

For more information about the app and its implementation details, see chapter 4.2.

4. D1L2: Additional features

This chapter depicts the two applications that come with grape_app, that are available for the D1L2 platform only. The application's implementations are not covered completely in the following section. This chapter rather focuses on particular features of the applications, since all grape_app application are similar to the structure depicted in chapter 2.2.

4.1 App_tripcomp

The app_tripcomp demo uses several features of the D1L2 board to implement a simple trip computer example, as shown in Figure 4-1.

Figure 4-1: Screenshot of app_tripcomp

The screen shows a background image, overlayn by several menu elements that can be selected by rotating the board's knob. The selected elements will be highlighted using a bluish color. The fan item can be selected for input by pressing the knob when the element is highlighted. The color will change to orange and the fan's speed can be controlled by rotating the knob.

Additional to the menu elements, the trip computer features a fuel gauge at the lower right. The gauge will slowly go lower and show a blinking warning sign when on a critical level. The fuel level will be reset to the highest level, after the hand has reached the lowest position.

The top screen shows a status bar for displaying information like the current date and tool tips for the selected menu elements. The bar below shows weather information by means of a symbol, the temperature and a short description.

4.1.1 TP GUI Implementation

To deal with the menu elements and other GUI elements on the screen in a uniform way, app_tripcomp uses a small framework called TP GUI.

The framework takes care of managing the GUI elements, showing the elements using the VOVIAL framework, highlighting and animating the elements, selecting elements and processing user inputs. The framework uses a callback function based approach to deal with events like VBlank and user inputs.

The TP GUI's source code can be found in "app_tripcomp_menu.h" and "app_tripcomp_menu.c".

TP GUI elements

The frameworks central data structure is the `tp_menu_Element_t` struct. A detailed description of the structure can be found in "app_tripcomp_menu.h". The structure's definition is shown in the following code:

```
struct tp_menu_Element_t
{
    tp_menu_Element_t      *Next;
    fw_vovial_Sprite_t     *SpriteConf;
    uint32_t                AddressNormal;
    uint32_t                AddressHighlighted;
    uint32_t               *AddressAnimation;
    uint32_t                AnimationFrameCount;
    uint32_t                AnimationCurrFrame;
    uint32_t                AnimationSpeed;
    int32_t                 IsFocusable;
    TP_OnFocus              OnFocus;
    TP_OnVBlank             OnVBlank;
    TP_OnInput              OnInput;
    char                   *Tip;
};
```

The structure is used as a wrapper around VOVIAL's `fw_vovial_Sprite_t` structure, which represents a single sprite, to add additional features needed in a simple GUI. The structure also allows to create a linked list of `tp_menu_Element_t` objects by means of a pointer to an object of the same structure (`Next`).

To implement highlighting and animation effects the structure has members to store pointers to image data and the current animation state.

Furthermore, the structure holds function pointers which are used as callback functions for certain events:

- **OnFocus**
  Called if the element gains or looses focus.
- **OnVBlank**
  Called if a VBlank occurrs.
- **OnInput**
  Called if user input is available for the element.

Creating GUI elements

When adding an element to TP GUI via a `TP_GUI_MenuAddElement` function call, the framework uses the element's `SpriteConf` parameter for subsequent call to the VOVIAL framework to create a new sprite:

```
FW_VOVIAL_CreateSprite(Element->SpriteConf);
```

VOVIAL then creates a new sprite. The surface used for the new sprite is already stored in the `fw_vovial_Sprite_t` object referenced by `SpriteConf`.

Since all sprites are initially disabled (invisible), the sprite has to be enabled by a call to `FW_VOVIAL_EnableSprite` to show it on the surface. This is done in `TP_GUI_Show` or `TP_GUI_MenuEnableElement`:

```
void TP_GUI_MenuEnableElement(tp_menu_Element_t *e)
{
    FW_VOVIAL_EnableSprite(e->SpriteConf);
}
```

Highlighting GUI elements

Highlightin of elements is implemented by switching a sprites image data pointer to the address of its highlighted counterpart. This can be done by calling `TP_GUI_MenuHighlightElement`. However, the main functionality is implemented in the function below:

```
static void locSpriteHighlight(tp_menu_Element_t *Element)
{
    FW_VOVIAL_SetSpriteBuffer(Element->SpriteConf,
                              (void*) Element->AddressHighlighted);
}
```

A call to `FW_VOVIAL_SetSpriteBuffer` sets the highlighted image data buffer for the given sprite.

Animating GUI elements

Animated elements have an array of image data pointer for every frame, referenced by the `AddressAnimation` member. On a vertical blank (VBlank) event, the framework calls the following function for animated elements:

```
static void locAnimate(tp_menu_Element_t *Element)
{
    static uint32_t Time = 0;

    if(0 != Element->AnimationSpeed)
    {
        Time = FW_OSAL_TimeGet();

        Element->AnimationCurrFrame =   (Time / Element->AnimationSpeed)
                                      % Element->AnimationFrameCount;

        FW_VOVIAL_SetSpriteBuffer(Element->SpriteConf,
                    (void*) Element->AddressAnimation[Element->AnimationCurrFrame]);
    }
}
```

The function selects the current animation frame depending on the animation speed and the current time (retrieved by `FW_OSAL_TimeGet`), to animate the element independent of the current overall framerate. Then, the sprite's image data buffer is changed to the current animation frame by calling `FW_VOVIAL_SetSpriteBuffer`.

Moving GUI elements

GUI elements can be moved by calling `TP_GUI_MenuMoveElement`. The function's code is shown below:

```
void TP_GUI_MenuMoveElement(tp_menu_Element_t *Element, uint32_t PosX,  uint32_t PosY,
uint32_t PosZ)
{
    FW_VOVIAL_MoveSprite(Element->SpriteConf, PosX, PosY, PosZ);
}
```

The function calls the underlying VOVIAL function to move the sprite to the given position. `PosZ` specifies the sprites Z postion within its surface. The function is used to generate the gas gauge hand's moving animation (see 4.1.2).

4.1.2 Trip Computer Implementation

To implement the trip computer example, the demo makes use of several D1L2 features. Therefore, the demo builds up the demo screen out of four layers. The four layers are depicted in Figure 4-2.

From back to front the layers are:

- **RLE layer**
  The RLE layer shows a RLE compressed background image using a VOVIAL RLE surface. This layer uses the RLE unit of the board's sprite engine (SPEA), to decompress RLE compressed image data to a virtual framebuffer.
- **Sprite layer**
  The sprite layer uses a VOVIAL sprite surface to show several GUI elements. It uses one layer of D1L2's sprite engine (SPEA).
- **Overlay sprite layer**
  The overlay sprite layer is also a VOVIAL sprite surface. The layer lies above the sprite layer to create sprite overlay effects, e.g. the gas gauge's hand and the fan's highlight element.
- **Frame buffer**
  The fonts are rendered to a frame buffer that has a size of 240*64 pixels. The frame buffer uses double buffering to avoid visible glitches while rendering the fonts.

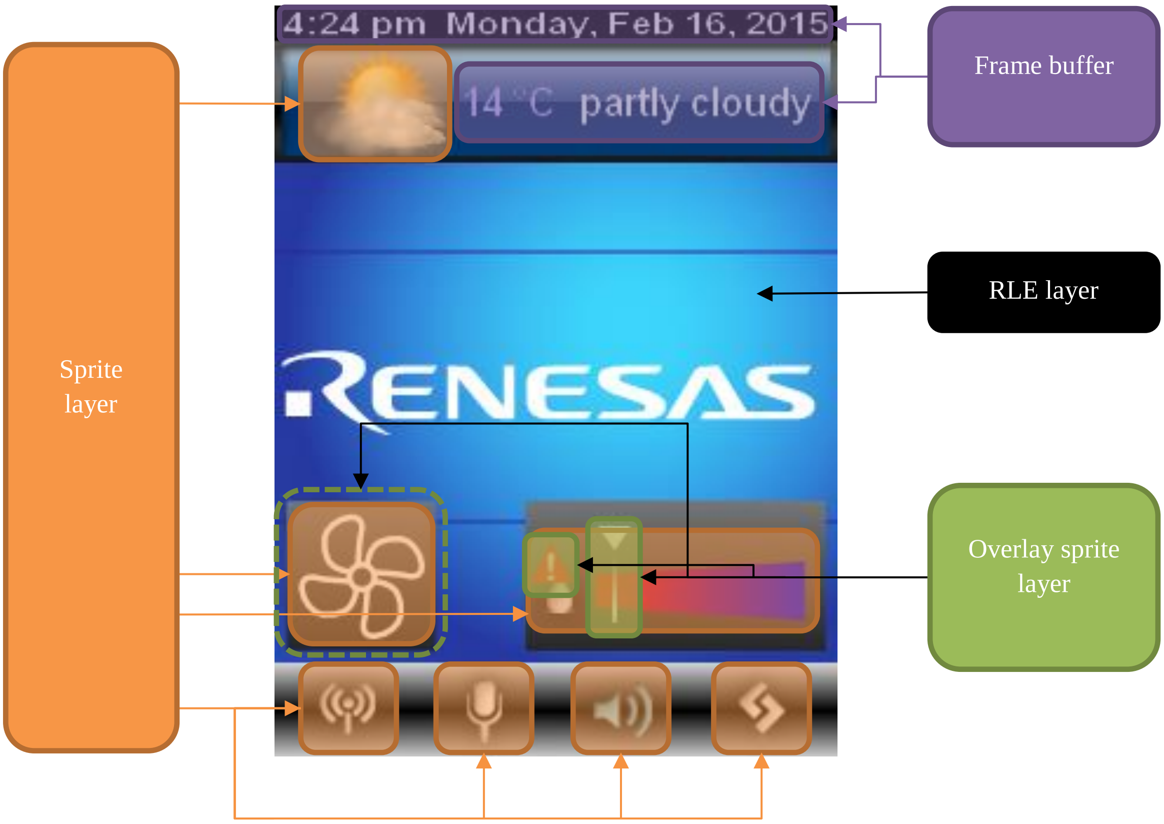

Figure 4-2: Layers of app_tripcomp

Creating the VOVIAL surfaces for the layers

To create the layers described above, VOVIAL's `FW_VOVIAL_CreateSurface` function is used in the `locInit` function, which can be found in "app_tripcomp.c". The function is called with the appropriate flags as follows:

```
locSurface = FW_VOVIAL_CreateSurface(
            LOC_VOVIAL_UNIT,
            FW_VOVIAL_ARGB8888,
            0,
            0,
            3,
            LOC_DISPLAY_STRIDE,
            LOC_DISPLAY_WIDTH,
            64,
            0xff,
            2,
            0);

    locSurfaceSprite = FW_VOVIAL_CreateSurface(
                LOC_VOVIAL_UNIT,
                FW_VOVIAL_ARGB8888,
                0,
                0,
                1,
                0,
                LOC_DISPLAY_WIDTH,
                LOC_DISPLAY_HEIGHT,
                0xff,
                1,
                FW_VOVIAL_CRFLAG_SPRITE);

    locSurfaceSpriteTop = FW_VOVIAL_CreateSurface(
                    LOC_VOVIAL_UNIT,
                    FW_VOVIAL_ARGB8888,
                    0,
                    0,
                    2,
                    0,
                    LOC_DISPLAY_WIDTH,
                    LOC_DISPLAY_HEIGHT,
                    0xff,
                    1,
                    FW_VOVIAL_CRFLAG_SPRITE);

    locSurfaceRLE = FW_VOVIAL_CreateSurface(
                LOC_VOVIAL_UNIT,
                FW_VOVIAL_RLE24RGB0888,
                0,
                0,
                0,
                LOC_DISPLAY_STRIDE,
                LOC_DISPLAY_WIDTH,
                LOC_DISPLAY_HEIGHT,
                0xff,
                1,
                FW_VOVIAL_CRFLAG_RLE);
```

The frame buffer `locSurface` is created with a height of 64 and two back buffers. The `Flags` parameter is set to 0, since a usual frame buffer should be created. Its z position is set to 3, in order to place it on top of all other surfaces.

Both sprite surfaces `locSurfaceSprite` and `locSurfaceSpriteTop` are created with the `FW_VOVIAL_CRFLAG_SPRITE` flag set. Their size is set to fit the whole screen. The stride will be set automatically, depending on the selected color format. The `PosZ` parameter is used to place `locSurfaceSpriteTop` over `locSurfaceSprite`, since `locSurfaceSpriteTop` is used to create overlay effects.

`locSurfaceRLE` is created with `the FW_VOVIAL_CRFLAG_RLE` flag. An RLE surface needs RLE image data that will be displayed. The image data is set via `FW_VOVIAL_SetBufferAddr`, after the Img_t objects were initialized in `locInit` as follows:

```
FW_VOVIAL_SetBufferAddr(locSurfaceRLE, 1, (void**) Img_tp_bg.Data);
```

Using the VRAM Wrapper

The D1x boards provide a hardware wrapper, supporting various pixel formats (e.g. ARGB6666, RGB666, RGB888), for their internal VRAM. The image data is wrapped transparently to ARGB8888 format, simply by reading the data from the wrapper address spaces.

In the Trip Computer Demo, this feature is used to save memory when using the sprite engine. All menu elements are converted to ARGB6666, which saves 1 byte per pixel (at the cost of color depth). The sprite engine can use them anyhow on an ARGB8888 surface with help of the VRAM wrapper.

Since the menu elements’ image data is stored in the serial flash, they have to be copied to the VRAM first. During initialization (`locInit`), the function shown below is called to do so:

```
void locMoveToVRAM(const Img_t *Image)
{
    static int32_t block_idx = 0;
    uint8_t *vram;
    int32_t i;
    uint32_t align, offset;

    /* Take 6 more bytes than needed to leave room for alignment. */
    vram = (uint8_t*) R_CDI_Alloc(&loc_VRAM_heap, Image->DataLength + 6);
    locAllocedVRAM[block_idx] = (void*) vram;
    block_idx = block_idx % LOC_VRAM_IMAGE_COUNT;

    /* Get the offset within VRAM. */
    offset = (uint32_t) vram - LOC_VRAM0;

    /* The data must be aligned to an address divisible by 6, since we are
     * dealing with a 3 bytes per pixel format and SPEA needs a 2 pixel
     * alignment. */
    align = 6 - (offset % 6);

    /* Make the address divisible by 3 */
    offset = offset + align;

    for(i = 0; i < Image->DataLength; ++i)
    {
        ((uint8_t*) offset)[i] = (*Image->Data)[i];
    }

    /* Compute address in wrapper address space. */
    offset = offset / 3 * 4;
    *Image->Data = (uint8_t*) (LOC_VRAM0_WRAP_ARGB6666 + offset);
}
```

Since the VRAM wrapper extends the pixel data to ARGB8888, it is crucial to store the pixel data with a suitable alignment. As a 3 byte per pixel color format is used and the sprite engine requires an alignment of 2 pixels, the data must be aligned to a VRAM offset divisible by 6. Note that this must be respected when allocating the memory via R_CDI. The allocated memory is increased by 6 byte to leave room for the alignment.

The offset is computed by subtracting the VRAM base address from the just allocated memory's address. Then, the offset is increased to match the 6 byte alignment.

In the final step, the address of the data in the wrapper address space has to be computed. The offset is divided by 3 to get the number of pixels (ARGB6666 = 3 bytes per pixel) and then multiplied by 4 (ARGB8888 = 4 bytes per pixel) to get the number of bytes in the 4 byte per pixel color space.

Setting up TP GUI elements and underlying sprites

The `SetupMenu` function in "app_tripcomp.c" sets up all TP GUI elements (see 4.1.1).

First, the underlying `fw_vovial_Sprite_t` objects for each element are initialized by calling `SetupSprite`. The sprites' image size can be taken from the `Img_t` object of the particular sprite image. The sprites' position is given as parameter to the function. In this case, the `Z` parameter distinguishes if the sprite is added to the overlay layer `locSurfaceSpriteTop (Z == 1)` or to underlying sprite layer `locSurfaceSprite (Z == 0)`. The function's code is given below:

```
static void SetupSprite(const Img_t *Image, fw_vovial_Sprite_t *Conf, uint16_t X,
uint16_t Y, uint16_t Z)
{
    if(Z==1)
    {
        Conf->Surface = locSurfaceSpriteTop;
    }
    else
    {
        Conf->Surface = locSurfaceSprite;
    }
    Conf->Status = FW_VOVIAL_SPRITESTATUS_NOT_INITIALIZED;
    Conf->Data = *Image->Data;
    Conf->Width = Image->Width;
    Conf->Height = Image->Height;
    Conf->PosX = X;
    Conf->PosY = Y;
    Conf->PosZ = 0;
}
```

In the following, the `tp_menu_Element_t` objects for each GUI element are set up using the `fw_vovial_Sprite_t` objects and the particular image data pointers. Finally, the GUI elements are added to the TP GUI menu by use of `TP_GUI_MenuAddElement`.

Creating a moving animation for the gas gauge hand

The gas gauge hand's moving animation is created by implementing a callback function for the `locElementFuelHand` object's VBlank event (see the structure's `OnVBlank` member). The `locOnVBlankFuelHand` callback function moves the hand sprite using the `TP_GUI_MenuMoveElement` function (see 4.1.1):

```
TP_GUI_MenuMoveElement(Element, Element->SpriteConf->PosX - 1,
                       Element->SpriteConf->PosY,
                       Element->SpriteConf->PosZ);
```

Since the callback is called on every VBlank, the function uses a static counter to move the hand's position only every 30 frames.

Creating a blinking animation for the warning sign

The blinking animation of the warning sign is also implemented using an `OnVBlank` callback function. The callback function can be found as `locOnVBlankWarningSign` in “app_tripcomp.c”.

The warning sign GUI element is simply disabled and enabled via the `TP_GUI_MenuDisableElement` and `TP_GUI_MenuEnableElement` functions of TP GUI (see 4.1.1) to generate the blinking animation.

Since the function is called on every VBlank, again a static counter is used to disable and enable the element every 40 frames.

Font rendering in the status bar

The font rendering is implemented in “tp_font.c”. The single characters are stored in a single texture, ordered by their ASCII code. This allows to compute each characters texture location by the use of its ASCII code. This is implemented in the `locGetCharRemapping` function.

The actual render operation for each character is implemented in `locWriteChar`. The function computes the source rectangle for the character in the font texture and renders it to the destination rectangle by a call to `R_DRW2D_TextureBlit`.

Since rendering is done in software on the D1L2, it is reasonable to reduce the render operations to a minimum. Therefore, the status bar is redrawn only if it changes. The behavior is implemented in `locStatusBarRender` of “app_tripcomp.c” by the use of a dirty flag `locStatusBarDirty`. As long as the dirty flag is greater than 0, the function renders to the status bar and decreases the dirty flag’s value. When setting the flag to 2, the text will be rendered in the next two render passes in order to fill both buffers of the `locSurface` surface.

Whenever the status bar text is changed, the dirty flag `locStatusBarDirty` has to be set to 2. This is implemented, for example, in `locStatusBarShowTip`, which can be called to show a tool tip in the status bar for a few seconds.

4.2 App_drw2dcpu

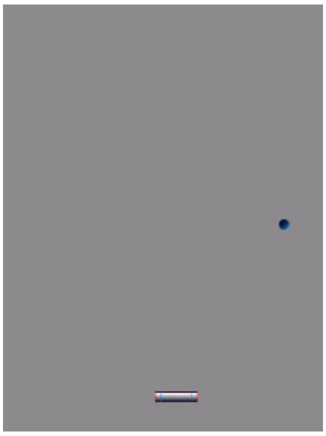

Figure 4-3: Screenshot of app_drw2dcpu

The app_drw2dcpu demo uses the Drw2D CPU implementation to implement a simple ball batting game as shown in Figure 4-3.

The demo shows a bat on the lower screen and a moving ball. The bat can be moved using the board’s knob.

4.2.1 Implementation

Initialization

To save memory, the application uses a frame buffer with a 16 bit color format (RGB565). Therefore, the VOVIAL surface is initialized with the format `FW_VOVIAL_RGB565` in `locInit`.

Drw2D frame buffer format must be set to the equal format, which is `R_DRW2D_PIXELFORMAT_RGB565`. This is done in the function `locSetRenderTarget`.

Enabling the fast render path

Since both textures used in this example are rendered without any transformation (e.g. scaling, rotating), the fast render path of the Drw2D software implementation can be enabled. This will skip some of the operations on the render path (e.g. convex polygon rasterization) and speed things up.

Setting the transform mode to `R_DRW2D_TRANSFORM_NONE` will enable the fast render path. The appropriate Drw2D function (`R_DRW2D_CtxTransformMode`) is called in `locAppTestThread`, before the thread's main loop is entered:

```
R_DRW2D_CtxTransformMode(locDrw2dDev, R_DRW2D_TRANSFORM_NONE);
```

> *Note that enabling the fast render path will enable/disable some of the Drw2D (CPU implementation) features:*
>
> *RLE decoding is AVAILABLE ONLY when using the fast render path.*
>
> *Bilinear filtering is NOT AVAILABLE when using the fast render path.*

Rendering the textures

Since the bat texture does not use an alpha channel, it can be converted to the RGB565 color format. However, the ball texture uses transparency for an anti aliasing effect in the texture and has to be stored in the ARGB8888 color format. The Drw2D software implementation can handle both color formats and converts the latter one to RGB565 on rendering (after alpha blending).

As described in chapter 3.1.1, the `IMG_Blit` calls in `locRender` render the two textures and keep the difference in color format completely transparent to user.

5. Bitmap Tool

The following chapter describes the RGBench Bitmap tool and its usage. The covered tasks are: how to convert images, how to integrate them into grape_app and what options concerning memory usage, input/output format and compression are available.

The chapter aims to make the user able to convert images and generate the code necessary to integrate them into grape_app according to the user's needs.

5.1 The Bitmap Tool Dialog

The Bitmap tool is part of the RGBench, which has to be started first. The "r_rgbench.exe" executable can be found in "vlib\app\util\d1mx_eco_system\p_rgbench\". RGBench starts with a tool selection dialog as shown in Figure 5-1.

Click on "Bitmap tool" to open the Bitmap tool dialog.

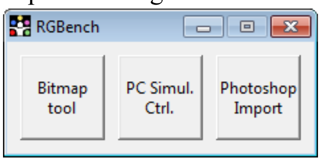

Figure 5-1: RGBench tool selection dialog

Appearance

The Bitmap tool consists of one dialog, containing control elements for all options that can be set. A screenshot of the dialog is shown in Figure 5-2.

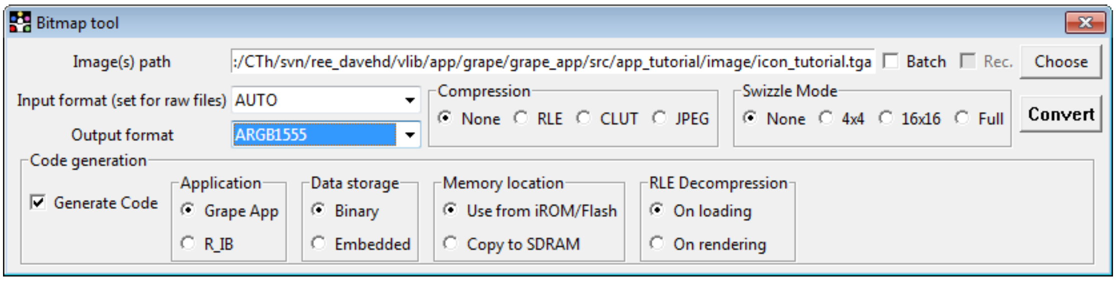

Figure 5-2: Bitmap tool dialog

Selecting the Input File(s)

The first row holds the user controls to set the input file or input files. At default, the “Batch” mode is deactivated and by clicking “Choose”, a single input image file can be selected.

When the “Batch” check box is activated, “Choose” opens a directory selection dialog. All files in the selected directory will be converted when clicking on “Convert”. Additionally the “Rec.” check box can be enabled. This enables the recursive batch mode, which selects all files in the chosen directory and all its deeper sub directories for conversion.

Choosing an Input Format

In the second row, the format of the input files can be chosen. It is recommended to set this to “AUTO”, except the input files are raw image files that do not carry any format information.

Choosing an Output Format

The combo box in the third row, allows to set the format of the output files. Note that the available output formats depend on the selected compression in the “Compression” frame.

Currently, there are four compression modes available:

- **None**: Do not use compression.
- **RLE**: Uses run length encoding to compress images. Works best for drawings with low color depth and long runs of the same color (e.g. GUI elements). RLE is rather useless for “noisy” images like photographs.
- **CLUT**: Uses a small look up table, containing an array of ARGB8888 values. Pixels in the original image are then replaced by their look up table index (which has 1, 4 or 8 bit)
- **JPEG**: Currently, this mode only supports a JPEG bypass. That means, that a given JPEG image directly written to the output file. Therefore, the input file has already to be a JPEG image. This option can be used for R_IB code generation only.

If no compression is enabled, the third row also allows to choose a swizzle mode in the “Swizzle Mode” frame. Swizzling increases the locality of pixels in a texture and thereby optimizes the utilization of caches in scenarios, where adjacent pixels have to be read (e.g. rotations, bilinear filtering). However, the rendering framework must support swizzled textures. There are four available swizzle modes:

- **None**: No texture swizzling.
- **4x4**: Divides the image in 4*4 sized swizzled squares. The image's width and height must be divisible by 4.
- **16x16**: Divides the image in 16*16 sized swizzled squares. The image's width and height must be divisible by 16.
- **Full**: Swizzles the whole image. The image's width and height must be a power of 2 (e.g. 2*2, 32*32, 256*256 …)

5.1.1 Code Generation Settings

The "Code generation" frame, as shown in Figure 5-3, holds all options that configure the Bitmap tool's code generation. If the "Generate Code" check box is enabled, the behavior of the tool differs to the general conversion mode (e.g. in output file name, output file target directory …).

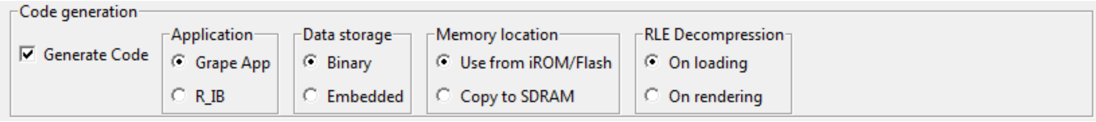

Figure 5-3: Code generation frame

Target Application

In the "Application" frame, you can choose the target application, for which you want to generate code. "Grape App" generates code and binary files for grape_app. "R_IB" generates code for application's that use the R_IB library.

Note that "R_IB" only works in batch mode. Moreover, if the "R_IB" radio button is enabled, all other code generation options are ignored.

Data Storage

To choose, where the images' binary data will be stored, use the "Data storage" frame. There are two storage modes available:

- **Binary**: The images' binary data will be stored in a separate binary (BIN file) per image. Therefore, a sub folder "bin" is created.
- **Embedded**: The images' binary data is directly stored in the C source file. No separate binary files are generated. For the source files, the tool creates a sub folder "src".

Memory Location

The image data can be used from the location where it was initially stored or copied to SDRAM before usage. Copying the image data to (expensive and therefore usually rare) SDRAM is reasonable in scenarios, where short access time and high throughput is desired or the image data is to be altered during run time.

Here, the Bitmap tool offers two options:

- **Use from iROM/Flash**: Leaves the image data in the internal ROM, resp. serial flash memory. (The location where the image data is initially stored can be chosen in the make file, see chapter 2.1.2.)
- **Copy to SDRAM**: Copies the image data to SDRAM when initializing the images. This is done, when calling Init2DImg (see "img.h" for details) for the corresponding image.

RLE Decompression

In the frame "RLE Decompression", the point in time when the RLE decompression of the image data is done can be chosen. There are two options available:

- **On loading**: Decompresses the image data during image initialization. This is done, when Init2DImg (see "img.h" for details) is called for the corresponding image.
- **On rendering**: Decompresses the image while rendering. This is reasonable when RLE hardware support is available (e.g. DHD).

Note that RLE decompression will slow rendering operations down significantly, when non-linear pixel access is necessary. This happens for instance if bilinear filtering or rotations are applied to the image data.

6. eFLASHLOAD

This chapter describes Renesas' eFLASHROM tool and how to use it to flash binary data to the D1Mx and D1L2 boards. The chapter covers how to configure the tool for the board to be flashed and how to flash a binary data, as generated by Grape_app's ROMFS building system (see chapter 2.3.1).

6.1 Starting the tool

After installing and starting the application, it will show its main dialog on the screen as shown in Figure 6-1.

The main dialog consists of three elements: a console on the left, status fields on the right and a menu bar of commands on the top.

The console shows status and progress information during the connecting and flashing process, as well as the executed commands. The status fields on the right shows information about the connected device and the used monitor and flash file. The monitor file is a small program that will be downloaded to the board and executed. Its purpose is to setup the board and serial flash and to communicate with the eFLASHLOAD tool running on the host computer.

The menu on the top shows the currently available commands that can be executed.

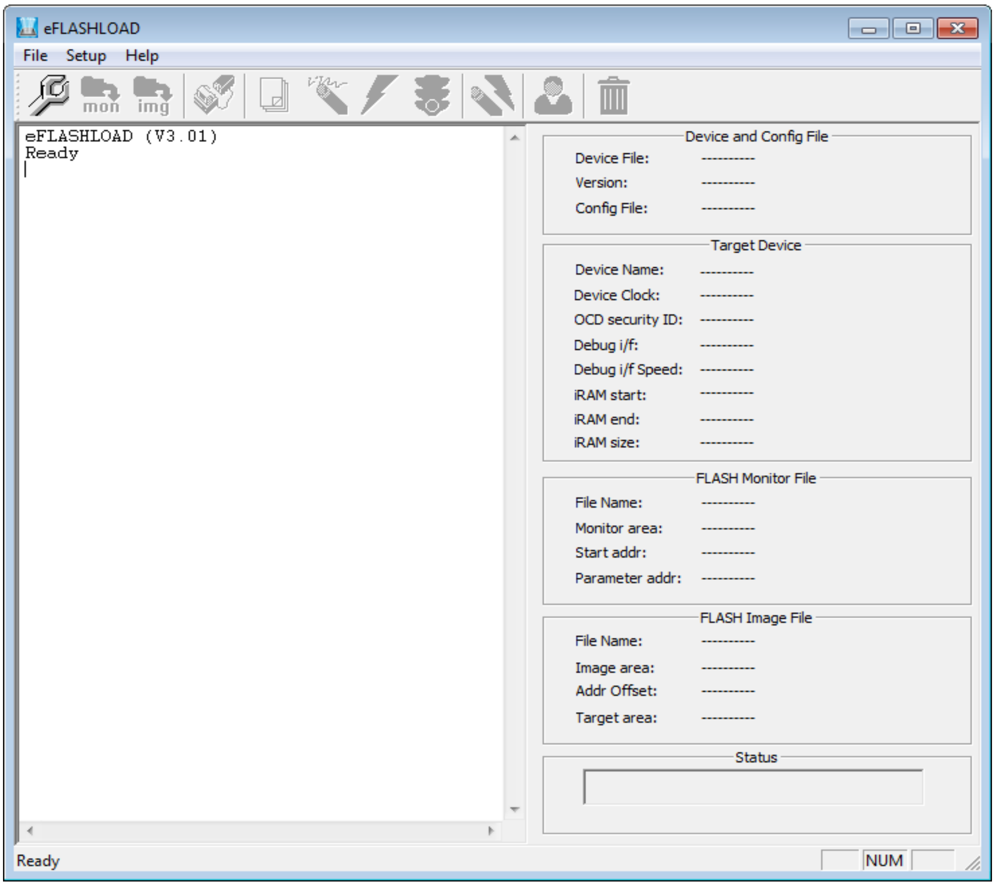

Figure 6-1: eFLASHLOAD main dialog

6.2 Flashing binary data

At first, the tool has to be configured for the D1Mx/D1L2 board. For this purpose, the tool comes with pre-built configuration files that are located in the tool's install directory (default: "C:\Program Files (x86)\Renesas Electronics\eFLASHLOAD_V301") in the folder "examples_config":

- Go to File/Load Config…
- In the file dialog, navigate to "examples_config" in the eFLASHLOAD install directory
- Select and open "FLM_D1M_R7F701412.xml" (works with both, D1Mx and D1L2)
- Click on the "Connect" button of the menu bar (see Figure 6-2)

- **Figure 6-2: Connect button after loading the config file**

eFLASHLOAD will now try to connect to the board, this may take a few seconds. After the connection is established, the status field on the lower right should show "Ready". The monitor program is now running on the board. The dialog should now look similar to the screenshot in Figure 6-3.

The connecting process may fail with an "RSU code verify error!" message. In such a case, unplug the executor/programmer from the host computer, power off and on the board, replug the executor/programmer and try it again.

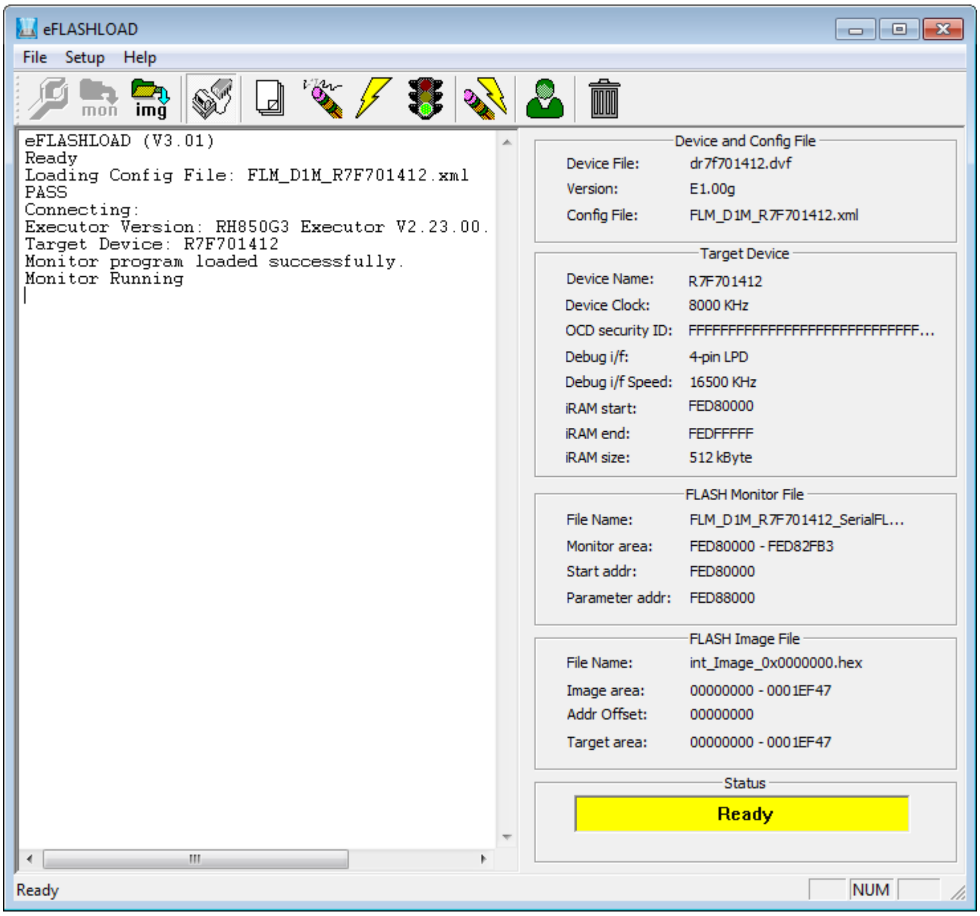

Figure 6-3: Dialog after succesfully connecting to the device

At last, the binary data file to be flashed has to be selected. To do so, proceed as follows:

- Click on the “Image Setup” button of the menu bar (see Figure 6-4)
- The “FLASH image settings” dialog will show up
- In the image file section, navigate to the binary data file to be flashed
  - For Grape_app’s ROMFS, the file can be found at “vlib\app\grape\grape_app\romfs\binary\platform\[*platform name*]\data_flash_sf.srec”
- Enter the offset for the data within the board’s flash memory
  - Grape_app’s ROMFS, expects the data to begin at 0x00000000
- Click on OK

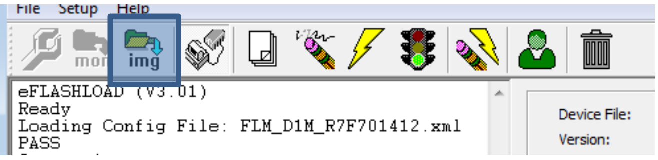

Figure 6-4: Image Setup button after connecting to the device

The eFLASHLOAD tool is now configured to flash the binary data file to the board’s serial flash memory. To start the flashing process, click on the “E.P.V.” button (the button is shown in Figure 6-5). This will flash the data in a threepart process. The flash memory is erased and programmed with the image data. Then, the data is verified by reading and comparing it with the binary image file.
If the process finishs successfully, the status bar on the lower right will show “Pass” on a green background.

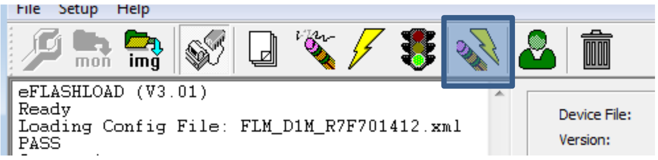

Figure 6-5: E.P.V. button after selecting the binary image to be flashed

Revision History	Grape_app User Manual User's Manual: Applocation

Rev.	Date	Description	
		Page	Summary
0.1	2015-01-08	-	Initial Release
0.2	2015-02-20	-	Adapted tutorial for D1L2 Added descriptions of platform specific grape_app features
0.3	2015-03-03	-	Added eFLASHLOAD chapter
0.4	2016-10-18	-	Delete D1Hx, small adaptions in generic chapters
0.5	2017-06-13	-	Update version.

RH850/D1x device family
Grape_app User Manual
User’s Manual: Application

Publication Date:	Rev.0.10	Jan 08, 2015
	Rev.0.50	June 13, 2017

Published by: Renesas Electronics Corporation

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HAL II Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 4.1

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口

http://www.renesas.com

※営業お問合せ窓口の住所は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス株式会社 〒135-0061 東京都江東区豊洲3-2-24（豊洲フォレシア）

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：https://www.renesas.com/contact/

Colophon 3.1

Grape_app User Manual

Renesas Electronics Corporation