User Manual

RZ/G2 Group

Application Note

Renesas Electronics
www.renesas.com

Rev.1.03 Jul 2020

How to Use This Manual

1. Purpose and Target Readers

This manual is provided to give users an understanding of concrete methods of developing software that runs on RZ/G2-group products. It's intended for users who are designing application software to run on systems designed around RZ/G2-group products. Applying the information in this manual requires knowledge of the fundamentals of software development, including operating systems and programming.

This manual is roughly divided into three subject areas and gives a description and concrete examples of each: namely, the development of applications to run under the Linux OS on products of the RZ/G2 group, and specifically from among such applications, the development of those for which the GStreamer framework is used, and of those for which the Qt framework is used.

Particular attention should be paid to the precautionary notes when using RZ/G2-group products. These notes occur within the body of the text, at the end of each section, and in the Usage Notes section.

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.

2. Abbreviations

The table below lists abbreviations used with the RZ/G2 group as a whole.

Abbreviation	Description
AHCI	Advanced Host Controller Interface
ALSA	Advanced Linux Sound Architecture
ATA	Advanced Technology Attachment
CPRM	Content Protection for Recordable Media
DMA	Direct Memory Access
DMAC	DMA Controller
DRM	Direct Rendering Manager
DU	Display Unit on RZ/G2
EHCI	Enhanced Host Controller Interface
eMMC	Embedded Multi Media Card
FB	Framebuffer
GLSL	OpenGL Shading Language
GPIO	General Purpose Input/Output interface
GPL	GNU General Public License
gPTP	Generalized Precision Time Protocol
GUI	Graphical User Interface
HSCIF	High Speed Serial Communications Interface with FIFO
I2C	Inter-Integrated Circuit
LGPL	GNU Lesser General Public License
MMC	Multi Media Card
MMCIF	Multi Media Card Interface H/W module
MSIOF	Clock-Synchronized Serial Interface with FIFO
MTD	Memory Technology Device
NCQ	Native Command Queuing
OHCI	Open Host Controller Interface
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PCIEC	PCIe host controller
PCM	Pulse Code Modulation
PTP	Precision Time Protocol
QSPI	Quad Serial Peripheral Interface
SATA	Serial Advanced Technology Attachment
SCIF	Serial Communications Interface with FIFO
SD	Secure Digital
SDIO	Secure Digital Input/Output
SPI	Serial Peripheral Interface
SRC	Sampling Rate Converter
SSI	Serial Sound Interface
USB	Universal Serial Bus
V4L2	Video for Linux2
VSPD	VSP for DU
xHCI	Extensible Host Controller Interface

Contents

1. Overview

This manual gives sample applications and a description of the development of applications to run under the Linux OS on products of the RZ/G2 group for reference.

The sample applications are assumed to be developed on the RZ/G2 platform from Renesas or under the Linux OS from the Yocto project. Moreover, the GStreamer or Qt framework is used in developing the sample applications, as the note is intended to help you to confirm the basic operations of applications for which the GStreamer or Qt framework is used.

Note: Until this version, these applications can run on RZ/G2E, RZ/G2N, RZ/G2M and RZ/G2H boards. Others will support later.

1.1 How to Use This Application Note

To confirm how to develop applications to run under the Linux OS on products of the RZ/G2 group and see descriptions of the sample programs for this, see section 2.

- 2. Linux Applications

To confirm how to use the GStreamer framework to develop applications to run under the Linux OS on products of the RZ/G2 group and see descriptions of the sample programs for this, see section 3.

- 3. GStreamer Applications

To confirm how to use the Qt framework to develop applications to run under the Linux OS on products of the RZ/G2 group and see descriptions of the sample programs for this, see section 4.

- 4. Qt Applications

Each section has two subsections. The first subsection gives a simple application as an example and describes the given application, including an overview, the development environment for creating the application, the source code, and how to build and execute the application. The latter subsection gives a description for each sample application in terms of the source code and its structure, how to obtain the source code, and results of execution.

1.2 Related Document for the RZ/G2 Group

For more details about RZ/G2 Linux BSP, SDK (**S**oftware **D**evelopment **K**it), please refer to the Release Note of RZ/G Verified Linux Package for 64-bit kernel (r01tu0277ej0104-rz-g (Release Note)).

2. Linux applications

This start-up guide explains how to use SDK to build some sample applications such as Hello World, Image Display, and Serial Communication for RZ/G2 Group. Please contact Renesas Electronics personnel who provided this product to you in the case of questions.

2.1 Hello World

This section describes *Hello World* application, how to cross-compile, and run it in Linux environment.

2.1.1 Development Environment

- Yocto 2.4.3
- Linux kernel CIP 4.19
- RZ/G2 Linux platform VLP 64.

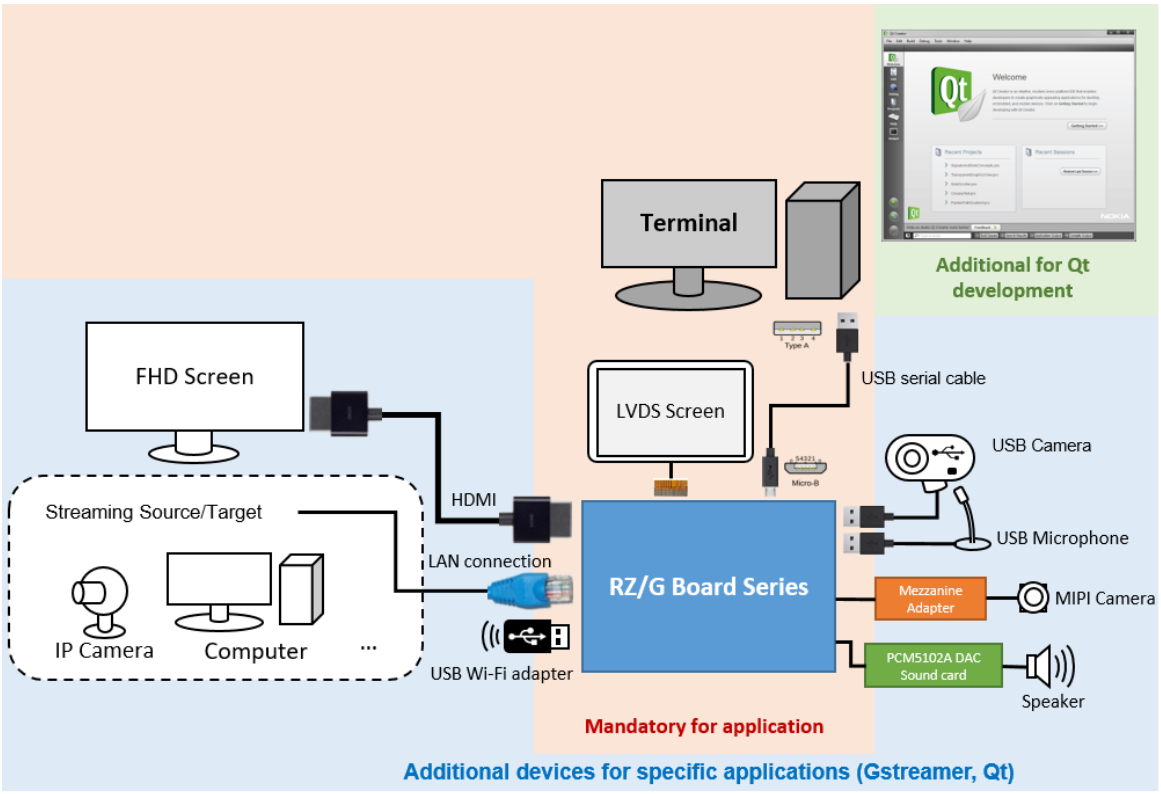


Figure 2.1 Equipment diagram

2.1.2 Application content

Content of **main.c**:

```
#include <stdio.h>
/* Display "Hello World" text on terminal software */
int main(int argc, char** argv)
{
    printf("\nHello World\n");
    return 0;
}
```

Walkthrough:

Include header file

```
#include <stdio.h>
```

The `#include <stdio.h>` is a preprocessor command. This command tells compiler to include the contents of `stdio.h` (standard input and output) file in the program.

The `stdio.h` file contains functions such as `scanf()` and `printf()` to take input and display output respectively.

If you use `printf()` without writing `#include <stdio.h>`, the program will not be compiled.

The main function

```
int main(int argc, char** argv);
```

It is the first function in your program that is executed when the application begins executing. Note that it has 2 arguments called `argc` and `argv` and returns a sign integer (`int`). Linux environment expects the program to return `0` on success and `-1` on failure.

The `argc` represents the number of command-line arguments.

The `argv` is a pointer to the first element of an array of `argc - 1` strings which are command-line arguments. Note that `argv[0]` is the name of this application.

The printf() function

```
printf("\nHello World\n");
```

It is a function which sends formatted output to the screen. In this program, `printf()` displays “Hello World” text (surrounded by 2 line breaks (`'\n'`)) on the monitor.

2.1.3 How to build and run Linux application

This section shows how to cross-compile and deploy Linux *Hello World* application.

2.1.3.1 How to extract SDK

Step 1. Install toolchain on a Host PC:

```
$  sudo sh ./poky-glibc-x86_64-core-image-weston-sdk-aarch64-toolchain-2.4.3.sh
```

Note:

sudo is optional in case user wants to extract SDK into a restricted directory (such as: /opt/).

If the installation is successful, the following messages will appear:

```
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
 $ . /opt/poky/ek874/environment-setup-aarch64-poky-linux
 $ . /opt/poky/ek874/environment-setup-armv7vehf-neon-pokymllib32-linux-gnueabi
```

Figure 2.2 SDK is successfully set up

Step 2. Set up cross-compile environment:

```
$  source /<Location in which SDK is extracted>/environment-setup-aarch64-poky-linux
```

Note:

User needs to run the above command once for each login session.

2.1.3.2 How to build and run Linux application

Step 1. Go to *linux-helloworld* directory:

```
$  cd $WORK/linux-helloworld
```

Step 2. Cross-compile:

```
$  make
```

Step 3. Copy all files inside this directory to `/usr/share` directory on the target board:

```
$  scp -r $WORK/linux-helloworld/ <username>@<board IP>:/usr/share/
```

Step 4. Run the application:

```
/usr/share/linux-helloworld/linux-helloworld
```

2.2 Application samples

The following table shows Linux applications in RZ/G2 platform.

Application name	Description
Image display	Display a bitmap image on Wayland desktop.
Serial communication	Send and receive data to/from PC via serial port.

Table 2.1 Linux sample applications

2.2.1 Image Display

Display a bitmap (bmp) image on Wayland desktop.

2.2.1.1 Source code

(1) Files

- main.c
- util.h/util.c

(2) main.c

```
#include "util.h"
#define ARG_NUMBER              2           /* Number input parameters on terminal */
#define RAW_IMAGE_FILENAME     "images.bin" /* Default output name */

struct wl_display *display=NULL;
struct wl_shell *shell=NULL;
struct wl_registry *registry=NULL;
struct wl_buffer *buffer=NULL;
struct wl_surface *surface=NULL;
struct wl_shell_surface *shell_surface=NULL;
struct wl_compositor *compositor=NULL;

static void registry_global(void *data,
                            struct wl_registry *registry, uint32_t name,
                            const char *interface, uint32_t version) {
    if (strcmp(interface, wl_compositor_interface.name) == 0)
        compositor = wl_registry_bind(registry, name, &wl_compositor_interface, 1);
    else if (strcmp(interface, wl_shm_interface.name) == 0)
        shm = wl_registry_bind(registry, name, &wl_shm_interface, 1);
    else if (strcmp(interface, wl_shell_interface.name) == 0)
        shell = wl_registry_bind(registry, name, &wl_shell_interface, 1);
}

/* registry callback for listener */
static const struct wl_registry_listener registry_listener = {
    .global = registry_global,
};

static void shell_surface_ping(void *data,
                           struct wl_shell_surface *shell_surface,
                           uint32_t serial) {
    wl_shell_surface_pong(shell_surface, serial);
}

static const struct wl_shell_surface_listener shell_surface_listener = {
    .ping = shell_surface_ping,
};

int main(int argc, char **argv)
{
    const char *filename;
    const char *raw_image_name = RAW_IMAGE_FILENAME;
    int image;
    int ret = 0;

    if (argc != ARG_NUMBER) {
        puts("Not enough arguments");
        puts("Usage:");
        puts("          ./linux-imagedisplay <filename>");
        puts("Example:");
        puts("          ./linux-imagedisplay 640x480.bmp");
        return -1;
    }

    filename = argv[1];
```

```
    /* Convert bmp file into bgrx8888 */
    ret = write_bmp_data(filename, raw_image_name);
    if (ret == -1) {
        perror("Input image processing failed!");
        return ret;
    }

    image = open(raw_image_name, O_RDWR);
    if (image < 0) {
        perror("Error opening surface image");
        return -1;
    }

    /* Connects to the display server */
    display = wl_display_connect(NULL);
    if (display == NULL) {
        perror("Connecting to display server failed!");
        close(image);
        return -1;
    }

    /* Get registry to work with global object and set listener */
    registry = wl_display_get_registry(display);
    wl_registry_add_listener(registry, &registry_listener, NULL);

    /* Send request to server */
    wl_display_roundtrip(display);

    /* compositor is already bind in registry callback */
    surface = wl_compositor_create_surface(compositor);
    if (surface == NULL)
        goto shell_destructor;

    shell_surface = wl_shell_get_shell_surface(shell, surface);
    if (shell_surface == NULL) {
       goto surface_destructor;
    }

     /* shell_surface config */
     wl_shell_surface_add_listener(shell_surface, &shell_surface_listener, 0);
     wl_shell_surface_set_toplevel(shell_surface);
     wl_shell_surface_set_user_data(shell_surface, surface);
     wl_surface_set_user_data(surface, NULL);

     /* create and bind buffer to surface */
     buffer = create_image_buffer(image);
     if (buffer == NULL)
         goto surface_destructor;

     wl_surface_attach(surface, buffer, 0, 0);
     wl_surface_commit(surface);

     while (1) {
         if (wl_display_dispatch(display) < 0) {
             perror("Main loop error");
             break;
         }
     }

     /* Clean up */
     wl_buffer_destroy(buffer);

surface_destructor:
     free_surface(shell_surface);

shell_destructor:
    wl_shell_destroy(shell);

registry_destructor:
    wl_shm_destroy(shm);
    wl_compositor_destroy(compositor);
    wl_registry_destroy(registry);

    wl_display_disconnect(display);
```

```
    close(image);

    return 0;
}
```

Walkthrough:

Include header file

```
#include "util.h"
```

The `util.h` file contains functions that allow us to convert the input image from `bmp` to raw `bgrx8888` format and create image buffer in shared memory pool.

```
if (argc != ARG_NUMBER) {
    puts("Not enough arguments");
    puts("Usage:");
    puts("          ./linux-imagedisplay <filename>");
    puts("Example:");
    puts("          ./linux-imagedisplay 640x480.bmp");
    return -1;
}

filename = argv[1];
```

This application requires 1 command-line argument which points to a bitmap image. If the condition is not met, the application will print an error message and its usage.

Process user input

```
#define RAW_IMAGE_FILENAME     "images.bin"
const char *raw_image_name = RAW_IMAGE_FILENAME;

write_bmp_data(filename, raw_image_name);
open(raw_image_name, O_RDWR);
```

This function converts the `filename` from `bitmap` format to `gbrx8888` format and then writes the output to `images.bin`.

Next, the application opens `images.bin` to create a buffer in Wayland's shared memory pool.

Interact with Wayland display server

```
display = wl_display_connect(NULL);
if (display == NULL) {
    perror("Connecting to display server failed!");
    close(image);
    return -1;
}

registry = wl_display_get_registry(display);
wl_registry_add_listener(registry, &registry_listener, NULL);

wl_display_roundtrip(display);
```

The application calls `wl_display_connect()` to connect to Wayland display server and uses `wl_registry_add_listener()` to request global objects (such as: `wl_compositor`, `wl_shm`, and `wl_shell`) from it.

```
surface = wl_compositor_create_surface(compositor);
if (surface == NULL)
    goto shell_destructor;

shell_surface = wl_shell_get_shell_surface(shell, surface);
```

```
if (shell_surface == NULL) {
   goto surface_destructor;
}

wl_shell_surface_add_listener(shell_surface, &shell_surface_listener, 0);
wl_shell_surface_set_toplevel(shell_surface);
wl_shell_surface_set_user_data(shell_surface, surface);
wl_surface_set_user_data(surface, NULL);
```

In this step, we call `wl_compositor_create_surface()` to create the `surface` (user interface) and keep it always on top and responsive.

```
/* create and bind buffer to surface */
buffer = create_image_buffer(image);
if (buffer == NULL)
    goto surface_destructor;
```

In this application, `create_image_buffer()` creates the `buffer` (from the `image`) which is shared between the client application and the display server, so no copies are involved. That is the main design philosophy of Wayland when dealing with graphics.

```
wl_surface_attach(surface, buffer, 0, 0);
wl_surface_commit(surface);
```

Now, we need to bind the `buffer` to the `surface` by using function `wl_surface_attach()`. Next, call `wl_surface_commit()` to notify Wayland server so that it can “draw” the input image at coordinate (0, 0) on the monitor.

```
while (1) {
    if (wl_display_dispatch(display) < 0) {
        perror("Main loop error");
        break;
    }
}
```

This code block holds the program until the user presses Ctrl-C.

Clean up

```
free_surface(shell_surface);

wl_shell_destroy(shell);

wl_shm_destroy(shm);
wl_compositor_destroy(compositor);
wl_registry_destroy(registry);

wl_display_disconnect(display);
close(image);
```

(3) util.h/util.c

```
#ifndef _UTIL_H
#define _UTIL_H

#include <stdio.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <wayland-client.h>

typedef uint32_t pixel;

typedef struct tagBITMAPFILEHEADER {
    unsigned short bfType;
    unsigned int bfSize;
    unsigned short bfReserved1;
    unsigned short bfReserved2;
    unsigned int bfOffBits;
} __attribute__((packed)) BITMAPFILEHEADER;

typedef struct tagBITMAPINFOHEADER {
   unsigned int biSize;
   int biWidth;
   int biHeight;
   unsigned short biPlanes;
   unsigned short biBitCount;
   unsigned int biCompression;
   unsigned int biSizeImage;
   int biXPelsPerMeter;
   int biYPelsPerMeter;
   unsigned int biClrUsed;
   unsigned int biClrImportant;
} __attribute__((packed)) BITMAPINFOHEADER;

BITMAPFILEHEADER bfh;
BITMAPINFOHEADER bih;

struct pool_data {
    int fd;
    pixel *memory;
    unsigned capacity;
    unsigned size;
};

extern struct wl_shm *shm;

/* Function declaration */
int write_bmp_data(const char *filename, const char *outfile);

struct wl_buffer *create_image_buffer(int image);

void free_surface(struct wl_shell_surface *shell_surface);

#endif /* _UTIL_H */
```

Walkthrough:

Global variable

```
typedef uint32_t pixel;
```

It is a 4-byte integer which represents a pixel.

Data structures

```
typedef struct tagBITMAPFILEHEADER {
    unsigned short bfType;
    unsigned int bfSize;
    unsigned short bfReserved1;
    unsigned short bfReserved2;
    unsigned int bfOffBits;
} __attribute__((packed)) BITMAPFILEHEADER;
```

This structure contains bitmap header structure, such as:

- Variable `bfType` (unsigned short): A 2-byte integer to represent the file type; must be `BM`.
- Variable `bfSize` (unsigned int): A 4-byte integer to specify the size, in bytes, of the bitmap file.
- Variable `bfReserved1` (unsigned short): A 2-byte integer to represent reserved area; must be `zero`.
- Variable `bfReserved2` (unsigned short): A 2-byte integer to represent reserved area; must be `zero`.
- Variable `bfOffBits` (unsigned int): A 4-byte integer to specify the offset, in bytes, from the beginning of the `BITMAPFILEHEADER` structure to the bitmap bits.

```
typedef struct tagBITMAPINFOHEADER {
    unsigned int biSize;
    int biWidth;
    int biHeight;
    unsigned short biPlanes;
    unsigned short biBitCount;
    unsigned int biCompression;
    unsigned int biSizeImage;
    int biXPelsPerMeter;
    int biYPelsPerMeter;
    unsigned int biClrUsed;
    unsigned int biClrImportant;
} __attribute__((packed)) BITMAPINFOHEADER;
```

This structure contains information about the dimension and color format of a device-independent bitmap (`DIB`):

- Variable `biSize` (unsigned int): A 4-byte integer to specify the number of bytes required by the structure.
- Variable `biWidth` (int): A 4-byte integer to specify the width of the bitmap, in pixels.
- Variable `biHeight` (int): A 4-byte integer to specify the height of the bitmap, in pixels.
- Variable `biPlanes` (unsigned short): A 2-byte integer to specify the number of planes for the target device. This value must be set to `1`.
- Variable `biBitCount` (unsigned short): A 2-byte integer to specify the number of bits per pixel (`bpp`).
- Variable `biCompression` (unsigned int): A 4-byte integer whose value is either `BI_RGB` (uncompressed RGB) or `BI_BITFIELDS` (uncompressed RGB with color masks).
- Variable `biSizeImage` (unsigned int): A 4-byte integer to specify the size, in bytes, of the image. This can be set to `0` for uncompressed RGB bitmaps.
- Variable `biXPelsPerMeter` (int): A 4-byte integer to specify the horizontal resolution, in pixels per meter, of the target device for the bitmap.
- Variable `biYPelsPerMeter` (int): A 4-byte integer to specify the vertical resolution, in pixels per meter, of the target device for the bitmap.
- Variable `biClrUsed` (unsigned int): A 4-byte integer to specify the number of color indices in the color table that are actually used by the bitmap.

- Variable `biClrImportant` (unsigned int): A 4-byte integer to specify the number of color indices that are considered important for displaying the bitmap. If this value is `zero`, all colors are important.

```
struct pool_data {
    int fd;
    pixel *memory;
    unsigned capacity;
    unsigned size;
};
```

This structure contains information to create image buffer in Wayland's shared memory pool:

- Variable `fd` (int): A file descriptor that "points" to the raw `bgrx8888` image file.
- Variable `memory` (`struct pixel*`): A buffer which maps to `fd` file descriptor.
- Variable `capacity` (unsigned int): A 4-byte integer to specify the size, in bytes, of `fd` file descriptor.
- Variable `size` (unsigned int): A 4-byte integer to specify the frame size, in bytes, of `fd` file descriptor. It is calculated by multiplying 4 (`BGRX`) x frame width x frame height.

APIs

```
int write_bmp_data(const char *filename, const char *outfile);
```

This function converts the `filename` from `bitmap` format to `gbrx8888` format and then writes the output to `outfile`.

```
struct wl_buffer *create_image_buffer(int image);
```

This function creates a buffer of the `image` (`outfile`) in Wayland's shared memory pool.

```
void free_surface(struct wl_shell_surface *shell_surface);
```

This function frees `wl_surface` and `wl_shell_surface` objects.

2.2.1.2 Result

Can display a bitmap image on Wayland desktop.

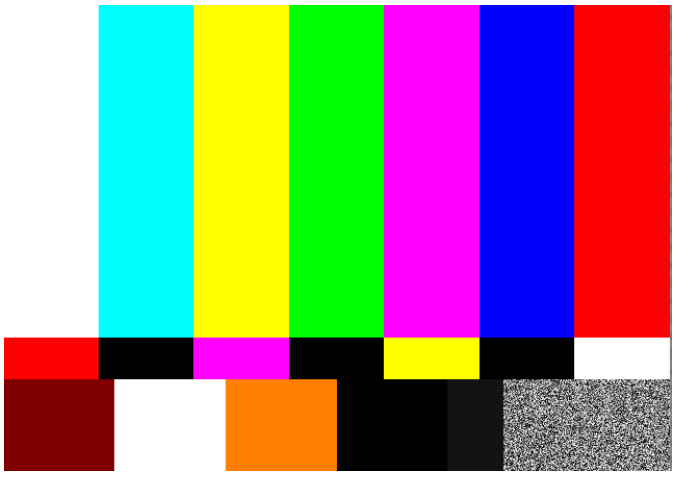


Figure 2.3 Image Display result

2.2.1.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

2.2.1.4 Special instruction

- Please put the bitmap image in `/home/root` location (on board).

2.2.2 Serial Communication

Send and receive data to/from PC via serial port.

2.2.2.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>

#define LENGTH_OF_SENDING_STRING 6                /* length of sending string */
#define LENGTH_OF_SENDING_STRING_WITH_LINE_FEED 7 /* length of sending string with Line Feed */
#define LENGTH_OF_RECEIVING_STRING 9              /* length of receiving string */
#define LINE_FEED 0x0a                            /* Line Feed */
#define GET_SERIAL_PORT_COMMAND    "dmesg | grep ttyS  > /home/root/serial_info.txt"
#define SERIAL_PORT_INFO_FILE      "/home/root/serial_info.txt"

static void get_serial_port(char* serial_port) {

    /* Initial variables */
    char *line = NULL;
    size_t len, read;
    FILE *fp;

    system(GET_SERIAL_PORT_COMMAND);
    fp = fopen(SERIAL_PORT_INFO_FILE, "rt");

    if(fp == NULL) {
        printf("Can't open serial info file.\n");
        exit(1);
    } else {
        while((read = getline(&line, &len, fp)) != -1 ) {
            char* p_start_serial   = strstr(line, "serial");
            if(p_start_serial != NULL) {
                char* p_end_serial     = strstr(line, "at");
                strcpy(serial_port, "/dev/");
                strncat(serial_port, p_start_serial+8, p_end_serial - 9 - p_start_serial);
            }
        }
    }
    fclose(fp);
}

void openport(void);
void readport(void);
int fd;

/* Open serial port */
void openport(void) {
    char device[50];
    struct termios oldtp, newtp;

    char serial_port[20];
    memset(serial_port, '\0', sizeof(serial_port));
    get_serial_port(serial_port);
    printf("Opening serial port: %s \n", serial_port);

    fd = open(serial_port, O_RDWR | O_NOCTTY | O_NDELAY);    /* File descriptor for the port */
    write(fd, "Send: ", LENGTH_OF_SENDING_STRING);
```

```
    if (fd < 0) {
        perror(device);         /* Could not open the port */
    }

    /* Reading data from the Port*/
    fcntl(fd, F_SETFL, 0);
    tcgetattr(fd, &oldtp);      /* save current serial port settings */
    bzero(&newtp, sizeof(newtp));

    /* Set the baud rates to 115200 */
    cfsetispeed(&newtp, B115200);
    cfsetospeed(&newtp, B115200);

    newtp.c_cflag = CRTSCTS | CS8 | CLOCAL | CREAD;     /* Enable receiver and set local mode */
    newtp.c_iflag = IGNPAR | ICRNL;
    newtp.c_oflag = 0;
    newtp.c_lflag = 0;

    newtp.c_cc[VINTR] = 0;      /* Ctrl-c */
    newtp.c_cc[VQUIT] = 0;      /* Ctrl-\ */
    newtp.c_cc[VERASE] = 0;     /* del */
    newtp.c_cc[VKILL] = 0;      /* @ */
    newtp.c_cc[VEOF] = 0;       /* Ctrl-d */
    newtp.c_cc[VTIME] = 0;      /* inter-character timer unused */
    newtp.c_cc[VMIN] = 0;       /* blocking or non blocking read until 1 character arrives */
}

/* Read/Write data from the port */
void readport(void) {
    char buff;
    char cmd[512];
    int n1, n2;
    int count = LENGTH_OF_RECEIVING_STRING;

    strcpy(cmd, "Receive: ");
    while (1) {
        n1 = read(fd, &buff, 1);     /* Read each byte */
        if (n1 < 0) {
            fputs("Read failed!\n", stderr);
            break;
        } else if (n1 == 0)
            break;
        cmd[count] = buff;
        count++;
        if (buff == LINE_FEED) {
            n2 = write(fd, cmd, count);
            if (n2 < 0)
                fputs("Write() of bytes failed!\n", stderr);
            write(fd, "\nSend: ", LENGTH_OF_SENDING_STRING_WITH_LINE_FEED);
            count = LENGTH_OF_RECEIVING_STRING;
        }
    }
    close(fd);                /* Close a serial port */
}

int main(int argc, char **argv) {
    openport();
    readport();
    return 0;
}
```

Walkthrough:

Get serial port

```
static void get_serial_port(char* serial_port);
```

This function gets a serial port (such as: /dev/ttySC0, or /dev/ttySC16...) and stores it in the `serial_port` parameter.

```
#define GET_SERIAL_PORT_COMMAND    "dmesg | grep ttyS  > /home/root/serial_info.txt"
system(GET_SERIAL_PORT_COMMAND);
```

This code block executes `GET_SERIAL_PORT_COMMAND` which dumps all messages related to the `ttyS` text to `/home/root/serial_info.txt`

```
#define SERIAL_PORT_INFO_FILE      "/home/root/serial_info.txt"
fp = fopen(SERIAL_PORT_INFO_FILE, "rt");

while((read = getline(&line, &len, fp)) != -1 ) {
    char* p_start_serial   = strstr(line, "serial");
    if(p_start_serial != NULL) {
        char* p_end_serial     = strstr(line, "at");
        strcpy(serial_port, "/dev/");
        strncat(serial_port, p_start_serial+8, p_end_serial - 9 - p_start_serial);
    }
}

fclose(fp);
```

Let's open `serial_info.txt` to extract the serial port. For each line in this file, if the application detects `serial` text in it, it will extract the serial port by using `strcpy()` and `strncat()` function and store it in `serial_port` parameter.

Open and configure serial port

```
void openport(void);
```

This function opens and configures the serial port.

```
fd = open(serial_port, O_RDWR | O_NOCTTY | O_NDELAY);    /* File descriptor for the port */
```

The `open()` function opens the `serial_port` with the following options:

- The `O_RDWR` flag tells Linux that this program would like to read and write to serial_port.
- The `O_NOCTTY` flag tells Linux that this program doesn't want to be the "controlling terminal" for that port. If you don't specify this then any input (such as keyboard abort signals and so forth) will affect your process. Programs like `getty` use this feature when starting the login process, but normally a user program does not want this behavior.
- The `O_NDELAY` flag tells Linux that this program doesn't care what state the `DCD` signal line is in - whether the other end of the port is up and running. If you do not specify this flag, your process will be put to sleep until the `DCD` signal line is the space voltage.

```
fcntl(fd, F_SETFL, 0);
```

This function sets the file status flag of `fd` to `0`. The file access mode and file creation flag are now ignored. It also resets every other flags: no append, no async, no direct, no atime, and no non-blocking.

```
bzero(&newtp, sizeof(newtp));
```

This function initializes `newtp` variable to contain settings (such as: baud rate) for the `serial_port`.

```
cfsetispeed(&newtp, B115200);
```

This function sets the input baud rate stored in `newtp` variable to `115200`.

```
cfsetospeed(&newtp, B115200);
```

This function sets the output baud rate stored in `newtp` variable to `115200`.

Send/receive data to/from serial port

```
void readport(void) {
    char buff;
    char cmd[512];
    int n1, n2;
    int count = LENGTH_OF_RECEIVING_STRING;

    strcpy(cmd, "Receive: ");
    while (1) {
        n1 = read(fd, &buff, 1);      /* Read each byte */
        if (n1 < 0) {
            fputs("Read failed!\n", stderr);
            break;
        } else if (n1 == 0)
            break;
        cmd[count] = buff;
        count++;
        if (buff == LINE_FEED) {
            n2 = write(fd, cmd, count);
            if (n2 < 0)
                fputs("Write() of bytes failed!\n", stderr);
            write(fd, "\nSend: ", LENGTH_OF_SENDING_STRING_WITH_LINE_FEED);
            count = LENGTH_OF_RECEIVING_STRING;
        }
    }
    close(fd);                /* Close a serial port */
}
```

The application receives 1 character (`buff`) at a time and put it to `cmd` array. If the `buff` is `LINE_FEED` (`0x0a`), we will send `cmd` array back to the `serial_port` (`fd`).

Sending data is easy. We just use the `write()` function. It returns the number of bytes sent or `-1` if an error occurred. The only error we will run into usually is `EIO` when a modem or data link drops the `Data Carrier Detect` (`DCD`) line. This condition will persist until you close the port.

To close the serial port, just use the `close()` function. This will set the `DTR` signal low which causes most modems to hang up.

2.2.2.2 Result

Can read/write data from serial port.

2.2.2.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3. GStreamer applications

GStreamer is a library for constructing graphs of media-handling components. The applications it supports range from simple Ogg/Vorbis playback, audio/video streaming to complex audio (mixing) and video (non-linear editing) processing.

GStreamer is released under the LGPL. This section explains how to create and run the GStreamer applications on Wayland Window System. The applications can take advantage of codec, filter technology, and hardware processing of RZ/G2 SoC by using Renesas GStreamer elements: `omxh264dec`, `omxh264enc`, `omxh265dec`, `vspfilter`, `vspmfilter`, `waylandsink`.

3.1 Hello World

GStreamer is a framework designed to handle multimedia flows. Media data travels from *source* elements (the producers) to *sink* elements (the consumers), passing through a series of intermediate elements performing all kinds of tasks. The set of all interconnected elements is called a *pipeline*.

Let's get started with this application. Instead of printing “hello world”, we are going to create a sound and send it to the audio output jack. You need to connect *PCM5102A DAC Sound Card* to the board, then plug a speaker or headphone in it to listen.

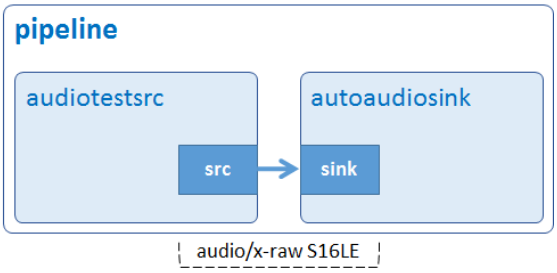


Figure 3.1 Hello World pipeline

3.1.1 Development Environment

- GStreamer: 1.12.2 (edited by Renesas).

Please refer to section **2.1.1** to prepare equipment for GStreamer applications.

3.1.2 Application content

Content of **main.c**:

```
#include <gst/gst.h>

int
main (int argc, char *argv[])
{
  GstElement *pipeline;
  GstBus *bus;
  GstMessage *msg;

  /* Initialization */
  gst_init (&argc, &argv);

  pipeline =
     gst_parse_launch ("audiotestsrc num-buffers=100 ! autoaudiosink", NULL);

  /* Set the pipeline to "playing" state */
  g_print ("Now playing: audiotestsrc\n");
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
```

```
  /* Wait for EOS */
  g_print ("Running...\n");
  bus = gst_element_get_bus (pipeline);
  msg =
      gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
      GST_MESSAGE_ERROR | GST_MESSAGE_EOS);

  /* Note that because input timeout is GST_CLOCK_TIME_NONE,
     the gst_bus_timed_pop_filtered() function will block forever until a
     matching message was posted on the bus (GST_MESSAGE_ERROR or
     GST_MESSAGE_EOS). */
  if (msg != NULL) {
    gst_message_unref (msg);
  }

  /* Out of the main loop, clean up nicely */
  gst_object_unref (bus);
  g_print ("Returned, stopping playback...\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);
  g_print ("Deleting pipeline...\n");
  gst_object_unref (GST_OBJECT (pipeline));
  g_print ("Completed. Goodbye!\n");

  return 0;
}
```

Walkthrough:

Include header file

```
#include <gst/gst.h>
```

You need to include `gst/gst.h` header file so that all functions and objects of GStreamer are properly defined.

Initialize GStreamer

```
gst_init (&argc, &argv);
```

The first thing that always needs to do is initializing GStreamer library by calling `gst_init()`. This function will initialize all internal structures, check what plug-ins are available, and execute any command-line options intended for GStreamer. The GStreamer command-line options can be passed as application arguments.

Create pipeline

```
pipeline = gst_parse_launch ("audiotestsrc num-buffers=100 ! autoaudiosink", NULL);
```

In GStreamer, you usually build the pipeline by manually assembling the individual elements like the sample applications in the next section. For this application, the pipeline is easy and you do not need any advanced features so you can take the shortcut: `gst_parse_launch()`. This function takes a textual representation of a pipeline and turns it into an actual pipeline.

Element `audiotestsrc` generates basic audio signals. It supports several different waveforms and allows to set the frequency and volume. The number of buffers to output before sending `EOS` (`End-of-Stream`) signal is set to `100`. If not, the audio will not stop unless you press Ctrl-C to terminate the program.

Element `autoaudiosink` is an audio sink that automatically detects an appropriate audio sink to use. In RZ/G2 platform, the audio sink is `alsasink`.

Play pipeline

```
gst_element_set_state (pipeline, GST_STATE_PLAYING);
```

Every pipeline has an associated state. To start audio playback, the pipeline needs to be set to `PLAYING` state.

Wait until error or EOS

```
bus = gst_element_get_bus (pipeline);
msg = gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE, GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
```

`gst_element_get_bus()` retrieves the pipeline's bus, then `gst_bus_timed_pop_filtered()` will block until you receive either an error or `EOS` through that bus.

Basically, the bus is an object responsible for delivering the generated messages (`GstMessage`) from the elements to the application. Note that the actual streaming of media is done in another thread. The application can always be stopped by pressing Ctrl-C in the console.

Clean up

```
if (msg != NULL) {
  gst_message_unref (msg);
}

/* Out of the main loop, clean up nicely */
gst_object_unref (bus);
gst_element_set_state (pipeline, GST_STATE_NULL);
gst_object_unref (GST_OBJECT (pipeline));
```

At last, we need to do tidy up correctly after the pipeline ends:

- `gst_bus_timed_pop_filtered()` returns a message (`GstMessage`) which needs to be freed with `gst_message_unref()`.
- Setting the pipeline to the `NULL` state will make sure it frees any resources it has allocated.
- `gst_element_get_bus()` adds a reference to the bus that must be freed with `gst_object_unref()`.

Note:

Always read the documentation of the functions you use, to know if you should free the objects they return after using them.

3.1.3 How to build and run Gstreamer application

This section shows how to cross-compile and deploy GStreamer *hello world* application.

3.1.3.1 How to extract SDK

Step 1. Install toolchain on a Host PC:

```
$  sudo sh ./poky-glibc-x86_64-core-image-weston-sdk-aarch64-toolchain-2.4.3.sh
```

Note:

sudo is optional in case user wants to extract SDK into a restricted directory (such as: /opt/)

If the installation is successful, the following messages will appear:

```
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
 $ . /opt/poky/ek874/environment-setup-aarch64-poky-linux
 $ . /opt/poky/ek874/environment-setup-armv7vehf-neon-pokymllib32-linux-gnueabi
```

Figure 3.2 SDK is successfully set up

Step 2. Set up cross-compile environment:

```
$  source /<Location in which SDK is extracted>/environment-setup-aarch64-poky-linux
```

Note:

User needs to run the above command once for each login session.

3.1.3.2 How to build and run GST application

Step 1. Go to *gst-helloworld* directory:

```
$  cd $WORK/gst-helloworld
```

Step 2. Cross-compile:

```
$  make
```

Step 3. Copy all files inside this directory to `/usr/share` directory on the target board:

```
$  scp -r $WORK/gst-helloworld/ <username>@<board IP>:/usr/share/
```

Step 4. Run the application:

```
/usr/share/gst-helloworld/gst-helloworld
```

3.2 Application samples

The following table shows multimedia applications in RZ/G2 platform ranging from playing, recording, scaling, streaming audio/video, to displaying multiple videos on multiple monitors.

Application name	Description
Audio Play	Play an Ogg/Vorbis audio file.
Video Play	Play an H.264/H.265 video file.
Audio Encode	Encode audio data from F32LE raw format to Ogg/Vorbis format.
Video Encode	Encode video data from NV12 raw format to H.264 format.
Audio Record	Record raw data from USB microphone, then store it in Ogg container.
Video Record	Display and record raw video from USB/MIPI camera, then store it in MP4 container.
Audio Video Record	Record raw data from USB microphone and USB/MIPI camera at the same time, then store them in MKV container.
Receive Streaming Video	Receive and display streaming video.
Send Streaming Video	Send streaming video.
Video Scale	Scale down an H.264 video, then store it in MP4 container.
Audio Player	A simple text-based Ogg/Vorbis audio player.
Video Player	A simple text-based MP4 video player.
Audio-Video Play	Play H.264 video and Ogg/Vorbis audio file independently.
File Play	Play an MP4 file.
Multiple Displays 1	Display 1 H.264/H.265 video simultaneously on HDMI and LCD monitors.
Multiple Displays 2	Display 2 H.264/H.265 videos simultaneously on HDMI and LCD monitors.
Overlapped Display	Display 3 overlapping H.264 videos.

Table 3.1 GStreamer sample applications

3.2.1 Audio Play

Play an Ogg/Vorbis audio file.

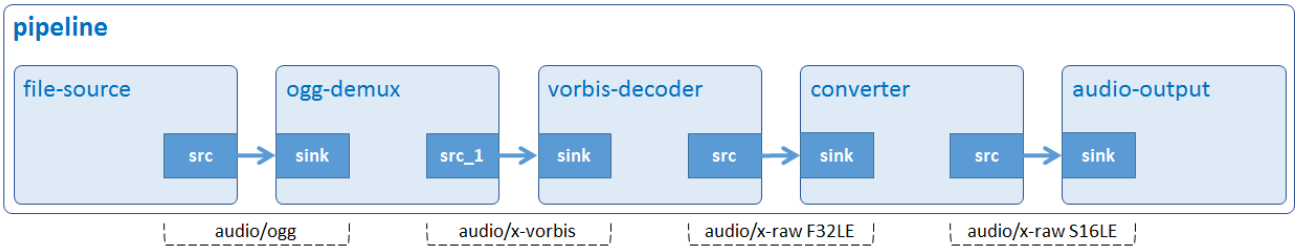


Figure 3.3 Audio Play pipeline

3.2.1.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <gst/gst.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <libgen.h>

#define FORMAT      "S16LE"
#define ARG_PROGRAM_NAME 0
#define ARG_INPUT 1
#define ARG_COUNT 2

static void
on_pad_added (GstElement * element, GstPad * pad, gpointer data)
{
  GstPad *sinkpad;
  GstElement *decoder = (GstElement *) data;

  /* We can now link this pad with the vorbis-decoder sink pad */
  g_print ("Dynamic pad created, linking demuxer/decoder\n");

  sinkpad = gst_element_get_static_pad (decoder, "sink");

  gst_pad_link (pad, sinkpad);

  gst_object_unref (sinkpad);
}

/*
 *
 * name: is_file_exist
 * Check if the input file exists or not?
 *
 */
bool
is_file_exist(const char *path)
{
  bool result = false;
  FILE *fd = NULL;

  if (path != NULL)
  {
    fd = fopen(path, "r");
```

```
    if (fd != NULL)
    {
      result = true;
      fclose(fd);
    }
  }

  return result;
}

/* get the extention of filename */
const char* get_filename_ext (const char *filename) {
  const char* dot = strrchr (filename, '.');
  if ((!dot) || (dot == filename))
  {
    g_print ("Invalid input file.\n");
    return "";
  }
  else
  {
    return dot + 1;
  }
}

int
main (int argc, char *argv[])
{
  GstElement *pipeline, *source, *demuxer, *decoder, *conv, *capsfilter, *sink;
  GstCaps *caps;
  GstBus *bus;
  GstMessage *msg;
  const char* ext;
  char* file_name;

  if (argc != ARG_COUNT)
  {
    g_print ("Invalid arugments.\n");
    g_print ("Format: %s <path to file> \n", argv[ARG_PROGRAM_NAME]);
    return -1;
  }

  const gchar *input_file = argv[ARG_INPUT];
  if (!is_file_exist(input_file))
  {
    g_printerr("Cannot find input file: %s. Exiting.\n", input_file);
    return -1;
  }

  file_name = basename (argv[ARG_INPUT]);
  ext = get_filename_ext (file_name);

  if (strcasecmp ("ogg", ext) != 0)
  {
    g_print ("Invalid extention.This application is used to play an ogg file.\n");
    return -1;
  }

  /* Initialization */
  gst_init (&argc, &argv);

  /* Create GStreamer elements */
  pipeline = gst_pipeline_new ("audio-play");
  source = gst_element_factory_make ("filesrc", "file-source");
  demuxer = gst_element_factory_make ("oggdemux", "ogg-demuxer");
  decoder = gst_element_factory_make ("vorbisdec", "vorbis-decoder");
  conv = gst_element_factory_make ("audioconvert", "converter");
  capsfilter = gst_element_factory_make ("capsfilter", "conv_capsfilter");
  sink = gst_element_factory_make ("autoaudiosink", "audio-output");

  if (!pipeline || !source || !demuxer || !decoder || !capsfilter || !conv
      || !sink) {
    g_printerr ("One element could not be created. Exiting.\n");
    return -1;
```

```
}

/* Set up the pipeline */

/* Set the input filename to the source element */
g_object_set (G_OBJECT (source), "location", input_file, NULL);

/* Set the caps option to the caps-filter element */
caps =
    gst_caps_new_simple ("audio/x-raw", "format", G_TYPE_STRING, FORMAT,
    NULL);
g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);


/* Add the elements into the pipeline */
/* file-source | ogg-demuxer | vorbis-decoder | converter | alsa-output */
gst_bin_add_many (GST_BIN (pipeline),
    source, demuxer, decoder, conv, capsfilter, sink, NULL);

/* Link the elements together */
/* file-source -> ogg-demuxer -> vorbis-decoder -> converter -> alsa-output */
if (gst_element_link (source, demuxer) != TRUE) {
  g_printerr ("Elements could not be linked.\n");
  gst_object_unref (pipeline);
  return -1;
}
if (gst_element_link_many (decoder, conv, capsfilter, sink, NULL) != TRUE) {
  g_printerr ("Elements could not be linked.\n");
  gst_object_unref (pipeline);
  return -1;
}

g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added), decoder);

/* note that the demuxer will be linked to the decoder dynamically.
   The reason is that Ogg may contain various streams (for example
   audio and video). The source pad(s) will be created at run time,
   by the demuxer when it detects the amount and nature of streams.
   Therefore we connect a callback function which will be executed
   when the "pad-added" is emitted. */

/* Set the pipeline to "playing" state */
g_print ("Now playing file %s:\n", input_file);
if (gst_element_set_state (pipeline,
       GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
  g_printerr ("Unable to set the pipeline to the playing state.\n");
  gst_object_unref (pipeline);
  return -1;
}

/* Waiting */
g_print ("Running...\n");
bus = gst_element_get_bus (pipeline);
msg =
    gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
    GST_MESSAGE_ERROR | GST_MESSAGE_EOS);

/* Note that because input timeout is GST_CLOCK_TIME_NONE,
   the gst_bus_timed_pop_filtered() function will block forever untill a
   matching message was posted on the bus (GST_MESSAGE_ERROR or
   GST_MESSAGE_EOS). */

/* Playback end. Clean up nicely */
if (msg != NULL) {
  GError *err;
  gchar *debug_info;

  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_ERROR:
      gst_message_parse_error (msg, &err, &debug_info);
      g_printerr ("Error received from element %s: %s.\n",
          GST_OBJECT_NAME (msg->src), err->message);
      g_printerr ("Debugging information: %s.\n",
          debug_info ? debug_info : "none");
```

```
      g_clear_error (&err);
      g_free (debug_info);
      break;
    case GST_MESSAGE_EOS:
      g_print ("End-Of-Stream reached.\n");
      break;
    default:
      /* We should not reach here because we only asked for ERRORs and EOS */
      g_printerr ("Unexpected message received.\n");
      break;
  }
  gst_message_unref (msg);
 }

 gst_object_unref (bus);
 g_print ("Returned, stopping playback...\n");
 gst_element_set_state (pipeline, GST_STATE_NULL);
 g_print ("Deleting pipeline...\n");
 gst_object_unref (GST_OBJECT (pipeline));
 g_print ("Program end!\n");
 return 0;
}
```

Walkthrough:

Input location

```
if (argc != ARG_COUNT)
{
 g_print ("Invalid arugments.\n");
 g_print ("Format: %s <path to file> \n", argv[ARG_PROGRAM_NAME]);
 return -1;
}
```

This application accepts one command-line argument which points to an Ogg/Vorbis file.

Create new pipeline

```
pipeline = gst_pipeline_new ("audio-play");
```

The `gst_pipeline_new()` function creates a new empty pipeline which is the top-level container with clocking and bus management functionality.

Create elements

```
source = gst_element_factory_make ("filesrc", "file-source");
demuxer = gst_element_factory_make ("oggdemux", "ogg-demuxer");
decoder = gst_element_factory_make ("vorbisdec", "vorbis-decoder");
conv = gst_element_factory_make ("audioconvert", "converter");
capsfilter = gst_element_factory_make ("capsfilter", "conv_capsfilter");
sink = gst_element_factory_make ("autoaudiosink", "audio-output");
```

To play an Ogg/Vorbis audio file, the following elements are used:

- Element `filesrc` reads data from a local file.
- Element `oggdemux` de-multiplexes Ogg files into their encoded audio and video components. In this case, only audio stream is available.
- Element `vorbisdec` decompresses a Vorbis stream to raw audio.
- Element `audioconvert` converts raw audio buffers between various possible formats depending on the given source pad and sink pad it links to.
- Element `capsfilter` specifies raw audio format `S16LE`.
- Element `autoaudiosink` automatically detects an appropriate audio sink (such as: `alsasink`).

Check elements

```
if (!pipeline || !source || !demuxer || !decoder || !capsfilter || !conv || !sink) {
  g_printerr ("One element could not be created. Exiting.\n");
  return -1;
}
```

If either `gst_element_factory_make()` or `gst_pipeline_new()` is unable to create an element, NULL will be returned. Next, the application prints error and exit.

Note that this statement is used for reference purpose only. If an element cannot be created, the application should use `gst_object_unref()` to free all created elements.

Set element's properties

```
g_object_set (G_OBJECT (source), "location", input_file, NULL);
```

The `g_object_set()` function is used to set the `location` property of `filesrc` (`source`) to an Ogg/Vorbis file.

```
caps = gst_caps_new_simple ("audio/x-raw", "format", G_TYPE_STRING, FORMAT, NULL);

g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);
```

Target audio format `S16LE` is added to a new cap (`gst_caps_new_simple`) which is then added to `caps` property of `capsfilter` (`g_object_set`). Then, `audioconvert` will use this element to convert audio format `F32LE` (of `vorbisdec`) to `S16LE` which is supported by sound driver.

Note that the `caps` should be freed with `gst_caps_unref()` if it is not used anymore.

Build pipeline

```
gst_bin_add_many (GST_BIN (pipeline), source, demuxer, decoder, conv, capsfilter, sink, NULL);

gst_element_link (source, demuxer);
gst_element_link_many (decoder, conv, capsfilter, sink, NULL);
```

This code block adds all elements to pipeline and then links them into separated groups as below:

- Group #1: `source` and `demuxer`.
- Group #2: `decoder`, `conv`, `capsfilter`, and `sink`.

The reason for the separation is that `demuxer` (`oggdemux`) contains no source pads at this point, so it cannot link to `decoder` (`vorbisdec`) until `pad-added` signal is emitted (see below).

Note that the order counts, because links must follow the data flow (this is, from source elements to sink elements).

Signal

```
g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added), decoder);
```

Signals are a crucial point in GStreamer. They allow you to be notified (by means of a callback) when something interesting has happened. Signals are identified by a name, and each element has its own signals.

In this application, `g_signal_connect()` is used to bind `pad-added` signal of `oggdemux` (`demuxer`) to callback function `pad_added_handler()` and `decoder` (`vorbisdec`). GStreamer does nothing with this element, it just forwards it to the callback.

Link oggdemux to vorbisdec

When `oggdemux` (`demuxer`) finally has enough information to start producing data, it will create source pads, and trigger the `pad-added` signal. At this point our callback will be called:

```
static void on_pad_added (GstElement * element, GstPad * pad, gpointer data);
```

The element parameter is the GstElement which triggered the signal. In this application, it is oggdemux. The first parameter of a signal handler is always the object that has triggered it.

The pad parameter is the GstPad that has just been added to the oggdemux. This is usually the pad to which we want to link.

The data parameter is the decoder (vorbisdec) we provided earlier when attaching to the signal.

```
GstPad *sinkpad;
GstElement *decoder = (GstElement *) data;

sinkpad = gst_element_get_static_pad (decoder, "sink");
gst_pad_link (pad, sinkpad);
gst_object_unref (sinkpad);
```

The application retrieves the sink pad of vorbisdec using gst_element_get_static_pad(), then uses gst_pad_link() to connect it to the source pad of oggdemux.

Note that sinkpad should be freed with gst_caps_unref() if it is not used anymore.

Play pipeline

```
gst_element_set_state (pipeline, GST_STATE_PLAYING);
```

Every pipeline has an associated state. To start audio playback, the pipeline needs to be set to PLAYING state.

Wait until error or EOS

```
bus = gst_element_get_bus (pipeline);
msg = gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE, GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
```

Now, the pipeline is running. gst_bus_timed_pop_filtered() waits for execution to end and returns a GstMessage which is either an error or an EOS (End-of-Stream) message.

Handle messages

```
if (msg != NULL) {
  GError *err;
  gchar *debug_info;

  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_ERROR:
      gst_message_parse_error (msg, &err, &debug_info);
      g_printerr ("Error received from element %s: %s.\n",
          GST_OBJECT_NAME (msg->src), err->message);
      g_printerr ("Debugging information: %s.\n",
          debug_info ? debug_info : "none");
      g_clear_error (&err);
      g_free (debug_info);
      break;
    case GST_MESSAGE_EOS:
      g_print ("End-Of-Stream reached.\n");
      break;
    default:
      /* We should not reach here because we only asked for ERRORs and EOS */
      g_printerr ("Unexpected message received.\n");
      break;
  }
  gst_message_unref (msg);
}
```

If the message is GST_MESSAGE_ERROR, the application will print the error message and debugging information.

If the message is GST_MESSAGE_EOS, the application will inform to users that the audio is finished.

After the message is handled, it should be un-referred by gst_message_unref().

Clean up

```
gst_object_unref (bus);

gst_element_set_state (pipeline, GST_STATE_NULL);
gst_object_unref (GST_OBJECT (pipeline));
```

The `gst_element_get_bus()` function added the `bus` that must be freed with `gst_object_unref()`.
Next, setting the `pipeline` to the `NULL` state will make sure it frees any resources it has allocated.
Finally, un-referencing the pipeline will destroy it, and all its contents.

3.2.1.2 Result

Can play audio through speaker/headphone.

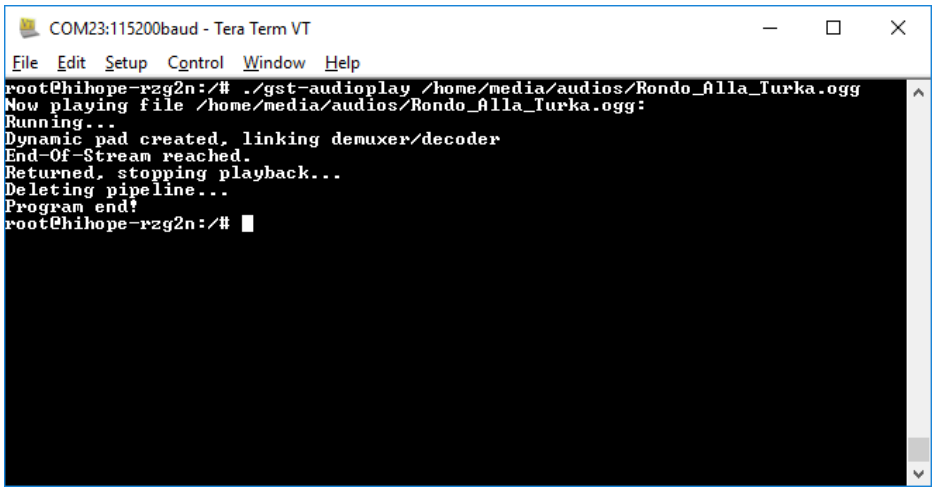


Figure 3.4 Audio Play result

3.2.1.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.1.4 Special instruction

- Download the input file at:
  https://upload.wikimedia.org/wikipedia/commons/b/bd/Rondo_Alla_Turka.ogg
- To set the playback volume: please use the `alsamixer` or `amixer` tool. It depends on the audio system on the specific board. Reference: https://en.wikipedia.org/wiki/Alsamixer

3.2.2 Video Play

Play an H.264/H.265 video file.

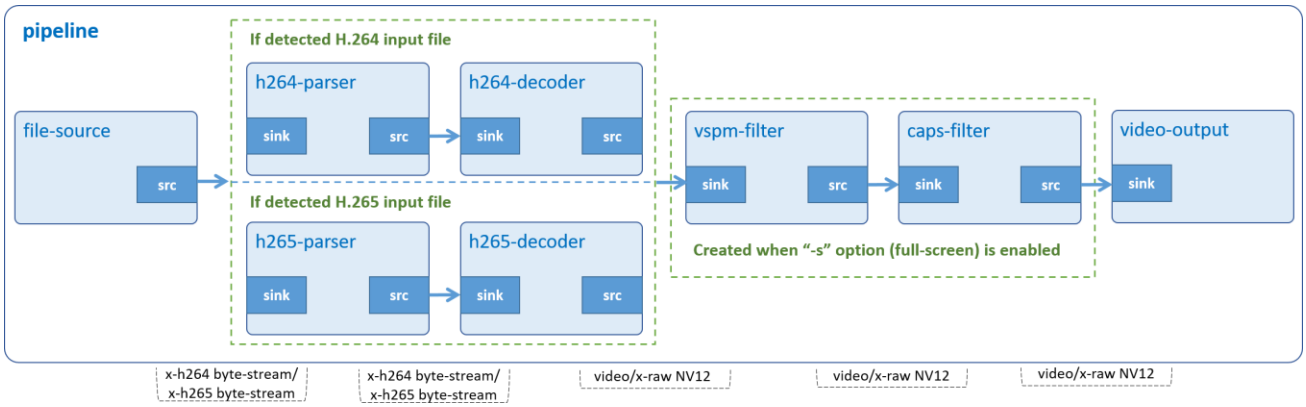


Figure 3.5 Video Play pipeline

3.2.2.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <stdio.h>
#include <string.h>
#include <gst/gst.h>
#include <stdlib.h>
#include <stdbool.h>
#include <wayland-client.h>
#include <strings.h>
#include <libgen.h>

#define ARG_PROGRAM_NAME 0
#define ARG_INPUT 1
#define ARG_SCALE 2
#define ARG_COUNT 3

int
main (int argc, char *argv[])
{
  struct wayland_t *wayland_handler = NULL;
  struct screen_t *main_screen = NULL;

  GstElement *pipeline, *source, *parser, *decoder, *filter, *capsfilter, *sink;
  GstCaps *caps;
  GstBus *bus;
  GstMessage *msg;
  bool fullscreen = false;
  const char* ext;
  char* file_name;

  if ((argc > ARG_COUNT) || (argc == 1) || ((argc == ARG_COUNT) && (strcmp (argv[ARG_SCALE], "-s"))))
  {
    g_print ("Error: Invalid arugments.\n");
    g_print ("Usage: %s <path to H264/H265 file> [-s]\n", argv[ARG_PROGRAM_NAME]);
    return -1;
  }

  /* Check -s option */
  if (argc == ARG_COUNT) {
```

```
  if (strcmp (argv[ARG_SCALE], "-s") == 0) {
    fullscreen = true;
  }
}

/* Get a list of available screen */
wayland_handler = get_available_screens();

/* Get main screen */
main_screen = get_main_screen(wayland_handler);
if (main_screen == NULL)
{
  g_printerr("Cannot find any available screens. Exiting.\n");
  destroy_wayland(wayland_handler);
  return -1;
}

/* Initialization */
gst_init (&argc, &argv);

const gchar *input_file = argv[ARG_INPUT];

if (!is_file_exist(input_file))
{
  g_printerr("Cannot find input file: %s. Exiting.\n", input_file);
  destroy_wayland(wayland_handler);
  return -1;
}

file_name = basename ((char*) input_file);
ext = get_filename_ext (file_name);

/* Check the extension and create parser, decoder */
if (strcasecmp ("h264", ext) == 0) {
  parser = gst_element_factory_make ("h264parse", "h264-parser");
  decoder = gst_element_factory_make ("omxh264dec", "h264-decoder");
}
else if (strcasecmp ("h265", ext) == 0)
{
  parser = gst_element_factory_make ("h265parse", "h265-parser");
  decoder = gst_element_factory_make ("omxh265dec", "h265-decoder");
}
else
{
  g_print ("Unsupported video type. H264/H265 format is required.\n");
  destroy_wayland(wayland_handler);
  return -1;
}

/* Create gstreamer elements */
pipeline = gst_pipeline_new ("video-play");
source = gst_element_factory_make ("filesrc", "file-source");
sink = gst_element_factory_make ("waylandsink", "video-output");

if (!pipeline || !source || !parser || !decoder || !sink) {
   g_printerr ("One element could not be created. Exiting.\n");
   destroy_wayland(wayland_handler);
   return -1;
}

/* Add all elements into the pipeline */
gst_bin_add_many (GST_BIN (pipeline), source,
    parser, decoder, sink, NULL);

/* Set input video file for source element */
g_object_set (G_OBJECT (source), "location", input_file, NULL);

/* Set position for displaying (0, 0) */
g_object_set (G_OBJECT (sink), "position-x", main_screen->x, "position-y",
    main_screen->y, NULL);

if (!fullscreen) {
  /* Link the elements together */
  if (gst_element_link_many (source, parser,
```

```
          decoder, sink, NULL) != TRUE) {
    g_printerr ("Elements could not be linked.\n");
    gst_object_unref (pipeline);
    destroy_wayland(wayland_handler);
    return -1;
  }
}
else {
  /* Create vspm-filter and caps-filter */
  filter = gst_element_factory_make ("vspmfilter", "vspm-filter");
  capsfilter = gst_element_factory_make ("capsfilter", "caps-filter");

  if (!filter || !capsfilter) {
    g_printerr ("One element could not be created. Exiting.\n");
    gst_object_unref (pipeline);
    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Set property "dmabuf-use" of vspmfilter to true */
  /* Without it, waylandsink will display broken video */
  g_object_set (G_OBJECT (filter), "dmabuf-use", TRUE, NULL);

  /* Create simple cap which contains video's resolution */
  caps = gst_caps_new_simple ("video/x-raw",
     "width", G_TYPE_INT, main_screen->width,
     "height", G_TYPE_INT, main_screen->height, NULL);

  /* Add cap to capsfilter element */
  g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
  gst_caps_unref (caps);

  /* Add filter, capsfilter into the pipeline */
  gst_bin_add_many (GST_BIN (pipeline), filter, capsfilter, NULL);

  /* Link the elements together */
  /* file-source -> parser -> decoder -> vspm-filter -> caps-filter -> video-output */
  if (gst_element_link_many (source, parser, decoder, filter, capsfilter, sink, NULL) != TRUE) {
    g_printerr ("Elements could not be linked.\n");
    gst_object_unref (pipeline);
    destroy_wayland(wayland_handler);
    return -1;
  }
}

/* Set the pipeline to "playing" state */
g_print ("Now playing: %s\n", input_file);
if (gst_element_set_state (pipeline,
       GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
  g_printerr ("Unable to set the pipeline to the playing state.\n");
  gst_object_unref (pipeline);
  destroy_wayland(wayland_handler);
  return -1;
}

g_print ("Running...\n");

/* Wait until error or EOS */
bus = gst_element_get_bus (pipeline);
msg =
    gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
    GST_MESSAGE_ERROR | GST_MESSAGE_EOS);

/* Note that because input timeout is GST_CLOCK_TIME_NONE,
   the gst_bus_timed_pop_filtered() function will block forever until a
   matching message was posted on the bus (GST_MESSAGE_ERROR or
   GST_MESSAGE_EOS). */

if (msg != NULL) {
  GError *err;
  gchar *debug_info;

  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_ERROR:
```

```
      gst_message_parse_error (msg, &err, &debug_info);
      g_printerr ("Error received from element %s: %s.\n",
          GST_OBJECT_NAME (msg->src), err->message);
      g_printerr ("Debugging information: %s.\n",
          debug_info ? debug_info : "none");
      g_clear_error (&err);
      g_free (debug_info);
      break;
    case GST_MESSAGE_EOS:
      g_print ("End-Of-Stream reached.\n");
      break;
    default:
      /* We should not reach here because we only asked for ERRORs and EOS */
      g_printerr ("Unexpected message received.\n");
      break;
  }
  gst_message_unref (msg);
 }

 /* Clean up "wayland_t" structure */
 destroy_wayland(wayland_handler);

 /* Free resources and change state to NULL */
 gst_object_unref (bus);
 g_print ("Returned, stopping playback...\n");
 gst_element_set_state (pipeline, GST_STATE_NULL);
 g_print ("Freeing pipeline...\n");
 gst_object_unref (GST_OBJECT (pipeline));
 g_print ("Completed. Goodbye!\n");

 return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.1 Audio Play.

Command-line argument

```
if ((argc > ARG_COUNT) || (argc == 1) || ((argc == ARG_COUNT) && (strcmp (argv[ARG_SCALE], "-s"))))
{
  g_print ("Error: Invalid arugments.\n");
  g_print ("Usage: %s <path to H264/H265 file> [-s]\n", argv[ARG_PROGRAM_NAME]);
  return -1;
}

/* Check -s option */
if (argc == ARG_COUNT) {
  if (strcmp (argv[ARG_SCALE], "-s") == 0) {
    fullscreen = true;
  }
}
```

This application accepts a command-line argument which points to an H.264/H.265 file.
If “-s” option is enabled, the video will be scaled to full-screen.

Create elements

```
if (strcasecmp ("h264", ext) == 0) {
  parser = gst_element_factory_make ("h264parse", "h264-parser");
  decoder = gst_element_factory_make ("omxh264dec", "h264-decoder");
} else if (strcasecmp ("h265", ext) == 0)
{
  parser = gst_element_factory_make ("h265parse", "h265-parser");
  decoder = gst_element_factory_make ("omxh265dec", "h265-decoder");
}

source = gst_element_factory_make ("filesrc", "file-source");
```

```
sink = gst_element_factory_make ("waylandsink", "video-output");

filter = gst_element_factory_make ("vspmfilter", "vspm-filter");
capsfilter = gst_element_factory_make ("capsfilter", "caps-filter");
```

To play an H.264/H.265 video file, the following elements are used:

- Element `filesrc` reads data from a local file.
- Element `h264parse` parses H.264 stream to AVC format which `omxh264dec` can recognize and process.
- Element `h265parse` parses H.265 stream to HEVC format which `omxh265dec` can recognize and process.
- Element `omxh264dec` decompresses H.264 stream to raw NV12-formatted video.
- Element `omxh265dec` decompresses H.265 stream to raw NV12-formatted video.
- Element `vspmfilter` handles video scaling.
- Element `capsfilter` contains screen resolution so that `vspmfilter` can scale video frames based on this value.
- Element `waylandsink` creates its own window and renders the decoded video frames to that.

Set element's properties

```
g_object_set (G_OBJECT (source), "location", input_file, NULL);
g_object_set (G_OBJECT (sink), "position-x", main_screen->x, "position-y", main_screen->y, NULL);
g_object_set (G_OBJECT (filter), "dmabuf-use", TRUE, NULL);
```

The `g_object_set()` function is used to set some element's properties, such as:

- The `location` property of `filesrc` element which points to an H.264/H.265 video file.
- The `position-x` and `position-y` properties of `waylandsink` element which point to (x, y) coordinate of Wayland desktop.
- The `dmabuf-use` property of `vspmfilter` element which is set to `true`. This disallows `dmabuf` to be output buffer. If it is not set, `waylandsink` will display broken video frames.

```
caps = gst_caps_new_simple ("video/x-raw", "width", G_TYPE_INT, main_screen->width,
                                           "height", G_TYPE_INT, main_screen->height, NULL);

g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);
```

The `gst_caps_new_simple()` function creates a new cap which holds screen resolution. This cap is then added to `caps` property of `capsfilter` (`g_object_set`) so that `vspmfilter` element can use these values to resize video frames to match screen resolution.

Note that the `caps` should be freed with `gst_caps_unref()` if it is not used anymore.

Build pipeline

```
gst_bin_add_many (GST_BIN (pipeline), source, parser, decoder, sink, NULL);

/*Not display video in full-screen*/
gst_element_link_many (source, parser, decoder, sink, NULL);

/*Display video in full-screen*/
gst_bin_add_many (GST_BIN (pipeline), filter, capsfilter, NULL);
gst_element_link_many (source, parser, decoder, filter, capsfilter, sink, NULL);
```

If "-s" option is enabled, the application will add and link GStreamer elements from `source`, `depayloader`, `parser`, `decoder`, `filter` and `capsfilter` to `sink`. Otherwise, it does not consider `filter` and `capsfilter` elements.

Note that the order counts, because links must follow the data flow (this is, from source elements to sink elements).

3.2.2.2 Result

Can display video on the monitor. Note that this application does not output audio.

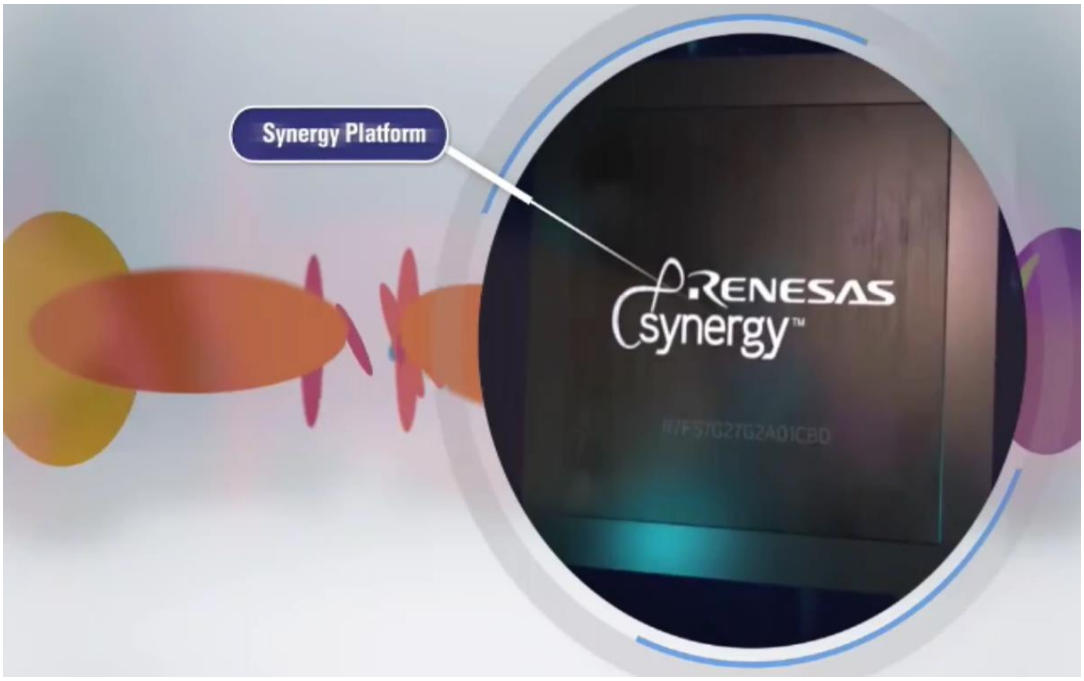


Figure 3.6 Video Play result

3.2.2.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.2.4 Special instruction

- Download input file at:

https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga1.h264

Note:

RZ/G2E platform does not support 2K and 4K video.

3.2.3 Audio Encode

Encode audio data from F32LE raw format to Ogg/Vorbis format.

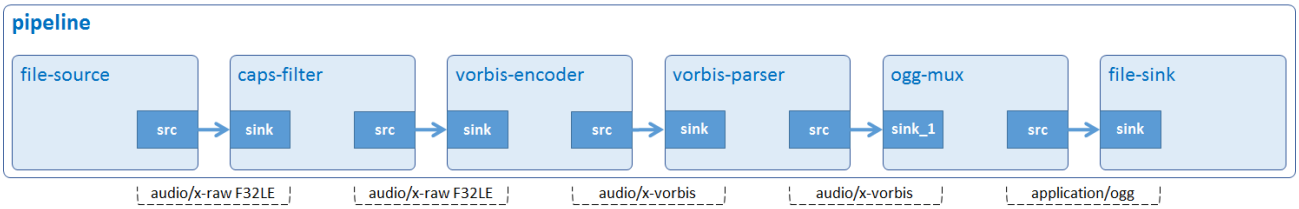


Figure 3.7 Audio Encode pipeline

3.2.3.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <gst/gst.h>
#include <stdio.h>
#include <stdbool.h>

#define BITRATE             128000      /* Bitrate averaging */
#define SAMPLE_RATE         44100       /* Sample rate  of audio file*/
#define CHANNEL             2           /* Channel*/
#define INPUT_FILE          "/home/media/audios/Rondo_Alla_Turka_F32LE_44100_stereo.raw"
#define OUTPUT_FILE         "/home/media/audios/ENCODE_Rondo_Alla_Turka.ogg"
#define FORMAT              "F32LE"

int
main (int argc, char *argv[])
{
  GstElement *pipeline, *source, *capsfilter, *encoder, *parser, *muxer, *sink;
  GstCaps *caps;
  GstBus *bus;
  GstMessage *msg;

  const gchar *input_file = INPUT_FILE;
  const gchar *output_file = OUTPUT_FILE;

  if (!is_file_exist(input_file))
  {
    g_printerr("Cannot find input file: %s. Exiting.\n", input_file);
    return -1;
  }

  /* Initialization */
  gst_init (&argc, &argv);

  /* Create GStreamer elements */
  pipeline = gst_pipeline_new ("audio-encode");
  source = gst_element_factory_make ("filesrc", "file-source");
  capsfilter = gst_element_factory_make ("capsfilter", "caps-filter");
  encoder = gst_element_factory_make ("vorbisenc", "vorbis-encoder");
  parser = gst_element_factory_make ("vorbisparse", "vorbis-parser");
  muxer = gst_element_factory_make ("oggmux", "OGG-muxer");
  sink = gst_element_factory_make ("filesink", "file-output");

  if (!pipeline || !source || !encoder || !capsfilter || !parser || !muxer
      || !sink) {
    g_printerr ("One element could not be created. Exiting.\n");
    return -1;
  }
```

```
/* Set up the pipeline */

/* Set the input filename to the source element */
g_object_set (G_OBJECT (source), "location", input_file, NULL);

/* Set the bitrate to 128kbps to the encode element */
g_object_set (G_OBJECT (encoder), "bitrate", BITRATE, NULL);

/* Set the output filename to the source element */
g_object_set (G_OBJECT (sink), "location", output_file, NULL);

/* Create a simple caps */
caps =
    gst_caps_new_simple ("audio/x-raw",
    "format", G_TYPE_STRING, FORMAT,
    "rate", G_TYPE_INT, SAMPLE_RATE, "channels", G_TYPE_INT, CHANNEL, NULL);

/* Set the caps option to the caps-filter element */
g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);

/* Add all elements into the pipeline */
/* file-source | vorbis-encoder | vorbis-parser | OGG-muxer | file-output */
gst_bin_add_many (GST_BIN (pipeline), source, capsfilter, encoder,
    parser, muxer, sink, NULL);

/* Link the elements together */
if (gst_element_link_many (source, capsfilter, encoder, parser, muxer,
       sink, NULL) != TRUE) {
  g_printerr ("Elements could not be linked.\n");
  gst_object_unref (pipeline);
  return -1;
}

/* Set the pipeline to "playing" state */
g_print ("Now start encode file: %s\n", input_file);
if (gst_element_set_state (pipeline,
        GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
   g_printerr ("Unable to set the pipeline to the playing state.\n");
   gst_object_unref (pipeline);
   return -1;
}

/* Waiting */
g_print ("Encoding...");
bus = gst_element_get_bus (pipeline);
msg =
    gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
    GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
gst_object_unref (bus);

/* Note that because input timeout is GST_CLOCK_TIME_NONE,
   the gst_bus_timed_pop_filtered() function will block forever untill a
   matching message was posted on the bus (GST_MESSAGE_ERROR or
   GST_MESSAGE_EOS). */

/* Encode end. Clean up nicely */
if (msg != NULL) {
  GError *err;
  gchar *debug_info;

   switch (GST_MESSAGE_TYPE (msg)) {
     case GST_MESSAGE_ERROR:
       gst_message_parse_error (msg, &err, &debug_info);
       g_printerr ("Error received from element %s: %s.\n",
           GST_OBJECT_NAME (msg->src), err->message);
       g_printerr ("Debugging information: %s.\n",
           debug_info ? debug_info : "none");
       g_clear_error (&err);
       g_free (debug_info);
       break;
     case GST_MESSAGE_EOS:
       g_print ("End-Of-Stream reached.\n");
       break;
```

```
      default:
        /* We should not reach here because we only asked for ERRORs and EOS */
        g_printerr ("Unexpected message received.\n");
        break;
    }
    gst_message_unref (msg);
  }
  g_print ("Returned, please wait for the writing file process....\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);

  g_print ("Deleting pipeline...\n");
  gst_object_unref (GST_OBJECT (pipeline));
  g_print ("Completed. The output file available at: %s\n", output_file);
  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.1 Audio Play.

Input/output location

```
#define INPUT_FILE       "/home/media/audios/Rondo_Alla_Turka_F32LE_44100_stereo.raw"
#define OUTPUT_FILE      "/home/media/audios/ENCODE_Rondo_Alla_Turka.ogg"
```

Note: You can create input file by following section 3.2.3.4 Special Instruction

Create elements

```
source = gst_element_factory_make ("filesrc", "file-source");
capsfilter = gst_element_factory_make ("capsfilter", "caps-filter");
encoder = gst_element_factory_make ("vorbisenc", "vorbis-encoder");
parser = gst_element_factory_make ("vorbisparse", "vorbis-parser");
muxer = gst_element_factory_make ("oggmux", "OGG-muxer");
sink = gst_element_factory_make ("filesink", "file-output");
```

To encode an audio file to `Vorbis` format, the following elements are used:

- Element `filesrc` reads data from a local file.
- Element `capsfilter` specifies raw audio format, channel, and bitrate.
- Element `vorbisenc` encodes raw float audio into a `Vorbis` stream.
- Element `vorbisparse` parses the header packets of the `Vorbis` stream and put them as the stream header in the caps.
- Element `oggmux` merges streams (audio and/or video) into Ogg files. In this case, only audio stream is available.
- Element `filesink` writes incoming data to a local file.

Set element's properties

```
g_object_set (G_OBJECT (source), "location", input_file, NULL);
g_object_set (G_OBJECT (encoder), "bitrate", BITRATE, NULL);
g_object_set (G_OBJECT (sink), "location", output_file, NULL);
```

The `g_object_set()` function is used to set some element's properties, such as:

- The `location` property of `filesrc` element which points to a raw audio file.
- The `bitrate` property of `vorbisenc` element which is set to `128 Kbps`. Note that this value is just for demonstration purpose only. Users can define other value which will affect output quality.
- The `location` property of `filesink` element which points to an output file.

```
caps = gst_caps_new_simple ("audio/x-raw", "format", G_TYPE_STRING, FORMAT,
                            "rate", G_TYPE_INT, SAMPLE_RATE, "channels", G_TYPE_INT, CHANNEL, NULL);

g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);
```

A `capsfilter` is needed between `filesrc` and `vorbisenc` because the `vorbisenc` element needs to know what raw audio format, sample rate, and channels of the incoming data stream are. In this application, audio file is formatted to `F32LE`, has sample rate `44.1 kHz` and `stereo channel`.

The `gst_caps_new_simple()` function creates a new cap which holds these values. This cap is then added to `caps` property of `capsfilter` element (`g_object_set`).

Note that the `caps` should be freed with `gst_caps_unref()` if it is not used anymore.

3.2.3.2 Result

An Ogg/Vorbis audio file will be generated after running this application.

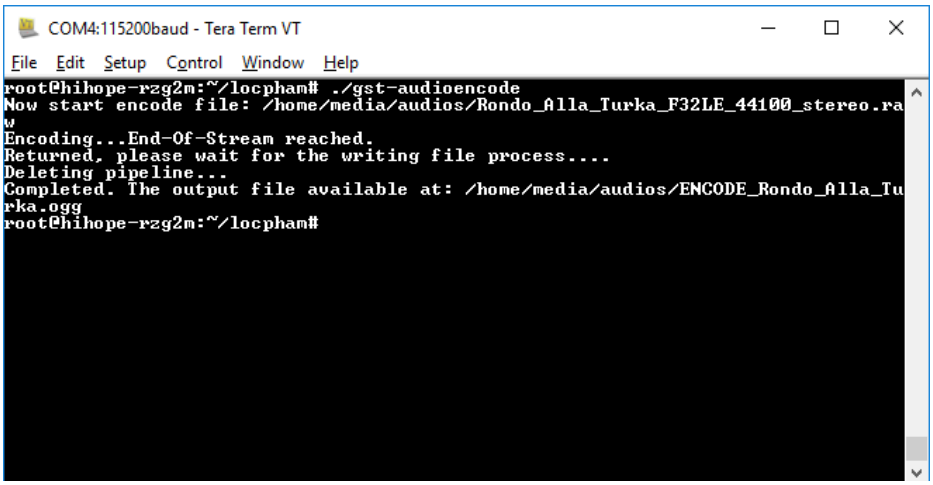


Figure 3.8 Audio Encode result

3.2.3.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual uploaded.

3.2.3.4 Special Instruction

Please put the input file at location `/home/media/audios` (on board).

Prepare raw audio file:

1. Download file `Rondo_Alla_Turka.ogg` at:
   https://upload.wikimedia.org/wikipedia/commons/b/bd/Rondo_Alla_Turka.ogg
   (128 Kbps, 44.1 kHz, stereo channel, and Vorbis audio format).
2. Run this command (on board) to convert this file to raw audio format (F32LE, 44.1 kHz, and stereo channel).

```
$ gst-launch-1.0 -e filesrc location=/home/media/audios/Rondo_Alla_Turka.ogg ! oggdemux
! vorbisdec ! audio/x-raw, format=F32LE, rate=44100, channels=2 ! filesink
location=/home/media/audios/Rondo_Alla_Turka_F32LE_44100_stereo.raw
```

Note that this process will take a while depending on the file size and processor speed.

To check the output file:

Option 1: `VLC` media player (https://www.videolan.org/vlc/index.html).

Option 2: Tool `gst-launch-1.0` (on board):

```
$ gst-launch-1.0 filesrc location=/home/media/audios/ENCODE_Rondo_Alla_Turka.ogg !
oggdemux ! vorbisdec ! audioconvert ! audio/x-raw, format=S16LE ! alsasink
```

3.2.4 Video Encode

Encode video data from NV12 raw format to H.264 format.

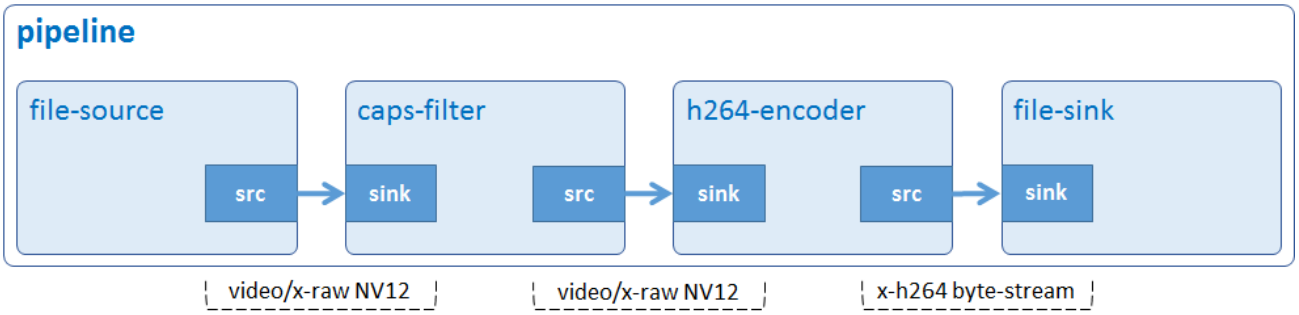


Figure 3.9 Video Encode pipeline

3.2.4.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <gst/gst.h>
#include <stdbool.h>
#include <stdio.h>

#define BITRATE          10485760
#define CONTROL_RATE     2   /* Constant bitrate */
#define BLOCKSIZE        576000 /* Blocksize of NV12 is width*heigh*3/2 */
#define INPUT_FILE       "/home/media/videos/h264-wvga-30.yuv"
#define OUTPUT_FILE      "/home/media/videos/ENCODE_h264-wvga-30.h264"
#define VIDEO_FORMAT     "NV12"
#define LVDS_WIDTH       800
#define LVDS_HEIGHT      480
#define FRAMERATE        30
#define TYPE_FRACTION    1

int
main (int argc, char *argv[])
{
  GstElement *pipeline, *source, *capsfilter, *encoder, *sink;
  GstCaps *caps;
  GstBus *bus;
  GstMessage *msg;

  const gchar *input_file = INPUT_FILE;
  const gchar *output_file = OUTPUT_FILE;

  if (!is_file_exist(input_file))
  {
    g_printerr("Cannot find input file: %s. Exiting.\n", input_file);
    return -1;
  }

  /* Initialization */
  gst_init (&argc, &argv);

  /* Create GStreamer elements */
  pipeline = gst_pipeline_new ("video-encode");
  source = gst_element_factory_make ("filesrc", "file-source");
  capsfilter = gst_element_factory_make ("capsfilter", "caps-filter");
  encoder = gst_element_factory_make ("omxh264enc", "H264-encoder");
```

```
  sink = gst_element_factory_make ("filesink", "file-output");

  if (!pipeline || !source || !capsfilter || !encoder || !sink) {
    g_printerr ("One element could not be created. Exiting.\n");
    return -1;
  }

  /* Set up the pipeline */

  /* Set the input filename to the source element, blocksize of NV12 is width*heigh*3/2 */
  g_object_set (G_OBJECT (source), "location", input_file, NULL);
  g_object_set (G_OBJECT (source), "blocksize", BLOCKSIZE, NULL);

  /* Set the caps option to the caps-filter element */
  caps =
      gst_caps_new_simple ("video/x-raw",
       "format", G_TYPE_STRING, VIDEO_FORMAT,
       "framerate", GST_TYPE_FRACTION, FRAMERATE, TYPE_FRACTION,
       "width", G_TYPE_INT, LVDS_WIDTH, "height", G_TYPE_INT, LVDS_HEIGHT, NULL); /* The raw video file
is in NV12 format, resolution 800x480*/
  g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
  gst_caps_unref (caps);

  /* Set the H.264 encoder options: constant control-rate, 10Mbps target bitrate to the encoder element*/
  g_object_set (G_OBJECT (encoder), "control-rate", CONTROL_RATE, NULL);
  g_object_set (G_OBJECT (encoder), "target-bitrate", BITRATE, NULL);

  /* Set the output filename to the sink element */
  g_object_set (G_OBJECT (sink), "location", output_file, NULL);

  /* Add all elements into the pipeline */
  /* file-source | caps-filter | H264-encoder | file-output */
  gst_bin_add_many (GST_BIN (pipeline), source, capsfilter, encoder, sink,
      NULL);

  /* Link the elements together */
  /* file-source -> caps-filter -> H264-encoder -> file-output */
  if (gst_element_link_many (source, capsfilter, encoder, sink, NULL) != TRUE) {
    g_printerr ("Elements could not be linked.\n");
    gst_object_unref (pipeline);
    return -1;
  }

  /* Set the pipeline to "playing" state */
  g_print ("Now encoding file: %s\n", input_file);
  if (gst_element_set_state (pipeline,
          GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
    g_printerr ("Unable to set the pipeline to the playing state.\n");
    gst_object_unref (pipeline);
    return -1;
  }

  /* Waiting */
  g_print ("Running...\n");
  bus = gst_element_get_bus (pipeline);
  msg =
      gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
      GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
  gst_object_unref (bus);

  /* Note that because input timeout is GST_CLOCK_TIME_NONE,
     the gst_bus_timed_pop_filtered() function will block forever untill a
     matching message was posted on the bus (GST_MESSAGE_ERROR or
     GST_MESSAGE_EOS). */

  /* Encode end. Clean up nicely */
  if (msg != NULL) {
    GError *err;
    gchar *debug_info;

    switch (GST_MESSAGE_TYPE (msg)) {
      case GST_MESSAGE_ERROR:
        gst_message_parse_error (msg, &err, &debug_info);
        g_printerr ("Error received from element %s: %s.\n",
```

```
          GST_OBJECT_NAME (msg->src), err->message);
      g_printerr ("Debugging information: %s.\n",
          debug_info ? debug_info : "none");
      g_clear_error (&err);
      g_free (debug_info);
      break;
    case GST_MESSAGE_EOS:
      g_print ("End-Of-Stream reached.\n");
      break;
     default:
       /* We should not reach here because we only asked for ERRORs and EOS */
       g_printerr ("Unexpected message received.\n");
       break;
   }
  gst_message_unref (msg);
 }

 g_print ("Returned, stopping conversion...\n");
 gst_element_set_state (pipeline, GST_STATE_NULL);
 g_print ("Deleting pipeline...\n");
 gst_object_unref (GST_OBJECT (pipeline));
 g_print ("Succeeded. Encoded file available at: %s\n", output_file);
 return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.1 Audio Play.

Input/output location

```
#define INPUT_FILE       "/home/media/videos/h264-wvga-30.yuv"
#define OUTPUT_FILE      "/home/media/videos/ENCODE_h264-wvga-30.h264"
```

Note: You can create input file by following section 3.2.4.4 Special Instruction.

Create elements

```
source = gst_element_factory_make ("filesrc", "file-source");
capsfilter = gst_element_factory_make ("capsfilter", "caps-filter");
encoder = gst_element_factory_make ("omxh264enc", "H264-encoder");
sink = gst_element_factory_make ("filesink", "file-output");
```

To encode a raw video file to H.264 video format, the following elements are used:

- Element `filesrc` reads data from a local file.
- Element `capsfilter` specifies raw video format, framerate, and resolution.
- Element `omxh264enc` encodes raw video into H.264 compressed data.
- Element `filesink` writes incoming data to a local file.

Set element properties

```
g_object_set (G_OBJECT (source), "location", input_file, NULL);
g_object_set (G_OBJECT (source), "blocksize", BLOCKSIZE, NULL);
g_object_set (G_OBJECT (encoder), "control-rate", CONTROL_RATE, NULL);
g_object_set (G_OBJECT (encoder), "target-bitrate", BITRATE, NULL);
g_object_set (G_OBJECT (sink), "location", output_file, NULL);
```

The `g_object_set()` function is used to set some element's properties, such as:

- The `location` property of `filesrc` and `filesink` elements which points to input and output file.
- The `blocksize` property of `filesrc` element which is calculated by multiplying 1.5 (`NV12`) x 800 (frame width) x 480 (frame height).
- The `control-rate` property of `omxh264enc` element which enables low latency video.

- The `target-bitrate` property of `omxh264enc` element which is set to 10 Mbps. The higher bitrate, the better quality.

```
caps = gst_caps_new_simple ("video/x-raw", "format", G_TYPE_STRING, VIDEO_FORMAT,
      "framerate", GST_TYPE_FRACTION, FRAMERATE, TYPE_FRACTION,
      "width", G_TYPE_INT, LVDS_WIDTH, "height", G_TYPE_INT, LVDS_HEIGHT, NULL);

g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);
```

A `capsfilter` is needed between `filesrc` and `omxh264enc` because `omxh264enc` element needs to know what raw video format, frame rate, and resolution of the incoming data stream are. In this application, the output video is `NV12` formatted, has 30 FPS, and resolution 800x480.

The `gst_caps_new_simple()` function creates a new cap which holds these values. This cap is then added to `caps` property of `capsfilter` element (`g_object_set`).

Note that the `caps` should be freed with `gst_caps_unref()` if it is not used anymore.

3.2.4.2 Result

An H.264 video file will be generated after running this application.

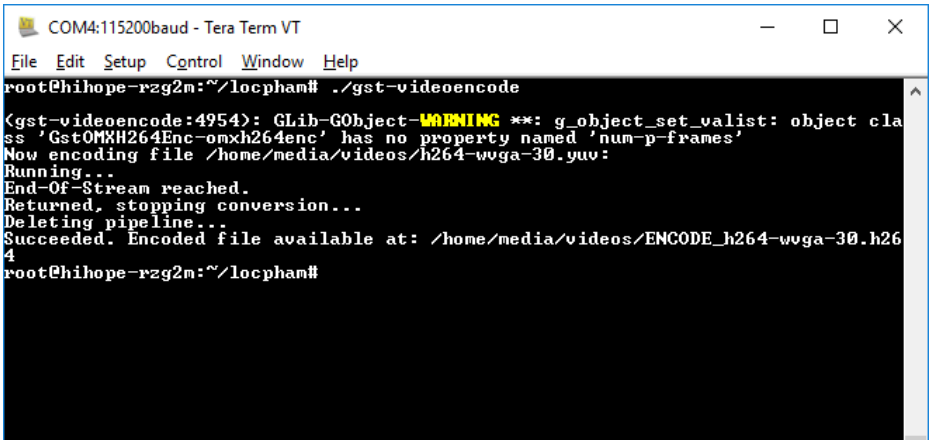


Figure 3.10 Video Encode result

3.2.4.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.4.4 Special Instruction

Prepare raw video file:

1. Download file `h264-wvga-30.mp4` at:

http://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/hmi-mmpoc-videos/h264-wvga-30.mp4

2. Run this command (on board) to convert this file to raw video format (NV12):

```
$ gst-launch-1.0 -e filesrc num-buffers=120 location=/home/media/videos/h264-wvga-30.mp4
! qtdemux ! h264parse ! omxh264dec no-copy=false ! filesink
location=/home/media/videos/h264-wvga-30.yuv
```

3. Finally, the pipeline will create the input file `h264-wvga-30.yuv` at location `/home/media/videos/`

Note:

Please remove `num-buffers` property if you would like to decode the whole video.

To check the output file:

Run this command on board:

```
$ gst-launch-1.0 filesrc location=/home/media/videos/ENCODE_h264-wvga-30.h264 !
h264parse ! omxh264dec ! waylandsink
```

3.2.5 Audio Record

Record raw data from USB microphone, then store it in Ogg container.

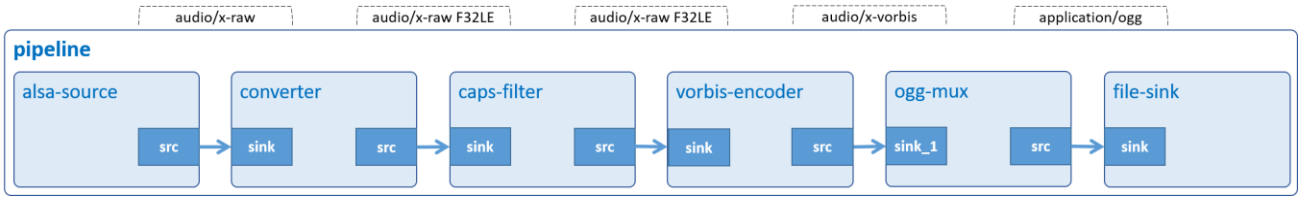


Figure 3.11 Audio Record pipeline

3.2.5.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <gst/gst.h>

#define BITRATE      128000     /* Bitrate averaging */
#define SAMPLE_RATE  48000  /* Sample rate  of audio file*/
#define CHANNEL      1          /* Channel*/
#define OUTPUT_FILE  "RECORD_microphone-mono.ogg"
#define FORMAT       "F32LE"
#define ARG_PROGRAM_NAME   0
#define ARG_DEVICE         1
#define ARG_COUNT          2

static GstElement *pipeline;

void
signalHandler (int signal)
{
  if (signal == SIGINT) {
    gst_element_send_event (pipeline, gst_event_new_eos ());
  }
}

static void
link_to_multiplexer (GstPad * tolink_pad, GstElement * mux)
{
  GstPad *pad;
  gchar *srcname, *sinkname;

  srcname = gst_pad_get_name (tolink_pad);
  pad = gst_element_get_compatible_pad (mux, tolink_pad, NULL);
  gst_pad_link (tolink_pad, pad);
  sinkname = gst_pad_get_name (pad);
  gst_object_unref (GST_OBJECT (pad));

  g_print ("A new pad %s was created and linked to %s\n", sinkname, srcname);
  g_free (sinkname);
  g_free (srcname);
}

int
main (int argc, char *argv[])
{
  GstElement *source, *converter, *convert_capsfilter, *encoder, *muxer, *sink;
  GstCaps *caps;
  GstBus *bus;
```

```
GstPad *srcpad;
GstMessage *msg;

const gchar *output_file = OUTPUT_FILE;

if (argc != ARG_COUNT) {
  g_print ("Error: Invalid arugments.\n");
  g_print ("Usage: %s <microphone device> \n", argv[ARG_PROGRAM_NAME]);
  return -1;
}

/* Initialization */
gst_init (&argc, &argv);

/* Create GStreamer elements */
pipeline = gst_pipeline_new ("audio-record");
source = gst_element_factory_make ("alsasrc", "alsa-source");
converter = gst_element_factory_make ("audioconvert", "audio-converter");
convert_capsfilter = gst_element_factory_make ("capsfilter", "convert_caps");
encoder = gst_element_factory_make ("vorbisenc", "vorbis-encoder");
muxer = gst_element_factory_make ("oggmux", "ogg-muxer");
sink = gst_element_factory_make ("filesink", "file-output");

if (!pipeline || !source || !converter || !convert_capsfilter || !encoder
    || !muxer || !sink) {
  g_printerr ("One element could not be created. Exiting.\n");
  return -1;
}

/* set input device (microphone) of the source element - alsasrc */
g_object_set (G_OBJECT (source), "device", argv[ARG_DEVICE], NULL);

/* set target bitrate of the encoder element - vorbisenc */
g_object_set (G_OBJECT (encoder), "bitrate", BITRATE, NULL);

/* set output file location of the sink element - filesink */
g_object_set (G_OBJECT (sink), "location", output_file, NULL);

/* create simple caps */
caps =
    gst_caps_new_simple ("audio/x-raw",
    "format", G_TYPE_STRING, FORMAT,
    "channels", G_TYPE_INT, CHANNEL, "rate", G_TYPE_INT, SAMPLE_RATE, NULL);

/* set caps property of capsfilters */
g_object_set (G_OBJECT (convert_capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);

/* add the elements into the pipeline */
gst_bin_add_many (GST_BIN (pipeline), source, converter, convert_capsfilter,
    encoder, muxer, sink, NULL);

/* link the elements together */
if (gst_element_link_many (source, converter, convert_capsfilter, encoder,
        NULL) != TRUE) {
  g_printerr ("Elements could not be linked.\n");
  gst_object_unref (pipeline);
  return -1;
}
if (gst_element_link (muxer, sink) != TRUE) {
  g_printerr ("Elements could not be linked.\n");
  gst_object_unref (pipeline);
  return -1;
}

/* link srcpad of vorbisenc to request pad of oggmux */
srcpad = gst_element_get_static_pad (encoder, "src");
link_to_multiplexer (srcpad, muxer);
gst_object_unref (srcpad);

/* Set the pipeline to "playing" state */
g_print ("Now recording, press [Ctrl] + [C] to stop...\n");
if (gst_element_set_state (pipeline,
        GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
```

```
    g_printerr ("Unable to set the pipeline to the playing state.\n");
    gst_object_unref (pipeline);
    return -1;
  }

  /* Handle signals gracefully. */
  signal (SIGINT, signalHandler);

  /* Wait until error or EOS */
  bus = gst_element_get_bus (pipeline);
  msg =
      gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
      GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
  gst_object_unref (bus);

  /* Note that because input timeout is GST_CLOCK_TIME_NONE,
     the gst_bus_timed_pop_filtered() function will block forever until a
     matching message was posted on the bus (GST_MESSAGE_ERROR or
     GST_MESSAGE_EOS). */

  if (msg != NULL) {
    GError *err;
    gchar *debug_info;

    switch (GST_MESSAGE_TYPE (msg)) {
      case GST_MESSAGE_ERROR:
        gst_message_parse_error (msg, &err, &debug_info);
        g_printerr ("Error received from element %s: %s.\n",
            GST_OBJECT_NAME (msg->src), err->message);
        g_printerr ("Debugging information: %s.\n",
            debug_info ? debug_info : "none");
        g_clear_error (&err);
        g_free (debug_info);
        break;
      case GST_MESSAGE_EOS:
        g_print ("End-Of-Stream reached.\n");
        break;
      default:
        /* We should not reach here because we only asked for ERRORs and EOS */
        g_printerr ("Unexpected message received.\n");
        break;
    }
    gst_message_unref (msg);
  }

  /* Clean up nicely */
  g_print ("Returned, stopping recording...\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);

  g_print ("Deleting pipeline...\n");
  gst_object_unref (GST_OBJECT (pipeline));
  g_print ("Succeeded. Please check output file: %s\n", output_file);
  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.1 Audio Play.

Output location

```
#define OUTPUT_FILE  "RECORD_microphone-mono.ogg"
const gchar *output_file = OUTPUT_FILE;
```

Command-line argument

```
if (argc != ARG_COUNT) {
  g_print ("Error: Invalid arugments.\n");
```

```
  g_print ("Usage: %s <microphone device> \n", argv[ARG_PROGRAM_NAME]);
  return -1;
}
```

This application accepts a command-line argument which points to a USB microphone (*hw:1,0*, for example).

Note: You can find this value by following section 3.2.5.4 Special Instruction.

Create elements

```
pipeline = gst_pipeline_new ("audio-record");
source = gst_element_factory_make ("alsasrc", "alsa-source");
converter = gst_element_factory_make ("audioconvert", "audio-converter");
convert_capsfilter = gst_element_factory_make ("capsfilter", "convert_caps");
encoder = gst_element_factory_make ("vorbisenc", "vorbis-encoder");
muxer = gst_element_factory_make ("oggmux", "ogg-muxer");
sink = gst_element_factory_make ("filesink", "file-output");
```

To record raw data from USB microphone then store it in Ogg container, the following elements are used:

- Element `alsasrc` reads data from an audio card using the ALSA API.
- Element `audioconvert` converts raw audio buffers to a format (such as: `F32LE`) which is understood by `vorbisenc`.
- Element `vorbisenc` encodes raw audio into a Vorbis stream.
- Element `oggmux` merges audio stream to Ogg container.
- Element `filesink` writes incoming data to a local file.

Set Element's properties

```
g_object_set (G_OBJECT (source), "device", argv[ARG_DEVICE], NULL);
g_object_set (G_OBJECT (encoder), "bitrate", BITRATE, NULL);
g_object_set (G_OBJECT (sink), "location", output_file, NULL);
```

The `g_object_set()` function is used to set some element's properties, such as:

- The `device` property of `alsasrc` element which points to a microphone device. Users will pass the device card as a command line argument to this application. Please refer to section 3.2.5.4 Special Instruction to find the value.
- The `location` property of `filesink` element which points to the output file.
- The `bitrate` property of `vorbisenc` element is used to specify encoding bit rate. The higher bitrate, the better quality.

```
caps = gst_caps_new_simple ("audio/x-raw", "format", G_TYPE_STRING, FORMAT,
           "channels", G_TYPE_INT, CHANNEL, "rate", G_TYPE_INT, SAMPLE_RATE, NULL);
g_object_set (G_OBJECT (convert_capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);
```

Capabilities (short: caps) describe the type of data which is streamed between two pads. This data includes raw audio format, channel, and sample rate.

The `gst_caps_new_simple()` function creates new caps which holds these values. These caps are then added to `caps` property of `capsfilter` elements (`g_object_set`).

Note that both `caps` should be freed with `gst_caps_unref()` if they are not used anymore.

Build pipeline

```
gst_bin_add_many (GST_BIN (pipeline), source, converter, convert_capsfilter,
    encoder, muxer, sink, NULL);
gst_element_link_many (source, converter, convert_capsfilter, encoder, NULL);
gst_element_link (muxer, sink);
```

The reason for the separation is that the sink pad of `oggmux` (`muxer`) cannot be created automatically but is only created on demand. This application uses self-defined function `link_to_multiplexer()` to link the sink pad to source pad of `vorbisenc` (`encoder`). That's why its sink pad is called `Request Pad`.

Note that the order counts, because links must follow the data flow (this is, from source elements to sink elements).

Link request pads

```
srcpad = gst_element_get_static_pad (encoder, "src");
link_to_multiplexer (srcpad, muxer);
gst_object_unref (srcpad);
```

This block gets the source pad (`srcpad`) of `vorbisenc` (`encoder`), then calls `link_to_multiplexer()` to link it to (the sink pad of) `oggmux` (`muxer`).

Note that the `srcpad` should be freed with `gst_object_unref()` if it is not used anymore.

```
static void link_to_multiplexer (GstPad * tolink_pad, GstElement * mux)
{
  pad = gst_element_get_compatible_pad (mux, tolink_pad, NULL);
  gst_pad_link (tolink_pad, pad);

  gst_object_unref (GST_OBJECT (pad));
}
```

This function uses `gst_element_get_compatible_pad()` to request a sink pad (`pad`) which is compatible with the source pad (`tolink_pad`) of `oggmux` (`mux`), then calls `gst_pad_link()` to link them together.

Note that the `pad` should be freed with `gst_object_unref()` if it is not used anymore.

Play pipeline

```
gst_element_set_state (pipeline, GST_STATE_PLAYING);
```

Every pipeline has an associated state. To start audio recording, the pipeline needs to be set to `PLAYING` state.

```
signal (SIGINT, signalHandler);
```

This application will stop recording if user presses Ctrl-C. To do so, it uses `signal()` to bind `SIGINT` (interrupt from keyboard) to `signalHandler()`.

To know how this function is implemented, please refer to the following code block:

```
void signalHandler (int signal)
{
  if (signal == SIGINT) {
    gst_element_send_event (pipeline, gst_event_new_eos ());
  }
}
```

It calls `gst_element_send_event()` to send `EOS (End-of-Stream)` signal (`gst_event_new_eos`) to the `pipeline`. This makes `gst_bus_timed_pop_filtered()` return. Finally, the program cleans up GStreamer objects and exits.

3.2.5.2 Result

An Ogg file will be generated after running this application.

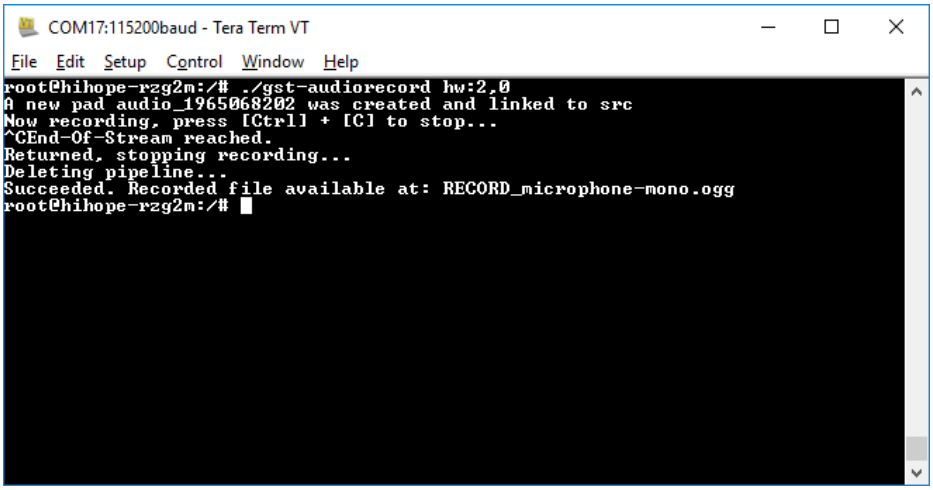


Figure 3.12 Audio Record result

3.2.5.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.5.4 Special Instruction

Run the following script to find microphone device card:

```
$ ./detect_USB_microphone.sh
```

Basically, this script analyzes the `/proc/asound/cards` file to get USB sound cards.

Note: This script can be used in combination with `gst-audiorecord` application.

```
$ ./gst-audiorecord $( ./detect_USB_microphone.sh )
```

For further information on how this script is implemented, please refer to the following code block:

```
#!/bin/bash

ALSA_DEV_FILE="/proc/asound/cards"

CMD_GET_USB_SND="cat $ALSA_DEV_FILE | awk '/[0-9]+ \\[[[:print:]]+\\]: USB-Audio - [[:print:]]+/ {
print \$0 }'"

CMD_GET_USB_SND_INDICES="$CMD_GET_USB_SND | awk '{ print \$1 }'"

USB_SND_INDICES="$( eval $CMD_GET_USB_SND_INDICES )"
if [ ! -z "$USB_SND_INDICES" ]
then
    for TEMP_INDEX in $USB_SND_INDICES
    do
        # Check if this USB sound card has microphone or not?
        HAS_MIC="$( amixer -D hw:$TEMP_INDEX scontrols | grep "Mic" )"

        if [ "$HAS_MIC" != "" ]
        then
            USB_SND_INDEX=$TEMP_INDEX
```

```
            DEVICE_NUMBER=${HAS_MIC: -1}
            echo "hw:$USB_SND_INDEX,$DEVICE_NUMBER"

            amixer -D hw:$USB_SND_INDEX set "Mic" 100% > /dev/null

            break
        fi
    done
fi
```

To check the output file:

Option 1: `VLC` media player (https://www.videolan.org/vlc/index.html).

Option 2: Tool `gst-launch-1.0` (on board):

```
$ gst-launch-1.0 filesrc location=RECORD_microphone-mono.ogg ! oggdemux !
vorbisdec ! audioconvert ! audioresample ! autoaudiosink
```

3.2.6 Video Record

Display and encode raw data from USB/MIPI camera to H.264 format, then store it in MP4 container when user presses Ctrl-C.

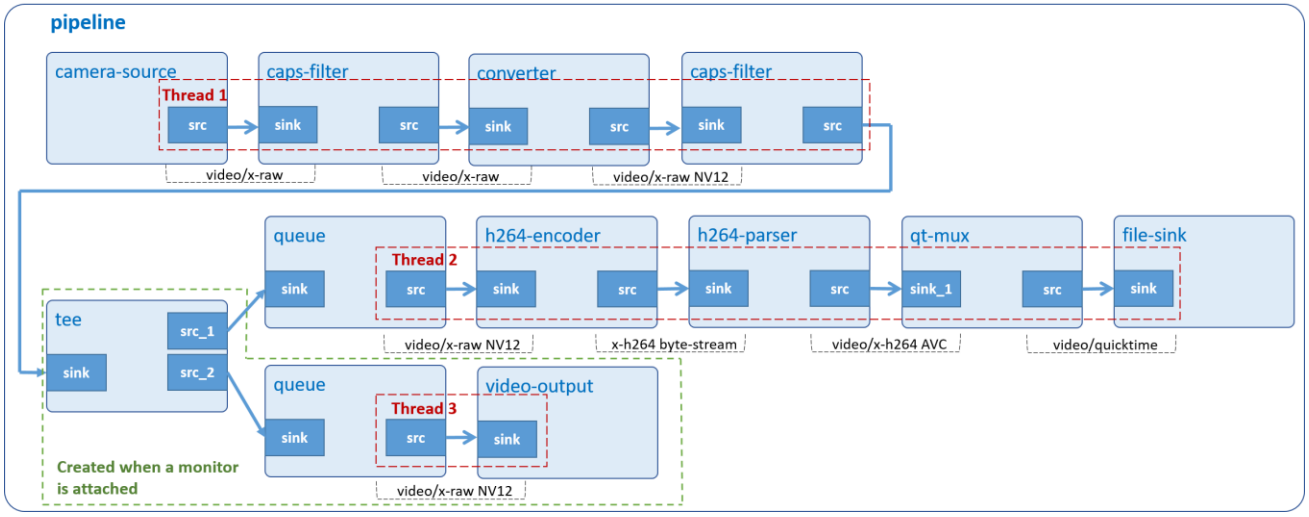


Figure 3.13 Video Record pipeline

3.2.6.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <gst/gst.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <wayland-client.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/videodev2.h>

#define MIPI_BITRATE_OMXH264ENC 40000000 /* Target bitrate of the encoder for MIPI camera */
#define USB_BITRATE_OMXH264ENC  10485760 /* Target bitrate of the encoder for USB camera */
#define VARIABLE_RATE      1
#define USB_WIDTH_SIZE     640        /* The output data of v4l2src in this application will be */
#define USB_HEIGHT_SIZE    480        /* a raw video with 640x480 size */
#define MIPI_WIDTH_SIZE    1280       /* The output data of v4l2src in this application will be */
#define MIPI_HEIGHT_SIZE   960        /* a raw video with 1280x960 size */
#define ARG_PROGRAM_NAME   0
#define ARG_DEVICE         1
#define ARG_WIDTH          2
#define ARG_HEIGHT         3
#define ARG_COUNT          4

static GstElement *pipeline;

enum camera_type {
  NO_CAMERA,
  MIPI_CAMERA,
  USB_CAMERA
```

```
};
/* Supported resolutions of MIPI camera */
const char *mipi_resolutions[] = {
  "1280x960",
  "1920x1080",
  "2592x1944",
  NULL,
};

/* Supported resolutions of USB camera */
const char *usb_resolutions[] = {
  "320x240",
  "640x480",
  "800x600",
  "960x720",
  "1280x720",
  "1280x960",
  NULL,
};

/* Check type of camera
 * return NO_CAMERA: Unsupported camera
 * return MIPI_CAMERA: MIPI camera detected
 * return USB_CAMERA: USB camera detected */
enum camera_type
check_camera_type (const char *device);

/* Print supported resolutions in console*/
void
print_supported_resolutions (char *resolution,
    const char* supported_resolutions[]);

/* Check resolution in program argument is supported or not
 * Supported resolutions are defined in
 * usb_resolutions and mipi_resolutions */
bool
check_resolution (char *resolution, const char *supported_resolutions[]);

/* Check resolution in program argument is supported or not
 * If not, display list of supported resolutions in console
 * else store resolution to width, height */
bool
get_resolution (char *arg_width, char *arg_height, int *width,
    int *height, enum camera_type camera);

void
signalHandler (int signal)
{
  if (signal == SIGINT) {
    gst_element_send_event (pipeline, gst_event_new_eos ());
  }
}

static void
link_to_multiplexer (GstPad * tolink_pad, GstElement * mux)
{
  GstPad *pad;
  gchar *srcname, *sinkname;

  srcname = gst_pad_get_name (tolink_pad);
  pad = gst_element_get_compatible_pad (mux, tolink_pad, NULL);
  gst_pad_link (tolink_pad, pad);
  sinkname = gst_pad_get_name (pad);
  gst_object_unref (GST_OBJECT (pad));

  g_print ("A new pad %s was created and linked to %s\n", sinkname, srcname);
  g_free (sinkname);
  g_free (srcname);
}

int
main (int argc, char *argv[])
{
```

```
struct wayland_t *wayland_handler = NULL;
bool display_video = true;
GstElement *source, *camera_capsfilter, *converter, *convert_capsfilter,
    *queue1, *encoder, *parser, *muxer, *filesink;
GstBus *bus;
GstMessage *msg;
GstPad *srcpad;
GstCaps *camera_caps, *convert_caps;
enum camera_type camera = NO_CAMERA;
int width = 0;
int height = 0;

const gchar *output_file = "RECORD-camera.mp4";

/* Get a list of available screen */
wayland_handler = get_available_screens();

if (wayland_handler->output == NULL) {
  display_video = false;
}
destroy_wayland(wayland_handler);

if ((argc != 2) && (argc != ARG_COUNT)) {
  g_print ("Error: Invalid arugments.\n");
  g_print ("Usage: %s <camera device> [width] [height]\n", argv[ARG_PROGRAM_NAME]);
  return -1;
}

/* Check type of camera and set default resolution */
camera = check_camera_type (argv[ARG_DEVICE]);
if (camera == NO_CAMERA) {
  return -1;
} else if (camera == MIPI_CAMERA) {
  width = MIPI_WIDTH_SIZE;
  height = MIPI_HEIGHT_SIZE;
} else {
  width = USB_WIDTH_SIZE;
  height = USB_HEIGHT_SIZE;
}

/* Parse resolution from program argument */
if (argc == ARG_COUNT) {
  char hostname[20];
  gethostname (hostname, 20);
  if (!strcmp (hostname, "ek874") && (camera == MIPI_CAMERA)) {
    g_print ("RZ/G2E only supports 1280x960 resolution.\n");
    g_print ("Set 1280x960 resolution as default.\n");
  } else {
    if (!get_resolution (argv[ARG_WIDTH], argv[ARG_HEIGHT],
              &width, &height, camera)) {
      return -1;
    }
  }
}

/* Initialization */
gst_init (&argc, &argv);

/* Create GStreamer elements */
pipeline = gst_pipeline_new ("video-record");
source = gst_element_factory_make ("v4l2src", "camera-source");
camera_capsfilter = gst_element_factory_make ("capsfilter", "camera_caps");
convert_capsfilter = gst_element_factory_make ("capsfilter", "convert_caps");
queue1 = gst_element_factory_make ("queue", "queue1");
encoder = gst_element_factory_make ("omxh264enc", "video-encoder");
parser = gst_element_factory_make ("h264parse", "h264-parser");
muxer = gst_element_factory_make ("qtmux", "mp4-muxer");
filesink = gst_element_factory_make ("filesink", "file-output");

if (camera == MIPI_CAMERA) {
  converter = gst_element_factory_make ("vspmfilter", "video-converter");
} else {
  converter = gst_element_factory_make ("videoconvert", "video-converter");
}
```

```
if (!pipeline || !source || !camera_capsfilter || !converter
    || !convert_capsfilter || !queue1 || !encoder || !parser
    || !muxer || !filesink) {
  g_printerr ("One element could not be created. Exiting.\n");
  return -1;
}

if (camera == MIPI_CAMERA) {
  /* Set property "dmabuf-use" of vspmfilter to true */
  /* Without it, waylandsink will display broken video */
  g_object_set (G_OBJECT (converter), "dmabuf-use", true, NULL);
  /* Set properties of the encoder element - omxh264enc */
  g_object_set (G_OBJECT (encoder), "target-bitrate", MIPI_BITRATE_OMXH264ENC,
      "control-rate", VARIABLE_RATE, "interval_intraframes", 14,
      "periodicty-idr", 2, "use-dmabuf", true, NULL);

  /* Create camera caps */
  camera_caps =
      gst_caps_new_simple ("video/x-raw", "format", G_TYPE_STRING, "UYVY",
          "width", G_TYPE_INT, width, "height", G_TYPE_INT, height, NULL);
} else {
  /* Set properties of the encoder element - omxh264enc */
  g_object_set (G_OBJECT (encoder), "target-bitrate", USB_BITRATE_OMXH264ENC,
      "control-rate", VARIABLE_RATE, NULL);

  /* Create camera caps */
  camera_caps =
      gst_caps_new_simple ("video/x-raw", "width", G_TYPE_INT, width,
          "height", G_TYPE_INT, height, NULL);
}

/* Set input video device file of the source element - v4l2src */
g_object_set (G_OBJECT (source), "device", argv[ARG_DEVICE], NULL);

/* Set output file location of the filesink element - filesink */
g_object_set (G_OBJECT (filesink), "location", output_file, NULL);

/* create convert caps */
convert_caps =
    gst_caps_new_simple ("video/x-raw", "format", G_TYPE_STRING, "NV12",
    NULL);

/* set caps property for capsfilters */
g_object_set (G_OBJECT (camera_capsfilter), "caps", camera_caps, NULL);
g_object_set (G_OBJECT (convert_capsfilter), "caps", convert_caps, NULL);

/* unref caps after usage */
gst_caps_unref (camera_caps);
gst_caps_unref (convert_caps);

/* Add elements into the pipeline */
gst_bin_add_many (GST_BIN (pipeline), source, camera_capsfilter, converter,
    convert_capsfilter, queue1, encoder, parser, muxer, filesink, NULL);

if (!display_video) {
  /* Link the elements together */
  if (gst_element_link_many (source, camera_capsfilter, converter,
          convert_capsfilter, queue1, encoder, parser, NULL) != TRUE) {
    g_printerr ("Elements could not be linked.\n");
    gst_object_unref (pipeline);
    return -1;
  }
} else {
  GstElement *tee, *queue2, *waylandsink;

  /* Create tee and waylandsink */
  tee = gst_element_factory_make ("tee", "tee");
  queue2 = gst_element_factory_make ("queue", "queue2");
  waylandsink = gst_element_factory_make ("waylandsink", "video-output");

  if (!tee || !queue2 || !waylandsink) {
    g_printerr ("One element could not be created. Exiting.\n");
    return -1;
```

```
    }

    /* Set position of video - waylandsink */
    g_object_set (G_OBJECT (waylandsink), "position-x", 0, "position-y", 0, NULL);

    /* Add tee and waylandsink into the pipeline */
    gst_bin_add_many (GST_BIN (pipeline), tee, queue2, waylandsink, NULL);

    /* Link the elements together */
    if (gst_element_link_many (source, camera_capsfilter, converter,
            convert_capsfilter, tee, NULL) != TRUE) {
      g_printerr ("Elements could not be linked.\n");
      gst_object_unref (pipeline);
      return -1;
    }
    if (gst_element_link_many (tee, queue1, encoder, parser, NULL) != TRUE) {
      g_printerr ("Elements could not be linked.\n");
      gst_object_unref (pipeline);
      return -1;
    }
    if (gst_element_link_many (tee, queue2, waylandsink, NULL) != TRUE) {
      g_printerr ("Elements could not be linked.\n");
      gst_object_unref (pipeline);
      return -1;
    }
  }

  /* link qtmuxer->filesink */
  if (gst_element_link (muxer, filesink) != TRUE) {
    g_printerr ("Elements could not be linked.\n");
    gst_object_unref (pipeline);
    return -1;
  }

  /* link srcpad of h264parse to request pad of qtmuxer */
  srcpad = gst_element_get_static_pad (parser, "src");
  link_to_multiplexer (srcpad, muxer);
  gst_object_unref (srcpad);

  /* Set the pipeline to "playing" state */
  g_print ("Now recording, press [Ctrl] + [C] to stop...\n");
  if (gst_element_set_state (pipeline,
          GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
    g_printerr ("Unable to set the pipeline to the playing state.\n");
    gst_object_unref (pipeline);
    return -1;
  }

  /* Handle signals gracefully. */
  signal (SIGINT, signalHandler);

  /* Wait until error or EOS */
  bus = gst_element_get_bus (pipeline);
  msg =
      gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
      GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
  gst_object_unref (bus);

  /* Note that because input timeout is GST_CLOCK_TIME_NONE,
     the gst_bus_timed_pop_filtered() function will block forever until a
     matching message was posted on the bus (GST_MESSAGE_ERROR or
     GST_MESSAGE_EOS). */

  if (msg != NULL) {
    GError *err;
    gchar *debug_info;

    switch (GST_MESSAGE_TYPE (msg)) {
      case GST_MESSAGE_ERROR:
        gst_message_parse_error (msg, &err, &debug_info);
        g_printerr ("Error received from element %s: %s.\n",
            GST_OBJECT_NAME (msg->src), err->message);
        g_printerr ("Debugging information: %s.\n",
            debug_info ? debug_info : "none");
```

```
        g_clear_error (&err);
        g_free (debug_info);
        break;
      case GST_MESSAGE_EOS:
        g_print ("End-Of-Stream reached.\n");
        break;
      default:
        /* We should not reach here because we only asked for ERRORs and EOS */
        g_printerr ("Unexpected message received.\n");
        break;
    }
    gst_message_unref (msg);
  }

  /* Clean up nicely */
  g_print ("Returned, stopping recording...\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);

  g_print ("Deleting pipeline...\n");
  gst_object_unref (GST_OBJECT (pipeline));
  g_print ("Succeeded. Please check output file: %s\n", output_file);
  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.1 Audio Play.

Output location

```
const gchar *output_file = "RECORD-camera.mp4";
```

Command-line argument

```
if ((argc != 2) && (argc != ARG_COUNT)) {
  g_print ("Error: Invalid arugments.\n");
  g_print ("Usage: %s <camera device> [width] [height]\n", argv[ARG_PROGRAM_NAME]);
  return -1;
}
camera = check_camera_type (argv[ARG_DEVICE]);
```

This application accepts a command-line argument which points to camera's device file (*/dev/video9*, for example). The type of camera is determined by function `check_camera_type`. Note: You can find this value by following section 3.2.6.4 Special Instruction.

User can enter `width` and `height` options to set camera's resolution.

Create elements

```
source = gst_element_factory_make ("v4l2src", "camera-source");
camera_capsfilter = gst_element_factory_make ("capsfilter", "camera_caps");
convert_capsfilter = gst_element_factory_make ("capsfilter", "convert_caps");
queue1 = gst_element_factory_make ("queue", "queue1");
encoder = gst_element_factory_make ("omxh264enc", "video-encoder");
parser = gst_element_factory_make ("h264parse", "h264-parser");
muxer = gst_element_factory_make ("qtmux", "mp4-muxer");
filesink = gst_element_factory_make ("filesink", "file-output");

if (camera == MIPI_CAMERA) {
  converter = gst_element_factory_make ("vspmfilter", "video-converter");
} else {
  converter = gst_element_factory_make ("videoconvert", "video-converter");
}

tee = gst_element_factory_make ("tee", "tee");
queue2 = gst_element_factory_make ("queue", "queue2");
waylandsink = gst_element_factory_make ("waylandsink", "video-output");
```

To display and record camera then store it in MP4 container, the following elements are used:

- Element `v4l2src` captures video from `V4L2` devices.
- Element `capsfilter` specifies raw video format, framerate, and resolution.
- Element `videoconvert` (used for USB camera) and element `vspmfilter` (used for MIPI camera) convert video frames to a format (such as: `NV12`) understood by `omxh264enc`.
- Element `tee` splits (video) data to multiple pads.
- Element `queue` (`queue1` and `queue2`) queues data until one of the limits specified by the `max-size-buffers`, `max-size-bytes`, and/or `max-size-time` properties has been reached. Any attempt to push more buffers into the queue will block the pushing thread until more space becomes available.
- Element `omxh264enc` encodes raw video into H.264 compressed data.
- Element `h264parse` connects `omxh264enc` to `qtmux`.
- Element `qtmux` merges H.264 byte stream to MP4 container.
- Element `filesink` writes incoming data to a local file.
- Element `waylandsink` creates its own window and renders the decoded video frames to that.

Set element's properties

```
g_object_set (G_OBJECT (source), "device", argv[ARG_DEVICE], NULL);
g_object_set (G_OBJECT (filesink), "location", output_file, NULL);

/*MIPI camera*/
g_object_set (G_OBJECT (converter), "dmabuf-use", true, NULL);
/* Set properties of the encoder element - omxh264enc */
g_object_set (G_OBJECT (encoder), "target-bitrate", MIPI_BITRATE_OMXH264ENC,
    "control-rate", VARIABLE_RATE, "interval_intraframes", 14,
    "periodicty-idr", 2, "use-dmabuf", true, NULL);

/*USB camera*/
g_object_set (G_OBJECT (encoder), "target-bitrate", USB_BITRATE_OMXH264ENC,
    "control-rate", VARIABLE_RATE, NULL);

g_object_set (G_OBJECT (waylandsink), "position-x", 0, "position-y", 0, NULL);
```

The `g_object_set()` function is used to set some element's properties, such as:

- The `device` property of `v4l2src` element which points to a camera device file. Users will pass the device file as a command line argument to this application. Please refer to section 3.2.6.4 Special Instruction to find the value.
- The `location` property of `filesink` element which points to MP4 output file.
- The `dmabuf-use` property of `vspmfilter` element which is set to `true`. This disallows `dmabuf` to be output buffer. If it is not set, `waylandsink` will display broken video frames.
- The `target-bitrate` property of `omxh264enc` element is used to specify encoding bit rate. The higher bitrate, the better quality.
- The `control-rate` property of `omxh264enc` element is used to specify birate control method which is variable bitrate method in this case.
- The `interval_intraframes` property of `omxh264enc` element is used to specify interval of coding intra frames.
- The `periodicty-idr` property of `omxh264enc` is used to specify periodicity of IDR frames.
- The position-x and position-y properties of waylandsink element which point to (x, y) coordinate of Wayland desktop.

```
/*MIPI camera*/
camera_caps = gst_caps_new_simple ("video/x-raw", "format", G_TYPE_STRING, "UYVY",
       "width", G_TYPE_INT, width, "height", G_TYPE_INT, height, NULL);

/*USB camera*/
camera_caps = gst_caps_new_simple ("video/x-raw", "width", G_TYPE_INT, width,
       "height", G_TYPE_INT, height, NULL);

convert_caps = gst_caps_new_simple ("video/x-raw", "format", G_TYPE_STRING, "NV12", NULL);
```

```
g_object_set (G_OBJECT (camera_capsfilter), "caps", camera_caps, NULL);
g_object_set (G_OBJECT (convert_capsfilter), "caps", convert_caps, NULL);

gst_caps_unref (camera_caps);
gst_caps_unref (convert_caps);
```

Capabilities (short: caps) describe the type of data which is streamed between two pads. This data includes raw video format, resolution, and framerate.

In this application, two caps are required, one specifies video resolution captured from `v4l2src`, the other specifies video format (`NV12`) generated from `converter`.

The `gst_caps_new_simple()` function creates new caps which holds these values. These caps are then added to `caps` property of `capsfilter` elements (`g_object_set`).

Note that both `camera_caps` and `convert_caps` should be freed with `gst_caps_unref()` if they are not used anymore.

Build pipeline

```
gst_bin_add_many (GST_BIN (pipeline), source, camera_capsfilter, converter,
    convert_capsfilter, queue1, encoder, parser, muxer, filesink, NULL);

/*Not display video on monitor*/
gst_element_link_many (source, camera_capsfilter, converter,
         convert_capsfilter, queue1, encoder, parser, NULL);

/*Display video on monitor*/
gst_bin_add_many (GST_BIN (pipeline), tee, queue2, waylandsink, NULL);
gst_element_link_many (source, camera_capsfilter, converter,
             convert_capsfilter, tee, NULL);
gst_element_link_many (tee, queue1, encoder, parser, NULL);
gst_element_link_many (tee, queue2, waylandsink, NULL);

gst_element_link (muxer, filesink);
```

In case of not displaying video on monitor, this code block adds elements to pipeline and then links them into separated groups as below:

- Group #1: `source`, `camera_capsfilter`, `converter`, `convert_capsfilter`, `queue1`, `encoder` and `parser`.
- Group #2: `muxer` and `filesink`.

The reason for the separation is that the sink pad of `qtmux` (`muxer`) cannot be created automatically but is only created on demand. This application uses self-defined function `link_to_multiplexer()` to link the sink pad to source pad of `h264parse` (`parser`). That's why its sink pad is called `Request Pad`.

Note that the order counts, because links must follow the data flow (this is, from source elements to sink elements).

In case of displaying video on monitor, this code block adds elements to pipeline and then links them into separated groups as below:

- Group #1: `source`, `camera_capsfilter`, `converter`, `convert_capsfilter` and `tee`.
- Group #2: `queue1`, `encoder` and `parser`.
- Group #3: `muxer` and `filesink`.
- Group #4: `queue2` and `waylandsink`.

Note that the order counts, because links must follow the data flow (this is, from source elements to sink elements).

Link request pads

```
srcpad = gst_element_get_static_pad (parser, "src");
link_to_multiplexer (srcpad, muxer);
gst_object_unref (srcpad);
```

This block gets the source pad (srcpad) of h264parse (parser), then calls link_to_multiplexer() to link it to the sink pad of qtmux (muxer).

Note that the srcpad should be freed with gst_object_unref() if it is not used anymore.

```
static void link_to_multiplexer (GstPad * tolink_pad, GstElement * mux)
{
  pad = gst_element_get_compatible_pad (mux, tolink_pad, NULL);
  gst_pad_link (tolink_pad, pad);

  gst_object_unref (GST_OBJECT (pad));
}
```

This function uses gst_element_get_compatible_pad() to request a sink pad (pad) which is compatible with the source pad (tolink_pad) of qtmux (mux), then calls gst_pad_link() to link them together.

Note that the pad should be freed with gst_object_unref() if it is not used anymore.

Play pipeline

```
gst_element_set_state (pipeline, GST_STATE_PLAYING);
```

Every pipeline has an associated state. To start webcam recording, the pipeline needs to be set to PLAYING state.

```
signal (SIGINT, signalHandler);
```

This application will stop recording if user presses Ctrl-C. To do so, it uses signal() to bind SIGINT (interrupt from keyboard) to signalHandler().

To know how this function is implemented, please refer to the following code block:

```
void signalHandler (int signal)
{
  if (signal == SIGINT) {
    gst_element_send_event (pipeline, gst_event_new_eos ());
  }
}
```

It calls gst_element_send_event() to send EOS (End-of-Stream) signal (gst_event_new_eos) to the pipeline. This makes gst_bus_timed_pop_filtered() return. Finally, the program cleans up GStreamer objects and exits.

3.2.6.2 Result

Video is displayed on monitor and an MP4 file will be generated after running this application. Note that it does not have audio.

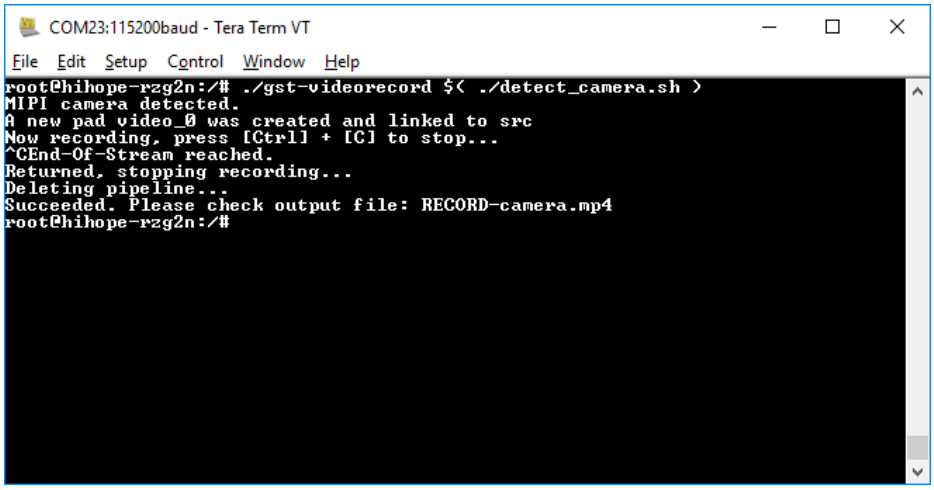


Figure 3.14 Video Record result

3.2.6.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.6.4 Special Instruction

Reference USB Camera:

Logitech USB HD Webcam C270 (model: V-U0018), Logitech USB HD 1080p Webcam C930E and Logitech USB UHD Webcam BRIO.

Reference MIPI Camera:

MIPI Mezzanine Adapter and OV5645 camera.

Run the following script to find camera device file:

```
$ ./detect_camera.sh
```

Basically, this script uses v4l2-ctl tool to read all information of device files (/dev/video8, for example) and find out if the device file has “Crop Capability Video Capture”. If the string is exist, the device file is available to use.

This script can be used in combination with gst-videorecord application.

```
$ ./gst-videorecord $( ./detect_camera.sh )
```

For further information on how this script is implemented, please refer to the following code block:

```
#!/bin/bash

# ---------- Define error code global variables ----------

ERR_NO_CAMERA=1
PROG_SUCCESS_CODE=0

# ---------- Define main function variables ----------

PROG_STAT=$PROG_SUCCESS_CODE
```

```
# ---------- Main function ----------

for DEV_NAME in $( ls -v /dev/video* )
do
  CHECK_CAMERA=$( v4l2-ctl -d $DEV_NAME --all | grep "Crop Capability Video Capture:" )

  if [ ! -z "$CHECK_CAMERA" ]
  then
    CAMERA_DEV=$DEV_NAME
    echo $CAMERA_DEV
    break
  fi
done

if [ -z $CAMERA_DEV ]
then
  PROG_STAT=$ERR_NO_CAMERA
fi

exit $PROG_STAT
```

Run the following script to initialize MIPI camera:

```
$ ./setup_MIPI_camera.sh
```

Basically, this script uses media-ctl tool to set up format of camera device. For RZ/G2E platform, the default resolution is 1280x960. For RZ/G2N platform, RZ/G2M platform and RZ/G2H platform, 3 acceptable resolutions are 1280x960, 1920x1080 and 2592x1944.

```
#!/bin/bash
vin=$(cat /sys/class/video4linux/video*/name | grep "VIN")
if [ -z "$vin" ]
then
  echo "No MIPI camera found"
else
  if [ "$HOSTNAME" = "ek874" ]
  then
    echo "RZ/G2E only supports 1280x960 resolution"
    echo "Set 1280x960 resolution as default"
    media-ctl -d /dev/media0 -r
    media-ctl -d /dev/media0 -l "'rcar_csi2 feaa0000.csi2':1 -> 'VIN4 output':0 [1]"
    media-ctl -d /dev/media0 -V "'rcar_csi2 feaa0000.csi2':1 [fmt:UYVY8_2X8/1280x960 field:none]"
    media-ctl -d /dev/media0 -V "'ov5645 3-003c':0 [fmt:UYVY8_2X8/1280x960 field:none]"
    echo "/dev/video0 is configured successfully with resolution 1280x960"
  elif ["$HOSTNAME"="hihope-rzg2n"] || ["$HOSTNAME"="hihope-rzg2m"] || ["$HOSTNAME"="hihope-rzg2h"]
  then
    if [ $# -ne 2 ]
    then
      echo "Illegal number of parameters"
      echo "Usage: ./setup_MIPI_camera.sh <width> <height>"
      exit 1
    fi

    if ([ "$1" != "1280" ] || [ "$2" != "960" ]) && ([ "$1" != "1920" ] || [ "$2" != "1080" ]) && ([ "$1" !=
"2592" ] || [ "$2" != "1944" ])
    then
      echo "Error: Unsupported resolution: "$1"x"$2""
      echo "Please try one of the following resolutions: 1280x960 1920x1080 2592x1944"
      exit 1
    fi
    media-ctl -d /dev/media0 -r
    media-ctl -d /dev/media0 -l "'rcar_csi2 fea80000.csi2':1 -> 'VIN0 output':0 [1]"
    media-ctl -d /dev/media0 -V "'rcar_csi2 fea80000.csi2':1 [fmt:UYVY8_2X8/"$1"x"$2" field:none]"
    media-ctl -d /dev/media0 -V "'ov5645 2-003c':0 [fmt:UYVY8_2X8/"$1"x"$2" field:none]"
    echo "/dev/video0 is configured successfully with resolution "$1"x"$2""
  else
    echo "Error: Unsupported board"
  fi
fi
```

To check the output file:

Option 1: `VLC` media player (https://www.videolan.org/vlc/index.html).

Option 2: Tool `gst-launch-1.0` (on board):

```
$ gst-launch-1.0 filesrc location=/home/media/videos/RECORD_USB-camera.mp4 ! qtdemux ! h264parse ! omxh264dec ! waylandsink
```

3.2.7 Audio Video Record

Record raw data from USB microphone and USB webcam or MIPI camera at the same time, then store them in MKV container.

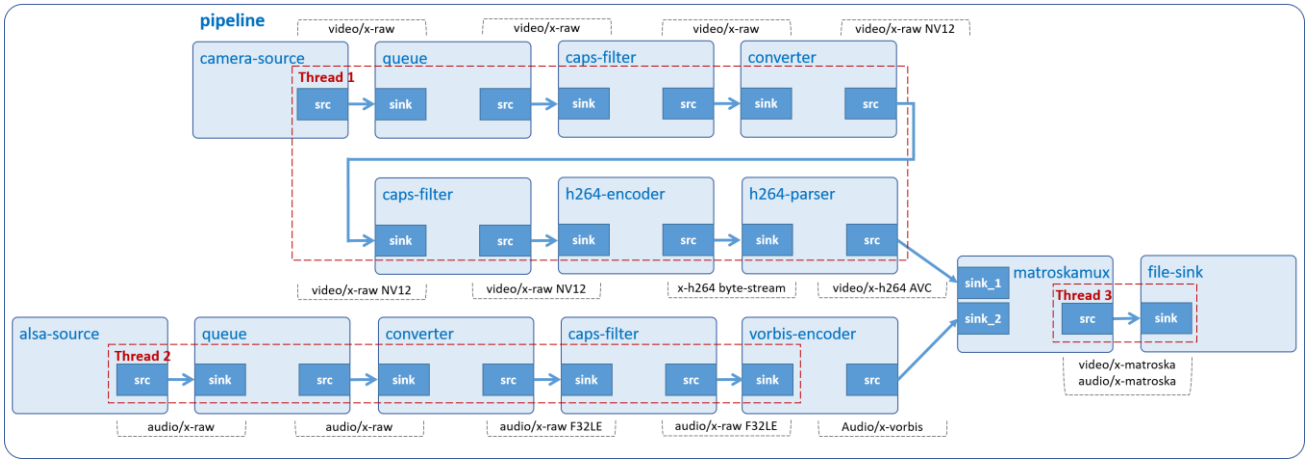


Figure 3.15 Audio Video Record pipeline

3.2.7.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <gst/gst.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/videodev2.h>

#define MIPI_BITRATE_OMXH264ENC 40000000 /* Target bitrate of the encoder for MIPI camera */
#define USB_BITRATE_OMXH264ENC  10485760 /* Target bitrate of the encoder for USB camera */
#define BITRATE_ALSASRC    128000      /* Target bitrate of the encoder element - alsasrc */
#define SAMPLE_RATE        48000  /* Sample rate  of audio file*/
#define CHANNEL            1           /* Channel*/
#define USB_WIDTH_SIZE         1280       /* The output data of v4l2src in this application will be */
#define USB_HEIGHT_SIZE        720        /* a raw video with 1280x720 size */
#define MIPI_WIDTH_SIZE    1280           /* The output data of v4l2src in this application will be */
#define MIPI_HEIGHT_SIZE   960            /* a raw video with 1280x960 size */
#define F_NV12             "NV12"
#define F_F32LE            "F32LE"
#define VARIABLE_RATE      1
#define OUTPUT_FILE        "RECORD_Multimedia.mkv"
#define ARG_PROGRAM_NAME   0
#define ARG_MICROPHONE     1
#define ARG_CAMERA         2
#define ARG_WIDTH          3
#define ARG_HEIGHT         4
#define ARG_COUNT          5

enum camera_type {
  NO_CAMERA,
```

```
  MIPI_CAMERA,
  USB_CAMERA
};

/* Supported resolutions of MIPI camera */
const char *mipi_resolutions[] = {
  "1280x960",
  "1920x1080",
  "2592x1944",
  NULL,
};

/* Supported resolutions of USB camera */
const char *usb_resolutions[] = {
  "320x240",
  "640x480",
  "800x600",
  "1280x720",
  NULL,
};

static GstElement *pipeline;

void
signalHandler (int signal)
{
  if (signal == SIGINT) {
    gst_element_send_event (pipeline, gst_event_new_eos ());
  }
}

static void
link_to_multiplexer (GstPad * tolink_pad, GstElement * mux)
{
  GstPad *pad;
  gchar *srcname, *sinkname;

  srcname = gst_pad_get_name (tolink_pad);
  pad = gst_element_get_compatible_pad (mux, tolink_pad, NULL);
  if (gst_pad_link (tolink_pad, pad) != GST_PAD_LINK_OK) {
    g_print ("cannot link request pad\n");
  }
  sinkname = gst_pad_get_name (pad);
  gst_object_unref (GST_OBJECT (pad));

  g_print ("A new pad %s was created and linked to %s\n", sinkname, srcname);
  g_free (sinkname);
  g_free (srcname);
}

/* Check type of camera
 * return NO_CAMERA: Unsupported camera
 * return MIPI_CAMERA: MIPI camera detected
 * return USB_CAMERA: USB camera detected */
enum camera_type
check_camera_type (const char *device);


/* Print supported resolutions in console*/
void
print_supported_resolutions (char *resolution, const char* supported_resolutions[]);

/* Check resolution in program argument is supported or not
 * Supported resolutions are defined in
 * usb_resolutions and mipi_resolutions */
bool
check_resolution (char *resolution, const char *supported_resolutions[]);

/* Check resolution in program argument is supported or not
 * If not, display list of supported resolutions in console
 * else store resolution to width, height */
bool
get_resolution (char *arg_width, char *arg_height, int *width, int *height, enum camera_type camera);
```

```
int
main (int argc, char *argv[])
{
  GstElement *cam_src, *cam_queue, *cam_capsfilter, *video_converter,
      *video_conv_capsfilter, *video_encoder, *video_parser, *audio_src,
      *audio_queue, *audio_converter, *audio_conv_capsfilter, *audio_encoder,
      *muxer, *sink;

  GstBus *bus;
  GstMessage *msg;
  GstPad *srcpad;
  GstCaps *cam_caps, *video_conv_caps, *audio_conv_caps;
  enum camera_type camera = NO_CAMERA;
  int width = 0;
  int height = 0;

  const gchar *output_file = OUTPUT_FILE;

  if ((argc != 3) && (argc != ARG_COUNT)) {
    g_print ("Error: Invalid arugments.\n");
    g_print ("Usage: %s <microphone device> <camera device> [width] [height]\n",
argv[ARG_PROGRAM_NAME]);
    return -1;
  }

  camera = check_camera_type (argv[ARG_CAMERA]);
  if (camera == NO_CAMERA) {
    return -1;
  } else if (camera == MIPI_CAMERA) {
    width = MIPI_WIDTH_SIZE;
    height = MIPI_HEIGHT_SIZE;
  } else {
    width = USB_WIDTH_SIZE;
    height = USB_HEIGHT_SIZE;
  }

  /* Parse resolution from program argument */
  if (argc == ARG_COUNT) {
    char hostname[20];
    gethostname (hostname, 20);
    if (!strcmp (hostname, "ek874") && (camera == MIPI_CAMERA)) {
      g_print ("RZ/G2E only supports 1280x960 resolution.\n");
      g_print ("Set 1280x960 resolution as default.\n");
    } else {
      if (!get_resolution (argv[ARG_WIDTH], argv[ARG_HEIGHT],
                &width, &height, camera)) {
        return -1;
      }
    }
  }

  /* Initialization */
  gst_init (&argc, &argv);

  /* Create GStreamer elements */
  pipeline = gst_pipeline_new ("audio-video-record");

  /* Video elements */
  cam_src = gst_element_factory_make ("v4l2src", "cam-src");
  cam_queue = gst_element_factory_make ("queue", "cam-queue");
  cam_capsfilter = gst_element_factory_make ("capsfilter", "cam_caps");
  video_conv_capsfilter =
      gst_element_factory_make ("capsfilter", "video-conv-caps");
  video_encoder = gst_element_factory_make ("omxh264enc", "video-encoder");
  video_parser = gst_element_factory_make ("h264parse", "h264-parser");

  if (camera == MIPI_CAMERA) {
    video_converter = gst_element_factory_make ("vspmfilter", "video-converter");
  } else {
    video_converter = gst_element_factory_make ("videoconvert", "video-converter");
  }

  /* Audio elements */
  audio_src = gst_element_factory_make ("alsasrc", "audio-src");
```

```
audio_queue = gst_element_factory_make ("queue", "audio-queue");
audio_converter = gst_element_factory_make ("audioconvert", "audio-conv");
audio_conv_capsfilter =
    gst_element_factory_make ("capsfilter", "audio-conv-caps");
audio_encoder = gst_element_factory_make ("vorbisenc", "audio-encoder");

/* container and output elements */
muxer = gst_element_factory_make ("matroskamux", "mkv-muxer");
sink = gst_element_factory_make ("filesink", "file-output");

if (!pipeline || !cam_src || !cam_queue || !cam_capsfilter || !video_converter
    || !video_conv_capsfilter || !video_encoder || !video_parser || !audio_src
    || !audio_queue || !audio_converter || !audio_conv_capsfilter
    || !audio_encoder || !muxer || !sink) {
  g_printerr ("One element could not be created. Exiting.\n");
  return -1;
}

/* Set properties */

/* for video elements */

if (camera == MIPI_CAMERA) {
  /* Set property "dmabuf-use" of vspmfilter to true */
  /* Without it, the output file will be broken video */
  g_object_set (G_OBJECT (video_converter), "dmabuf-use", true, NULL);
  /* Set properties of the encoder element - omxh264enc */
  g_object_set (G_OBJECT (video_encoder), "target-bitrate", MIPI_BITRATE_OMXH264ENC,
      "control-rate", VARIABLE_RATE, "interval_intraframes", 14,
      "periodicty-idr", 2, NULL);

  /* Create camera caps */
  cam_caps =
      gst_caps_new_simple ("video/x-raw", "format", G_TYPE_STRING, "UYVY",
          "width", G_TYPE_INT, width, "height", G_TYPE_INT, height, NULL);
} else {
  /* Set properties of the encoder element - omxh264enc */
  g_object_set (G_OBJECT (video_encoder), "target-bitrate", USB_BITRATE_OMXH264ENC,
      "control-rate", VARIABLE_RATE, NULL);

  /* Create camera caps */
  cam_caps =
      gst_caps_new_simple ("video/x-raw", "width", G_TYPE_INT, width,
          "height", G_TYPE_INT, height, NULL);
}

/* Set input video device file of the source element - v4l2src */
g_object_set (G_OBJECT (cam_src), "device", argv[ARG_CAMERA], NULL);

/* for audio elements */

/* set input device (microphone) of the source element - alsasrc */
g_object_set (G_OBJECT (audio_src), "device", argv[ARG_MICROPHONE], NULL);

/* set target bitrate of the encoder element - vorbisenc */
g_object_set (G_OBJECT (audio_encoder), "bitrate", BITRATE_ALSASRC, NULL);

/* Set output file location of the sink element - filesink */
g_object_set (G_OBJECT (sink), "location", output_file, NULL);

video_conv_caps =
    gst_caps_new_simple ("video/x-raw", "format", G_TYPE_STRING, F_NV12,
    NULL);
audio_conv_caps =
    gst_caps_new_simple ("audio/x-raw",
    "format", G_TYPE_STRING, F_F32LE,
    "channels", G_TYPE_INT, CHANNEL, "rate", G_TYPE_INT, SAMPLE_RATE, NULL);

/* set caps property for capsfilters */
g_object_set (G_OBJECT (cam_capsfilter), "caps", cam_caps, NULL);
g_object_set (G_OBJECT (video_conv_capsfilter), "caps", video_conv_caps,
    NULL);
g_object_set (G_OBJECT (audio_conv_capsfilter), "caps", audio_conv_caps,
    NULL);
```

```
/* unref caps after usage */
gst_caps_unref (cam_caps);
gst_caps_unref (video_conv_caps);
gst_caps_unref (audio_conv_caps);

/* Add all elements into the pipeline */
gst_bin_add_many (GST_BIN (pipeline), cam_src, cam_queue, cam_capsfilter,
    video_converter, video_conv_capsfilter, video_encoder, video_parser,
    audio_src, audio_queue, audio_converter, audio_conv_capsfilter,
    audio_encoder, muxer, sink, NULL);

/* Link the elements together */
if (gst_element_link_many (cam_src, cam_queue, cam_capsfilter,
        video_converter, video_conv_capsfilter, video_encoder, video_parser,
        NULL) != TRUE) {
  g_printerr ("Camera record elements could not be linked.\n");
  gst_object_unref (pipeline);
  return -1;
}
if (gst_element_link_many (audio_src, audio_queue, audio_converter,
        audio_conv_capsfilter, audio_encoder, NULL) != TRUE) {
  g_printerr ("Audio record elements could not be linked.\n");
  gst_object_unref (pipeline);
  return -1;
}
if (gst_element_link (muxer, sink) != TRUE) {
  g_printerr ("muxer and sink could not be linked.\n");
  gst_object_unref (pipeline);
  return -1;
}

/* link srcpad of h264parse to request pad of matroskamux */
srcpad = gst_element_get_static_pad (video_parser, "src");
link_to_multiplexer (srcpad, muxer);
gst_object_unref (srcpad);

/* link srcpad of vorbisenc to request pad of matroskamux */
srcpad = gst_element_get_static_pad (audio_encoder, "src");
link_to_multiplexer (srcpad, muxer);
gst_object_unref (srcpad);

/* Set the pipeline to "playing" state */
g_print ("Now recording, press [Ctrl] + [C] to stop...\n");
if (gst_element_set_state (pipeline,
        GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
  g_printerr ("Unable to set the pipeline to the playing state.\n");
  gst_object_unref (pipeline);
  return -1;
}

/* Handle signals gracefully. */
signal (SIGINT, signalHandler);

/* Wait until error or EOS */
bus = gst_element_get_bus (pipeline);
msg =
    gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
    GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
gst_object_unref (bus);

/* Note that because input timeout is GST_CLOCK_TIME_NONE,
   the gst_bus_timed_pop_filtered() function will block forever until a
   matching message was posted on the bus (GST_MESSAGE_ERROR or
   GST_MESSAGE_EOS). */

if (msg != NULL) {
  GError *err;
  gchar *debug_info;

  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_ERROR:
      gst_message_parse_error (msg, &err, &debug_info);
      g_printerr ("Error received from element %s: %s.\n",
          GST_OBJECT_NAME (msg->src), err->message);
      g_printerr ("Debugging information: %s.\n",
```

```
            debug_info ? debug_info : "none");
        g_clear_error (&err);
        g_free (debug_info);
        break;
      case GST_MESSAGE_EOS:
        g_print ("End-Of-Stream reached.\n");
        break;
      default:
        /* We should not reach here because we only asked for ERRORs and EOS */
        g_printerr ("Unexpected message received.\n");
        break;
    }
    gst_message_unref (msg);
  }

  /* Clean up nicely */
  g_print ("Returned, stopping recording...\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);

  g_print ("Deleting pipeline...\n");
  gst_object_unref (GST_OBJECT (pipeline));
  g_print ("Succeeded. Please check output file: %s\n", output_file);
  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.1 Audio Play, 3.2.5. Audio Record and 3.2.6. Video Record

Output location

```
#define OUTPUT_FILE       "RECORD_Multimedia.mkv"
const gchar *output_file = OUTPUT_FILE;
```

Command-line argument

```
if ((argc != 3) && (argc != ARG_COUNT)) {
  g_print ("Error: Invalid arugments.\n");
  g_print ("Usage: %s <microphone device> <camera device> [width] [height]\n", argv[ARG_PROGRAM_NAME]);
  return -1;
}
```

This application accepts 2 command-line argument:

- The first points to USB microphone device card (hw:1,0, for example). Note: You can find this value by following section 3.2.5.4 Special Instruction.

- The second points to USB camera/MIPI camera device file (/dev/video9, for example). Note: You can find this value by following section 3.2.6.4 Special Instruction.

Optional-argument `width` and `height` are used to set the resolution of camera.

Create elements

```
cam_src = gst_element_factory_make ("v4l2src", "cam-src");
cam_queue = gst_element_factory_make ("queue", "cam-queue");
cam_capsfilter = gst_element_factory_make ("capsfilter", "cam_caps");
video_conv_capsfilter =
    gst_element_factory_make ("capsfilter", "video-conv-caps");
video_encoder = gst_element_factory_make ("omxh264enc", "video-encoder");
video_parser = gst_element_factory_make ("h264parse", "h264-parser");

if (camera == MIPI_CAMERA) {
  video_converter = gst_element_factory_make ("vspmfilter", "video-converter");
} else {
  video_converter = gst_element_factory_make ("videoconvert", "video-converter");
}
```

```
audio_src = gst_element_factory_make ("alsasrc", "audio-src");
audio_queue = gst_element_factory_make ("queue", "audio-queue");
audio_converter = gst_element_factory_make ("audioconvert", "audio-conv");
audio_conv_capsfilter =
    gst_element_factory_make ("capsfilter", "audio-conv-caps");
audio_encoder = gst_element_factory_make ("vorbisenc", "audio-encoder");

muxer = gst_element_factory_make ("matroskamux", "mkv-muxer");
sink = gst_element_factory_make ("filesink", "file-output");
```

To record raw data from USB microphone and USB webcam or MIPI camera at the same time, then store them in MKV container, the following elements are used:

- Element `v4l2src` captures video from `V4L2` devices.
- Element `queue` (`cam_queue` and `audio_queue`) queues data until one of the limits specified by the `max-size-buffers`, `max-size-bytes`, and/or `max-size-time` properties has been reached. Any attempt to push more buffers into the queue will block the pushing thread until more space becomes available.
- Element `capsfilter` specifies raw video format, framerate, and resolution.
- Element `videoconvert` (used for USB camera) and element `vspmfilter` (used for MIPI camera) converts video frames to a format (such as: `NV12`) understood by `omxh264enc`.
- Element `omxh264enc` encodes raw video into H.264 compressed data.
- Element `h264parse` connects `omxh264enc` to `qtmux`.
- Element `alsasrc` reads data from an audio card using the ALSA API.
- Element `audioconvert` converts raw audio buffers to a format (such as: `F32LE`) understood by `vorbisenc`.
- Element `vorbisenc` encodes raw audio into a Vorbis stream.
- Element `matroskamux` merges audio stream and video stream to MKV container.
- Element `filesink` writes incoming data to a local file.

Set Element's properties

```
g_object_set (G_OBJECT (video_converter), "dmabuf-use", true, NULL);
g_object_set (G_OBJECT (video_encoder), "target-bitrate", MIPI_BITRATE_OMXH264ENC,
    "control-rate", VARIABLE_RATE, "interval_intraframes", 14, "periodicty-idr", 2, NULL);
g_object_set (G_OBJECT (video_encoder), "target-bitrate", USB_BITRATE_OMXH264ENC,
    "control-rate", VARIABLE_RATE, NULL);
g_object_set (G_OBJECT (cam_src), "device", argv[ARG_CAMERA], NULL);

g_object_set (G_OBJECT (audio_src), "device", argv[ARG_MICROPHONE], NULL);
g_object_set (G_OBJECT (audio_encoder), "bitrate", BITRATE_ALSASRC, NULL);

g_object_set (G_OBJECT (sink), "location", output_file, NULL);
```

The `g_object_set()` function is used to set some element's properties, such as:

- The `device` property of `v4l2src` element which points to camera's device file. Users will pass the device file as a command line argument to this application. Please refer to section 3.2.6.4 Special Instruction to find the value.
- The `location` property of `filesink` element which points to MKV output file.
- The `dmabuf-use` property of `vspmfilter` element which is set to `true`. This disallows `dmabuf` to be output buffer. If it is not set, the output file will be broken.
- The `target-bitrate` property of `omxh264enc` element is used to specify encoding bit rate. The higher bitrate, the better quality.
- The `control-rate` property of `omxh264enc` element is used to specify birate control method which is variable bitrate method in this case.
- The `interval_intraframes` property of `omxh264enc` element is used to specify interval of coding intra frames.
- The `periodicty-idr` property of `omxh264enc` is used to specify periodicity of IDR frames.
- The `device` property of `alsasrc` element which points to a microphone device. Users will pass the device card as a command line argument to this application. Please refer to section 3.2.5.4 Special Instruction to find the value.

- The bitrate property of vorbisenc element is used to specify encoding bit rate. The higher bitrate, the better quality.

```
/* MIPI camera*/
cam_caps = gst_caps_new_simple ("video/x-raw", "format", G_TYPE_STRING, "UYVY",
          "width", G_TYPE_INT, width, "height", G_TYPE_INT, height, NULL);
/* USB camera */
cam_caps = gst_caps_new_simple ("video/x-raw", "width", G_TYPE_INT, width,
          "height", G_TYPE_INT, height, NULL);

video_conv_caps = gst_caps_new_simple ("video/x-raw", "format", G_TYPE_STRING, F_NV12, NULL);
audio_conv_caps = gst_caps_new_simple ("audio/x-raw", "format", G_TYPE_STRING, F_F32LE,
    "channels", G_TYPE_INT, CHANNEL, "rate", G_TYPE_INT, SAMPLE_RATE, NULL);

g_object_set (G_OBJECT (cam_capsfilter), "caps", cam_caps, NULL);
g_object_set (G_OBJECT (video_conv_capsfilter), "caps", video_conv_caps, NULL);
g_object_set (G_OBJECT (audio_conv_capsfilter), "caps", audio_conv_caps, NULL);

gst_caps_unref (cam_caps);
gst_caps_unref (video_conv_caps);
gst_caps_unref (audio_conv_caps);
```

Capabilities (short: caps) describe the type of data which is streamed between two pads.

The gst_caps_new_simple() function creates new caps which holds these values. These caps are then added to caps property of capsfilter elements (g_object_set).

Note that both caps should be freed with gst_caps_unref() if they are not used anymore.

Build pipeline

```
gst_bin_add_many (GST_BIN (pipeline), cam_src, cam_queue, cam_capsfilter, video_converter,
    video_conv_capsfilter, video_encoder, video_parser, audio_src, audio_queue,
    audio_converter, audio_conv_capsfilter, audio_encoder, muxer, sink, NULL);

gst_element_link_many (cam_src, cam_queue, cam_capsfilter,
        video_converter, video_conv_capsfilter, video_encoder, video_parser,NULL);

gst_element_link_many (audio_src, audio_queue, audio_converter,
        audio_conv_capsfilter, audio_encoder, NULL);

gst_element_link (muxer, sink);
```

The reason for the separation is that the sink pad of matroskamux (muxer) cannot be created automatically but is only created on demand. This application uses self-defined function link_to_multiplexer() to link the sink pad to source pad of vorbisenc (encoder) and h264parse (parser). That's why its sink pad is called Request Pad.

Note that the order counts, because links must follow the data flow (this is, from source elements to sink elements).

Link request pads

```
srcpad = gst_element_get_static_pad (video_parser, "src");
link_to_multiplexer (srcpad, muxer);
gst_object_unref (srcpad);

srcpad = gst_element_get_static_pad (audio_encoder, "src");
link_to_multiplexer (srcpad, muxer);
gst_object_unref (srcpad);
```

This block gets the source pad (srcpad) of vorbisenc (encoder) and h264parse (parser), then calls link_to_multiplexer() to link them to the sink pad of matroskamux (muxer).

Note that the srcpad should be freed with gst_object_unref() if it is not used anymore.

```
static void link_to_multiplexer (GstPad * tolink_pad, GstElement * mux)
{
  pad = gst_element_get_compatible_pad (mux, tolink_pad, NULL);
  gst_pad_link (tolink_pad, pad);
```

```
  gst_object_unref (GST_OBJECT (pad));
}
```

This function uses `gst_element_get_compatible_pad()` to request a sink pad (`pad`) which is compatible with the source pad (`tolink_pad`) of `matroskamux` (`mux`), then calls `gst_pad_link()` to link them together.

Note that the `pad` should be freed with `gst_object_unref()` if it is not used anymore.

Play pipeline

```
gst_element_set_state (pipeline, GST_STATE_PLAYING);
```

Every pipeline has an associated state. To start audio recording, the pipeline needs to be set to `PLAYING` state.

```
signal (SIGINT, signalHandler);
```

This application will stop recording if user presses Ctrl-C. To do so, it uses `signal()` to bind `SIGINT` (interrupt from keyboard) to `signalHandler()`.

To know how this function is implemented, please refer to the following code block:

```
void signalHandler (int signal)
{
  if (signal == SIGINT) {
    gst_element_send_event (pipeline, gst_event_new_eos ());
  }
}
```

It calls `gst_element_send_event()` to send `EOS (End-of-Stream)` signal (`gst_event_new_eos`) to the `pipeline`. This makes `gst_bus_timed_pop_filtered()` return. Finally, the program cleans up GStreamer objects and exits.

3.2.7.2 Result

An MKV file will be generated after running this application.

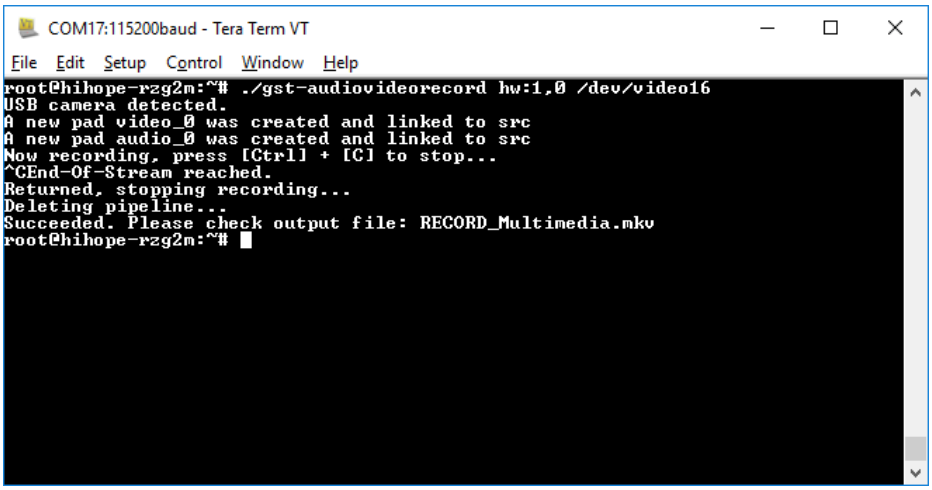


Figure 3.16 Audio Video Record result

3.2.7.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.7.4 Special Instruction

Recommended USB cameras:

Option 1: Logitech USB HD Webcam C270 (model: V-U0018).

Option 2: Logitech USB HD 1080p Webcam C930E.

Option 3**:** Logitech USB UHD Webcam BRIO.

Recommended MIPI camera:

MIPI Mezzanine Adapter and OV5645 camera.

To check the output file:

Option 1: `VLC` media player (https://www.videolan.org/vlc/index.html).

Option 2: Tool `gst-launch-1.0` (on board):

```
$ gst-launch-1.0 filesrc location=RECORD_Multimedia.mkv ! matroskademux name=d d. ! queue max-size-time=10000000000 ! vorbisdec ! audioconvert ! audioresample ! autoaudiosink d. ! queue ! h264parse ! omxh264dec ! waylandsink
```

3.2.8 Receive Streaming Video

Receive and display streaming video.

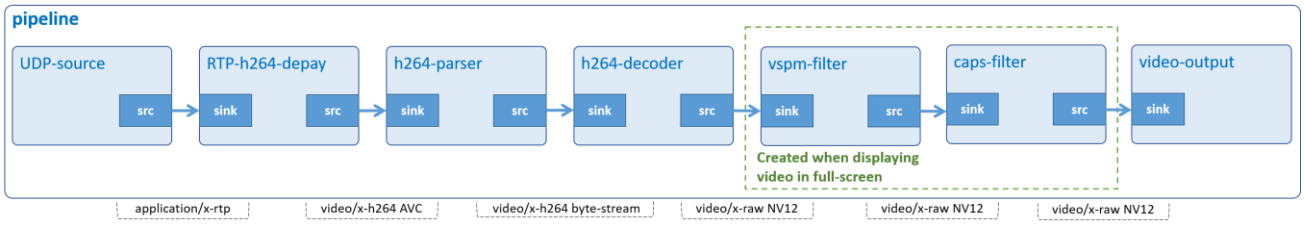


Figure 3.17 Receive Streaming Video pipeline

(1) Files

- main.c

(2) main.c

```
#include <stdio.h>
#include <string.h>
#include <gst/gst.h>
#include <stdlib.h>
#include <stdbool.h>
#include <wayland-client.h>
#include <libgen.h>

#define PORT 5000

static GMainLoop *main_loop;

void
signalHandler (int signal)
{
  if (signal == SIGINT) {
    g_main_loop_quit (main_loop);
    g_print ("\n");
  }
}

/* The function bus_call() will be called when a message is received */
static gboolean
bus_call (GstBus * bus, GstMessage * msg, gpointer data)
{
  GMainLoop *loop = (GMainLoop *) data;

  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_ERROR:{
      gchar *debug;
      GError *error;

      gst_message_parse_error (msg, &error, &debug);
      g_free (debug);

      g_printerr ("Error: %s\n", error->message);
      g_error_free (error);

      g_main_loop_quit (loop);
      break;
    }
    default:
      /* Don't care other message */
      break;
  }
  return TRUE;
}

int
```

```
main (int argc, char *argv[])
{
  bool fullscreen = false;
  if ((argc > 2) || ((argc == 2) && (strcmp (argv[1], "-s")))) {
    g_print ("Error: Invalid arguments.\n");
    g_print ("Usage: %s [-s]\n", argv[0]);
    return -1;
  }

  /* Check -s option */
  if (argc == 2) {
    if (strcmp (argv[1], "-s") == 0) {
      fullscreen = true;
    }
  }

  struct wayland_t *wayland_handler = NULL;
  struct screen_t *main_screen = NULL;

  GstElement *pipeline, *source, *depayloader, *parser,
      *decoder, *filter, *capsfilter, *sink;
  GstCaps *caps;
  GstBus *bus;
  guint bus_watch_id;


  /* Get a list of available screen */
  wayland_handler = get_available_screens();

  /* Get main screen */
  main_screen = get_main_screen(wayland_handler);
  if (main_screen == NULL)
  {
    g_printerr("Cannot find any available screens. Exiting.\n");
    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Initialization */
  gst_init (&argc, &argv);
  main_loop = g_main_loop_new (NULL, FALSE);

  /* Create GStreamer elements */
  pipeline = gst_pipeline_new ("receive-streaming");
  source = gst_element_factory_make ("udpsrc", "udp-src");
  depayloader = gst_element_factory_make ("rtph264depay", "h264-depay");
  parser = gst_element_factory_make ("h264parse", "h264-parser");
  decoder = gst_element_factory_make ("omxh264dec", "h264-decoder");
  sink = gst_element_factory_make ("waylandsink", "video-sink");

  if (!pipeline || !source || !depayloader || !parser
      || !decoder || !sink) {
    g_printerr ("One element could not be created. Exiting.\n");
    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Add all elements into the pipeline */
  gst_bin_add_many (GST_BIN (pipeline), source, depayloader,
      parser, decoder, sink, NULL);

  bus = gst_element_get_bus (pipeline);
  bus_watch_id = gst_bus_add_watch (bus, bus_call, main_loop);
  gst_object_unref (bus);

  /* For valid RTP packets encapsulated in GstBuffers,
     we use the caps with mime type application/x-rtp.
     Select listening port: 5000 */
  caps = gst_caps_new_empty_simple ("application/x-rtp");
  g_object_set (G_OBJECT (source), "port", PORT, "caps", caps, NULL);

  /* Set max-lateness maximum number of nanoseconds that a buffer can be late
     before it is dropped (-1 unlimited).
     Generate Quality-of-Service events upstream to FALSE */
```

```
  g_object_set (G_OBJECT (sink), "max-lateness", -1, "qos", FALSE, NULL);

  /* Set position for displaying (0, 0) */
  g_object_set (G_OBJECT (sink), "position-x", main_screen->x, "position-y", main_screen->y, NULL);

  /* unref cap after use */
  gst_caps_unref (caps);

  if (!fullscreen) {
    /* Link the elements together */
    if (gst_element_link_many (source, depayloader, parser,
            decoder, sink, NULL) != TRUE) {
      g_printerr ("Elements could not be linked.\n");
      gst_object_unref (pipeline);
      destroy_wayland(wayland_handler);
      return -1;
    }
  }
  else {
    /* Create 2 more elements */
    filter = gst_element_factory_make ("vspmfilter", "vspm-filter");
    capsfilter = gst_element_factory_make ("capsfilter", "caps-filter");

    if (!filter || !capsfilter) {
      g_printerr ("One element could not be created. Exiting.\n");
      gst_object_unref (pipeline);
      destroy_wayland(wayland_handler);
      return -1;
    }

    /* Set property "dmabuf-use" of vspmfilter to true */
    /* Without it, waylandsink will display broken video */
    g_object_set (G_OBJECT (filter), "dmabuf-use", TRUE, NULL);

    /* Create simple cap which contains video's resolution */
    caps = gst_caps_new_simple ("video/x-raw",
        "width", G_TYPE_INT, main_screen->width,
        "height", G_TYPE_INT, main_screen->height, NULL);

    /* Add cap to capsfilter element */
    g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
    gst_caps_unref (caps);

    /* Add elements into the pipeline */
    gst_bin_add_many (GST_BIN (pipeline), filter, capsfilter, NULL);

    /* Link the elements together */
    if (gst_element_link_many (source, depayloader, parser,
            decoder, filter, capsfilter, sink, NULL) != TRUE) {
      g_printerr ("Elements could not be linked.\n");
      gst_object_unref (pipeline);
      destroy_wayland(wayland_handler);
      return -1;
    }
  }

  /* Set the pipeline to "playing" state */
  g_print ("Now receiving...\n");
  if (gst_element_set_state (pipeline,
          GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
    g_printerr ("Unable to set the pipeline to the playing state.\n");
    gst_object_unref (pipeline);
    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Handle signals gracefully. */
  signal (SIGINT, signalHandler);

  /* Iterate */
  g_print ("Running...\n");
  g_main_loop_run (main_loop);

  /* Clean up "wayland_t" structure */
```

```
  destroy_wayland(wayland_handler);

  /* Clean up nicely */
  g_print ("Returned, stopping receiving...\n");
  g_source_remove (bus_watch_id);
  g_main_loop_unref (main_loop);

  gst_element_set_state (pipeline, GST_STATE_NULL);

  g_print ("Deleting pipeline...\n");
  gst_object_unref (GST_OBJECT (pipeline));
  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.1 Audio Play and section 3.2.2 Video Play

Create event loop

```
main_loop = g_main_loop_new (NULL, FALSE);
```

Function `g_main_loop_new()` creates a new `GMainLoop` structure with default context (`GMainContext`).

Basically, the main event loop manages all the available sources of events. To allow multiple independent sets of sources to be handled in different threads, each source is associated with a `GMainContext`. A `GMainContext` can only be running in a single thread, but sources can be added to it and removed from it from other threads.

Create elements

```
source = gst_element_factory_make ("udpsrc", "udp-src");
depayloader = gst_element_factory_make ("rtph264depay", "h264-depay");
parser = gst_element_factory_make ("h264parse", "h264-parser");
decoder = gst_element_factory_make ("omxh264dec", "h264-decoder");
filter = gst_element_factory_make ("vspmfilter", "vspm-filter");
capsfilter = gst_element_factory_make ("capsfilter", "caps-filter");
sink = gst_element_factory_make ("waylandsink", "video-sink");
```

To receive and display streaming video, the following elements are used:

- Element `udpsrc` reads UDP packets from the network.
- Element `rtph264depay` extracts H.264 video from RTP packets.
- Element `h264parse` parses H.264 video from `byte stream` format to `AVC` format which `omxh264dec` can process.
- Element `omxh264dec` decompresses H.264 stream to raw NV12-formatted video.
- Element `vspmfilter` handles video scaling.
- Element `capsfilter` contains screen resolution so that `vspmfilter` can scale video frames based on this value.
- Element `waylandsink` creates its own window and renders the decoded video frames to that.

Set element's properties

```
caps = gst_caps_new_empty_simple ("application/x-rtp");

g_object_set (G_OBJECT (source), "port", PORT, "caps", caps, NULL);
gst_caps_unref (caps);

g_object_set (G_OBJECT (sink), "max-lateness", -1, "qos", FALSE, NULL);
g_object_set (G_OBJECT (sink), "position-x", main_screen->x, "position-y", main_screen->y, NULL);
g_object_set (G_OBJECT (filter), "dmabuf-use", TRUE, NULL);
```

The `g_object_set()` function is used to set some element's properties, such as:

- The `port` property of `udpsrc` element which is set to port `5000`.
- The `caps` property of `udpsrc` element which specifies streaming format RTP (`application/x-rtp`).

- The max-lateness property of waylandsink element which specifies maximum number of nanoseconds that a buffer can be late before it is dropped. This means setting -1 to this property will cause waylandsink to wait the buffer forever.
- The qos property of waylandsink element which disable Quality-of-Service events upstream.
- The position-x and position-y properties of waylandsink element which point to (x, y) coordinate of Wayland desktop.
- The dmabuf-use property of vspmfilter element which is set to true. This disallows dmabuf to be output buffer. If it is not set, waylandsink will display broken video frames.

```
caps = gst_caps_new_simple ("video/x-raw", "width", G_TYPE_INT, main_screen->width,
       "height", G_TYPE_INT, main_screen->height, NULL);

g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);
```

The gst_caps_new_simple() function creates a new cap which holds screen resolution. This cap is then added to caps property of capsfilter (g_object_set) so that vspmfilter element can use these values to resize video frames to match screen resolution.

Note that the caps should be freed with gst_caps_unref() if it is not used anymore.

Add bus watch and handle messages (GstMessage)

```
bus = gst_element_get_bus (pipeline)
bus_watch_id = gst_bus_add_watch (bus, bus_call, main_loop);
gst_object_unref (bus);
```

The gst_element_get_bus() function returns the bus of the pipeline.

Basically, bus is the object responsible for delivering messages (GstMessage) generated by the elements, in order and to the application thread. This last point is important, because the actual streaming of media is done in another thread than the application.

In this application, the messages will be extracted from the bus asynchronously, using self-defined function bus_call(). Note that the bus should be freed with gst_caps_unref() if it is not used anymore.

To know how bus_call() is implemented, please refer to the following code block:

```
static gboolean
bus_call (GstBus * bus, GstMessage * msg, gpointer data)
{
  GMainLoop *loop = (GMainLoop *) data;

  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_ERROR:{
      gchar *debug;
      GError *error;

      gst_message_parse_error (msg, &error, &debug);
      g_free (debug);

      g_printerr ("Error: %s\n", error->message);
      g_error_free (error);

      g_main_loop_quit (loop);
      break;
    }
    default:
      /* Don't care other message */
      break;
  }
  return TRUE;
}
```

This function will be called when a message is received. If an error occurs, it will call `g_main_loop_quit()` to stop `GMainLoop`. This makes `g_main_loop_run()` return. Finally, the application cleans up GSteamer objects and exits.

Playing pipeline

```
gst_element_set_state (pipeline, GST_STATE_PLAYING);
```

Every pipeline has an associated state. To start receiving and displaying streaming video, the pipeline needs to be set to `PLAYING` state.

Stop pipeline

```
signal (SIGINT, signalHandler);
```

The application uses `signal()` to bind `SIGINT` (interrupt from keryboard) to `signalHandler()` function.
To know how this function is implemented, please refer to the following code block:

```
void signalHandler (int signal)
{
  if (signal == SIGINT) {
    g_main_loop_quit (main_loop);
    g_print ("\n");
  }
}
```

Run main loop

```
g_main_loop_run (main_loop);
```

This function runs main loop until `g_main_loop_quit()` is called on the loop (context `NULL`). In other words, it will make the context check if anything it watches for has happened. For example, when a message has been posted on the bus (`gst_element_get_bus`), the default main context will automatically call `bus_call()` to notify the message.

3.2.8.2 Result

This application waits for and displays streaming video. Note that it does not output audio.

Figure 3.18 Receive Streaming Video result

3.2.8.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.8.4 Special Instruction

- Please configure IPv4 address (as below) before running this application:

```
$ ifconfig <Ethernet Interface> <IPv4 address>
```

- For example:

```
$ ifconfig eth0 192.168.5.20
```

Note:

The application might drop frames when receiving high-bitrate videos because the decoder (omxh264dec) does not work properly. This issue is being investigated.

3.2.9 Send Streaming Video

Stream video.

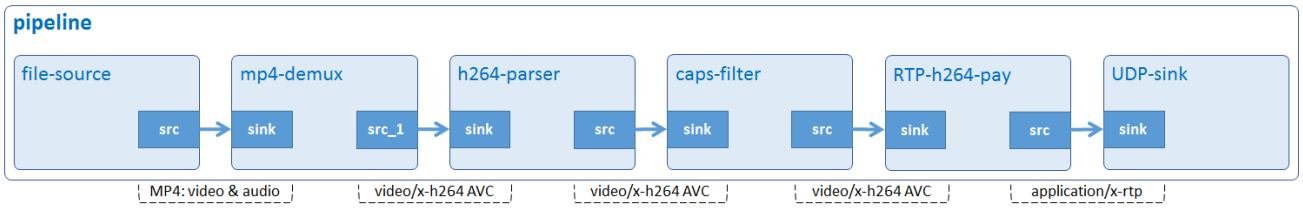


Figure 3.19 Send Streaming Video pipeline

3.2.9.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <gst/gst.h>
#include <stdbool.h>
#include <stdio.h>
#include <strings.h>
#include <libgen.h>
#include <string.h>
#include <arpa/inet.h>

#define PORT         5000
#define PAYLOAD_TYPE  96
#define TIME          3  /* Send SPS and PPS Insertion every 3 second */

/* Macros for program's arguments */
#define ARG_PROGRAM_NAME 0
#define ARG_IP_ADDRESS   1
#define ARG_INPUT        2
#define ARG_COUNT        3

static void
on_pad_added (GstElement * element, GstPad * pad, gpointer data)
{
  GstPad *sinkpad;
  GstElement *decoder = (GstElement *) data;

  /* We can now link this pad with the H.264-decoder sink pad */
  g_print ("Dynamic pad created, linking demuxer/decoder\n");

  sinkpad = gst_element_get_static_pad (decoder, "sink");

  gst_pad_link (pad, sinkpad);

  gst_object_unref (sinkpad);
}

bool isValidIpAddress(char *ipAddress) {
  struct sockaddr_in sa;
  int result = inet_pton(AF_INET, ipAddress, &(sa.sin_addr));
  return result != 0;
}

int
main (int argc, char *argv[])
{
  GstElement *pipeline, *source, *demuxer, *parser, *parser_capsfilter, *payloader, *sink;
  GstBus *bus;
```

```
GstMessage *msg;
GstCaps *parser_caps;
const char* ext;
char* file_name;

const gchar *input_file = argv[ARG_INPUT];
if (!is_file_exist(input_file))
{
  g_printerr("Cannot find input file: %s. Exiting.\n", input_file);
  return -1;
}

if (!isValidIpAddress (argv[ARG_IP_ADDRESS])) {
  g_print ("IP is not valid\n");
  return -1;
}

file_name = basename ((char*) input_file);
ext = get_filename_ext (file_name);

if (strcasecmp ("mp4", ext) != 0) {
  g_print ("Unsupported video type. MP4 format is required\n");
  return -1;
}

/* Initialization */
gst_init (&argc, &argv);

/* Create GStreamer elements */
pipeline = gst_pipeline_new ("send-streaming");
source = gst_element_factory_make ("filesrc", "file-src");
demuxer = gst_element_factory_make ("qtdemux", "mp4-demuxer");
parser = gst_element_factory_make ("h264parse", "h264-parser");
parser_capsfilter =
    gst_element_factory_make ("capsfilter", "parser-capsfilter");
payloader = gst_element_factory_make ("rtph264pay", "h264-payloader");
sink = gst_element_factory_make ("udpsink", "stream-output");

if (!pipeline || !source || !demuxer || !parser || !parser_capsfilter
    || !payloader || !sink) {
  g_printerr ("One element could not be created. Exiting.\n");
  return -1;
}

/* Set input video device file of the source element - v4l2src */
g_object_set (G_OBJECT (source), "location", input_file, NULL);

/* Send SPS and PPS Insertion every 3 second.
     https://www.quora.com/What-are-SPS-and-PPS-in-video-codecs
   For H.264 AVC, dynamic payload is used.
     https://en.wikipedia.org/wiki/RTP_audio_video_profile
     http://www.freesoft.org/CIE/RFC/1890/29.htm */
/* The stream is H.264 AVC so that the payload type must be a dynamic type (96-127).
   The value 96 is used as a default value */
g_object_set (G_OBJECT (payloader), "pt", PAYLOAD_TYPE, "config-interval", TIME, NULL);

/* Listening to port 5000 of the host */
g_object_set (G_OBJECT (sink), "host", argv[ARG_IP_ADDRESS], "port", PORT, NULL);

/* create simple caps */
parser_caps =
    gst_caps_new_simple ("video/x-h264", "stream-format", G_TYPE_STRING,
    "avc", "alignment", G_TYPE_STRING, "au", NULL);

/* set caps property for capsfilters */
g_object_set (G_OBJECT (parser_capsfilter), "caps", parser_caps, NULL);

/* unref caps after usage */
gst_caps_unref (parser_caps);

/* Add all elements into the pipeline */
gst_bin_add_many (GST_BIN (pipeline), source, demuxer, parser,
    parser_capsfilter, payloader, sink, NULL);

/* Link the elements together */
```

```
  if (gst_element_link (source, demuxer) != TRUE) {
    g_printerr ("Elements could not be linked.\n");
    gst_object_unref (pipeline);
    return -1;
  }
  if (gst_element_link_many (parser, parser_capsfilter, payloader, sink,
          NULL) != TRUE) {
    g_printerr ("Elements could not be linked.\n");
    gst_object_unref (pipeline);
    return -1;
  }

  /* Dynamic link */
  g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added), parser);

  /* Set the pipeline to "playing" state */
  g_print ("Now streaming: %s...\n", input_file);
  if (gst_element_set_state (pipeline,
          GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
    g_printerr ("Unable to set the pipeline to the playing state.\n");
    gst_object_unref (pipeline);
    return -1;
  }

  /* Wait until error or EOS */
  bus = gst_element_get_bus (pipeline);
  msg =
      gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
      GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
  gst_object_unref (bus);

  /* Note that because input timeout is GST_CLOCK_TIME_NONE,
     the gst_bus_timed_pop_filtered() function will block forever until a
     matching message was posted on the bus (GST_MESSAGE_ERROR or
     GST_MESSAGE_EOS). */

  if (msg != NULL) {
    GError *err;
    gchar *debug_info;

    switch (GST_MESSAGE_TYPE (msg)) {
      case GST_MESSAGE_ERROR:
        gst_message_parse_error (msg, &err, &debug_info);
        g_printerr ("Error received from element %s: %s.\n",
            GST_OBJECT_NAME (msg->src), err->message);
        g_printerr ("Debugging information: %s.\n",
            debug_info ? debug_info : "none");
        g_clear_error (&err);
        g_free (debug_info);
        break;
      case GST_MESSAGE_EOS:
        g_print ("End-Of-Stream reached.\n");
        break;
      default:
        /* We should not reach here because we only asked for ERRORs and EOS */
        g_printerr ("Unexpected message received.\n");
        break;
    }
    gst_message_unref (msg);
  }

  /* Clean up nicely */
  g_print ("Returned, stopping streaming...\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);
  g_print ("Deleting pipeline...\n");
  gst_object_unref (GST_OBJECT (pipeline));
  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.1 Audio Play.

Command-line argument

```
if (argc != ARG_COUNT) {
  g_print ("Invalid arugments.\n");
  g_print ("Format: %s <IP address> <path to MP4>.\n", argv[ARG_PROGRAM_NAME]);

  return -1;
}

if (!isValidIpAddress (argv[ARG_IP_ADDRESS])) {
  g_print ("IP is not valid\n");
  return -1;
}
```

This application accepts two command-line arguments as below:

- IPv4 address: If invalid, the application prints error and exits. Please refer to section 3.2.9.4 Special Instruction to set and pass the address to it.
- A MP4 file's location.

Create elements

```
source = gst_element_factory_make ("filesrc", "file-src");
demuxer = gst_element_factory_make ("qtdemux", "mp4-demuxer");
parser = gst_element_factory_make ("h264parse", "h264-parser");
parser_capsfilter = gst_element_factory_make ("capsfilter", "parser-capsfilter");
payloader = gst_element_factory_make ("rtph264pay", "h264-payloader");
sink = gst_element_factory_make ("udpsink", "stream-output");
```

To stream H.264 video, the following elements are used:

- Element `filesrc` reads data from a local file.
- Element `qtdemux` de-multiplexes an `MP4` file into audio and video stream.
- Element `h264parse` parses H.264 video from byte-stream format to `AVC` format.
- Element `capsfilter` specifies video format `video/x-h264`, `AVC`, and `au` alignment.
- Element `rtph264pay` payload-encodes H264 video into `RTP` packets.
- Element `udpsink` sends UDP packets to the network

Set element's properties

```
g_object_set (G_OBJECT (source), "location", input_file, NULL);
g_object_set (G_OBJECT (payloader), "pt", PAYLOAD_TYPE, "config-interval", TIME, NULL);
g_object_set (G_OBJECT (sink), "host", argv[ARG_IP_ADDRESS], "port", PORT, NULL);
```

The `g_object_set()` function is used to set some element's properties, such as:

- The `location` property of `filesrc` element which points to an MP4 file.
- The `pt` property of `rtph264pay` element which is the payload type of the packets. Because the input stream is `H.264 AVC`, this property must be in dynamic range from 96 to 127. For reference purpose, it is set to `96`.
- The `config-interval` property of `rtph264pay` element which is the interval time to insert SPS and PPS. They contain data that are required by H.264 decoder. If lost, the receiver (such as: Receive Streaming Video) cannot reconstruct video frames. To avoid this issue, SPS and PPS will be sent for every 3 seconds.
- The `host` and `port` properties of `udpsink` element which are the IPv4 address and port to send the packets to. In this application, the port is hard-coded to `5000` while the address is provided by users.

```
parser_caps = gst_caps_new_simple ("video/x-h264", "stream-format", G_TYPE_STRING,
                                   "avc", "alignment", G_TYPE_STRING, "au", NULL);

g_object_set (G_OBJECT (parser_capsfilter), "caps", parser_caps, NULL);
gst_caps_unref (parser_caps);
```

Capabilities (short: caps) describe the type of data which is streamed between two pads.

This application creates new cap (`gst_caps_new_simple`) which specifies data format `video/x-h264`, `AVC`, and `au` alignment. This cap is then added to `caps` property of `capsfilter` element (`g_object_set`).

Next, `capsfilter` is linked between `h264parse` and `rtph264pay` to negotiate the data format flowing through these 2 elements.

Note that the cap should be freed with `gst_caps_unref()` if it is not used anymore.

3.2.9.2 Result

This application streams video at location /home/media/videos/sintel_trailer-720p.mp4.

Note that it does not stream audio.

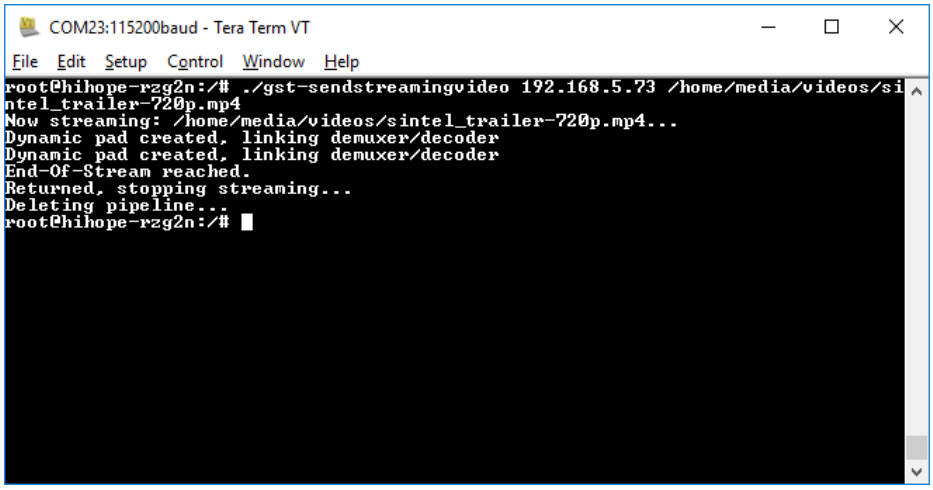


Figure 3.20 Send Streaming Video result

3.2.9.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.9.4 Special Instruction

- Please configure IPv4 address (as below) before running this application:

```
$ ifconfig <Ethernet Interface> <IP address>
```

- For example:

```
$ ifconfig eth0 192.168.5.25
```

- Input an IP address (192.168.5.20, for example) to which you would like to stream video like below:

```
$ gst-sendstreamingvideo 192.168.5.20 /home/media/videos/sintel_trailer-720p.mp4
```

- Download the input file at: https://download.blender.org/durian/trailer/sintel_trailer-720p.mp4

3.2.10 Video Scale

Scale down an H.264 video, then store it in MP4 container.

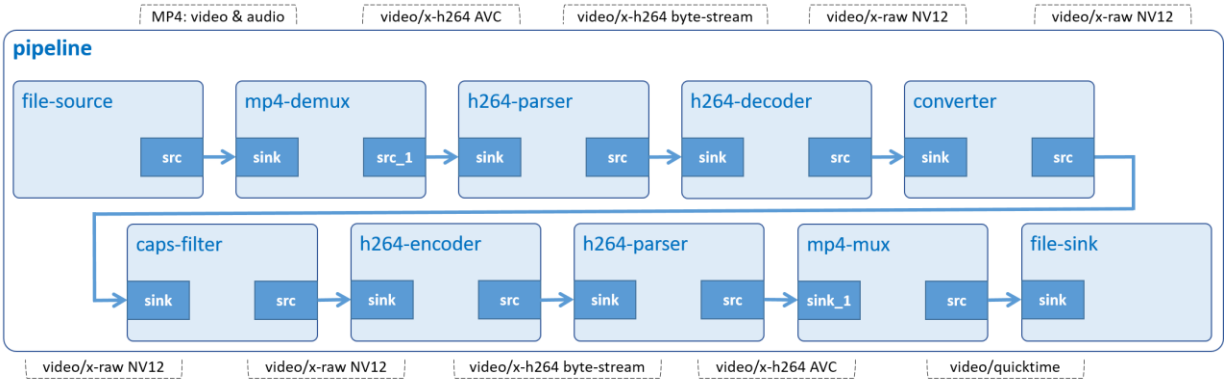


Figure 3.21 Video Scale pipeline

3.2.10.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <gst/gst.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <libgen.h>
#include <math.h>

#define BITRATE_OMXH264ENC 40000000 /* Target bitrate of the encoder element - omxh264enc */
#define OUTPUT_FILE        "SCALE_video.mp4"
#define ARG_PROGRAM_NAME   0
#define ARG_SCALE_RATIO    1
#define ARG_INPUT          2
#define ARG_COUNT          3

typedef struct tag_user_data
{
  GstElement *parser1;
  GstElement *capsfilter;
  float scale_ratio;
} UserData;

static void
on_pad_added (GstElement * element, GstPad * pad, gpointer data)
{
  GstPad *sinkpad;
  GstCaps *new_pad_caps = NULL;
  GstStructure *new_pad_struct = NULL;
  const gchar *new_pad_type = NULL;
  UserData *puser_data = (UserData *) data;
  GstCaps *scale_caps;
  int width;
  int height;
  int scaled_width;
  int scaled_height;
```

```
  /* Gets the capabilities that pad can produce or consume */
  new_pad_caps = gst_pad_query_caps (pad, NULL);

  /* Gets the structure in caps */
  new_pad_struct = gst_caps_get_structure (new_pad_caps, 0);

  /* Get the name of structure */
  new_pad_type = gst_structure_get_name (new_pad_struct);

  g_print ("Received new pad '%s' from '%s': %s\n", GST_PAD_NAME (pad),
      GST_ELEMENT_NAME (element), new_pad_type);

  if (g_str_has_prefix (new_pad_type, "video/x-h264")) {

    /* Get width of video */
    gst_structure_get_int (new_pad_struct, "width", &width);

    /* Get height of video */
    gst_structure_get_int (new_pad_struct, "height", &height);

    g_print ("Resolution of original video: %dx%d\n", width, height);

    scaled_width = (int) (width * puser_data->scale_ratio);

    scaled_height = (int) (height * puser_data->scale_ratio);

    g_print ("Now scaling video to resolution %dx%d...\n", scaled_width, scaled_height);

    /* create simple caps */
    scale_caps =
        gst_caps_new_simple ("video/x-raw", "width", G_TYPE_INT, scaled_width, "height",
        G_TYPE_INT, scaled_height, NULL);

    /* set caps property for capsfilters */
    g_object_set (G_OBJECT (puser_data->capsfilter), "caps", scale_caps, NULL);

    /* unref caps after usage */
    gst_caps_unref (scale_caps);
    /* We can now link this pad with the H.264 parser sink pad */
    g_print ("Dynamic pad created, linking demuxer/parser\n");

    sinkpad = gst_element_get_static_pad (puser_data->parser1, "sink");
    /* Link the input pad and the new request pad */
    gst_pad_link (pad, sinkpad);

    gst_object_unref (sinkpad);

  } else {
    g_printerr ("Unexpected pad received.\n");
  }

  if (new_pad_caps != NULL) {
    gst_caps_unref (new_pad_caps);
  }
}

/* Function is used to link request pad and a static pad */
static void
link_to_multiplexer (GstPad * tolink_pad, GstElement * mux)
{
  GstPad *pad;
  gchar *srcname, *sinkname;

  srcname = gst_pad_get_name (tolink_pad);
  pad = gst_element_get_compatible_pad (mux, tolink_pad, NULL);
  /* Link the input pad and the new request pad */
  gst_pad_link (tolink_pad, pad);
  sinkname = gst_pad_get_name (pad);
  gst_object_unref (GST_OBJECT (pad));

  g_print ("A new pad %s was created and linked to %s\n", sinkname, srcname);
  g_free (sinkname);
  g_free (srcname);
}
```

```
int
main (int argc, char *argv[])
{
  GstElement *pipeline, *source, *demuxer, *parser1, *decoder,
     *filter, *capsfilter, *encoder, *parser2, *muxer, *sink;
  GstBus *bus;
  GstMessage *msg;
  GstPad *srcpad;
  const char* ext;
  char* file_name;
  UserData user_data;
  float scale_ratio;

  const gchar *output_file = OUTPUT_FILE;

  if (argc != ARG_COUNT)
  {
    g_print ("Invalid arugments.\n");
    g_print ("Usage: %s <scale ratio (0, 1]> <MP4 file> \n", argv[ARG_PROGRAM_NAME]);
    return -1;
  }

  scale_ratio = roundoff (atof (argv[ARG_SCALE_RATIO]), 1);

  if ((scale_ratio > 1) || (scale_ratio <= 0 ))
  {
    g_print ("Invalid arugments.\n");
    g_print ("Usage: %s <scale ratio (0, 1]> <MP4 file> \n", argv[ARG_PROGRAM_NAME]);
    return -1;
  }

  const gchar *input_file = argv[ARG_INPUT];

  if (!is_file_exist(input_file))
  {
    g_printerr("Cannot find input file: %s. Exiting.\n", input_file);
    return -1;
  }

  file_name = basename (argv[ARG_INPUT]);
  ext = get_filename_ext (file_name);

  if (strcasecmp ("mp4", ext) != 0)
  {
    g_print ("Unsupported video type. MP4 format is required.\n");
    return -1;
  }

  /* Initialization */
  gst_init (&argc, &argv);

  /* Create GStreamer elements */
  pipeline = gst_pipeline_new ("video-scale");
  source = gst_element_factory_make ("filesrc", "video-src");
  demuxer = gst_element_factory_make ("qtdemux", "mp4-demuxer");
  parser1 = gst_element_factory_make ("h264parse", "h264-parser-1");
  decoder = gst_element_factory_make ("omxh264dec", "video-decoder");
  filter = gst_element_factory_make ("vspfilter", "video-filter");
  capsfilter = gst_element_factory_make ("capsfilter", "capsfilter");
  encoder = gst_element_factory_make ("omxh264enc", "video-encoder");
  parser2 = gst_element_factory_make ("h264parse", "h264-parser-2");
  muxer = gst_element_factory_make ("qtmux", "mp4-muxer");
  sink = gst_element_factory_make ("filesink", "file-output");

  if (!pipeline || !source || !demuxer || !parser1 || !decoder || !filter
      || !capsfilter || !encoder || !parser2 || !muxer || !sink) {
    g_printerr ("One element could not be created. Exiting.\n");
    return -1;
  }

  /* Set input video device file of the source element - v4l2src */
  g_object_set (G_OBJECT (source), "location", input_file, NULL);

  /* Set target-bitrate property of the encoder element - omxh264enc */
```

```
g_object_set (G_OBJECT (encoder), "target-bitrate", BITRATE_OMXH264ENC,
    "control-rate", 1, NULL);

/* Set output file location of the sink element - filesink */
g_object_set (G_OBJECT (sink), "location", output_file, NULL);

/* Add all elements into the pipeline */
gst_bin_add_many (GST_BIN (pipeline), source, demuxer, parser1, decoder,
    filter, capsfilter, encoder, parser2, muxer, sink, NULL);

/* Link the elements together */
if (gst_element_link (source, demuxer) != TRUE) {
  g_printerr ("Elements could not be linked.\n");
  gst_object_unref (pipeline);
  return -1;
}
if (gst_element_link_many (parser1, decoder, filter, capsfilter, encoder,
       parser2, NULL) != TRUE) {
  g_printerr ("Elements could not be linked.\n");
  gst_object_unref (pipeline);
  return -1;
}

if (gst_element_link (muxer, sink) != TRUE) {
  g_printerr ("Elements could not be linked.\n");
  gst_object_unref (pipeline);
  return -1;
}

user_data.parser1 = parser1;
user_data.capsfilter = capsfilter;
user_data.scale_ratio = scale_ratio;

/* Dynamic link */
g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added), &user_data);

/* link srcpad of h264parse to request pad of qtmuxer */
srcpad = gst_element_get_static_pad (parser2, "src");
link_to_multiplexer (srcpad, muxer);
gst_object_unref (srcpad);

/* Set the pipeline to "playing" state */
g_print ("Running...\n");
if (gst_element_set_state (pipeline,
       GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
  g_printerr ("Unable to set the pipeline to the playing state.\n");
  gst_object_unref (pipeline);
  return -1;
}

/* Wait until error or EOS */
bus = gst_element_get_bus (pipeline);
msg =
    gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
    GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
gst_object_unref (bus);

/* Note that because input timeout is GST_CLOCK_TIME_NONE,
   the gst_bus_timed_pop_filtered() function will block forever until a
   matching message was posted on the bus (GST_MESSAGE_ERROR or
   GST_MESSAGE_EOS). */

if (msg != NULL) {
  GError *err;
  gchar *debug_info;

  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_ERROR:
      gst_message_parse_error (msg, &err, &debug_info);
      g_printerr ("Error received from element %s: %s.\n",
          GST_OBJECT_NAME (msg->src), err->message);
      g_printerr ("Debugging information: %s.\n",
          debug_info ? debug_info : "none");
      g_clear_error (&err);
      g_free (debug_info);
```

```
      break;
    case GST_MESSAGE_EOS:
      g_print ("End-Of-Stream reached.\n");
      break;
    default:
      /* We should not reach here because we only asked for ERRORs and EOS */
      g_printerr ("Unexpected message received.\n");
      break;
  }
  gst_message_unref (msg);
}

/* Clean up nicely */
g_print ("Returned, stopping scaling...\n");
gst_element_set_state (pipeline, GST_STATE_NULL);

g_print ("Deleting pipeline...\n");
gst_object_unref (GST_OBJECT (pipeline));
g_print ("Succeeded. Please check output file: %s\n", output_file);
return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.6 Video Record and 3.2.1 Audio Play.

Command-line argument

```
if (argc != ARG_COUNT)
{
  g_print ("Invalid arugments.\n");
  g_print ("Usage: %s <scale ratio (0, 1]> <MP4 file> \n", argv[ARG_PROGRAM_NAME]);
  return -1;
}

scale_ratio = roundoff (atof (argv[ARG_SCALE_RATIO]), 1);

if ((scale_ratio > 1) || (scale_ratio <= 0 ))
{
  g_print ("Invalid arugments.\n");
  g_print ("Usage: %s <scale ratio (0, 1]> <MP4 file> \n", argv[ARG_PROGRAM_NAME]);
  return -1;
}
```

This application accepts two command-line arguments as below:

- Scale ratio: If it is not in range (0, 1], the application prints error and exits.
- An MP4 file's location.

Elements creation

```
source = gst_element_factory_make ("filesrc", "video-src");
demuxer = gst_element_factory_make ("qtdemux", "mp4-demuxer");
parser1 = gst_element_factory_make ("h264parse", "h264-parser-1");
decoder = gst_element_factory_make ("omxh264dec", "video-decoder");
filter = gst_element_factory_make ("vspfilter", "video-filter");
capsfilter = gst_element_factory_make ("capsfilter", "capsfilter");
encoder = gst_element_factory_make ("omxh264enc", "video-encoder");
parser2 = gst_element_factory_make ("h264parse", "h264-parser-2");
muxer = gst_element_factory_make ("qtmux", "mp4-muxer");
sink = gst_element_factory_make ("filesink", "file-output");
```

To scale down an H.264 video and store it in MP4 container, the following elements are needed:

- Element `filesrc` reads data from a local file.
- Element `qtdemux` de-multiplexes an MP4 file into audio and video stream.
- Element `omxh264dec` decompresses H.264 stream to raw NV12-formatted video.

- Element vspfilter handles video scaling.
- Element capsfilter contains resolution so that vspfilter will scale video frames based on this value.
- Element omxh264enc encodes raw video into H.264 compressed data.
- Element h264parse parses H.264 video from byte stream format to AVC format which omxh264dec can process.
- Element qtmux merges H.264 byte stream to MP4 container.
- Element filesink writes incoming data to a local file.

Set element's properties

```
g_object_set (G_OBJECT (source), "location", input_file, NULL);
g_object_set (G_OBJECT (encoder), "target-bitrate", BITRATE_OMXH264ENC, "control-rate", 1, NULL);
g_object_set (G_OBJECT (sink), "location", output_file, NULL);
```

The g_object_set() function is used to set some element's properties, such as:

- The location property of filesrc element which points to an MP4 input file.
- The target-bitrate property of omxh264enc element which is set to 40 Mbps. The higher bitrate, the better quality.
- The control-rate property of omxh264enc element is used to specify birate control method which is variable bitrate method in this case.
- The location property of filesink element which points to MP4 output file.

```
scale_caps =
    gst_caps_new_simple ("video/x-raw", "width", G_TYPE_INT, scaled_width, "height",
    G_TYPE_INT, scaled_height, NULL);

g_object_set (G_OBJECT (puser_data->capsfilter), "caps", scale_caps, NULL);
gst_caps_unref (scale_caps);
```

Capabilities (short: caps) describe the type of data which is streamed between two pads. This data includes raw video format, resolution, and framerate.

The gst_caps_new_simple() function creates a new cap (conv_caps) which holds output's resolution. This cap is then added to caps property of capsfilter (g_object_set) so that vspfilter will use these values to resize video frames.

Note that the scale_caps should be freed with gst_caps_unref() if it is not used anymore.

Get input file's information

```
new_pad_caps = gst_pad_query_caps (pad, NULL);
new_pad_struct = gst_caps_get_structure (new_pad_caps, 0);

gst_structure_get_int (new_pad_struct, "width", &width);
gst_structure_get_int (new_pad_struct, "height", &height);
```

This code block gets the capabilities of pad, finds the structure in new_pad_caps then gets resolution of video.

3.2.10.2 Result

In this case, a 640x360 MP4 file will be generated after running this application. Note that it does not have audio.

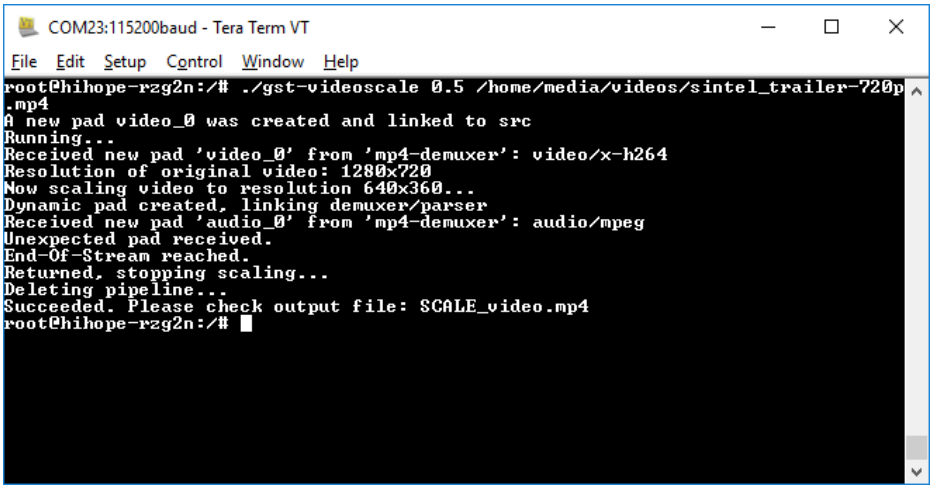


Figure 3.22 Video Scale result

Note: If the output's resolution is abnormal, this application will not work.

3.2.10.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.10.4 Special Instruction

- Download the input file at: https://download.blender.org/durian/trailer/sintel_trailer-720p.mp4.

3.2.11 Audio Player

A simple text-based audio player that can play Ogg/Vorbis files. Supported features: play, pause, stop, resume, seek, and display audio file(s). You can input a single file or a whole directory.

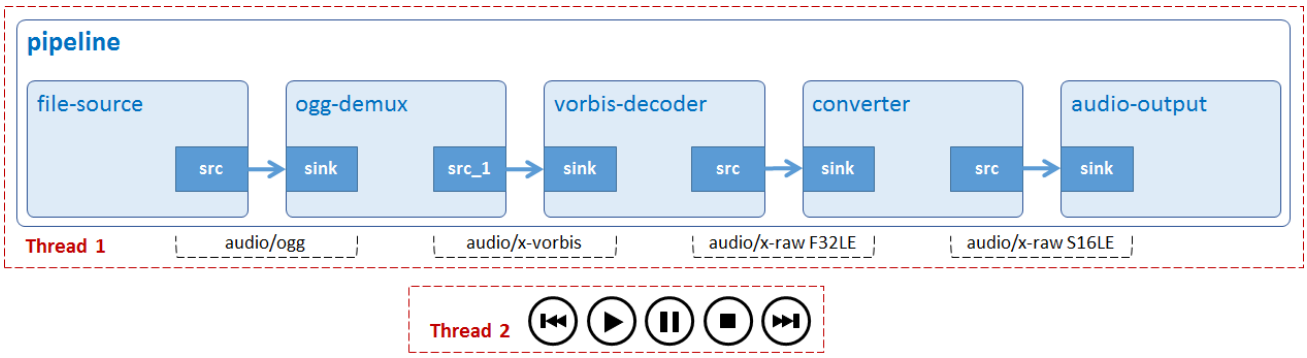


Figure 3.23 Audio Player pipeline

3.2.11.1 Source code

(1) Files

- main.c
- player.h/player.c

(2) main.c

```
#include <gst/gst.h>
#include <pthread.h>
#include "player.h"             /* player UI APIs */

#define SKIP_POSITION               (gint64) 5000000000       /* 5s */
#define NORMAL_PLAYING_RATE         (gdouble) 1.0
#define GET_SECOND_FROM_NANOSEC(x)  (x / 1000000000)
#define FORMAT                      "S16LE"
#define TIME                        60

typedef struct tag_user_data
{
  GMainLoop *loop;
  GstElement *pipeline;
  GstElement *source;
  GstElement *demuxer;
  GstElement *decoder;
  GstElement *converter;
  GstElement *capsfilter;
  GstElement *sink;
  gint64 audio_length;
} UserData;

/* Private helper functions */
static gboolean seek_to_time (GstElement * pipeline, gint64 time_nanoseconds);
static gint64 get_current_play_position (GstElement * pipeline);
GstState wait_for_state_change_completed (GstElement * pipeline);
void play_new_file (UserData * data, gboolean refresh_console_message);

/* Local variables */
gboolean ui_thread_die = FALSE;
pthread_t id_ui_thread = 0;
pthread_t id_autoplay_thread = 0;
pthread_cond_t cond_gst_data = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex_gst_data = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_t mutex_ui_data = PTHREAD_MUTEX_INITIALIZER;

/* Call back functions */
static void
on_pad_added (GstElement * element, GstPad * pad, gpointer data)
{
  pthread_mutex_lock (&mutex_gst_data);
  GstPad *sinkpad;
  UserData *puser_data = (UserData *) data;

  LOGD ("inside on_pad_added\n");
  /* Add back audio_queue, audio_decoder and audio_sink */
  gst_bin_add_many (GST_BIN (puser_data->pipeline),
     puser_data->decoder, puser_data->converter, puser_data->capsfilter,
     puser_data->sink, NULL);
  /* Link decoder and converter */
  if (gst_element_link_many (puser_data->decoder, puser_data->converter,
        puser_data->capsfilter, puser_data->sink, NULL) != TRUE) {
    g_print ("Elements could not be linked.\n");
  }

  /* Link demuxer and decoder */
  sinkpad = gst_element_get_static_pad (puser_data->decoder, "sink");
  if (GST_PAD_LINK_OK != gst_pad_link (pad, sinkpad)) {
    g_print ("Link Failed");
  }
  gst_object_unref (sinkpad);

  /* Change newly added element to ready state is required */
  gst_element_set_state (puser_data->decoder, GST_STATE_PLAYING);
  gst_element_set_state (puser_data->converter, GST_STATE_PLAYING);
  gst_element_set_state (puser_data->capsfilter, GST_STATE_PLAYING);
  gst_element_set_state (puser_data->sink, GST_STATE_PLAYING);

  /* Signal the dynamic pad linked */
  pthread_cond_signal (&cond_gst_data);
  pthread_mutex_unlock (&mutex_gst_data);
}

void
sync_to_play_new_file (UserData * data, gboolean refresh_console_message)
{
  pthread_mutex_lock (&mutex_gst_data);
  play_new_file (data, refresh_console_message);
  /* Wait on_pad_added completed in main thread */
  pthread_cond_wait (&cond_gst_data, &mutex_gst_data);
  /* Wait the state become PLAYING to get the audio length */
  gst_element_get_state (data->pipeline, NULL, NULL, GST_CLOCK_TIME_NONE);
  gst_element_query_duration (data->pipeline, GST_FORMAT_TIME,
      &(data->audio_length));
  pthread_mutex_unlock (&mutex_gst_data);
}

static void *
auto_play_thread_func (void *data)
{
  sync_to_play_new_file ((UserData *) data, TRUE);
  pthread_exit (NULL);
  return NULL;
}

static gboolean
bus_call (GstBus * bus, GstMessage * msg, gpointer data)
{
  GMainLoop *loop = ((UserData *) data)->loop;
  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_ERROR:{
      gchar *debug;
      GError *error;
      gst_message_parse_error (msg, &error, &debug);
      g_free (debug);
      g_printerr ("Error: %s\n", error->message);
      g_error_free (error);
      fclose (stdin);
```

```
      g_main_loop_quit (loop);
      break;
    }
    case GST_MESSAGE_EOS:{
      /* Auto play next file when EOS reach if possible */
      pthread_mutex_lock (&mutex_ui_data);
      gboolean ret = request_update_file_path (1);
      pthread_mutex_unlock (&mutex_ui_data);
      if (ret) {
        /* Need to call play_new_file() from another thread to avoid DEADLOCK */
        if (pthread_create (&id_autoplay_thread, NULL, auto_play_thread_func,
                (UserData *) data)) {
          LOGE ("pthread_create autoplay failed\n");
        }
      }
      break;
    }
    default:
      break;
  }
  return TRUE;
}

static void *
check_user_command_loop (void *data)
{
  GstElement *pipeline;
  gint64 audio_length;
  GstState current_state;

  while (!ui_thread_die) {
    pthread_mutex_lock (&mutex_gst_data);
    pipeline = ((UserData *) data)->pipeline;
    current_state = wait_for_state_change_completed (pipeline);
    audio_length = ((UserData *) data)->audio_length;
    pthread_mutex_unlock (&mutex_gst_data);

    UserCommand cmd = get_user_command ();
    switch (cmd) {
      case QUIT:
        g_main_loop_quit (((UserData *) data)->loop);
        pthread_exit (NULL);
        return NULL;
        break;

      case PAUSE_PLAY:
        if (GST_STATE_PLAYING == current_state) {
          gst_element_set_state (pipeline, GST_STATE_PAUSED);
          g_print ("PAUSED!\n");
        } else if (GST_STATE_PAUSED == current_state) {
          gst_element_set_state (pipeline, GST_STATE_PLAYING);
          g_print ("PLAYED!\n");
        } else {
          g_print ("Cannot PAUSE/PLAY\n");
        }
        break;

      case STOP:{
        /* Seek to the begining */
        if (GST_STATE_PLAYING == current_state) {
          gst_element_set_state (pipeline, GST_STATE_PAUSED);
        }
        if (seek_to_time (pipeline, 0)) {
          g_print ("STOP\n");
        }
        break;
      }

      case REPLAY:
        if (seek_to_time (pipeline, 0)) {
          /* Seek to the begining */
          if (GST_STATE_PAUSED == current_state) {
            gst_element_set_state (pipeline, GST_STATE_PLAYING);
          }
```

```
        g_print ("REPLAYED!\n");
      }
      break;

    case FORWARD:{
      gint64 pos = get_current_play_position (pipeline);
      /* calculate the forward position */
      pos = pos + SKIP_POSITION;
      if (pos >= audio_length) {
        pos = audio_length;
      }
      if (seek_to_time (pipeline, pos)) {
        g_print ("current: %02d:%02d\n",
            (int) (GET_SECOND_FROM_NANOSEC (pos) / TIME),
            (int) (GET_SECOND_FROM_NANOSEC (pos) % TIME));
      }
      break;
    }

    case REWIND:{
      gint64 pos = get_current_play_position (pipeline);
      /* calculate the rewind position */
      pos = pos - SKIP_POSITION;
      if (pos < 0) {
        pos = 0;
      }
      if (seek_to_time (pipeline, pos)) {
        g_print ("current: %02d:%02d\n",
            (int) (GET_SECOND_FROM_NANOSEC (pos) / TIME),
            (int) (GET_SECOND_FROM_NANOSEC (pos) % TIME));
      }
      break;
    }

    case LIST:{
      pthread_mutex_lock (&mutex_ui_data);
      update_file_list ();
      pthread_mutex_unlock (&mutex_ui_data);
      break;
    }

    case PREVIOUS:{
      pthread_mutex_lock (&mutex_ui_data);
      gboolean ret = request_update_file_path (-1);
      pthread_mutex_unlock (&mutex_ui_data);
      if (ret) {
        sync_to_play_new_file ((UserData *) data, FALSE);
      }
      break;
    }

    case NEXT:{
      pthread_mutex_lock (&mutex_ui_data);
      gboolean ret = request_update_file_path (1);
      pthread_mutex_unlock (&mutex_ui_data);
      if (ret) {
        sync_to_play_new_file ((UserData *) data, FALSE);
      }
      break;
    }

    case PLAYFILE:{
      pthread_mutex_lock (&mutex_ui_data);
      gboolean ret = request_update_file_path (0);
      pthread_mutex_unlock (&mutex_ui_data);
      if (ret) {
        sync_to_play_new_file ((UserData *) data, FALSE);
      }
      break;
    }

    case HELP:
      print_supported_command ();
      break;
```

```
      default:
        break;
    }
  }
  return NULL;
}

/* Private helper functions */
static gboolean
seek_to_time (GstElement * pipeline, gint64 time_nanoseconds)
{
  if (!gst_element_seek (pipeline, NORMAL_PLAYING_RATE, GST_FORMAT_TIME,
          GST_SEEK_FLAG_FLUSH, GST_SEEK_TYPE_SET, time_nanoseconds,
          GST_SEEK_TYPE_NONE, GST_CLOCK_TIME_NONE)) {
    g_print ("Seek failed!\n");
    return FALSE;
  }
  return TRUE;
}

static gint64
get_current_play_position (GstElement * pipeline)
{
  gint64 pos;

  if (gst_element_query_position (pipeline, GST_FORMAT_TIME, &pos)) {
    return pos;
  } else
    return -1;
}

GstState
wait_for_state_change_completed (GstElement * pipeline)
{
  GstState current_state = GST_STATE_VOID_PENDING;
  GstStateChangeReturn state_change_return = GST_STATE_CHANGE_ASYNC;
  do {
    state_change_return =
        gst_element_get_state (pipeline, &current_state, NULL,
        GST_CLOCK_TIME_NONE);
  } while (state_change_return == GST_STATE_CHANGE_ASYNC);
  return current_state;
}

void
play_new_file (UserData * data, gboolean refresh_console_message)
{
  GstElement *decoder = data->decoder;
  GstElement *converter = data->converter;
  GstElement *capsfilter = data->capsfilter;
  GstElement *sink = data->sink;
  gboolean ret = FALSE;

  /* Seek to start and flush all old data */
  gst_element_seek_simple (data->pipeline, GST_FORMAT_TIME, GST_SEEK_FLAG_FLUSH, 0);
  gst_element_set_state (data->pipeline, GST_STATE_READY);

  /* wait until the changing is complete */
  gst_element_get_state (data->pipeline, NULL, NULL, GST_CLOCK_TIME_NONE);

  /* Keep the element to still exist after removing */
  decoder = gst_bin_get_by_name (GST_BIN (data->pipeline), "vorbis-decoder");
  if (NULL != decoder) {
    ret = gst_bin_remove (GST_BIN (data->pipeline), data->decoder);
    LOGD ("gst_bin_remove decoder from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
  }
  /* Keep the element to still exist after removing */
  converter = gst_bin_get_by_name (GST_BIN (data->pipeline), "converter");
  if (NULL != converter) {
    ret = gst_bin_remove (GST_BIN (data->pipeline), data->converter);
    LOGD ("gst_bin_remove converter from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
```

```
  }
  /* Keep the element to still exist after removing */
  capsfilter = gst_bin_get_by_name (GST_BIN (data->pipeline), "conv_capsfilter");
  if (NULL != capsfilter) {
    ret = gst_bin_remove (GST_BIN (data->pipeline), data->capsfilter);
    LOGD ("gst_bin_remove capsfilter from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
  }
  /* Keep the element to still exist after removing */
  sink = gst_bin_get_by_name (GST_BIN (data->pipeline), "audio-output");
  if (NULL != sink) {
    ret = gst_bin_remove (GST_BIN (data->pipeline), data->sink);
    LOGD ("gst_bin_remove sink from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
  }

  /* Update file location */
  g_object_set (G_OBJECT (data->source), "location", get_current_file_path (), NULL);

  /* Set the pipeline to "playing" state */
  print_current_selected_file (refresh_console_message);
  gst_element_set_state (data->pipeline, GST_STATE_PLAYING);
}

int
main (int argc, char *argv[])
{
  /* Check input arguments */
  if (argc != 2) {
    g_printerr ("Usage: %s <Ogg/Vorbis filename or directory>\n", argv[0]);
    return -1;
  }

  /* Get dir_path and file_path if possible */
  if (!inject_dir_path_to_player (argv[1])) {
    return -1;
  }

  GMainLoop *loop;
  UserData user_data;

  GstElement *pipeline, *source, *demuxer, *decoder, *conv, *capsfilter, *sink;
  GstCaps *caps;
  GstBus *bus;
  guint bus_watch_id;

  /* Initialization */
  gst_init (NULL, NULL);
  loop = g_main_loop_new (NULL, FALSE);

  /* Create GStreamer elements instead of demuxer */
  pipeline = gst_pipeline_new ("audio-player");
  source = gst_element_factory_make ("filesrc", "file-source");
  demuxer = gst_element_factory_make ("oggdemux", "ogg-demuxer");
  decoder = gst_element_factory_make ("vorbisdec", "vorbis-decoder");
  conv = gst_element_factory_make ("audioconvert", "converter");
  capsfilter = gst_element_factory_make ("capsfilter", "conv_capsfilter");
  sink = gst_element_factory_make ("alsasink", "audio-output");

  if (!pipeline || !source || !demuxer || !decoder || !conv || !capsfilter
      || !sink) {
    g_printerr ("One element could not be created. Exiting.\n");
    return -1;
  }

  /* Set the caps option to the caps-filter element */
  caps =
      gst_caps_new_simple ("audio/x-raw", "format", G_TYPE_STRING, FORMAT,
      NULL);
  g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
  gst_caps_unref (caps);
```

```
  /* we add all elements into the pipeline and link them instead of demuxer */
  gst_bin_add_many (GST_BIN (pipeline), source, demuxer, decoder, conv,
      capsfilter, sink, NULL);
  if (gst_element_link (source, demuxer) != TRUE) {
    g_printerr ("Elements could not be linked.\n");
    gst_object_unref (pipeline);
    return -1;
  }
  if (gst_element_link_many (decoder, conv, capsfilter, sink, NULL) != TRUE) {
    g_printerr ("Elements could not be linked.\n");
    gst_object_unref (pipeline);
    return -1;
  }

  /* Construct user data */
  user_data.loop = loop;
  user_data.pipeline = pipeline;
  user_data.source = source;
  user_data.demuxer = demuxer;
  user_data.decoder = decoder;
  user_data.converter = conv;
  user_data.capsfilter = capsfilter;
  user_data.sink = sink;
  user_data.audio_length = 0;

  g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added),
      &user_data);

  /* Set up the pipeline */
  /* we add a message handler */
  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  bus_watch_id = gst_bus_add_watch (bus, bus_call, &user_data);
  gst_object_unref (bus);

  /* Show the media file list */
  update_file_list ();

  if (try_to_update_file_path ()) {
    sync_to_play_new_file (&user_data, FALSE);
  }

  /* Handle user input thread */
  if (pthread_create (&id_ui_thread, NULL, check_user_command_loop, &user_data)) {
    LOGE ("pthread_create failed\n");
    goto exit;
  }

  /* Iterate */
  g_main_loop_run (loop);

  /* Out of the main loop, clean up nicely */
  g_print ("Returned, stopping playback...\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);

  g_print ("Wait for UI thread terminated.\n");
  ui_thread_die = TRUE;
  pthread_join (id_ui_thread, NULL);
  if (id_autoplay_thread != 0) {
    g_print ("Wait Autoplay thread terminated.\n");
    pthread_join (id_autoplay_thread, NULL);
  }

exit:
  g_print ("Deleting pipeline...\n");
  gst_object_unref (GST_OBJECT (pipeline));
  g_source_remove (bus_watch_id);
  g_main_loop_unref (loop);

  g_print ("Program end!\n");
  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.14 File Play and 3.2.1 Audio Play.

Include player.h

```
#include "player.h"             /* player UI APIs */
```

Header file `player.h` contains functions that allow us to retrieve Ogg/Vorbis file path(s) from program parameter `argv[1]`.

```
#define SKIP_POSITION               (gint64) 5000000000       /* 5s */
#define NORMAL_PLAYING_RATE         (gdouble) 1.0
```

The `SKIP_POSITION` macro defines the time interval (in nanosecond) to seek audio backwards and forwards.

```
#define NORMAL_PLAYING_RATE         (gdouble) 1.0
```

GStreamer pipeline also supports changing audio playback speed. By default, the speed is set normal (`1.0`).

UserData structure

```
typedef struct tag_user_data
{
  GMainLoop *loop;
  GstElement *pipeline;
  GstElement *source;
  GstElement *demuxer;
  GstElement *decoder;
  GstElement *converter;
  GstElement *capsfilter;
  GstElement *sink;
  gint64 audio_length;
} UserData;
```

This structure contains:

- Variable `loop` (`GMainLoop`): An opaque data type to represent the main event loop of a Glib application.
- Variable `pipeline` (`GstElement`): A pointer to GStreamer pipeline which contains connected audio elements.
- Variable `source` (`GstElement`): A GStreamer element to read data from a local file.
- Variable `demuxer` (`GstElement`): A GStreamer element to de-multiplex Ogg/Vorbis files into their encoded audio and video components. In this case, only audio stream is available.
- Variable `decoder` (`GstElement`): A GStreamer element to decompress a Vorbis stream to raw audio.
- Variable `converter` (`GstElement`): A GStreamer element to convert raw audio buffers between various possible formats depending on the given source pad and sink pad it links to.
- Variable `capsfilter` (`GstElement`): A GStreamer element to specify raw audio format `S16LE`.
- Variable `sink` (`GstElement`): A GStreamer element to automatically detect an appropriate audio sink, in this case `alsasink`.
- Variable `audio_length` (`qint64`): An 8-byte integer variable to represent audio duration.

Thread IDs

```
pthread_t id_ui_thread = 0;
pthread_t id_autoplay_thread = 0;
```

Variable `id_ui_thread` contains ID of text-based UI thread which handles inputs from user.

Variable `id_autoplay_thread` contains ID of auto-play thread which automatically plays the next audio file if the current one has just finished.

Validate user input

```
if (argc != 2) {
  g_printerr ("Usage: %s <Ogg/Vorbis filename or directory>\n", argv[0]);
  return -1;
}
```

This application accepts one command-line argument which points to an Ogg/Vorbis file or a whole directory.

Process user input

```
if (!inject_dir_path_to_player (argv[1])) {
  return -1;
}
```

This function retrieves an absolute path of the Ogg/Vorbis file or directory.

Audio pipeline

```
source = gst_element_factory_make ("filesrc", "file-source");
demuxer = gst_element_factory_make ("oggdemux", "ogg-demuxer");
decoder = gst_element_factory_make ("vorbisdec", "vorbis-decoder");
conv = gst_element_factory_make ("audioconvert", "converter");
capsfilter = gst_element_factory_make ("capsfilter", "conv_capsfilter");
sink = gst_element_factory_make ("alsasink", "audio-output");

gst_bin_add_many (GST_BIN (pipeline), source, demuxer, decoder, conv, capsfilter, sink, NULL);
gst_element_link (source, demuxer);
gst_element_link_many (decoder, conv, capsfilter, sink, NULL);

user_data.loop = loop;
user_data.pipeline = pipeline;
user_data.source = source;
user_data.demuxer = demuxer;
user_data.decoder = decoder;
user_data.converter = conv;
user_data.capsfilter = capsfilter;
user_data.sink = sink;
user_data.audio_length = 0;

g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added), &user_data);

bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
bus_watch_id = gst_bus_add_watch (bus, bus_call, &user_data);
gst_object_unref (bus);
```

Basically, this pipeline is just like Audio Play except it uses `gst_bus_add_watch()` instead of `gst_bus_timed_pop_filtered()` to receive messages (such as: error or `EOS` (`End-of-Stream`)) from `bus_call()` asynchronously.

Audio duration

```
user_data.audio_length = 0;
```

At this point, the `pipeline` is not running (`NULL`), so it is not safe to query audio duration. Let's just assign `audio_length` to `0` for now. We will find its value in `sync_to_play_new_file()` when the pipeline is in `PLAYING` state.

Process user input (cont.)

```
update_file_list ();
```

If the input is a path to a directory, this code block will get the number of files whose extension is `.ogg` inside the directory.

```
if (try_to_update_file_path ()) {
```

```
  sync_to_play_new_file (&user_data, FALSE);
}
```

The `try_to_update_file_path()` function will find the first audio file to play. If successful, it will call `sync_to_play_new_file()` (with argument `user_data`) to set the `pipeline` to `PLAYING` state, then retrieves audio duration.

Start text-based UI thread

```
if (pthread_create (&id_ui_thread, NULL, check_user_command_loop, &user_data)) {
  LOGE ("pthread_create failed\n");
  goto exit;
}
```

The `pthread_create()` function starts a new thread. Note that the new thread will handle UI inputs from users (such as: play, pause, stop, resume, seek, and display audio files). It starts execution by invoking `check_user_command_loop()`. Note that `user_data` is passed as the sole argument of this function.

Start auto-play thread

```
if (pthread_create (&id_autoplay_thread, NULL, auto_play_thread_func, (UserData *) data)) {
  LOGE ("pthread_create autoplay failed\n");
}
```

The `id_autoplay_thread` thread will be executed when the current song receives `EOS` (`End-of-Stream`) signal.

Clean up

```
pthread_join (id_ui_thread, NULL);
pthread_join (id_autoplay_thread, NULL);
```

The `pthread_join()` function waits for both `id_ui_thread` and `id_autoplay_thread` to terminate. If that thread has already terminated, it will return immediately.

Play audio pipeline

```
void
play_new_file (UserData * data, gboolean refresh_console_message)
{
  /* Seek to start and flush all old data */
  gst_element_seek_simple (data->pipeline, GST_FORMAT_TIME, GST_SEEK_FLAG_FLUSH, 0);
  gst_element_set_state (data->pipeline, GST_STATE_READY);

  /* Wait until the changing is complete */
  gst_element_get_state (data->pipeline, NULL, NULL, GST_CLOCK_TIME_NONE);

  decoder = gst_bin_get_by_name (GST_BIN (data->pipeline), "vorbis-decoder");
  gst_bin_remove (GST_BIN (data->pipeline), data->decoder);

  converter = gst_bin_get_by_name (GST_BIN (data->pipeline), "converter");
  gst_bin_remove (GST_BIN (data->pipeline), data->converter);

  capsfilter = gst_bin_get_by_name (GST_BIN (data->pipeline), "conv_capsfilter");
  gst_bin_remove (GST_BIN (data->pipeline), data->capsfilter);

  sink = gst_bin_get_by_name (GST_BIN (data->pipeline), "audio-output");
  gst_bin_remove (GST_BIN (data->pipeline), data->sink);

  /* Update file location */
  g_object_set (G_OBJECT (data->source), "location", get_current_file_path (), NULL);

  /* Set the pipeline to "playing" state */
  print_current_selected_file (refresh_console_message);
```

```
  gst_element_set_state (data->pipeline, GST_STATE_PLAYING);
}
```

This function seeks `pipeline` to the beginning of playback position and changes its state to `READY` to prepare for the next audio file (`get_current_file_path`).

Next, it removes audio elements from `pipeline`, such as: `vorbisdec` (`decoder`), `audioconvert` (`converter`), `capsfilter`, and `autoaudiosink` (`sink`) from `pipeline` everytime `location` property of `filesrc` (`source`) changes (`g_object_set`). After this step, the pipeline only contains upstream elements, such as: `filesrc` and `oggdemux`.

Finally, the pipeline is set to `PLAYING` state. This will later call `on_pad_added()` asynchronously.

Note:

We need to call `gst_bin_get_by_name()` to keep the elements existing after we call `get_bin_remove()`.

Function on_pad_added()

```
static void on_pad_added (GstElement * element, GstPad * pad, gpointer data)
{
  gst_bin_add_many (GST_BIN (puser_data->pipeline),
      puser_data->decoder, puser_data->converter, puser_data->capsfilter, puser_data->sink, NULL);

  gst_element_link_many (puser_data->decoder, puser_data->converter,
        puser_data->capsfilter, puser_data->sink, NULL)

  /* Link demuxer and decoder */
  sinkpad = gst_element_get_static_pad (puser_data->decoder, "sink");
  if (GST_PAD_LINK_OK != gst_pad_link (pad, sinkpad)) {
    g_print ("Link Failed");
  }
  gst_object_unref (sinkpad);

  /* Change newly added element to ready state is required */
  gst_element_set_state (puser_data->decoder, GST_STATE_PLAYING);
  gst_element_set_state (puser_data->converter, GST_STATE_PLAYING);
  gst_element_set_state (puser_data->capsfilter, GST_STATE_PLAYING);
  gst_element_set_state (puser_data->sink, GST_STATE_PLAYING);
}
```

This function adds and links (again) these audio elements to `pipeline`. Note that their states should be in `PLAYING` state to synchronize with upstream elements.

Function bus_call()

```
static gboolean bus_call (GstBus * bus, GstMessage * msg, gpointer data)
{
  GMainLoop *loop = ((UserData *) data)->loop;
  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_ERROR:{
      g_main_loop_quit (loop);
      break;
    }
    case GST_MESSAGE_EOS:{
      gboolean ret = request_update_file_path (1);
      if (ret) {
        if (pthread_create (&id_autoplay_thread, NULL, auto_play_thread_func, (UserData *) data)) {
          LOGE ("pthread_create autoplay failed\n");
        }
      }
      break;
    }
    default:
      break;
  }
  return TRUE;
}
```

This function will be called when a message is received from bus.

If the message is GST_MESSAGE_ERROR, the application will call g_main_loop_quit() to stop loop. This makes g_main_loop_run() return. Finally, the application cleans up GSteamer objects and exits.

If the message is GST_MESSAGE_EOS, the application will execute id_autoplay_thread thread (pthread_create) to play the next audio file (request_update_file_path) automatically.

Thread handler auto_play_thread_func()

```
static void * auto_play_thread_func (void *data)
{
  sync_to_play_new_file ((UserData *) data, TRUE);
  pthread_exit (NULL);
  return NULL;
}
```

Basically, it will call sync_to_play_new_file() and exit.

Note:

The bus_call() function cannot call sync_to_play_new_file() directly because it is blocking the pipeline. This makes it unable to play new audio file.
This is the reason why we have to call sync_to_play_new_file() from another thread to avoid deadlock.

Playing pipeline

```
void sync_to_play_new_file (UserData * data, gboolean refresh_console_message)
{
  play_new_file (data, refresh_console_message);

  gst_element_get_state (data->pipeline, NULL, NULL, GST_CLOCK_TIME_NONE);
  gst_element_query_duration (data->pipeline, GST_FORMAT_TIME, &(data->audio_length));
}
```

This function reconfigures and runs existing pipeline (play_new_file) to play next audio file, then gets audio duration when the pipeline is in PLAYING state.

Text-based UI thread handler

```
static void * check_user_command_loop (void *data)
{
  GstElement *pipeline;
  gint64 audio_length;
  GstState current_state;

  while (!ui_thread_die) {
    pipeline = ((UserData *) data)->pipeline;
    current_state = wait_for_state_change_completed (pipeline);
    audio_length = ((UserData *) data)->audio_length;

    UserCommand cmd = get_user_command ();
    switch (cmd) {}
  }
}
```

This handler waits for (get_user_command) and executes input commands from user.

```
case QUIT:
  g_main_loop_quit (((UserData *) data)->loop);
  pthread_exit (NULL);
  return NULL;
  break;
```

Command QUIT calls g_main_loop_quit() to stop the loop and UI thread from running. This makes g_main_loop_run() return. Finally, the application cleans up GSteamer objects, and exits.

```
case PAUSE_PLAY:
  if (GST_STATE_PLAYING == current_state) {
    gst_element_set_state (pipeline, GST_STATE_PAUSED);
  } else if (GST_STATE_PAUSED == current_state) {
    gst_element_set_state (pipeline, GST_STATE_PLAYING);
  } else {
    g_print ("Cannot PAUSE/PLAY\n");
  }
  break;
```

Command PAUSE_PLAY calls gst_element_set_state() to toggle the pipeline between PLAYING and PAUSED state.

```
case STOP:{
  if (GST_STATE_PLAYING == current_state) {
    gst_element_set_state (pipeline, GST_STATE_PAUSED);
  }
  if (seek_to_time (pipeline, 0)) {
    g_print ("STOP\n");
  }
  break;
}
```

Command STOP sets the pipeline to PAUSED, then seeks it to the beginning of playback position.

```
case REPLAY:
  if (seek_to_time (pipeline, 0)) {
    if (GST_STATE_PAUSED == current_state) {
      gst_element_set_state (pipeline, GST_STATE_PLAYING);
    }
    g_print ("REPLAYED!\n");
  }
  break;
```

Command REPLAY seeks the pipeline to beginning of playback position. Also, it will resume the audio if it is pausing.

```
case FORWARD:{
  gint64 pos = get_current_play_position (pipeline);

  pos = pos + SKIP_POSITION;
  if (pos >= audio_length) {
    pos = audio_length;
  }
  if (seek_to_time (pipeline, pos)) {
    g_print ("current: %02d:%02d\n",
        (int) (GET_SECOND_FROM_NANOSEC (pos) / TIME),
        (int) (GET_SECOND_FROM_NANOSEC (pos) % TIME));
  }
  break;
}
```

Command FORWARD adds 5 seconds (defined in SKIP_POSITION) to the current position and seeks forwards.

```
case REWIND:{
  gint64 pos = get_current_play_position (pipeline);
  pos = pos - SKIP_POSITION;
  if (pos < 0) {
    pos = 0;
  }
  if (seek_to_time (pipeline, pos)) {
    g_print ("current: %02d:%02d\n",
        (int) (GET_SECOND_FROM_NANOSEC (pos) / TIME),
        (int) (GET_SECOND_FROM_NANOSEC (pos) % TIME));
  }
  break;
}
```

Command REWIND removes 5 seconds (defined in SKIP_POSITION) from the current position and seeks

backwards.

```
case LIST:{
  update_file_list ();
  break;
}
```

Command `LIST` calls `update_file_list()` to update the number of Ogg/Vorbis files.

```
case PREVIOUS:{
  gboolean ret = request_update_file_path (-1);
  if (ret) {
    sync_to_play_new_file ((UserData *) data, FALSE);
  }
  break;
}
```

Command `PREVIOUS` retrieves the location of previous audio file, then plays it.

```
case NEXT:{
  gboolean ret = request_update_file_path (1);
  if (ret) {
    sync_to_play_new_file ((UserData *) data, FALSE);
  }
  break;
}
```

Command `NEXT` retrieves the location of next audio file, then plays it.

```
case PLAYFILE:{
  gboolean ret = request_update_file_path (0);
  if (ret) {
    sync_to_play_new_file ((UserData *) data, FALSE);
  }
  break;
}
```

Command `PLAYFILE` repeats the current file.

```
case HELP:
  print_supported_command ();
  break;
```

Command `HELP` displays a short option summary.

(3) player.h/player.c

```
#include <gst/gst.h>
#include <stdbool.h>          /* bool type */
#include <stdio.h>            /* getline() API */
#include <stdlib.h>           /* free(), atoi() API */
#include <sys/stat.h>         /* stat(), struct stat */
#include <dirent.h>           /* struct dirent */
#include <libgen.h>           /* basename */
#include <limits.h>           /* PATH_MAX */

//#define DEBUG_LOG
#ifdef DEBUG_LOG
#define LOGD(...)   g_print(__VA_ARGS__)
#else
#define LOGD(...)
#endif

#define LOGI(...)   g_print(__VA_ARGS__)
#define LOGE(...)   g_printerr(__VA_ARGS__)

#define FILE_SUFFIX ".ogg"
```

```
typedef enum tag_user_command
{
  INVALID = 0,
  REPLAY,
  PAUSE_PLAY,
  STOP,
  FORWARD,
  REWIND,
  HELP,
  QUIT,
  LIST,
  PREVIOUS,
  NEXT,
  PLAYFILE,
} UserCommand;

gchar *get_current_file_path (void);

gboolean inject_dir_path_to_player (const gchar * path);
void update_file_list (void);
gboolean try_to_update_file_path (void);
gboolean request_update_file_path (gint32 offset_file);

UserCommand get_user_command (void);
void print_supported_command (void);
void print_current_selected_file (gboolean refresh_console_message);
```

Walkthrough:

Macros

```
#define FILE_SUFFIX ".ogg"
```

The `FILE_SUFFIX` macro defines the file extension that is supported by the `pipeline`. In this application, it only accepts audio files whose extension are `.ogg`.

```
#define DEBUG_LOG
```

If `DEBUG_LOG` is defined, the application will print out debugging messages.

Static variables

```
static gchar dir_path[PATH_MAX];
```

It contains an absolute path to the directory inputted by user. The value is retrieved by calling `inject_dir_path_to_player()`.

```
static gchar file_path[PATH_MAX];
```

It contains an absolute path to the file inputted by user or to the current file in the list. The value is retrieved by calling `inject_dir_path_to_player()`.

```
static guint32 current_file_no = 0;
```

It is an index which points to the current audio file (in the playlist). The value will be updated by calling `request_update_file_path()`.

Note that this index starts from `1` and will always be `1` if user inputs an Ogg/Vorbis audio file, not a whole directory.

```
static guint32 last_file_count = 0;
```

It contains the number of Ogg/Vorbis files. The value is retrieved by `update_file_list()`.

Note that this variable will always be 1 if user inputs an Ogg/Vorbis audio file, not a whole directory.

APIs

```
gchar *get_current_file_path (void);
```

This function returns variable `file_path` (see above).

```
gboolean inject_dir_path_to_player (const gchar * path);
```

If the input is a file, this function will get and store its absolute path in `file_path` and `dir_path` variables.
If the input is a directory, this function will get and store its absolute path in `dir_path` variable.

```
void update_file_list (void);
```

This function gets the number of Ogg/Vorbis files and stores them in `last_file_count` variable.

```
gboolean try_to_update_file_path (void);
```

This function gets file path without scanning variable `dir_path`. It can help the program play audio immediately if user inputs an Ogg/Vorbis file.

```
gboolean request_update_file_path (gint32 offset_file);
```

If the `offset_file` is `0`, this function will get an absolute path of the current audio file.
If the `offset_file` is `-1`, this function will get an absolute path of the previous audio file.
If the `offset_file` is `1`, this function will get an absolute path of the next audio file.

```
UserCommand get_user_command (void);
```

This function waits for and then executes input commands from user, or plays audio based on the order of audio files in the playlist.

```
void print_supported_command (void);
```

This function displays a short option summary.

```
void print_current_selected_file (gboolean refresh_console_message);
```

This function prints a name and index of the current audio file.

3.2.11.2 Result

Can play Ogg/Vorbis files. User can use some playback features (such as: start, stop, pause, resume...).

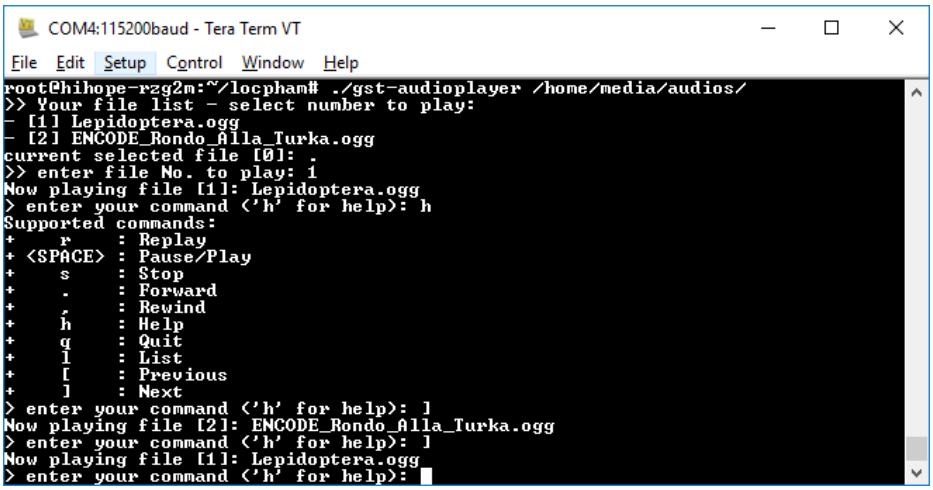


Figure 3.24 Audio Player result

3.2.11.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.11.4 Special Instruction

- Download the input files at: http://file-examples.com/index.php/sample-audio-files/sample-ogg-download/

3.2.12 Video Player

A simple text-based video player that can play MP4 files. Supported features: play, pause, stop, resume, and seek video files. You can input a single file or a whole directory.

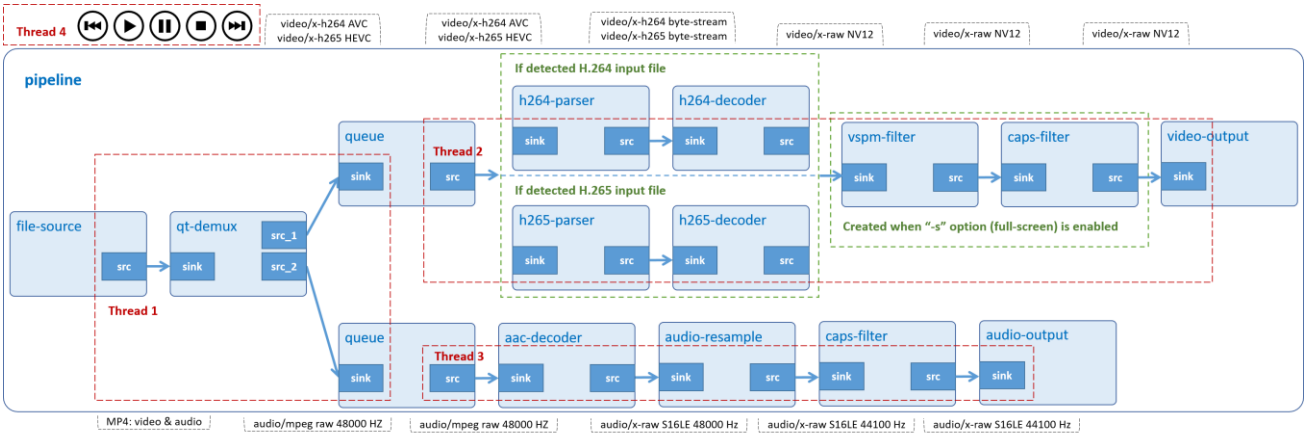


Figure 3.25 Video Player pipeline

3.2.12.1 Source code

(1) Files

- main.c
- player.h/player.c

(2) main.c

```
#include <stdio.h>
#include <string.h>
#include <gst/gst.h>
#include <pthread.h>
#include "player.h"             /* player UI APIs */
#include <stdbool.h>
#include <wayland-client.h>

#define SKIP_POSITION (gint64)          5000000000       /* 5s */
#define NORMAL_PLAYING_RATE (gdouble) 1.0
#define GET_SECOND_FROM_NANOSEC(x) (x / 1000000000)
#define ONE_MINUTE                      60               /* 1 minute = 60 seconds */
#define AUDIO_SAMPLE_RATE               44100

typedef struct tag_user_data
{
  GMainLoop *loop;
  GstElement *pipeline;
  GstElement *source;
  GstElement *demuxer;
  GstElement *audio_queue;
  GstElement *audio_decoder;
  GstElement *audio_resample;
  GstElement *audio_capsfilter;
  GstElement *audio_sink;
  GstElement *video_queue;
  GstElement *video_parser;
  GstElement *video_decoder;
  GstElement *video_filter;
  GstElement *video_capsfilter;
  GstElement *video_sink;
  gint64 media_length;
  struct screen_t *main_screen;
```

```
  bool fullscreen;
} UserData;

/* Private helper functions */
static gboolean seek_to_time (GstElement * pipeline, gint64 time_nanoseconds);
static gint64 get_current_play_position (GstElement * pipeline);
GstState wait_for_state_change_completed (GstElement * pipeline);
void play_new_file (UserData * data, gboolean refresh_console_message);

/* Local variables */
gboolean ui_thread_die = FALSE;
pthread_t id_ui_thread = 0;
pthread_t id_autoplay_thread = 0;
pthread_cond_t cond_gst_data = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex_gst_data = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_ui_data = PTHREAD_MUTEX_INITIALIZER;

/* Call back functions */
static void
on_pad_added (GstElement * element, GstPad * pad, gpointer data)
{
  GstCaps *video_caps = NULL;
  GstPad *sinkpad = NULL;
  GstCaps *new_pad_caps = NULL;
  GstStructure *new_pad_struct = NULL;
  const gchar *new_pad_type = NULL;
  UserData *puser_data = (UserData *) data;
  struct screen_t *main_screen = puser_data->main_screen;

  new_pad_caps = gst_pad_query_caps (pad, NULL);
  new_pad_struct = gst_caps_get_structure (new_pad_caps, 0);
  new_pad_type = gst_structure_get_name (new_pad_struct);

  GstState currentState;
  GstState pending;

  LOGD ("Received new pad '%s' from '%s': %s\n", GST_PAD_NAME (pad),
      GST_ELEMENT_NAME (element), new_pad_type);

  pthread_mutex_lock (&mutex_gst_data);

  if (g_str_has_prefix (new_pad_type, "audio")) {
    /* Need to set Gst State to PAUSED before change state from NULL to PLAYING */
    gst_element_get_state(puser_data->audio_queue, &currentState, &pending, GST_CLOCK_TIME_NONE);
    if(currentState == GST_STATE_NULL){
      gst_element_set_state (puser_data->audio_queue, GST_STATE_PAUSED);
    }
    gst_element_get_state(puser_data->audio_decoder, &currentState, &pending, GST_CLOCK_TIME_NONE);
    if(currentState == GST_STATE_NULL){
      gst_element_set_state (puser_data->audio_decoder, GST_STATE_PAUSED);
    }
    gst_element_get_state(puser_data->audio_resample, &currentState, &pending, GST_CLOCK_TIME_NONE);
    if(currentState == GST_STATE_NULL){
      gst_element_set_state (puser_data->audio_resample, GST_STATE_PAUSED);
    }
    gst_element_get_state(puser_data->audio_capsfilter, &currentState, &pending,
GST_CLOCK_TIME_NONE);
    if(currentState == GST_STATE_NULL){
      gst_element_set_state (puser_data->audio_capsfilter, GST_STATE_PAUSED);
    }
    gst_element_get_state(puser_data->audio_sink, &currentState, &pending, GST_CLOCK_TIME_NONE);
    if(currentState == GST_STATE_NULL){
      gst_element_set_state (puser_data->audio_sink, GST_STATE_PAUSED);
    }

    /* Add back audio_queue, audio_decoder, audio_resample, audio_capsfilter, and audio_sink */
    gst_bin_add_many (GST_BIN (puser_data->pipeline),
        puser_data->audio_queue, puser_data->audio_decoder,
        puser_data->audio_resample, puser_data->audio_capsfilter,
        puser_data->audio_sink, NULL);

    /* Link audio_queue +++ audio_decoder +++ audio_resample +++ audio_capsfilter +++ audio_sink */
    if (gst_element_link_many (puser_data->audio_queue,
            puser_data->audio_decoder, puser_data->audio_resample,
            puser_data->audio_capsfilter, puser_data->audio_sink, NULL) != TRUE) {
```

```
      g_print
          ("audio_queue, audio_decoder, audio_resample, audio_capsfilter, and audio_sink could not be
linked.\n");
    }

    /* In case link this pad with the AAC-decoder sink pad */
    sinkpad = gst_element_get_static_pad (puser_data->audio_queue, "sink");
    if (GST_PAD_LINK_OK != gst_pad_link (pad, sinkpad)) {
      g_print ("Audio link failed\n");
    } else {
      LOGD ("Audio pad linked!\n");
    }
    gst_object_unref (sinkpad);
    if (new_pad_caps != NULL)
      gst_caps_unref (new_pad_caps);

    /* Change the pipeline to PLAYING state
     * TODO: The below statement temporarily fixes issue "Unable to pause the video".
     * The root cause is still unknown.
     */
    gst_element_set_state (puser_data->pipeline, GST_STATE_PLAYING);
  } else if (g_str_has_prefix (new_pad_type, "video")) {
    /* Recreate waylandsink */
    if (NULL == puser_data->video_sink) {
      puser_data->video_sink =
          gst_element_factory_make ("waylandsink", "video-output");
      LOGD ("Re-create gst_element_factory_make video_sink: %s\n",
          (NULL == puser_data->video_sink) ? ("FAILED") : ("SUCCEEDED"));

      /* Set position for displaying (0, 0) */
      g_object_set (G_OBJECT (puser_data->video_sink), "position-x", main_screen->x, "position-y",
main_screen->y, NULL);
    }

    /* Create decoder and parser for H264 video */
    if (g_str_has_prefix (new_pad_type, "video/x-h264")) {
      /* Recreate video-parser */
      if (NULL == puser_data->video_parser) {
        puser_data->video_parser =
            gst_element_factory_make ("h264parse", "h264-parser");
        LOGD ("Re-create gst_element_factory_make video_parser: %s\n",
            (NULL == puser_data->video_parser) ? ("FAILED") : ("SUCCEEDED"));
      }
      /* Recreate video-decoder */
      if (NULL == puser_data->video_decoder) {
        puser_data->video_decoder =
            gst_element_factory_make ("omxh264dec", "omxh264-decoder");
        LOGD ("Re-create gst_element_factory_make video_decoder: %s\n",
            (NULL == puser_data->video_decoder) ? ("FAILED") : ("SUCCEEDED"));
      }
    /* Create decoder and parser for H265 video */
    } else if (g_str_has_prefix (new_pad_type, "video/x-h265")) {
      /* Recreate video-parser */
      if (NULL == puser_data->video_parser) {
        puser_data->video_parser =
            gst_element_factory_make ("h265parse", "h265-parser");
        LOGD ("Re-create gst_element_factory_make video_parser: %s\n",
            (NULL == puser_data->video_parser) ? ("FAILED") : ("SUCCEEDED"));
      }
      /* Recreate video-decoder */
      if (NULL == puser_data->video_decoder) {
        puser_data->video_decoder =
            gst_element_factory_make ("omxh265dec", "omxh265-decoder");
        LOGD ("Re-create gst_element_factory_make video_decoder: %s\n",
            (NULL == puser_data->video_decoder) ? ("FAILED") : ("SUCCEEDED"));
      }
    } else {
      g_print("Unsupported video format\n");
      g_main_loop_quit (puser_data->loop);
    }

    /* Recreate vspmfilter and capsfilter in case of scaling full screen */
    if (puser_data->fullscreen) {
      /* Recreate vspmfilter */
      if (NULL == puser_data->video_filter) {
```

```
      puser_data->video_filter =
          gst_element_factory_make ("vspmfilter", "video-filter");
      LOGD ("Re-create gst_element_factory_make : %s\n",
          (NULL == puser_data->video_filter) ? ("FAILED") : ("SUCCEEDED"));

      g_object_set (G_OBJECT (puser_data->video_filter), "dmabuf-use", TRUE, NULL);
    }

    /* Recreate capsfilter */
    if (NULL == puser_data->video_capsfilter) {
      puser_data->video_capsfilter =
          gst_element_factory_make ("capsfilter", "video-capsfilter");
      LOGD ("Re-create gst_element_factory_make video_capsfilter: %s\n",
          (NULL == puser_data->video_capsfilter) ? ("FAILED") : ("SUCCEEDED"));

      /* Create simple cap which contains video's resolution */
      video_caps = gst_caps_new_simple ("video/x-raw",
          "width", G_TYPE_INT, main_screen->width,
          "height", G_TYPE_INT, main_screen->height, NULL);

      /* Add cap to capsfilter element */
      g_object_set (G_OBJECT (puser_data->video_capsfilter), "caps", video_caps, NULL);
      gst_caps_unref (video_caps);
    }
  }

  /* Need to set Gst State to PAUSED before change state from NULL to PLAYING */
  gst_element_get_state(puser_data->video_queue, &currentState, &pending, GST_CLOCK_TIME_NONE);
  if(currentState == GST_STATE_NULL){
    gst_element_set_state (puser_data->video_queue, GST_STATE_PAUSED);
  }
  gst_element_get_state(puser_data->video_parser, &currentState, &pending, GST_CLOCK_TIME_NONE);
  if (currentState == GST_STATE_NULL){
    gst_element_set_state (puser_data->video_parser, GST_STATE_PAUSED);
  }
  gst_element_get_state(puser_data->video_decoder, &currentState, &pending, GST_CLOCK_TIME_NONE);
  if(currentState == GST_STATE_NULL){
    gst_element_set_state (puser_data->video_decoder, GST_STATE_PAUSED);
  }

  /* Change state of vspmfilter and capsfilter in case of scaling full screen */
  if (puser_data->fullscreen) {
    gst_element_get_state(puser_data->video_filter, &currentState, &pending, GST_CLOCK_TIME_NONE);
    if(currentState == GST_STATE_NULL){
      gst_element_set_state (puser_data->video_filter, GST_STATE_PAUSED);
    }
    gst_element_get_state(puser_data->video_capsfilter, &currentState, &pending,
GST_CLOCK_TIME_NONE);
    if(currentState == GST_STATE_NULL){
      gst_element_set_state (puser_data->video_capsfilter, GST_STATE_PAUSED);
    }
  }

  gst_element_get_state(puser_data->video_sink, &currentState, &pending, GST_CLOCK_TIME_NONE);
  if(currentState == GST_STATE_NULL){
    gst_element_set_state (puser_data->video_sink, GST_STATE_PAUSED);
  }

  /* Add back video_queue, video_parser, video_decoder, and video_sink */
  gst_bin_add_many (GST_BIN (puser_data->pipeline), puser_data->video_queue,
      puser_data->video_parser, puser_data->video_decoder, puser_data->video_sink, NULL);

  /* Add back video_filter and video_capsfilter in case of scaling full screen */
  if (puser_data->fullscreen) {
    gst_bin_add_many (GST_BIN (puser_data->pipeline),
        puser_data->video_filter, puser_data->video_capsfilter, NULL);
  }

  /* Link video_queue -> video_parser -> video_decoder -> video_filter -> video_capsfilter -> video_sink
   * in case of scaling full screen.
   * Link video_queue -> video_parser -> video_decoder -> video_sink in case of not scaling full screen
*/
  if (puser_data->fullscreen) {
    if (gst_element_link_many (puser_data->video_queue, puser_data->video_parser,
```

```
            puser_data->video_decoder, puser_data->video_filter, puser_data->video_capsfilter,
            puser_data->video_sink, NULL) != TRUE) {
       g_print ("video_queue, video_parser, video_decoder, video_filter, video_capsfilter, and
video_sink could not be linked.\n");
      }
    } else {
     if (gst_element_link_many (puser_data->video_queue, puser_data->video_parser,
            puser_data->video_decoder, puser_data->video_sink, NULL) != TRUE) {
       g_print ("video_queue, video_parser, video_decoder and video_sink could not be linked.\n");
     }
    }

    /* In case link this pad with the omxh264-decoder sink pad */
    sinkpad = gst_element_get_static_pad (puser_data->video_queue, "sink");
    if (GST_PAD_LINK_OK != gst_pad_link (pad, sinkpad)) {
      g_print ("Video link failed\n");
    } else {
      LOGD ("Video pad linked!\n");
    }
    gst_object_unref (sinkpad);
    if (new_pad_caps != NULL)
      gst_caps_unref (new_pad_caps);

    /* Change the pipeline to PLAYING state
     * TODO: The below statement temporarily fixes issue "Unable to pause the video".
     * The root cause is still unknown.
     */
    gst_element_set_state (puser_data->pipeline, GST_STATE_PLAYING);
  }

  /* Signal the dynamic pad linked */
  pthread_mutex_unlock (&mutex_gst_data);
}

static void
no_more_pads (GstElement * element, gpointer data)
{
  pthread_mutex_lock (&mutex_gst_data);
  pthread_cond_signal (&cond_gst_data);
  pthread_mutex_unlock (&mutex_gst_data);
}

/* This function will be call from the UI thread to replay or play next/previous file */
void
sync_to_play_new_file (UserData * data, gboolean refresh_console_message)
{
  pthread_mutex_lock (&mutex_gst_data);
  play_new_file (data, refresh_console_message);
  /* Wait for all pad added - no_more_pads signaled */
  pthread_cond_wait (&cond_gst_data, &mutex_gst_data);
  /* Wait the state become PLAYING to get the video length */
  gst_element_get_state (data->pipeline, NULL, NULL, GST_CLOCK_TIME_NONE);
  gst_element_query_duration (data->pipeline, GST_FORMAT_TIME,
      &(data->media_length));
  pthread_mutex_unlock (&mutex_gst_data);
}

static void *
auto_play_thread_func (void *data)
{
  sync_to_play_new_file ((UserData *) data, TRUE);
  pthread_exit (NULL);
  return NULL;
}

static gboolean
bus_call (GstBus * bus, GstMessage * msg, gpointer data)
{
  GMainLoop *loop = ((UserData *) data)->loop;
  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_ERROR:{
      gchar *debug;
      GError *error;
      gst_message_parse_error (msg, &error, &debug);
```

```
      g_free (debug);
      g_printerr ("Error: %s\n", error->message);
      g_error_free (error);
      fclose (stdin);
      g_main_loop_quit (loop);
      break;
    }
    case GST_MESSAGE_EOS:{
      /* Auto play next file when EOS reach if possible */
      pthread_mutex_lock (&mutex_ui_data);
      gboolean ret = request_update_file_path (1);
      pthread_mutex_unlock (&mutex_ui_data);
      if (ret) {
        /* Need to call play_new_file() from another thread to avoid DEADLOCK */
        if (pthread_create (&id_autoplay_thread, NULL, auto_play_thread_func,
              (UserData *) data)) {
          LOGE ("pthread_create autoplay failed\n");
        }
      }
      break;
    }
    default:
      break;
  }
  return TRUE;
}

static void *
check_user_command_loop (void *data)
{
  GstElement *pipeline;
  gint64 media_length;
  GstState current_state;

  while (!ui_thread_die) {
    pthread_mutex_lock (&mutex_gst_data);
    pipeline = ((UserData *) data)->pipeline;
    current_state = wait_for_state_change_completed (pipeline);
    media_length = ((UserData *) data)->media_length;
    pthread_mutex_unlock (&mutex_gst_data);

    UserCommand cmd = get_user_command ();
    switch (cmd) {
      case QUIT:
        g_main_loop_quit (((UserData *) data)->loop);
        pthread_exit (NULL);
        return NULL;
        break;

      case PAUSE_PLAY:
        if (GST_STATE_PLAYING == current_state) {
          gst_element_set_state (pipeline, GST_STATE_PAUSED);
          g_print ("PAUSED!\n");
        } else if (GST_STATE_PAUSED == current_state) {
          gst_element_set_state (pipeline, GST_STATE_PLAYING);
          g_print ("PLAYED!\n");
        } else {
          g_print ("Cannot PAUSE/PLAY\n");
        }
        break;

      case STOP:{

        /* Seek to the begining */
        if (GST_STATE_PLAYING == current_state) {
          gst_element_set_state (pipeline, GST_STATE_PAUSED);
        }
        if (seek_to_time (pipeline, 0)) {
          g_print ("STOP\n");
        }
        break;
      }

      case REPLAY:
        if (seek_to_time (pipeline, 0)) {
```

```
      /* Seek to the begining */
      if (GST_STATE_PAUSED == current_state) {
        gst_element_set_state (pipeline, GST_STATE_PLAYING);
      }
      g_print ("REPLAYED!\n");
    }
    break;

  case FORWARD:{
    gint64 pos = get_current_play_position (pipeline);
    /* calculate the forward position */
    pos = pos + SKIP_POSITION;
    if (pos >= media_length) {
      pos = media_length;
    }
    if (seek_to_time (pipeline, pos)) {
      g_print ("current: %02d:%02d\n",
          (int) (GET_SECOND_FROM_NANOSEC (pos) / ONE_MINUTE),
          (int) (GET_SECOND_FROM_NANOSEC (pos) % ONE_MINUTE));
    }
    break;
  }

  case REWIND:{
    gint64 pos = get_current_play_position (pipeline);
    /* calculate the rewind position */
    pos = pos - SKIP_POSITION;
    if (pos < 0) {
      pos = 0;
    }
    if (seek_to_time (pipeline, pos)) {
      g_print ("current: %02d:%02d\n",
          (int) (GET_SECOND_FROM_NANOSEC (pos) / ONE_MINUTE),
          (int) (GET_SECOND_FROM_NANOSEC (pos) % ONE_MINUTE));
    }
    break;
  }

  case LIST:{
    pthread_mutex_lock (&mutex_ui_data);
    update_file_list ();
    pthread_mutex_unlock (&mutex_ui_data);
    break;
  }

  case PREVIOUS:{
    pthread_mutex_lock (&mutex_ui_data);
    gboolean ret = request_update_file_path (-1);
    pthread_mutex_unlock (&mutex_ui_data);
    if (ret) {
      sync_to_play_new_file ((UserData *) data, FALSE);
    }
    break;
  }

  case NEXT:{
    pthread_mutex_lock (&mutex_ui_data);
    gboolean ret = request_update_file_path (1);
    pthread_mutex_unlock (&mutex_ui_data);
    if (ret) {
      sync_to_play_new_file ((UserData *) data, FALSE);
    }
    break;
  }

  case PLAYFILE:{
    pthread_mutex_lock (&mutex_ui_data);
    gboolean ret = request_update_file_path (0);
    pthread_mutex_unlock (&mutex_ui_data);
    if (ret) {
      sync_to_play_new_file ((UserData *) data, FALSE);
    }
    break;
  }
```

```
      case HELP:
        print_supported_command ();
        break;

      default:
        break;
    }
  }
  return NULL;
}

/* Private helper functions */
static gboolean
seek_to_time (GstElement * pipeline, gint64 time_nanoseconds)
{
  if (!gst_element_seek (pipeline, NORMAL_PLAYING_RATE, GST_FORMAT_TIME,
         GST_SEEK_FLAG_FLUSH, GST_SEEK_TYPE_SET, time_nanoseconds,
         GST_SEEK_TYPE_NONE, GST_CLOCK_TIME_NONE)) {
    g_print ("Seek failed!\n");
    return FALSE;
  }
  return TRUE;
}

static gint64
get_current_play_position (GstElement * pipeline)
{
  gint64 pos;

  if (gst_element_query_position (pipeline, GST_FORMAT_TIME, &pos)) {
    return pos;
  } else
    return -1;
}

GstState
wait_for_state_change_completed (GstElement * pipeline)
{
  GstState current_state = GST_STATE_VOID_PENDING;
  GstStateChangeReturn state_change_return = GST_STATE_CHANGE_ASYNC;
  do {
    state_change_return =
        gst_element_get_state (pipeline, &current_state, NULL,
        GST_CLOCK_TIME_NONE);
  } while (state_change_return == GST_STATE_CHANGE_ASYNC);
  return current_state;
}

/* Seek to beginning and flush all the data in the current pipeline to prepare for the next file */
void
play_new_file (UserData * data, gboolean refresh_console_message)
{
  GstElement *pipeline = data->pipeline;
  GstElement *source = data->source;
  GstElement *video_queue = data->video_queue;
  GstElement *audio_resample = data->audio_resample;
  GstElement *audio_capsfilter = data->audio_capsfilter;
  GstElement *audio_queue = data->audio_queue;
  GstElement *audio_decoder = data->audio_decoder;
  GstElement *audio_sink = data->audio_sink;
  bool fullscreen = data->fullscreen;
  gboolean ret = FALSE;

  GstState currentState;
  GstState pending;

  gst_element_get_state(pipeline, &currentState, &pending, GST_CLOCK_TIME_NONE);
  if(currentState == GST_STATE_PLAYING){
    gst_element_set_state (pipeline, GST_STATE_PAUSED);
    /* Seek to start and flush all old data */
    gst_element_seek_simple (pipeline, GST_FORMAT_TIME, GST_SEEK_FLAG_FLUSH, 0);
  }

  LOGD ("gst_element_set_state pipeline to NULL\n");
```

```
  gst_element_set_state (pipeline, GST_STATE_NULL);

  /* wait until the changing is complete */
  gst_element_get_state (pipeline, NULL, NULL, GST_CLOCK_TIME_NONE);

  /* Unlink and remove audio elements */
  audio_queue = gst_bin_get_by_name (GST_BIN (pipeline), "audio-queue");       /* Keep the element to
still exist after removing */
  if (NULL != audio_queue) {
    ret = gst_bin_remove (GST_BIN (pipeline), data->audio_queue);
    LOGD ("gst_bin_remove audio_queue from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
  }
  audio_decoder = gst_bin_get_by_name (GST_BIN (pipeline), "aac-decoder");     /* Keep the element to
still exist after removing */
  if (NULL != audio_decoder) {
    ret = gst_bin_remove (GST_BIN (pipeline), data->audio_decoder);
    LOGD ("gst_bin_remove audio_decoder from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
  }
  audio_resample = gst_bin_get_by_name (GST_BIN (pipeline), "audio-resample");
  if (NULL != audio_resample)
  {
    ret = gst_bin_remove (GST_BIN (pipeline), data->audio_resample);
    LOGD ("gst_bin_remove audio_resample from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
  }
  audio_capsfilter = gst_bin_get_by_name (GST_BIN (pipeline), "audio-capsfilter");
  if (NULL != audio_capsfilter)
  {
    ret = gst_bin_remove (GST_BIN (pipeline), data->audio_capsfilter);
    LOGD ("gst_bin_remove audio_capsfilter from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
  }
  audio_sink = gst_bin_get_by_name (GST_BIN (pipeline), "audio-output");       /* Keep the element to
still exist after removing */
  if (NULL != audio_sink) {
    ret = gst_bin_remove (GST_BIN (pipeline), data->audio_sink);
    LOGD ("gst_bin_remove audio_sink from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
  }

  /* Unlink and remove video elements */
  video_queue = gst_bin_get_by_name (GST_BIN (pipeline), "video-queue");       /* Keep the element to
still exist after removing */
  if (NULL != video_queue) {
    ret = gst_bin_remove (GST_BIN (pipeline), data->video_queue);
    LOGD ("gst_bin_remove video_queue from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
  }

  /* Remove video-parser completely */
  if (data->video_parser != NULL) {
    ret = gst_bin_remove (GST_BIN (pipeline), data->video_parser);
    LOGD ("gst_bin_remove video_parser from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
    if (TRUE == ret) {
      data->video_parser = NULL;
    }
  }

  /* Remove video-decoder completely */
  if (data->video_decoder != NULL) {
    ret = gst_bin_remove (GST_BIN (pipeline), data->video_decoder);
    LOGD ("gst_bin_remove video_decoder from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
    if (TRUE == ret) {
     data->video_decoder = NULL;
    }
  }

  if (fullscreen) {
    /* Remove vspmfilter completely */
    if (data->video_filter != NULL) {
```

```
      ret = gst_bin_remove (GST_BIN (pipeline), data->video_filter);
      LOGD ("gst_bin_remove video_filter from pipeline: %s\n",
          (ret) ? ("SUCCEEDED") : ("FAILED"));
      if (TRUE == ret) {
        data->video_filter = NULL;
      }
    }

    /* Remove capsfilter completely */
    if (data->video_capsfilter != NULL) {
      ret = gst_bin_remove (GST_BIN (pipeline), data->video_capsfilter);
      LOGD ("gst_bin_remove video_capsfilter from pipeline: %s\n",
          (ret) ? ("SUCCEEDED") : ("FAILED"));
      if (TRUE == ret) {
        data->video_capsfilter = NULL;
      }
    }
  }

  /* Remove waylandsink completely */
  if (data->video_sink != NULL) {
    ret = gst_bin_remove (GST_BIN (pipeline), data->video_sink);
    LOGD ("gst_bin_remove video_sink from pipeline: %s\n",
        (ret) ? ("SUCCEEDED") : ("FAILED"));
    if (TRUE == ret) {
      data->video_sink = NULL;
    }
  }

  /* Update file location */
  g_object_set (G_OBJECT (source), "location", get_current_file_path (), NULL);

  /* Set the pipeline to "playing" state */
  print_current_selected_file (refresh_console_message);
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
}

int
main (int argc, char *argv[])
{
  struct wayland_t *wayland_handler = NULL;
  struct screen_t *temp = NULL;
  struct screen_t main_screen;

  /* Check input arguments */
  if ((argc != 2) && (argc != 3)) {
    g_printerr ("Usage: %s <MP4 filename or directory>\n", argv[0]);
    g_printerr ("Usage: %s -s <MP4 filename or directory> to scale full screen\n", argv[0]);
    return -1;
  }

  if (argc == 3) {
    if (strcmp ("-s", argv[1]) != 0) {
      g_printerr ("Usage: %s <MP4 filename or directory>\n", argv[0]);
      g_printerr ("Usage: %s -s <MP4 filename or directory> to scale full screen\n", argv[0]);
      return -1;
    }
  }

  /* Get a list of available screen */
  wayland_handler = get_available_screens();

  /* Get main screen */
  temp = get_main_screen(wayland_handler);
  if (temp == NULL)
  {
    g_printerr("Cannot find any available screens. Exiting.\n");

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Prepare "main_screen" variable */
  memcpy(&main_screen, temp, sizeof(struct screen_t));
  destroy_wayland(wayland_handler);
```

```
/* Get dir_path and file_path if possible */
if (argc == 2) {
  if (!inject_dir_path_to_player (argv[1])) {
    return -1;
  }
} else if (argc == 3) {
  if (!inject_dir_path_to_player (argv[2])) {
    return -1;
  }
}

GMainLoop *loop;
UserData user_data;

GstElement *pipeline, *source, *demuxer;
GstElement *video_queue, *video_sink;
GstElement *audio_queue, *audio_decoder, *audio_resample,
          *audio_capsfilter, *audio_sink;

if (argc == 3) {
  user_data.fullscreen = true;
} else {
  user_data.fullscreen = false;
}

GstCaps *caps;
GstBus *bus;
guint bus_watch_id;

/* Initialization */
gst_init (NULL, NULL);
loop = g_main_loop_new (NULL, FALSE);

/* Create GStreamer elements instead of waylandsink */
pipeline = gst_pipeline_new ("video-player");
source = gst_element_factory_make ("filesrc", "file-source");
demuxer = gst_element_factory_make ("qtdemux", "qt-demuxer");
/* elements for Video thread */
video_queue = gst_element_factory_make ("queue", "video-queue");
video_sink = NULL;
/* elements for Audio thread */
audio_queue = gst_element_factory_make ("queue", "audio-queue");
audio_decoder = gst_element_factory_make ("faad", "aac-decoder");
audio_resample = gst_element_factory_make("audioresample", "audio-resample");
audio_capsfilter = gst_element_factory_make ("capsfilter", "audio-capsfilter");
audio_sink = gst_element_factory_make ("alsasink", "audio-output");

if (!pipeline || !source || !demuxer || !video_queue
    || !audio_queue || !audio_decoder || !audio_resample
    || !audio_capsfilter || !audio_sink) {
  g_printerr ("One element could not be created. Exiting.\n");
  return -1;
}

/* Create simple cap which contains audio's sample rate */
caps = gst_caps_new_simple ("audio/x-raw",
    "rate", G_TYPE_INT, AUDIO_SAMPLE_RATE, NULL);

/* Add cap to capsfilter element */
g_object_set (G_OBJECT (audio_capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);

/* Add all created elements into the pipeline */
gst_bin_add_many (GST_BIN (pipeline), source, demuxer,
    video_queue, audio_queue, audio_decoder,
    audio_resample, audio_capsfilter, audio_sink, NULL);

/* Link the elements together:
   - file-source -> qt-demuxer
 */
if (gst_element_link (source, demuxer) != TRUE) {
  g_printerr ("File source could not be linked.\n");
  gst_object_unref (pipeline);
  return -1;
```

```
  }

  /* Construct user data */
  user_data.loop = loop;
  user_data.pipeline = pipeline;
  user_data.source = source;
  user_data.demuxer = demuxer;
  user_data.audio_queue = audio_queue;
  user_data.audio_decoder = audio_decoder;
  user_data.audio_resample = audio_resample;
  user_data.audio_capsfilter = audio_capsfilter;
  user_data.audio_sink = audio_sink;
  user_data.video_queue = video_queue;
  user_data.video_parser = NULL;
  user_data.video_decoder = NULL;
  user_data.video_sink = video_sink;
  user_data.media_length = 0;
  user_data.main_screen = &main_screen;

  if (user_data.fullscreen) {
    user_data.video_filter = NULL;
    user_data.video_capsfilter = NULL;
  }

  /* Set up the pipeline */
  /* we add a message handler */
  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  bus_watch_id = gst_bus_add_watch (bus, bus_call, &user_data);
  gst_object_unref (bus);

  g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added),
      &user_data);
  g_signal_connect (demuxer, "no-more-pads", G_CALLBACK (no_more_pads), NULL);

  /* Show the media file list */
  update_file_list ();

  if (try_to_update_file_path ()) {
    sync_to_play_new_file (&user_data, FALSE);
  }

  /* Handle user input thread */
  if (pthread_create (&id_ui_thread, NULL, check_user_command_loop, &user_data)) {
    LOGE ("pthread_create failed\n");
    goto exit;
  }

  /* Iterate */
  g_main_loop_run (loop);

  /* Out of the main loop, clean up nicely */
  g_print ("Returned, stopping playback...\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);

  g_print ("Wait for UI thread terminated.\n");
  ui_thread_die = TRUE;
  pthread_join (id_ui_thread, NULL);
  if (id_autoplay_thread != 0) {
    g_print ("Wait Autoplay thread terminated.\n");
    pthread_join (id_autoplay_thread, NULL);
  }

exit:
  g_print ("Deleting pipeline...\n");
  gst_object_unref (GST_OBJECT (pipeline));
  g_source_remove (bus_watch_id);
  g_main_loop_unref (loop);

  g_print ("Program end!\n");
  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.11 Audio Player and 3.2.14 File Play.

UserData structure

```
typedef struct tag_user_data
{
  GMainLoop *loop;
  GstElement *pipeline;
  GstElement *source;
  GstElement *demuxer;
  GstElement *audio_queue;
  GstElement *audio_decoder;
  GstElement *audio_resample;
  GstElement *audio_capsfilter;
  GstElement *audio_sink;
  GstElement *video_queue;
  GstElement *video_parser;
  GstElement *video_decoder;
  GstElement *video_filter;
  GstElement *video_capsfilter;
  GstElement *video_sink;
  gint64 media_length;
  struct screen_t *main_screen;
  bool fullscreen;
} UserData;
```

This structure contains:

- Variable `loop` (`GMainLoop`): An opaque data type to represent the main event loop of a Glib application.
- Variable `pipeline` (`GstElement`): A GStreamer pipeline which contains connected video elements.
- Variable `source` (`GstElement`): A GStreamer element to read data from a local file.
- Variable `demuxer` (`GstElement`): A GStreamer element to de-multiplex an MP4 file into audio and video streams.
- Variable `audio_queue` and `video_queue` (`GstElement`): A GStreamer element to queue data until one of the limits specified by the `max-size-buffers`, `max-size-bytes`, and/or `max-size-time` properties has been reached. Any attempt to push more buffers into the queue will block the pushing thread until more space becomes available.
- Variable `audio_decoder` (`GstElement`): A GStreamer element to decompress MPEG-2/4 AAC stream to raw S16LE-formatted audio.
- Variable `audio_resample` (`GstElement`): A GStreamer element to resample raw audio buffers to different sample rates using a configurable windowing function to enhance quality.
- Variable `audio_capsfilter` (`GstElement`): A GStreamer element to contain target sample rate `44.1 kHz`. Variable `audio_resample` will resample audio based on this value.
- Variable `audio_sink` (`GstElement`): A GStreamer element to render audio samples using the ALSA audio API.
- Variable `video_parser` (`GstElement`): A GStreamer element to parse H.264 stream to AVC format which `omxh264dec` can recognize and process.
- Variable `video_decoder` (`GstElement`): A GStreamer element to decompress H.264 stream to raw NV12-formatted video.
- Variable `video_filter` (`GstElement`): A GStreamer element to handle video scaling.
- Variable `video_capfilter` (GstElement): A GStreamer element to contain screen resolution.
- Variable `video_sink` (`GstElement`): A GStreamer element to create its own window and renders the decoded video frames to that.
- Variable `media_length` (`qint64`): An 8-byte integer variable to represent video duration.
- Variable `main_screen` (`screen_t`): A pointer to screen_t structure to contain monitor information, such as: (x, y), width, and height.

- Variable `fullscreen` (`bool`): A boolean variable to enable full-screen mode.

Video pipeline

```
source = gst_element_factory_make ("filesrc", "file-source");
demuxer = gst_element_factory_make ("qtdemux", "qt-demuxer");
video_queue = gst_element_factory_make ("queue", "video-queue");
video_sink = NULL;

audio_queue = gst_element_factory_make ("queue", "audio-queue");
audio_decoder = gst_element_factory_make ("faad", "aac-decoder");
audio_resample = gst_element_factory_make("audioresample", "audio-resample");
audio_capsfilter = gst_element_factory_make ("capsfilter", "audio-capsfilter");
audio_sink = gst_element_factory_make ("alsasink", "audio-output");

caps = gst_caps_new_simple ("audio/x-raw", "rate", G_TYPE_INT, AUDIO_SAMPLE_RATE, NULL);

g_object_set (G_OBJECT (audio_capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);

gst_bin_add_many (GST_BIN (pipeline), source, demuxer,
    video_queue, audio_queue, audio_decoder,
    audio_resample, audio_capsfilter, audio_sink, NULL);

gst_element_link (source, demuxer)

/* Construct user data */
user_data.loop = loop;
user_data.pipeline = pipeline;
user_data.source = source;
user_data.demuxer = demuxer;
user_data.audio_queue = audio_queue;
user_data.audio_decoder = audio_decoder;
user_data.audio_resample = audio_resample;
user_data.audio_capsfilter = audio_capsfilter;
user_data.audio_sink = audio_sink;
user_data.video_queue = video_queue;
user_data.video_parser = NULL;
user_data.video_decoder = NULL;
user_data.video_sink = video_sink;
user_data.media_length = 0;
user_data.main_screen = &main_screen;

if (user_data.fullscreen) {
  user_data.video_filter = NULL;
  user_data.video_capsfilter = NULL;
}

/* Set up the pipeline */
/* we add a message handler */
bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
bus_watch_id = gst_bus_add_watch (bus, bus_call, &user_data);
gst_object_unref (bus);

g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added), &user_data);
g_signal_connect (demuxer, "no-more-pads", G_CALLBACK (no_more_pads), NULL);
```

Basically, this pipeline is just like File Play except it uses `gst_bus_add_watch()` instead of `gst_bus_timed_pop_filtered()` to receive messages (such as: error or `EOS` (`End-of-Stream`)) from `bus_call()` asynchronously.

Note:

The `video_parser`, `video_decoder`, `waylandsink`, `vspmfilter` and `capsfilter` element will be created at runtime and removed right before playing new video file to reset their settings. If not, the `pipeline` cannot play the video.
That's why they are not created in `main()`.

Function on_pad_added()

```
static void on_pad_added (GstElement * element, GstPad * pad, gpointer data)
{  if (g_str_has_prefix (new_pad_type, "audio")) {
    gst_element_set_state (puser_data->audio_queue, GST_STATE_PAUSED);
    gst_element_set_state (puser_data->audio_decoder, GST_STATE_PAUSED);
    gst_element_set_state (puser_data->audio_resample, GST_STATE_PAUSED);
    gst_element_set_state (puser_data->audio_capsfilter, GST_STATE_PAUSED);
    gst_element_set_state (puser_data->audio_sink, GST_STATE_PAUSED);

    /* Add back audio_queue, audio_decoder, audio_resample, audio_capsfilter, and audio_sink */
    gst_bin_add_many (GST_BIN (puser_data->pipeline),
        puser_data->audio_queue, puser_data->audio_decoder,
        puser_data->audio_resample, puser_data->audio_capsfilter,
        puser_data->audio_sink, NULL);

    /* Link audio_queue +++ audio_decoder +++ audio_resample +++ audio_capsfilter +++ audio_sink */
    gst_element_link_many (puser_data->audio_queue,
        puser_data->audio_decoder, puser_data->audio_resample,
        puser_data->audio_capsfilter, puser_data->audio_sink, NULL);

    /* In case link this pad with the AAC-decoder sink pad */
    sinkpad = gst_element_get_static_pad (puser_data->audio_queue, "sink");
    if (GST_PAD_LINK_OK != gst_pad_link (pad, sinkpad)) {
      g_print ("Audio link failed\n");
    } else {
      LOGD ("Audio pad linked!\n");
    }

    gst_object_unref (sinkpad);
    if (new_pad_caps != NULL)
      gst_caps_unref (new_pad_caps);

   /* Change the pipeline to PLAYING state */
   gst_element_set_state (puser_data->pipeline, GST_STATE_PLAYING);
}
```

If the `pad` is an audio pad, the application will set all audio elements, such as: `audio_queue` (`queue`), `audio_decoder` (`faad`), `audio_resample` (`audioresample`), `audio_capsfilter` (`capsfilter`), and `audio_sink` (`alsasink`) to `PAUSED` state, then link, and add them to the `pipeline`.

Note that we have to link `demuxer` (`qtdemux`) to `audio_queue` (`queue`) manually (`gst_pad_link`) just like File Play.

Finally, the application sets the `pipeline` to `PLAYING` state.

```
static void on_pad_added (GstElement * element, GstPad * pad, gpointer data)
{
  if (g_str_has_prefix (new_pad_type, "video")) {
    /* Recreate waylandsink */
    puser_data->video_sink = gst_element_factory_make ("waylandsink", "video-output");

    g_object_set (G_OBJECT (puser_data->video_sink), "position-x", main_screen->x,
                                                     "position-y", main_screen->y, NULL);

  if (g_str_has_prefix (new_pad_type, "video/x-h264")) {
    /* Recreate video-parser */
    puser_data->video_parser = gst_element_factory_make ("h264parse", "h264-parser");
    /* Recreate video-decoder */
    puser_data->video_decoder = gst_element_factory_make ("omxh264dec", "omxh264-decoder");
  } else if (g_str_has_prefix (new_pad_type, "video/x-h265")) {
    /* Recreate video-parser */
    puser_data->video_parser = gst_element_factory_make ("h265parse", "h265-parser");
    /* Recreate video-decoder */
    puser_data->video_decoder = gst_element_factory_make ("omxh265dec", "omxh265-decoder");
  }

   /* Recreate vspmfilter and capsfilter in case of scaling full screen */
  if (puser_data->fullscreen) {
     puser_data->video_filter = gst_element_factory_make ("vspmfilter", "video-filter");
     g_object_set (G_OBJECT (puser_data->video_filter), "dmabuf-use", TRUE, NULL);
```

```
    puser_data->video_capsfilter = gst_element_factory_make ("capsfilter", "video-capsfilter");
    video_caps = gst_caps_new_simple ("video/x-raw",
         "width", G_TYPE_INT, main_screen->width,
         "height", G_TYPE_INT, main_screen->height, NULL);    g_object_set (G_OBJECT
(puser_data->video_capsfilter), "caps", video_caps, NULL);
    gst_caps_unref (video_caps);
  }

  /* Need to set Gst State to PAUSED before change state from NULL to PLAYING */
  gst_element_set_state (puser_data->video_queue, GST_STATE_PAUSED);
  gst_element_set_state (puser_data->video_parser, GST_STATE_PAUSED);
  gst_element_set_state (puser_data->video_decoder, GST_STATE_PAUSED);
  if (puser_data->fullscreen) {
    gst_element_set_state (puser_data->video_filter, GST_STATE_PAUSED);
    gst_element_set_state (puser_data->video_capsfilter, GST_STATE_PAUSED);
  }
  gst_element_set_state (puser_data->video_sink, GST_STATE_PAUSED);

  gst_bin_add_many (GST_BIN (puser_data->pipeline), puser_data->video_queue,
    puser_data->video_parser, puser_data->video_decoder, puser_data->video_sink, NULL);

  /* Add back video_filter and video_capsfilter in case of scaling full screen */
  if (puser_data->fullscreen) {
    gst_bin_add_many (GST_BIN (puser_data->pipeline),
       puser_data->video_filter, puser_data->video_capsfilter, NULL);
  }

  if (puser_data->fullscreen) {
    if (gst_element_link_many (puser_data->video_queue, puser_data->video_parser,
          puser_data->video_decoder, puser_data->video_filter, puser_data->video_capsfilter,
          puser_data->video_sink, NULL) != TRUE) {
      g_print ("video_queue, video_parser, video_decoder, video_filter, video_capsfilter, and "
               "video_sink could not be linked.\n");
    }
  } else {
    if (gst_element_link_many (puser_data->video_queue, puser_data->video_parser,
          puser_data->video_decoder, puser_data->video_sink, NULL) != TRUE) {
      g_print ("video_queue, video_parser, video_decoder and video_sink could not be linked.\n");
    }
  }

  /* In case link this pad with the decoder sink pad */
  sinkpad = gst_element_get_static_pad (puser_data->video_queue, "sink");
  if (GST_PAD_LINK_OK != gst_pad_link (pad, sinkpad)) {
    g_print ("Video link failed\n");
  } else {
    LOGD ("Video pad linked!\n");
  }

  gst_object_unref (sinkpad);
  if (new_pad_caps != NULL)
    gst_caps_unref (new_pad_caps);

  /* Change the pipeline to PLAYING state */
  gst_element_set_state (puser_data->pipeline, GST_STATE_PLAYING);
}
```

If the `pad` is a video pad, the application will create and configure `video_parser` (`h264parse/h265parse`), `video_decoder` (`omxh264dec/omxh265dec`), `video_filter` (`vspmfilter`), `video_capsfilter` (`capsfilter`) (`video_filter` and `video_capsfilter` are optional) for the new video file. Then, it sets them along with other video elements, such as: `video_queue` (`queue`), and `video_sink` (`waylandsink`) to `PAUSED` state. Next, it will link and add these elements to the `pipeline`.

Note that we have to link `demuxer` (`qtdemux`) to `video_queue` (`queue`) manually (`gst_pad_link`) just like File Play.

Finally, the application sets the `pipeline` to `PLAYING` state.

(3) player.h/player.c

```
#include <gst/gst.h>
#include <stdbool.h>            /* bool type */
#include <stdio.h>              /* getline() API */
#include <stdlib.h>             /* free(), atoi() API */
#include <sys/stat.h>           /* stat(), struct stat */
#include <dirent.h>             /* struct dirent */
#include <libgen.h>             /* basename */
#include <limits.h>             /* PATH_MAX */


//#define DEBUG_LOG
#ifdef DEBUG_LOG
#define LOGD(...)    g_print(__VA_ARGS__)
#else
#define LOGD(...)
#endif

#define LOGI(...)    g_print(__VA_ARGS__)
#define LOGE(...)    g_printerr(__VA_ARGS__)

#define FILE_SUFFIX  ".mp4"
typedef enum tag_user_command
{
  INVALID = 0,
  REPLAY,
  PAUSE_PLAY,
  STOP,
  FORWARD,
  REWIND,
  HELP,
  QUIT,
  LIST,
  PREVIOUS,
  NEXT,
  PLAYFILE,
} UserCommand;

gchar *get_current_file_path (void);

gboolean inject_dir_path_to_player (const gchar * path);
void update_file_list (void);
gboolean try_to_update_file_path (void);
gboolean request_update_file_path (gint32 offset_file);

UserCommand get_user_command (void);
void print_supported_command (void);
void print_current_selected_file (gboolean refresh_console_message);
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.11 Audio Player.

Macro

```
#define FILE_SUFFIX  ".mp4"
```

The `FILE_SUFFIX` macro defines the file extension that is supported by the `pipeline`. In this application, it only accepts video files whose extension are `.mp4`.

3.2.12.2 Result

Can play MP4 files. User can use some playback features (such as: start, stop, pause, resume...).

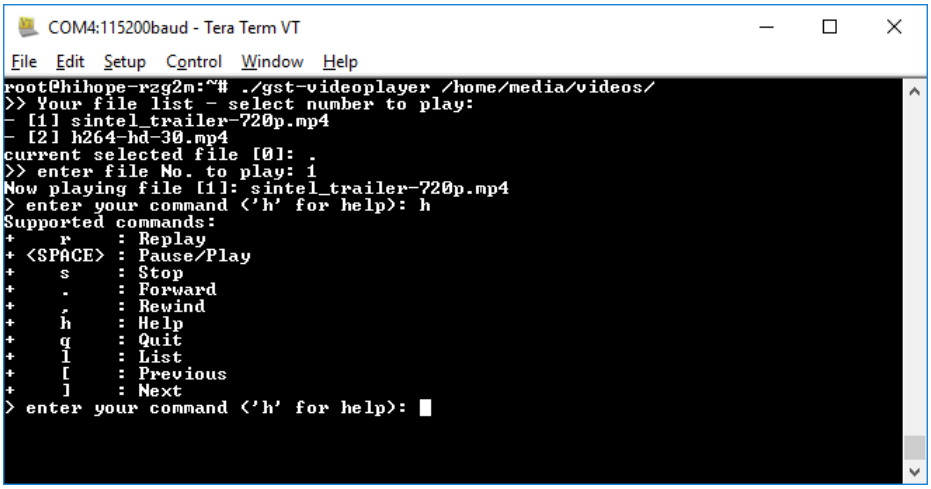


Figure 3.26 Video Player result

3.2.12.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.12.4 Special Instruction

- Download the input files at:
  https://download.blender.org/durian/trailer/sintel_trailer-720p.mp4
  http://www-stg.renesas.com/ja-jp/media/products/microcontrollers-microprocessors/rz/rzg/hmi-mmpoc-videos/h264-wvga-30.mp4
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/qt-videos/renesas-bigideasforeveryspace.mp4

Note:

RZ/G2E platform does not support 2K and 4K video.

3.2.13 Audio Video Play

Play H.264 video and Ogg/Vorbis audio file independently.

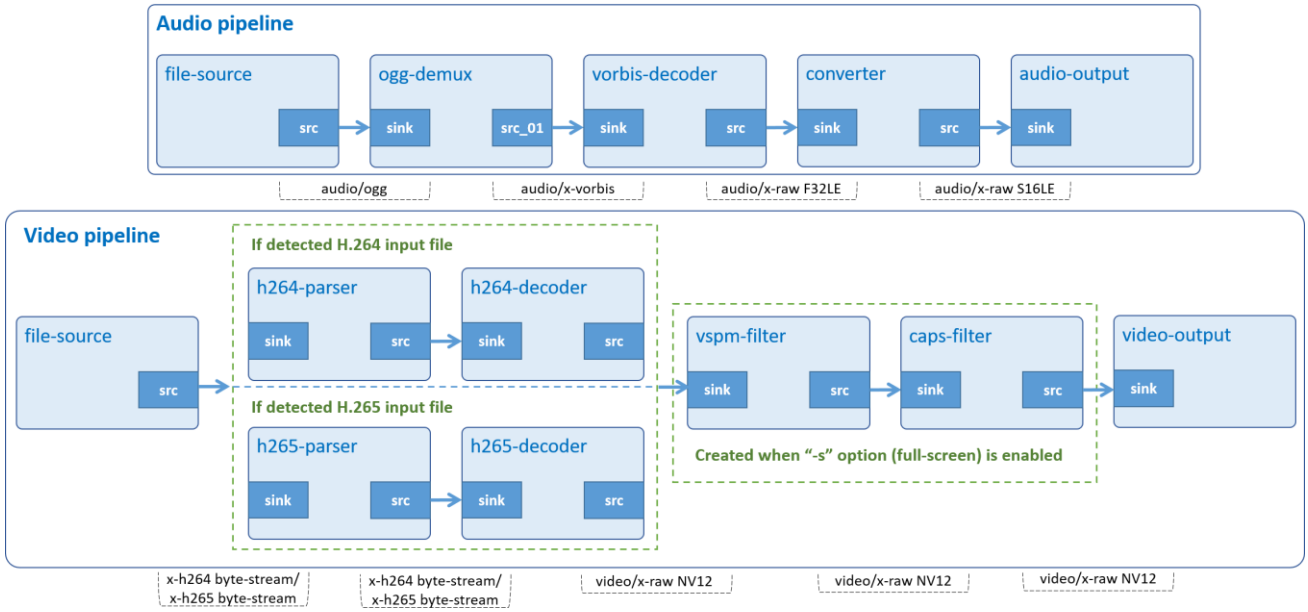


Figure 3.27 Audio-Video Play pipeline

3.2.13.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <gst/gst.h>
#include <stdbool.h>
#include <stdio.h>
#include <wayland-client.h>
#include <stdlib.h>
#include <string.h>
#include <libgen.h>

#define FORMAT           "S16LE"
#define ARG_PROGRAM_NAME 0
#define ARG_INPUT_AUDIO  1
#define ARG_INPUT_VIDEO  2
#define ARG_SCALE        3
#define ARG_COUNT        4

typedef struct _CustomData
{
  GMainLoop *loop;
  int loop_reference;
  GMutex mutex;
  const char *video_ext;
  struct wayland_t *wayland_handler;
  struct screen_t *main_screen;
  bool fullscreen;
} CustomData;

static void
try_to_quit_loop (CustomData * p_shared_data)
{
```

```
  g_mutex_lock (&p_shared_data->mutex);
  --(p_shared_data->loop_reference);
  if (0 == p_shared_data->loop_reference) {
    g_main_loop_quit ((p_shared_data->loop));
  }
  g_mutex_unlock (&p_shared_data->mutex);
}

static gboolean
bus_call (GstBus * bus, GstMessage * msg, gpointer data)
{
  GstElement *pipeline = (GstElement *) GST_MESSAGE_SRC (msg);
  gchar *pipeline_name = GST_MESSAGE_SRC_NAME (msg);

  switch (GST_MESSAGE_TYPE (msg)) {

    case GST_MESSAGE_EOS:
      g_print ("End-Of-Stream reached.\n");

      g_print ("Stoping %s...\n", pipeline_name);
      gst_element_set_state (pipeline, GST_STATE_NULL);
      g_print ("Deleting %s...\n", pipeline_name);
      gst_object_unref (GST_OBJECT (pipeline));
      try_to_quit_loop ((CustomData *) data);
      break;

    case GST_MESSAGE_ERROR:{
      gchar *debug_info;
      GError *err;
      gst_message_parse_error (msg, &err, &debug_info);
      g_printerr ("Error received from element %s: %s.\n",
          pipeline_name, err->message);
      g_printerr ("Debugging information: %s.\n",
          debug_info ? debug_info : "none");
      g_clear_error (&err);
      g_free (debug_info);

      g_print ("Stoping %s...\n", pipeline_name);
      gst_element_set_state (pipeline, GST_STATE_NULL);
      g_print ("Deleting %s...\n", pipeline_name);
      gst_object_unref (GST_OBJECT (pipeline));
      try_to_quit_loop ((CustomData *) data);
      break;
    }
    default:
      /* Don't care other message */
      break;
  }
  return TRUE;
}

static void
on_pad_added (GstElement * element, GstPad * pad, gpointer data)
{
  GstPad *sinkpad;
  GstElement *decoder = (GstElement *) data;

  /* We can now link this pad with the vorbis-decoder sink pad */
  g_print ("Dynamic pad created, linking demuxer/decoder\n");

  sinkpad = gst_element_get_static_pad (decoder, "sink");

  gst_pad_link (pad, sinkpad);

  gst_object_unref (sinkpad);
}

guint
create_audio_pipeline (GstElement ** p_audio_pipeline, const gchar * input_file,
    CustomData * data)
{
  GstBus *bus;
  guint audio_bus_watch_id;
  GstElement *audio_source, *audio_demuxer, *audio_decoder, *audio_converter,
      *audio_capsfilter, *audio_sink;
```

```
  GstCaps *capsfilter;

  /* Create GStreamer elements for audio play */
  *p_audio_pipeline = gst_pipeline_new ("audio-play");
  audio_source = gst_element_factory_make ("filesrc", "audio-file-source");
  audio_demuxer = gst_element_factory_make ("oggdemux", "ogg-demuxer");
  audio_decoder = gst_element_factory_make ("vorbisdec", "vorbis-decoder");
  audio_converter = gst_element_factory_make ("audioconvert", "converter");
  audio_capsfilter = gst_element_factory_make ("capsfilter", "convert_caps");
  audio_sink = gst_element_factory_make ("alsasink", "audio-output");

  if (!*p_audio_pipeline || !audio_source || !audio_demuxer || !audio_decoder
      || !audio_converter || !audio_capsfilter || !audio_sink) {
    g_printerr ("One audio element could not be created. Exiting.\n");
    return 0;
  }

  /* Adding a message handler for audio pipeline */
  bus = gst_pipeline_get_bus (GST_PIPELINE (*p_audio_pipeline));
  audio_bus_watch_id = gst_bus_add_watch (bus, bus_call, data);
  gst_object_unref (bus);

  /* Add all elements into the audio pipeline */
  /* file-source | ogg-demuxer | vorbis-decoder | converter | capsfilter | alsa-output */
  gst_bin_add_many (GST_BIN (*p_audio_pipeline),
      audio_source, audio_demuxer, audio_decoder, audio_converter,
      audio_capsfilter, audio_sink, NULL);

  /* Set up for the audio pipeline */
  /* Set the input file location of the file source element */
  g_object_set (G_OBJECT (audio_source), "location", input_file, NULL);

  /* we set the caps option to the caps-filter element */
  capsfilter =
      gst_caps_new_simple ("audio/x-raw", "format", G_TYPE_STRING, FORMAT,
      NULL);
  g_object_set (G_OBJECT (audio_capsfilter), "caps", capsfilter, NULL);
  gst_caps_unref (capsfilter);

  /* we link the elements together */
  /* file-source -> ogg-demuxer ~> vorbis-decoder -> converter -> capsfilter -> alsa-output */
  if (!gst_element_link (audio_source, audio_demuxer)) {
    g_printerr ("Audio elements could not be linked.\n");
    gst_object_unref (*p_audio_pipeline);
    return 0;
  }

  if (!gst_element_link_many (audio_decoder, audio_converter, audio_capsfilter,
        audio_sink, NULL)) {
    g_printerr ("Audio elements could not be linked.\n");
    gst_object_unref (*p_audio_pipeline);
    return 0;
  }

  g_signal_connect (audio_demuxer, "pad-added", G_CALLBACK (on_pad_added),
      audio_decoder);

  return audio_bus_watch_id;
}

guint
create_video_pipeline (GstElement ** p_video_pipeline, const gchar * input_file,
    CustomData * data)
{
  GstBus *bus;
  guint video_bus_watch_id;
  GstElement *video_source, *video_parser, *video_decoder, *video_sink;

  /* Create GStreamer elements for video play */
  *p_video_pipeline = gst_pipeline_new ("video-play");
  video_source = gst_element_factory_make ("filesrc", "video-file-source");
  video_sink = gst_element_factory_make ("waylandsink", "video-output");

  if (strcasecmp ("h264", data->video_ext) == 0) {
    video_parser = gst_element_factory_make ("h264parse", "h264-parser");
```

```
  video_decoder = gst_element_factory_make ("omxh264dec", "h264-decoder");
} else {
  video_parser = gst_element_factory_make ("h265parse", "h265-parser");
  video_decoder = gst_element_factory_make ("omxh265dec", "h265-decoder");
}

if (!*p_video_pipeline || !video_source || !video_parser || !video_decoder
    || !video_sink) {
  g_printerr ("One video element could not be created. Exiting.\n");
  return 0;
}

/* Adding a message handler for video pipeline */
bus = gst_pipeline_get_bus (GST_PIPELINE (*p_video_pipeline));
video_bus_watch_id = gst_bus_add_watch (bus, bus_call, data);
gst_object_unref (bus);

/* Add elements into the video pipeline */
/* file-source | parser | decoder | video-output */
gst_bin_add_many (GST_BIN (*p_video_pipeline), video_source, video_parser,
    video_decoder, video_sink, NULL);

/* Set up for the video pipeline */
/* Set the input file location of the file source element */
g_object_set (G_OBJECT (video_source), "location", input_file, NULL);

if (!data->fullscreen) {
  /* Link the elements together */
  /* file-source -> parser -> decoder -> video-output */
  if (!gst_element_link_many (video_source, video_parser, video_decoder,
        video_sink, NULL)) {
    g_printerr ("Video elements could not be linked.\n");
    gst_object_unref (*p_video_pipeline);
    destroy_wayland(data->wayland_handler);
    return 0;
  }
} else {
  GstElement *video_filter, *video_capsfilter;
  GstCaps *caps;

  /* Create vspm-filter and caps-filter */
  video_filter = gst_element_factory_make ("vspmfilter", "vspm-filter");
  video_capsfilter = gst_element_factory_make ("capsfilter", "caps-filter");

  if (!video_filter || !video_capsfilter) {
    g_printerr ("One element could not be created. Exiting.\n");
    gst_object_unref (*p_video_pipeline);
    destroy_wayland(data->wayland_handler);
    return 0;
  }

  /* Set property "dmabuf-use" of vspmfilter to true */
  /* Without it, waylandsink will display broken video */
  g_object_set (G_OBJECT (video_filter), "dmabuf-use", TRUE, NULL);

  /* Create simple cap which contains video's resolution */
  caps = gst_caps_new_simple ("video/x-raw",
      "width", G_TYPE_INT, data->main_screen->width,
      "height", G_TYPE_INT, data->main_screen->height, NULL);

  /* Add cap to capsfilter element */
  g_object_set (G_OBJECT (video_capsfilter), "caps", caps, NULL);
  gst_caps_unref (caps);

  /* Add filter, capsfilter into the pipeline */
  gst_bin_add_many (GST_BIN (*p_video_pipeline), video_filter,
      video_capsfilter, NULL);

  /* Link the elements together */
  /* file-source -> parser -> decoder -> vspm-filter
   * |-> caps-filter -> video-output */
  if (gst_element_link_many (video_source, video_parser, video_decoder,
        video_filter, video_capsfilter, video_sink, NULL) != TRUE) {
    g_printerr ("Elements could not be linked.\n");
```

```
      gst_object_unref (*p_video_pipeline);
      destroy_wayland(data->wayland_handler);
      return -1;
    }
  }

  return video_bus_watch_id;
}

bool
play_pipeline (GstElement * pipeline, CustomData * p_shared_data)
{
  bool result = true;

  g_mutex_lock (&p_shared_data->mutex);
  ++(p_shared_data->loop_reference);
  if (gst_element_set_state (pipeline,
        GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
    g_printerr ("Unable to set the pipeline to the playing state.\n");
    --(p_shared_data->loop_reference);
    gst_object_unref (pipeline);
    result = false;
  }
  g_mutex_unlock (&p_shared_data->mutex);
  return result;
}

int
main (int argc, char *argv[])
{
  struct wayland_t *wayland_handler = NULL;
  struct screen_t *main_screen = NULL;

  CustomData shared_data;
  GstElement *audio_pipeline;
  GstElement *video_pipeline;
  guint audio_bus_watch_id;
  guint video_bus_watch_id;
  const char *audio_ext, *video_ext;
  char* file_name;

  if (((argc != 3) && (argc != ARG_COUNT))
      || ((argc == ARG_COUNT) && (strcmp (argv[ARG_SCALE], "-s")))) {
    g_printerr ("Error: Invalid arugments.\n");
    g_printerr ("Usage: %s <OGG file> <H264/H265 file> [-s]\n", argv[ARG_PROGRAM_NAME]);
    return -1;
  }

  /* Check full-screen option */
  if (argc == ARG_COUNT) {
    shared_data.fullscreen = true;
  }

  /* Get a list of available screen */
  wayland_handler = get_available_screens();

  /* Get main screen */
  main_screen = get_main_screen(wayland_handler);
  if (main_screen == NULL)
  {
    g_printerr("Cannot find any available screens. Exiting.\n");

    destroy_wayland(wayland_handler);
    return -1;
  }

  const gchar *input_audio_file = argv[ARG_INPUT_AUDIO];
  const gchar *input_video_file = argv[ARG_INPUT_VIDEO];

  /* Check input file */
  if (!is_file_exist(input_audio_file) || !is_file_exist(input_video_file))
  {
    g_printerr("Make sure the following files exist:\n");
    g_printerr("  %s\n", input_audio_file);
```

```
  g_printerr("  %s\n", input_video_file);
  return -1;
}

/* Get extension of input file */
file_name = basename ((char*) input_audio_file);
audio_ext = get_filename_ext (file_name);
file_name = basename ((char*) input_video_file);
video_ext = get_filename_ext (file_name);

/* Check extension of input file */
if ((strcasecmp ("ogg", audio_ext) != 0)
    || ((strcasecmp ("h264", video_ext) != 0)
    && (strcasecmp ("h265", video_ext) != 0))) {
  g_printerr ("Error: Unsupported input type.\n");
  g_printerr ("OGG audio format and H264/H265 video format are required.\n");
  return -1;
}

/* Initialization */
gst_init (&argc, &argv);
shared_data.loop = g_main_loop_new (NULL, FALSE);
shared_data.loop_reference = 0;
shared_data.video_ext = video_ext;
shared_data.wayland_handler = wayland_handler;
shared_data.main_screen = main_screen;
g_mutex_init (&shared_data.mutex);

/* Create pipelines */
audio_bus_watch_id =
    create_audio_pipeline (&audio_pipeline, input_audio_file, &shared_data);
video_bus_watch_id =
    create_video_pipeline (&video_pipeline, input_video_file, &shared_data);

/* Set the audio pipeline to "playing" state */
if (audio_bus_watch_id) {
  if (play_pipeline (audio_pipeline, &shared_data) == true) {
    g_print ("Now playing audio file: %s\n", input_audio_file);
  } else {
    g_printerr ("Unable to play audio file: %s\n", input_audio_file);
    destroy_wayland (wayland_handler);
    g_source_remove (audio_bus_watch_id);
    g_source_remove (video_bus_watch_id);
    return -1;
  }
} else {
  g_printerr ("Unable to create audio pipeline\n");
  destroy_wayland (wayland_handler);
  g_source_remove (audio_bus_watch_id);
  g_source_remove (video_bus_watch_id);
  return -1;
}

/* Set the video pipeline to "playing" state */
if (video_bus_watch_id) {
  if (play_pipeline (video_pipeline, &shared_data) == true) {
    g_print ("Now playing video file: %s\n", input_video_file);
  } else {
    g_printerr ("Unable to play video file: %s\n", input_video_file);
    destroy_wayland (wayland_handler);
    g_source_remove (audio_bus_watch_id);
    g_source_remove (video_bus_watch_id);
    return -1;
  }
} else {
  g_printerr ("Unable to create video pipeline\n");
  destroy_wayland(wayland_handler);
  g_source_remove (audio_bus_watch_id);
  g_source_remove (video_bus_watch_id);
  return -1;
}

/* Iterate */
g_print ("Running...\n");
```

```
  g_main_loop_run (shared_data.loop);

  /* Clean up "wayland_t" structure */
  destroy_wayland(wayland_handler);

  /* Out of loop. Clean up nicely */
  g_source_remove (audio_bus_watch_id);
  g_source_remove (video_bus_watch_id);
  g_main_loop_unref (shared_data.loop);
  g_mutex_clear (&shared_data.mutex);
  g_print ("Program end!\n");

  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.1 Audio Play and 3.2.2 Video Play.

Command-line argument

```
  if (((argc != 3) && (argc != ARG_COUNT))
      || ((argc == ARG_COUNT) && (strcmp (argv[ARG_SCALE], "-s")))) {
    g_printerr ("Error: Invalid arugments.\n");
    g_printerr ("Usage: %s <OGG file> <H264/H265 file> [-s]\n", argv[ARG_PROGRAM_NAME]);
    return -1;
  }
```

This application accepts 2 command-line arguments which point to an Ogg/Vorbis file and an H.264/H.265 file. If "-s" option is enabled, video will be scaled to full-screen.

CustomData structure

```
typedef struct _CustomData
{
  GMainLoop *loop;
  int loop_reference;
  GMutex mutex;
  const char *video_ext;
  struct wayland_t *wayland_handler;
  struct screen_t *main_screen;
  bool fullscreen;
} CustomData;
```

This structure contains:

- Variable `loop` (`GMainLoop`): An opaque data type to represent the main event loop of a Glib application.
- Variable `loop_reference` (`int`): An integer variable to represent the number of `PLAYING` pipelines available. It is managed by `mutex` structure which controls GStreamer object release and program termination.
- Variable `mutex` (`GMutex`): An opaque data type to represent mutex (mutual exclusion). It can be used to protect data from critical section.
- Variable `video_ext` (`char`): A string variable to represent video extension.
- Variable `wayland_handler` (`wayland_t`): A pointer to wayland_t structure to contain list of monitors.
- Variable `main_screen` (`screen_t`): A pointer to screen_t structure to contain monitor information, such as: (x, y), width, and height.
- Variable `fullscreen` (`qint64`): A boolean variable to enable full-screen mode.

Initialize CustomData structure

```
shared_data.loop = g_main_loop_new (NULL, FALSE);
```

This function creates a new GMainLoop structure with default (NULL) context (GMainContext).

Basically, the main event loop manages all the available sources of events. To allow multiple independent sets of sources to be handled in different threads, each source is associated with a GMainContext. A GMainContext can only be running in a single thread, but sources can be added to it and removed from it from other threads.

The application will use GMainLoop to catch events and signals from 2 independent GStreamer pipelines. One plays a video and the other plays an Ogg/Vorbis audio file.

```
shared_data.loop_reference = 0;
```

At this point, variable loop_reference is set to 0 to indicate that there are no running pipelines available.

```
g_mutex_init (&shared_data.mutex);
```

This function initializes a GMutex so that it can be used. The structure protects loop_reference from read/write access of GStreamer threads.

Please use g_mutex_clear() if the mutex is no longer needed.

```
shared_data.video_ext = video_ext;
```

Variable video_ext contains the extension of video input file to create suitable video_parser and video_decoder

```
shared_data.main_screen = main_screen;
```

Variable main_screen contains the resolution of sreen to scale video to full-sreen.

Audio pipeline

```
guint create_audio_pipeline (GstElement** p_audio_pipeline, const gchar* input_file, CustomData* data);
```

Basically, the audio pipeline is just like Audio Play except it uses gst_bus_add_watch() instead of gst_bus_timed_pop_filtered() to receive messages (such as: error or EOS (End-of-Stream)) from bus_call() asynchronously.

```
bus = gst_pipeline_get_bus (GST_PIPELINE (*p_audio_pipeline));
audio_bus_watch_id = gst_bus_add_watch (bus, bus_call, data);
gst_object_unref (bus);
```

Note that the bus should be freed with gst_caps_unref() if it is not used anymore.

Video pipeline

```
guint create_video_pipeline (GstElement ** p_video_pipeline, const gchar * input_file, CustomData* data)
```

Basically, the video pipeline is just like Video Play except it uses gst_bus_add_watch() instead of gst_bus_timed_pop_filtered() to receive messages (such as: error or EOS (End-of-Stream)) from bus_call() asynchronously.

```
bus = gst_pipeline_get_bus (GST_PIPELINE (*p_video_pipeline));
video_bus_watch_id = gst_bus_add_watch (bus, bus_call, data);
gst_object_unref (bus);
```

Note that the bus should be freed with gst_caps_unref() if it is not used anymore.

Function bus_call()

```
static gboolean bus_call (GstBus * bus, GstMessage * msg, gpointer data)
```

This function will be called if either audio or video pipeline posts messages to `bus`.

```
try_to_quit_loop ((CustomData *) data);
```

Note that `bus_call()` only processes error and `EOS` messages. Moreover, no matter what the messages are, it will eventally call `try_to_quit_loop()` to try exiting main loop.

Create pipelines

```
create_audio_pipeline (&audio_pipeline, input_audio_file, &shared_data);
create_video_pipeline (&video_pipeline, input_video_file, &shared_data);
```

This code block creates 2 pipelines, one plays Ogg/Vorbis audio and the other displays MP4.

Play pipelines

```
int main (int argc, char *argv[])
{
  play_pipeline (audio_pipeline, &shared_data);
  play_pipeline (video_pipeline, &shared_data);
}
bool play_pipeline (GstElement * pipeline, CustomData * p_shared_data)
{
  g_mutex_lock (&p_shared_data->mutex);
  ++(p_shared_data->loop_reference);
  if (gst_element_set_state (pipeline,
        GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
    g_printerr ("Unable to set the pipeline to the playing state.\n");
    --(p_shared_data->loop_reference);
    gst_object_unref (pipeline);
  }
  g_mutex_unlock (&p_shared_data->mutex);
}
```

Basically, this function sets the state of pipeline to `PLAYING`. If successful, it will increases `loop_reference` to indicate that there is 1 more running pipeline. Note that the variable must be `2` for this application to play both audio and video.

Run main loop

```
g_main_loop_run (shared_data.loop);
```

This function runs main loop until `g_main_loop_quit()` is called on the loop (context `NULL`). In other words, it will make the context check if anything it watches for has happened. For example, when a message has been posted on the bus (`gst_element_get_bus`), the default main context will automatically call `bus_call()` to notify the message.

Stop pipelines

```
static void try_to_quit_loop (CustomData * p_shared_data)
{
  g_mutex_lock (&p_shared_data->mutex);
  --(p_shared_data->loop_reference);
  if (0 == p_shared_data->loop_reference) {
    g_main_loop_quit ((p_shared_data->loop));
  }
  g_mutex_unlock (&p_shared_data->mutex);
}
```

The main event loop will stop only if variable `loop_reference` reaches to `0`. This means the application will exit when both audio and video pipeline stopped. Also note that `mutex` is used to prevent GStreamer threads from reading incorrect value of `loop_reference`.

3.2.13.2 Result

Can play audio (through USB microphone) and display video on the monitor.

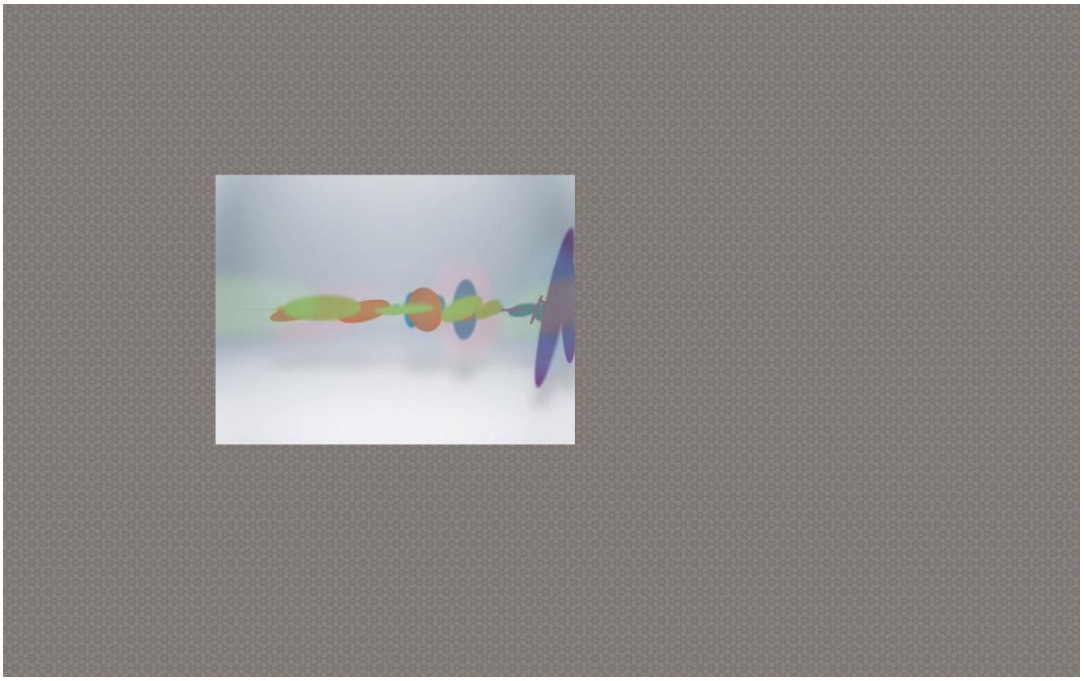


Figure 3.28 Audio Video Play result

3.2.13.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.13.4 Special Instruction

- Download audio file Rondo_Alla_Turka.ogg at:
  https://upload.wikimedia.org/wikipedia/commons/b/bd/Rondo_Alla_Turka.ogg
- Download video file vga1.h264 at:
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga1.h264
- To set the playback volume: please use the `alsamixer` or `amixer` tool. It depends on the audio system on the specific board. Reference https://en.wikipedia.org/wiki/Alsamixer

Note:

RZ/G2E platform does not support 2K and 4K video.

3.2.14 File Play

Play an MP4 file.

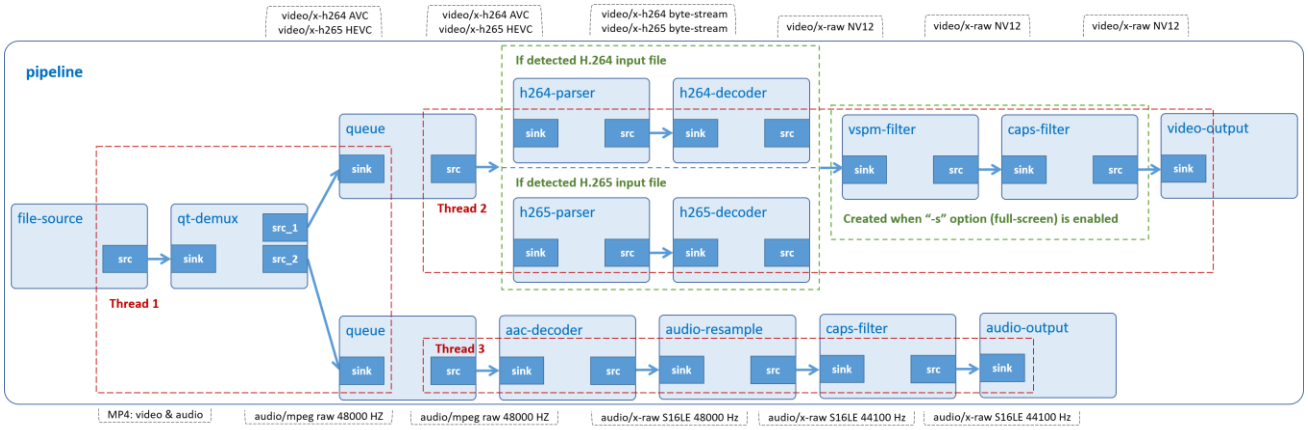


Figure 3.29 File Play pipeline

3.2.14.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <stdio.h>
#include <string.h>
#include <gst/gst.h>
#include <stdlib.h>
#include <stdbool.h>
#include <wayland-client.h>
#include <strings.h>
#include <libgen.h>

#define ARG_PROGRAM_NAME     0
#define ARG_INPUT            1
#define ARG_SCALE            2
#define ARG_COUNT            3
#define INDEX                0
#define AUDIO_SAMPLE_RATE    44100

/* Structure to contain decoder queue information, so we can pass it to callbacks */
typedef struct _CustomData
{
  GstElement *pipeline;
  GstElement *video_queue;
  GstElement *video_parser;
  GstElement *video_decoder;
  GstElement *filter;
  GstElement *video_capsfilter;
  GstElement *video_sink;
  GstElement *audio_queue;
  bool fullscreen;
  struct screen_t *main_screen;
} CustomData;

static void
on_pad_added (GstElement * element, GstPad * pad, gpointer data)
{
  CustomData *user_data = (CustomData *) data;
  GstPad *sinkpad;
```

```
GstCaps *new_pad_caps = NULL;
GstStructure *new_pad_struct = NULL;
const gchar *new_pad_type = NULL;
GstCaps *caps;

new_pad_caps = gst_pad_query_caps (pad, NULL);
new_pad_struct = gst_caps_get_structure (new_pad_caps, INDEX);
new_pad_type = gst_structure_get_name (new_pad_struct);
/* NOTE:
   gst_pad_query_caps: increase the ref count, need to be unref late.
   gst_caps_get_structure: no need to free or unref, it belongs to the GstCaps.
   gst_structure_get_name: no need to free or unref.
 */

g_print ("Received new pad '%s' from '%s': %s\n", GST_PAD_NAME (pad),
    GST_ELEMENT_NAME (element), new_pad_type);

if (g_str_has_prefix (new_pad_type, "audio")) {
  /* In case link this pad with the AAC-decoder sink pad */
  sinkpad = gst_element_get_static_pad (user_data->audio_queue, "sink");
  gst_pad_link (pad, sinkpad);
  gst_object_unref (sinkpad);
  g_print ("Audio pad linked!\n");
} else if (g_str_has_prefix (new_pad_type, "video")) {
  if (g_str_has_prefix (new_pad_type, "video/x-h264")) {
    user_data->video_parser =
        gst_element_factory_make ("h264parse", "h264-parser");
    user_data->video_decoder =
        gst_element_factory_make ("omxh264dec", "omxh264-decoder");
  } else if (g_str_has_prefix (new_pad_type, "video/x-h265")) {
    user_data->video_parser =
        gst_element_factory_make ("h265parse", "h265-parser");
    user_data->video_decoder =
        gst_element_factory_make ("omxh265dec", "omxh265-decoder");
  } else {
    g_print ("Unsupported video format\n");
  }

  if (!user_data->video_parser || !user_data->video_decoder) {
    g_printerr ("One element could not be created. Exiting.\n");
  }

  gst_bin_add_many (GST_BIN (user_data->pipeline),
    user_data->video_parser, user_data->video_decoder, NULL);

  /* Need to set Gst State to PAUSED before change state from NULL to PLAYING */
  gst_element_set_state (user_data->video_parser, GST_STATE_PAUSED);
  gst_element_set_state (user_data->video_decoder, GST_STATE_PAUSED);

  if (!user_data->fullscreen) {
    /* Link the elements together:
    - video-queue -> parser -> decoder -> video-output
    */
    if (gst_element_link_many (user_data->video_queue, user_data->video_parser,
           user_data->video_decoder, user_data->video_sink, NULL) != TRUE) {
       g_printerr ("Video elements could not be linked.\n");
    }
  } else {
    user_data->filter = gst_element_factory_make ("vspmfilter", "vspm-filter");
    user_data->video_capsfilter = gst_element_factory_make ("capsfilter", "video-capsfilter");

    if (!user_data->filter || !user_data->video_capsfilter) {
       g_printerr ("One element could not be created. Exiting.\n");
    }

    /* Need to set Gst State to PAUSED before change state from NULL to PLAYING */
    gst_element_set_state (user_data->filter, GST_STATE_PAUSED);
    gst_element_set_state (user_data->video_capsfilter, GST_STATE_PAUSED);

    /* Set property "dmabuf-use" of vspmfilter to true */
    /* Without it, waylandsink will display broken video */
    g_object_set (G_OBJECT (user_data->filter), "dmabuf-use", TRUE, NULL);

    /* Create simple cap which contains video's resolution */
    caps = gst_caps_new_simple ("video/x-raw",
```

```
          "width", G_TYPE_INT, user_data->main_screen->width,
          "height", G_TYPE_INT, user_data->main_screen->height, NULL);

      /* Add cap to capsfilter element */
      g_object_set (G_OBJECT (user_data->video_capsfilter), "caps", caps, NULL);
      gst_caps_unref (caps);

      /* Add filter, capsfilter into the pipeline */
      gst_bin_add_many (GST_BIN (user_data->pipeline), user_data->filter, user_data->video_capsfilter,
NULL);

      /* Link the elements together:
        - video-queue -> parser -> decoder -> video-filter -> capsfilter -> video-output
      */

      if (gst_element_link_many (user_data->video_queue, user_data->video_parser,
user_data->video_decoder, user_data->filter,
              user_data->video_capsfilter, user_data->video_sink, NULL) != TRUE) {
        g_printerr ("Video elements could not be linked.\n");
      }
    }

    /* In case link this pad with the decoder sink pad */
    sinkpad = gst_element_get_static_pad (user_data->video_queue, "sink");
    gst_pad_link (pad, sinkpad);
    gst_object_unref (sinkpad);
    g_print ("Video pad linked!\n");
  } else {
    g_printerr ("Unexpected pad received.\n");
  }

  if (new_pad_caps != NULL) {
    gst_caps_unref (new_pad_caps);
  }
}

int
main (int argc, char *argv[])
{
  struct wayland_t *wayland_handler = NULL;
  struct screen_t *main_screen = NULL;

  GstElement *pipeline, *source, *demuxer;
  GstElement *video_queue, *video_sink;
  GstElement *audio_queue, *audio_decoder, *audio_resample,
            *audio_capsfilter, *audio_sink;
  CustomData user_data;
  GstCaps *caps;
  GstBus *bus;
  GstMessage *msg;
  bool fullscreen = false;
  const char* ext;
  char* file_name;

  const gchar *input_file = argv[ARG_INPUT];
  if ((argc > ARG_COUNT) || (argc == 1) || ((argc == ARG_COUNT) && (strcmp (argv[ARG_SCALE], "-s"))))
{
    g_print ("Error: Invalid arugments.\n");
    g_print ("Usage: %s <path to MP4 file> [-s]\n", argv[ARG_PROGRAM_NAME]);
    return -1;
  }

  /* Check full-screen option */
  if (argc == ARG_COUNT) {
      fullscreen = true;
  }

  if (!is_file_exist(input_file))
  {
    g_printerr("Cannot find input file: %s. Exiting.\n", input_file);
    return -1;
   }

  file_name = basename ((char*) input_file);
```

```
  ext = get_filename_ext (file_name);
  if (strcasecmp ("mp4", ext) != 0) {
    g_print ("Unsupported video type. MP4 format is required\n");
    return -1;
  }

  /* Get a list of available screen */
  wayland_handler = get_available_screens();

  /* Get main screen */
  main_screen = get_main_screen(wayland_handler);
  if (main_screen == NULL)
  {
    g_printerr("Cannot find any available screens. Exiting.\n");

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Initialisation */
  gst_init (&argc, &argv);

  /* Create gstreamer elements */
  pipeline = gst_pipeline_new ("file-play");
  source = gst_element_factory_make ("filesrc", "file-source");
  demuxer = gst_element_factory_make ("qtdemux", "qt-demuxer");
  /* elements for Video thread */
  video_queue = gst_element_factory_make ("queue", "video-queue");
  video_sink = gst_element_factory_make ("waylandsink", "video-output");
  /* elements for Audio thread */
  audio_queue = gst_element_factory_make ("queue", "audio-queue");
  audio_decoder = gst_element_factory_make ("faad", "aac-decoder");
  audio_resample = gst_element_factory_make("audioresample", "audio-resample");
  audio_capsfilter = gst_element_factory_make ("capsfilter", "audio-capsfilter");
  audio_sink = gst_element_factory_make ("alsasink", "audio-output");

  if (!pipeline || !source || !demuxer || !video_queue || !video_sink
      || !audio_queue || !audio_decoder || !audio_resample
      || !audio_capsfilter || !audio_sink) {
    g_printerr ("One element could not be created. Exiting.\n");

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Set up the pipeline */

  /* Set the input filename to the source element */
  g_object_set (G_OBJECT (source), "location", input_file, NULL);

  /* Set position for displaying (0, 0) */
  g_object_set (G_OBJECT (video_sink), "position-x", main_screen->x, "position-y", main_screen->y,
NULL);

  /* Create simple cap which contains audio's sample rate */
  caps = gst_caps_new_simple ("audio/x-raw",
      "rate", G_TYPE_INT, AUDIO_SAMPLE_RATE, NULL);

  /* Add cap to capsfilter element */
  g_object_set (G_OBJECT (audio_capsfilter), "caps", caps, NULL);
  gst_caps_unref (caps);

  /* Add elements into the pipeline:
     file-source | qt-demuxer
     <1> | video-queue | omxh264-decoder | video-output
     <2> | audio-queue | aac-decoder | audio-resample | capsfilter | audio-output
   */
  gst_bin_add_many (GST_BIN (pipeline), source, demuxer, video_queue, video_sink,
      audio_queue, audio_decoder, audio_resample, audio_capsfilter,
      audio_sink, NULL);

  /* Link the elements together:
     - file-source -> qt-demuxer
  */
```

```
if (gst_element_link (source, demuxer) != TRUE) {
  g_printerr ("Source and demuxer could not be linked.\n");
  gst_object_unref (pipeline);

  destroy_wayland(wayland_handler);
  return -1;
}

/* Link the elements together:
   - audio-queue -> aac-decoder -> audio-resample -> capsfilter -> audio-output
*/
if (gst_element_link_many (audio_queue, audio_decoder, audio_resample,
        audio_capsfilter, audio_sink, NULL) != TRUE) {
  g_printerr ("Audio elements could not be linked.\n");
  gst_object_unref (pipeline);

  destroy_wayland(wayland_handler);
  return -1;
}

user_data.main_screen = main_screen;
user_data.fullscreen = fullscreen;
user_data.pipeline = pipeline;
user_data.video_sink = video_sink;
user_data.audio_queue = audio_queue;
user_data.video_queue = video_queue;
g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added),
    &user_data);

/* note that the demuxer will be linked to the decoder dynamically.
   The reason is that Qt may contain various streams (for example
   video, audio and subtitle). The source pad(s) will be created at run time,
   by the demuxer when it detects the amount and nature of streams.
   Therefore we connect a callback function which will be executed
   when the "pad-added" is emitted. */

/* set the piline to "playing" state */
g_print ("Now playing: %s\n", input_file);
if (gst_element_set_state (pipeline,
        GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
  g_printerr ("Unable to set the pipeline to the playing state.\n");
  gst_object_unref (pipeline);

  destroy_wayland(wayland_handler);
  return -1;
}
/* Iterate */
g_print ("Running...\n");
bus = gst_element_get_bus (pipeline);
msg =
    gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
    GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
gst_object_unref (bus);

/* Note that because input timeout is GST_CLOCK_TIME_NONE,
   the gst_bus_timed_pop_filtered() function will block forever untill a
   matching message was posted on the bus (GST_MESSAGE_ERROR or
   GST_MESSAGE_EOS). */

/* Playback end. Clean up nicely */
if (msg != NULL) {
  GError *err;
  gchar *debug_info;

  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_ERROR:
      gst_message_parse_error (msg, &err, &debug_info);
      g_printerr ("Error received from element %s: %s.\n",
          GST_OBJECT_NAME (msg->src), err->message);
      g_printerr ("Debugging information: %s.\n",
          debug_info ? debug_info : "none");
      g_clear_error (&err);
      g_free (debug_info);
      break;
```

```
      case GST_MESSAGE_EOS:
        g_print ("End-Of-Stream reached.\n");
        break;
      default:
        /* We should not reach here because we only asked for ERRORs and EOS */
        g_printerr ("Unexpected message received.\n");
        break;
    }
    gst_message_unref (msg);
  }

  /* Clean up "wayland_t" structure */
  destroy_wayland(wayland_handler);

  /* Clean up nicely */
  g_print ("Returned, stopping playback\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);

  g_print ("Deleting pipeline\n");
  gst_object_unref (GST_OBJECT (pipeline));

  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.2 Video Play and 3.2.1 Audio Play.

Command-line argument

```
if ((argc > ARG_COUNT) || (argc == 1) || ((argc == ARG_COUNT) && (strcmp (argv[ARG_SCALE], "-s"))))
{
    g_print ("Error: Invalid arugments.\n");
    g_print ("Usage: %s <path to MP4 file> [-s]\n", argv[ARG_PROGRAM_NAME]);
    return -1;
}
```

This application accepts a command-line argument which points to an MP4 file.
If "-s" option is enabled, the video will be scaled to full-screen.

CustomData structure

```
typedef struct _CustomData
{
  GstElement *pipeline;
  GstElement *video_queue;
  GstElement *video_parser;
  GstElement *video_decoder;
  GstElement *filter;
  GstElement *video_capsfilter;
  GstElement *video_sink;
  GstElement *audio_queue;
  bool fullscreen;
  struct screen_t *main_screen;
} CustomData;
```

This structure contains:

- Variable `pipeline` (`GstElement`): A GStreamer pipeline which contains connected video elements.
- Variable `audio_queue` and `video_queue` (`GstElement`): A GStreamer element to queue data until one of the limits specified by the `max-size-buffers`, `max-size-bytes`, and/or `max-size-time` properties has been reached. Any attempt to push more buffers into the queue will block the pushing thread until more space becomes available.
- Variable `video_parser` (`GstElement`): A GStreamer element to parse H.264/H.265 stream to format which `video_decoder` can recognize and process.
- Variable `video_decoder` (`GstElement`): A GStreamer element to decompress H.264/H.265 stream to raw

NV12-formatted video.

- Variable `filter` (`GstElement`): A GStreamer element to handle video scaling.
- Variable `video_capfilter` (GstElement): A GStreamer element to contain screen resolution.
- Variable `video_sink` (`GstElement`): A GStreamer element to create its own window and renders the decoded video frames to that.
- Variable `fullscreen` (`bool`): A boolean variable to enable full-screen mode.
- Variable `main_screen` (`screen_t`): A pointer to screen_t structure to contain monitor information, such as: (x, y), width, and height.

Create elements

```
source = gst_element_factory_make ("filesrc", "file-source");
demuxer = gst_element_factory_make ("qtdemux", "qt-demuxer");
video_queue = gst_element_factory_make ("queue", "video-queue");
video_sink = gst_element_factory_make ("waylandsink", "video-output");

audio_queue = gst_element_factory_make ("queue", "audio-queue");
audio_decoder = gst_element_factory_make ("faad", "aac-decoder");
audio_resample = gst_element_factory_make("audioresample", "audio-resample");
audio_capsfilter = gst_element_factory_make ("capsfilter", "audio-capsfilter");
audio_sink = gst_element_factory_make ("alsasink", "audio-output");

static void on_pad_added (GstElement * element, GstPad * pad, gpointer data) {
  if (g_str_has_prefix (new_pad_type, "video/x-h264")) {
    user_data->video_parser =
         gst_element_factory_make ("h264parse", "h264-parser");
    user_data->video_decoder =
         gst_element_factory_make ("omxh264dec", "omxh264-decoder");
  } else if (g_str_has_prefix (new_pad_type, "video/x-h265")) {
    user_data->video_parser =
         gst_element_factory_make ("h265parse", "h265-parser");
    user_data->video_decoder =
         gst_element_factory_make ("omxh265dec", "omxh265-decoder");
  }

  user_data->filter = gst_element_factory_make ("vspmfilter", "vspm-filter");
  user_data->video_capsfilter = gst_element_factory_make ("capsfilter", "video-capsfilter");
}
```

To play an MP4 video, the following elements are needed:

- Element `filesrc` reads data from a local file.
- Element `qtdemux` de-multiplexes an MP4 file into audio and video stream.
- Element `queue` (`audio_queue` and `video_queue`) queues data until one of the limits specified by the `max-size-buffers`, `max-size-bytes`, and/or `max-size-time` properties has been reached. Any attempt to push more buffers into the queue will block the pushing thread until more space becomes available.
- Element `h264parse` parses H.264 stream to format which `omxh264dec` can recognize and process.
- Element `h265parse` parses H.265 stream to format which `omxh265dec` can recognize and process.
- Element `omxh264dec` decompresses H.264 stream to raw NV12-formatted video.
- Element `omxh265dec` decompresses H.265 stream to raw NV12-formatted video.
- Element `waylandsink` creates its own window and renders the decoded video frames to that.
- Element `faad` decompresses MPEG-2/4 AAC stream to raw S16LE-formatted audio.
- Element `audioresample` resamples raw audio buffers to different sample rates using a configurable windowing function to enhance quality.
- Element `capsfilter` (`audio_capfilter`) contains target sample rate 44100 Hz so that `audioresample` can resample audio based on this value.
- Element `alsasink` renders audio samples using the ALSA audio API.
- Element `vspmfilter` handles video scaling.
- Element `capsfilter` contains screen resolution so that `vspmfilter` can scale video frames based on this value.

Set element's properties

```
g_object_set (G_OBJECT (source), "location", input_file, NULL);
g_object_set (G_OBJECT (filter), "dmabuf-use", TRUE, NULL);
g_object_set(G_OBJECT(video_sink),"position-x",main_screen->x,"position-y",main_screen->y,NULL);

caps = gst_caps_new_simple ("video/x-raw", "width", G_TYPE_INT, main_screen->width,
                                           "height", G_TYPE_INT, main_screen->height, NULL);

g_object_set (G_OBJECT (video_capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);
caps = gst_caps_new_simple ("audio/x-raw", "rate", G_TYPE_INT, AUDIO_SAMPLE_RATE, NULL);
g_object_set (G_OBJECT (audio_capsfilter), "caps", caps, NULL);
gst_caps_unref (caps);
```

The `g_object_set()` function is used to set some element's properties, such as:

- The `location` property of `filesrc` element which points to an MP4 file.
- The `dmabuf-use` property of `vspmfilter` element which is set to `true`. This disallows `dmabuf` to be output buffer. If it is not set, `waylandsink` will display broken video frames.
- The `position-x` and `position-y` properties of `waylandsink` element which point to (x, y) coordinate of Wayland desktop.
- The `caps` property of `capsfilter` (`video_capsfilter`) element which specifies output video resolution.
- The `caps` property of `capsfilter` (`audio_capsfilter`) element which specifies output audio sample rate 44100 Hz.

Link elements

```
gst_element_link (source, demuxer);
gst_element_link_many (audio_queue, audio_decoder, audio_resample,
                       audio_capsfilter, audio_sink, NULL);

/*Not display video in full-screen*/
gst_element_link_many (user_data->video_queue, user_data->video_parser,
            user_data->video_decoder, user_data->video_sink, NULL);

/*Display video in full-screen*/
gst_element_link_many (user_data->video_queue, user_data->video_parser, user_data->video_decoder,
            user_data->filter, user_data->video_capsfilter, user_data->video_sink, NULL);
```

Because `demuxer` (`qtdemux`) contains no source pads at this point, this element cannot link to either `video_queue` (`queue`) or `audio_queue` (`queue`) until its `pad-added` signal is emitted (see below).

If "-s" option is enabled, the application will add and link GStreamer elements from `video_queue`, `video_parser`, `video_decoder`, `filter`, and `video_capsfilter` to `video_sink`. Otherwise, it does not consider `filter` and `video_capsfilter` elements.

Note that the order counts, because links must follow the data flow (this is, from source elements to sink elements).

Signal

```
CustomData user_data;
user_data.audio_queue = audio_queue;
user_data.video_queue = video_queue;
g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added), &user_data);
```

Signals are a crucial point in GStreamer. They allow you to be notified (by means of a callback) when something interesting has happened. Signals are identified by a name, and each element has its own signals.

In this application, `g_signal_connect()` is used to bind `pad-added` signal of `qtdemux` (`demuxer`) to callback function `on_pad_added()` and `user_data` pointer. GStreamer does nothing with this pointer, it just forwards it to the callback.

Link qtdemux to audio_queue and video_queue

When `qtdemux` (`demuxer`) element finally has enough information to start producing data, it will create source

pads, and trigger the pad-added signal. At this point our callback will be called:

```
static void on_pad_added (GstElement * element, GstPad * pad, gpointer data)
```

The element parameter is the GstElement which triggered the signal. In this application, it is qtdemux. The first parameter of a signal handler is always the object that has triggered it.

The pad element is the GstPad that has just been added to the qtdemux element. This is usually the pad to which we want to link.

The data element is the user_data pointer which we provided earlier when attaching to the signal.

```
GstCaps *new_pad_caps = NULL;
GstStructure *new_pad_struct = NULL;
const gchar *new_pad_type = NULL;
new_pad_caps = gst_pad_query_caps (pad, NULL);
new_pad_struct = gst_caps_get_structure (new_pad_caps, INDEX);
new_pad_type = gst_structure_get_name (new_pad_struct);
```

The gst_pad_query_caps() function gets the current capabilities of the pad (that is, the kind of data it currently outputs), wrapped in a GstCaps structure. A pad can offer many capabilities, and hence GstCaps can contain many GstStructure, each representing a different capability. The current caps on a pad will always have a single GstStructure and represent a single media format, or if there are no current caps yet, NULL will be returned.

This application retrieves the first GstStructure with gst_caps_get_structure().

Finally, gst_structure_get_name() recovers the name of the structure, which contains the main description of the format (video/x-h264 or audio/mpeg, for example).

```
if (g_str_has_prefix (new_pad_type, "audio")) {

  sinkpad = gst_element_get_static_pad (user_data->audio_queue, "sink");
  gst_pad_link (pad, sinkpad);

  gst_object_unref (sinkpad);
  g_print ("Audio pad linked!\n");

} else if (g_str_has_prefix (new_pad_type, "video")) {

  sinkpad = gst_element_get_static_pad (user_data->video_queue, "sink");
  gst_pad_link (pad, sinkpad);

  gst_object_unref (sinkpad);
  g_print ("Video pad linked!\n");
}
```

If the name contains audio, the application retrieves the sink pad of audio_queue (queue) using gst_element_get_static_pad(), then uses gst_pad_link() to connect it to the source pad of qtdemux.

The procedure is the same if the name contains video.

Note that sinkpad should be freed with gst_caps_unref() if it is not used anymore.

```
  if (new_pad_caps != NULL) {
    gst_caps_unref (new_pad_caps);
  }
```

This code block un-refers new_pad_caps if it exists and is not used anymore.

3.2.14.2 Result

Can play MP4 file.

Figure 3.30 File Play result

3.2.14.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.14.4 Special Instruction

- Download the input file at: https://download.blender.org/durian/trailer/sintel_trailer-720p.mp4
- To set the playback volume, please use the `alsamixer` or `amixer` tool. It depends on the audio system on the specific board. Reference https://en.wikipedia.org/wiki/Alsamixer

Note:

RZ/G2E platform does not support 2K and 4K video.

3.2.15 Multiple Display 1

Display 1 H.264/H.265 video simultaneously on HDMI and LCD monitors.

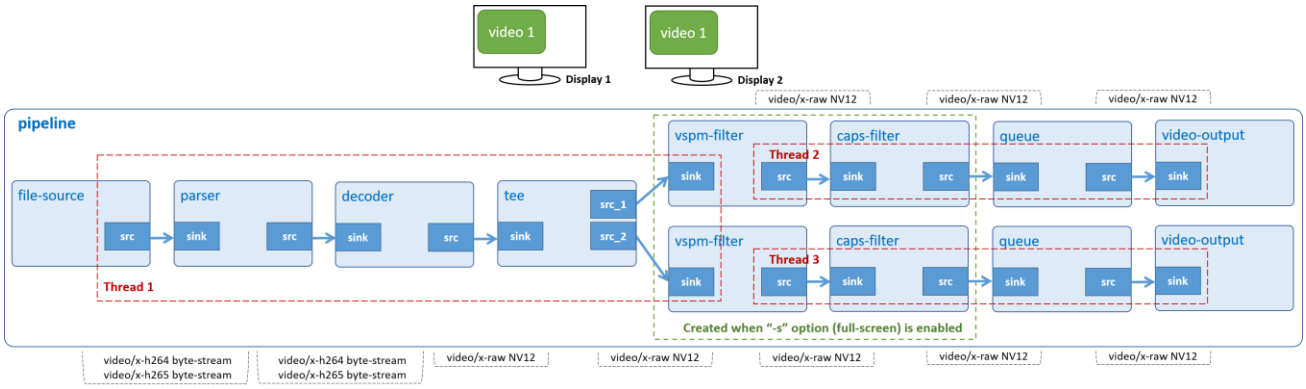


Figure 3.31 Multiple Display 1 pipeline

3.2.15.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <stdio.h>
#include <string.h>
#include <gst/gst.h>
#include <stdlib.h>
#include <stdbool.h>
#include <wayland-client.h>
#include <strings.h>
#include <libgen.h>

#define ARG_PROGRAM_NAME     0
#define ARG_INPUT            1
#define ARG_SCALE            2
#define ARG_COUNT            3
#define REQUIRED_SCREEN_NUMBERS 2
#define PRIMARY_SCREEN_INDEX 0
#define SECONDARY_SCREEN_INDEX 1

int
main (int argc, char *argv[])
{
  struct wayland_t *wayland_handler = NULL;
  struct screen_t *screens[REQUIRED_SCREEN_NUMBERS];
  int screen_numbers = 0;
  bool fullscreen = false;
  const char* ext;
  char* file_name;

  const char *input_video_file = argv[ARG_INPUT];

  GstElement *pipeline, *source, *parser, *decoder, *tee;
  GstElement *filter_1, *capsfilter_1, *queue_1, *video_sink_1;
  GstElement *filter_2, *capsfilter_2, *queue_2, *video_sink_2;

  GstCaps *caps_1, *caps_2;
  GstPad *req_pad_1, *sink_pad, *req_pad_2;
  GstBus *bus;
  GstMessage *msg;
```

```
  GstPadTemplate *tee_src_pad_template;

  if ((argc > ARG_COUNT) || (argc == 1) || ((argc == ARG_COUNT) && (strcmp (argv[ARG_SCALE], "-s"))))
{
    g_print ("Error: Invalid arugments.\n");
    g_print ("Usage: %s <path to H264 file> [-s]\n", argv[ARG_PROGRAM_NAME]);
    return -1;
  }

  /* Check full-screen option */
  if (argc == ARG_COUNT) {
    fullscreen = true;
  }

  file_name = basename ((char*) input_video_file);
  ext = get_filename_ext (file_name);

  /* Get a list of available screen */
  wayland_handler = get_available_screens();
  if (wayland_handler == NULL)
  {
    g_printerr("Cannot detect monitors. Exiting.\n");
    return -1;
  }

  screen_numbers = wl_list_length(&(wayland_handler->screens));
  if (screen_numbers < REQUIRED_SCREEN_NUMBERS)
  {
    g_printerr("Detected %d monitors.\n", screen_numbers);
    g_printerr("Must have at least %d monitors to run the app. Exiting.\n", REQUIRED_SCREEN_NUMBERS);

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Check input file */
  if (!is_file_exist(input_video_file))
  {
    g_printerr("Cannot find input file: %s. Exiting.\n", input_video_file);

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Extract required monitors */
  get_required_monitors(wayland_handler, screens, REQUIRED_SCREEN_NUMBERS);

  /* Initialization */
  gst_init (&argc, &argv);

  /* Check the extension and create parser, decoder */
  if (strcasecmp ("h264", ext) == 0) {
    parser = gst_element_factory_make ("h264parse", "h264-parser");
    decoder = gst_element_factory_make ("omxh264dec", "h264-decoder");
  } else if (strcasecmp ("h265", ext) == 0) {
    parser = gst_element_factory_make ("h265parse", "h265-parser");
    decoder = gst_element_factory_make ("omxh265dec", "h265-decoder");
  } else {
    g_print ("Unsupported video type. H264/H265 format is required.\n");
    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Create GStreamer elements */
  pipeline = gst_pipeline_new ("multiple-display");
  source = gst_element_factory_make ("filesrc", "file-source");
  tee = gst_element_factory_make ("tee", "tee-element");

  /* Elements for Video Display 1 */
  queue_1 = gst_element_factory_make ("queue", "queue-1");
  video_sink_1 = gst_element_factory_make ("waylandsink", "video-output-1");

  /* Elements for Video Display 2 */
  queue_2 = gst_element_factory_make ("queue", "queue-2");
  video_sink_2 = gst_element_factory_make ("waylandsink", "video-output-2");
```

```
if (!pipeline || !source || !parser || !decoder || !tee
    || !queue_1 || !video_sink_1 || !queue_2 || !video_sink_2) {
  g_printerr ("One element could not be created. Exiting.\n");

  destroy_wayland(wayland_handler);
  return -1;
}

/* Set up the pipeline */

/* Set the input file location to the source element */
g_object_set (G_OBJECT (source), "location", input_video_file, NULL);

/* Set display position and size for Display 1  */
g_object_set (G_OBJECT (video_sink_1),
     "position-x", screens[PRIMARY_SCREEN_INDEX]->x,
     "position-y", screens[PRIMARY_SCREEN_INDEX]->y, NULL);

/* Set display position and size for Display 2 */
g_object_set (G_OBJECT (video_sink_2),
     "position-x", screens[SECONDARY_SCREEN_INDEX]->x,
     "position-y", screens[SECONDARY_SCREEN_INDEX]->y, NULL);

/* Add all elements into the pipeline */
gst_bin_add_many (GST_BIN (pipeline), source, parser, decoder, tee,
    queue_1, video_sink_1, queue_2, video_sink_2, NULL);

/* Link elements together */
if (gst_element_link_many (source, parser, decoder, tee, NULL) != TRUE) {
  g_printerr ("Source elements could not be linked.\n");
  gst_object_unref (pipeline);

  destroy_wayland(wayland_handler);
  return -1;
}

if (!fullscreen) {
  if (gst_element_link_many (queue_1, video_sink_1, NULL) != TRUE) {
    g_printerr ("Elements of Video Display-1 could not be linked.\n");
    gst_object_unref (pipeline);

    destroy_wayland(wayland_handler);
    return -1;
  }
  if (gst_element_link_many (queue_2, video_sink_2, NULL) != TRUE) {
    g_printerr ("Elements of Video Display-2 could not be linked.\n");
    gst_object_unref (pipeline);

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Get a src pad template of Tee */
  tee_src_pad_template =
     gst_element_class_get_pad_template (GST_ELEMENT_GET_CLASS (tee),
     "src_%u");

  /* Get request pad and manually link for Video Display 1 */
  req_pad_1 = gst_element_request_pad (tee, tee_src_pad_template, NULL, NULL);
  sink_pad = gst_element_get_static_pad (queue_1, "sink");
  if (gst_pad_link (req_pad_1, sink_pad) != GST_PAD_LINK_OK) {
    g_print ("tee link failed!\n");
  }
  gst_object_unref (sink_pad);

  /* Get request pad and manually link for Video Display 2 */
  req_pad_2 = gst_element_request_pad (tee, tee_src_pad_template, NULL, NULL);
  sink_pad = gst_element_get_static_pad (queue_2, "sink");
  if (gst_pad_link (req_pad_2, sink_pad) != GST_PAD_LINK_OK) {
    g_print ("tee link failed!\n");
  }
  gst_object_unref (sink_pad);
} else {
  filter_1 = gst_element_factory_make ("vspmfilter", "vspm-filter-1");
```

```
    capsfilter_1 = gst_element_factory_make ("capsfilter", "caps-filter-1");
    filter_2 = gst_element_factory_make ("vspmfilter", "vspm-filter-2");
    capsfilter_2 = gst_element_factory_make ("capsfilter", "caps-filter-2");

    if (!filter_1 || !capsfilter_1 || !filter_2 || !capsfilter_2) {
    g_printerr ("One element could not be created. Exiting.\n");
    gst_object_unref (pipeline);
    destroy_wayland(wayland_handler);
    return -1;
    }

    /* Set property "dmabuf-use" of vspmfilter to true */
    /* Without it, waylandsink will display broken video */
    g_object_set (G_OBJECT (filter_1), "dmabuf-use", TRUE, NULL);
    g_object_set (G_OBJECT (filter_2), "dmabuf-use", TRUE, NULL);

    /* Create simple cap which contains video's resolutions */
    caps_1 = gst_caps_new_simple ("video/x-raw",
        "width", G_TYPE_INT, screens[PRIMARY_SCREEN_INDEX]->width,
        "height", G_TYPE_INT, screens[PRIMARY_SCREEN_INDEX]->height, NULL);

    caps_2 = gst_caps_new_simple ("video/x-raw",
        "width", G_TYPE_INT, screens[SECONDARY_SCREEN_INDEX]->width,
        "height", G_TYPE_INT, screens[SECONDARY_SCREEN_INDEX]->height, NULL);

    /* Add caps_1 to capsfilter_1 element */
    g_object_set (G_OBJECT (capsfilter_1), "caps", caps_1, NULL);
    gst_caps_unref (caps_1);

    /* Add caps_2 to capsfilter_2 element */
    g_object_set (G_OBJECT (capsfilter_2), "caps", caps_2, NULL);
    gst_caps_unref (caps_2);

    /* Add filter, capsfilter into the pipeline */
    gst_bin_add_many (GST_BIN (pipeline), filter_1, capsfilter_1, filter_2, capsfilter_2, NULL);

    if (gst_element_link_many (filter_1, capsfilter_1, queue_1, video_sink_1,
            NULL) != TRUE) {
      g_printerr ("Elements of Video Display-1 could not be linked.\n");
      gst_object_unref (pipeline);
      destroy_wayland(wayland_handler);
      return -1;
    }
    if (gst_element_link_many (filter_2, capsfilter_2, queue_2, video_sink_2,
            NULL) != TRUE) {
      g_printerr ("Elements of Video Display-2 could not be linked.\n");
      gst_object_unref (pipeline);
      destroy_wayland(wayland_handler);
      return -1;
    }

    /* Get a src pad template of Tee */
    tee_src_pad_template =
        gst_element_class_get_pad_template (GST_ELEMENT_GET_CLASS (tee),
        "src_%u");

    /* Get request pad and manually link for Video Display 1 */
    req_pad_1 = gst_element_request_pad (tee, tee_src_pad_template, NULL, NULL);
    sink_pad = gst_element_get_static_pad (filter_1, "sink");
    if (gst_pad_link (req_pad_1, sink_pad) != GST_PAD_LINK_OK) {
      g_print ("tee link failed!\n");
    }
    gst_object_unref (sink_pad);

    /* Get request pad and manually link for Video Display 2 */
    req_pad_2 = gst_element_request_pad (tee, tee_src_pad_template, NULL, NULL);
    sink_pad = gst_element_get_static_pad (filter_2, "sink");
    if (gst_pad_link (req_pad_2, sink_pad) != GST_PAD_LINK_OK) {
      g_print ("tee link failed!\n");
    }
    gst_object_unref (sink_pad);
  }
  /* Set the pipeline to "playing" state */
  g_print ("Now playing:\n");
```

```
  if (gst_element_set_state (pipeline,
        GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
    g_printerr ("Unable to set the pipeline to the playing state.\n");
    gst_object_unref (pipeline);

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Iterate */
  g_print ("Running...\n");
  bus = gst_element_get_bus (pipeline);
  msg =
      gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
      GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
  gst_object_unref (bus);

  /* Note that because input timeout is GST_CLOCK_TIME_NONE,
     the gst_bus_timed_pop_filtered() function will block forever until a
     matching message was posted on the bus (GST_MESSAGE_ERROR or
     GST_MESSAGE_EOS). */

  /* Playback end. Handle the message */
  if (msg != NULL) {
    GError *err;
    gchar *debug_info;

    switch (GST_MESSAGE_TYPE (msg)) {
      case GST_MESSAGE_ERROR:
        gst_message_parse_error (msg, &err, &debug_info);
        g_printerr ("Error received from element %s: %s.\n",
            GST_OBJECT_NAME (msg->src), err->message);
        g_printerr ("Debugging information: %s.\n",
            debug_info ? debug_info : "none");
        g_clear_error (&err);
        g_free (debug_info);
        break;
      case GST_MESSAGE_EOS:
        g_print ("End-Of-Stream reached.\n");
        break;
      default:
        /* We should not reach here because we only asked for ERRORs and EOS */
        g_printerr ("Unexpected message received.\n");
        break;
    }
    gst_message_unref (msg);
  }

  /* Seek to start and flush all old data */
  gst_element_seek_simple (pipeline, GST_FORMAT_TIME, GST_SEEK_FLAG_FLUSH, 0);

  /* Clean up "wayland_t" structure */
  destroy_wayland(wayland_handler);

  /* Clean up nicely */
  gst_element_release_request_pad (tee, req_pad_1);
  gst_element_release_request_pad (tee, req_pad_2);
  gst_object_unref (req_pad_1);
  gst_object_unref (req_pad_2);

  g_print ("Returned, stopping playback\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);

  g_print ("Deleting pipeline\n");
  gst_object_unref (GST_OBJECT (pipeline));

  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.2 Video Play.

Command-line argument

```
if ((argc > ARG_COUNT) || (argc == 1) || ((argc == ARG_COUNT) && (strcmp (argv[ARG_SCALE], "-s"))))
{
    g_print ("Error: Invalid arugments.\n");
    g_print ("Usage: %s <path to H264/H265 file> [-s]\n", argv[ARG_PROGRAM_NAME]);
    return -1;
}
```

This application accepts a command-line argument which points to an H.264/H.265 file.
If "-s" option is enabled, video will be scaled to full-screen.

Create elements

```
source = gst_element_factory_make ("filesrc", "file-source");
tee = gst_element_factory_make ("tee", "tee-element");

queue_1 = gst_element_factory_make ("queue", "queue-1");
video_sink_1 = gst_element_factory_make ("waylandsink", "video-output-1");

queue_2 = gst_element_factory_make ("queue", "queue-2");
video_sink_2 = gst_element_factory_make ("waylandsink", "video-output-2");

if (strcasecmp ("h264", ext) == 0) {
  parser = gst_element_factory_make ("h264parse", "h264-parser");
  decoder = gst_element_factory_make ("omxh264dec", "h264-decoder");
} else if (strcasecmp ("h265", ext) == 0) {
  parser = gst_element_factory_make ("h265parse", "h265-parser");
  decoder = gst_element_factory_make ("omxh265dec", "h265-decoder");
}

filter_1 = gst_element_factory_make ("vspmfilter", "vspm-filter-1");
capsfilter_1 = gst_element_factory_make ("capsfilter", "caps-filter-1");
filter_2 = gst_element_factory_make ("vspmfilter", "vspm-filter-2");
capsfilter_2 = gst_element_factory_make ("capsfilter", "caps-filter-2");
```

To play an H.264/H.265 video on 2 displays, the following elements are used:

- Element `filesrc` reads data from a local file.
- Element `h264parse` parses H.264 stream to format which `omxh264dec` can recognize and process.
- Element `h265parse` parses H.265 stream to format which `omxh265dec` can recognize and process.
- Element `omxh264dec` decompresses H.264 stream to raw NV12-formatted video.
- Element `omxh265dec` decompresses H.265 stream to raw NV12-formatted video.
- Element `tee` splits (video) data to multiple pads.
- Element `queue` (`queue_1` and `queue_2`) queues data until one of the limits specified by the `max-size-buffers`, `max-size-bytes`, and/or `max-size-time` properties has been reached. Any attempt to push more buffers into the queue will block the pushing thread until more space becomes available.
- Element `vspmfilter` (`filter_1` and `filter_2`) handles video scaling.
- Element `capsfilter` (`capsfilter_1` and `capsfilter_2`) contains screen resolution so that `vspmfilter` can scale video frames based on this value.
- Element `waylandsink` (`video_sink_1` and `video_sink_2`) creates its own window and renders the decoded video frames to that.

Set element's properties

```
g_object_set (G_OBJECT (source), "location", input_video_file, NULL);

g_object_set (G_OBJECT (video_sink_1),
    "position-x", screens[PRIMARY_SCREEN_INDEX]->x,
    "position-y", screens[PRIMARY_SCREEN_INDEX]->y, NULL);

g_object_set (G_OBJECT (video_sink_2),
```

```
    "position-x", screens[SECONDARY_SCREEN_INDEX]->x,
    "position-y", screens[SECONDARY_SCREEN_INDEX]->y, NULL);

g_object_set (G_OBJECT (filter_1), "dmabuf-use", TRUE, NULL);
g_object_set (G_OBJECT (filter_2), "dmabuf-use", TRUE, NULL);
```

The `g_object_set()` function is used to set some element's properties, such as:

- The `location` property of `filesrc` element which points to an H.264/H.265 video file.
- The `position-x` and `position-y` property of `video_sink_1 (waylandsink)` which point to the origin coordinate of main Wayland desktop. In this application, it will always be (0, 0).
- The `position-x` and `position-y` property of `video_sink_2 (waylandsink)` which point to the origin coordinate of secondary Wayland desktop. In this application, it depends on the width of main desktop.
- The `dmabuf-use` property of `vspmfilter` element which is set to `true`. This disallows `dmabuf` to be output buffer. If it is not set, `waylandsink` will display broken video frames.

```
caps_1 = gst_caps_new_simple ("video/x-raw",
    "width", G_TYPE_INT, screens[PRIMARY_SCREEN_INDEX]->width,
    "height", G_TYPE_INT, screens[PRIMARY_SCREEN_INDEX]->height, NULL);

caps_2 = gst_caps_new_simple ("video/x-raw",
    "width", G_TYPE_INT, screens[SECONDARY_SCREEN_INDEX]->width,
    "height", G_TYPE_INT, screens[SECONDARY_SCREEN_INDEX]->height, NULL);

g_object_set (G_OBJECT (capsfilter_1), "caps", caps_1, NULL);
gst_caps_unref (caps_1);

g_object_set (G_OBJECT (capsfilter_2), "caps", caps_2, NULL);
gst_caps_unref (caps_2);
```

Capabilities (short: caps) describe the type of data which is streamed between two pads. This data includes raw video format, resolution.

In this application, two caps are required (`gst_caps_new_simple`), one specifies main screen's resolution and the other specifies secondary screen's resolution. These caps are then added to `caps` property of `capsfilter` elements (`g_object_set`) so that `vspmfilter` elements can use these values to resize video frames.

Note that the cap should be freed with `gst_caps_unref()` if it is not used anymore.

Build pipeline

```
gst_bin_add_many (GST_BIN (pipeline), source, parser, decoder, tee,
     queue_1, video_sink_1, queue_2, video_sink_2, NULL);

gst_element_link_many (source, parser, decoder, tee, NULL)

/*Not display video in full-screen*/
gst_element_link_many (queue_1, video_sink_1, NULL);
gst_element_link_many (queue_2, video_sink_2, NULL);

/*Display video in full-screen*/
gst_bin_add_many (GST_BIN (pipeline), filter_1, capsfilter_1, filter_2, capsfilter_2, NULL);
gst_element_link_many (filter_1, capsfilter_1, queue_1, video_sink_1, NULL);
gst_element_link_many (filter_2, capsfilter_2, queue_2, video_sink_2, NULL);
```

This code block adds all elements to `pipeline` and then links them into separated groups as below:

- Group #1: `source`, `parser`, `decoder`, and `tee`.
- Group #2: `filter_1`, `capsfilter_1`, `queue_1`, and `video_sink_1`.
- Group #3: `filter_2`, `capsfilter_2`, `queue_2`, and `video_sink_2`.

Note: `filter_1`, `capsfilter_1`, `filter_2` and `capsfilter_2` are optional.

The reason for the separation is that `tee` element contains no initial source pads: they need to be requested manually and then `tee` adds them. That is why these source pads are called `Request Pads`. In this way, an input stream can be

replicated any number of times.

Also, to request (or release) pads in the PLAYING or PAUSED states, you need to take additional cautions (pad blocking) which are not described in this manual. It is safe to request (or release) pads in the NULL or READY states, though.

Link source pad (request pads of tee)

```
tee_src_pad_template = gst_element_class_get_pad_template (GST_ELEMENT_GET_CLASS (tee), "src_%u");

/*Not display video in full-screen*/
/* Get request pad and manually link for Video Display 1 */
req_pad_1 = gst_element_request_pad (tee, tee_src_pad_template, NULL, NULL);

sink_pad = gst_element_get_static_pad (queue_1, "sink");

if (gst_pad_link (req_pad_1, sink_pad) != GST_PAD_LINK_OK) {
  g_print ("tee link failed!\n");
}
gst_object_unref (sink_pad);

/* Get request pad and manually link for Video Display 2 */
req_pad_2 = gst_element_request_pad (tee, tee_src_pad_template, NULL, NULL);

sink_pad = gst_element_get_static_pad (queue_2, "sink");

if (gst_pad_link (req_pad_2, sink_pad) != GST_PAD_LINK_OK) {
  g_print ("tee link failed!\n");
}
gst_object_unref (sink_pad);

/*Display video in full-screen*/
/* Get request pad and manually link for Video Display 1 */
req_pad_1 = gst_element_request_pad (tee, tee_src_pad_template, NULL, NULL);

sink_pad = gst_element_get_static_pad (filter_1, "sink");

if (gst_pad_link (req_pad_1, sink_pad) != GST_PAD_LINK_OK) {
  g_print ("tee link failed!\n");
}
gst_object_unref (sink_pad);

/* Get request pad and manually link for Video Display 2 */
req_pad_2 = gst_element_request_pad (tee, tee_src_pad_template, NULL, NULL);

sink_pad = gst_element_get_static_pad (filter_2, "sink");

if (gst_pad_link (req_pad_2, sink_pad) != GST_PAD_LINK_OK) {
  g_print ("tee link failed!\n");
}
gst_object_unref (sink_pad);
```

To link Request Pads, they need to be obtained by “requesting” them from tee element. Note that it might be able to produce different kinds of Request Pads, so, when requesting them, the desired Pad Template name must be provided. In the documentation for the tee element, we see that it has two pad templates named sink (for its sink pads) and src_%u (for the source pad (Request Pads)). We request two source pads from the tee (for video branches) with gst_element_get_request_pad().

We then obtain the sink pads from queue/vspmfilter elements to which these Request Pads need to be linked using gst_element_get_static_pad(). Finally, we link the pads with gst_pad_link().

Note that the sink pads need to be released with gst_object_unref() if they are not used anymore.

Free tee element

```
gst_element_release_request_pad (tee, req_pad_1);
gst_element_release_request_pad (tee, req_pad_2);
gst_object_unref (req_pad_1);
gst_object_unref (req_pad_2);
```

The `gst_element_release_request_pad()` function releases the pads from tee, but it still needs to be un-referenced (freed) with `gst_object_unref()`.

3.2.15.2 Result

Can display 1 video simultaneously on HDMI and LCD monitors.

Figure 3.32 Multiple Display 1 result

3.2.15.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.15.4 Special Instruction

- Download the input file at:
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga1.h264

Note:

RZ/G2E platform supports playing 2 1920x1080, 30 fps videos simultaneously
RZ/G2H, RZ/G2M and RZ/G2N platform support playing 2 2560x1440, 30 fps videos simultaneously

3.2.16 Multiple Display 2

Display 2 H.264/H.265 videos simultaneously on HDMI and LCD monitors.

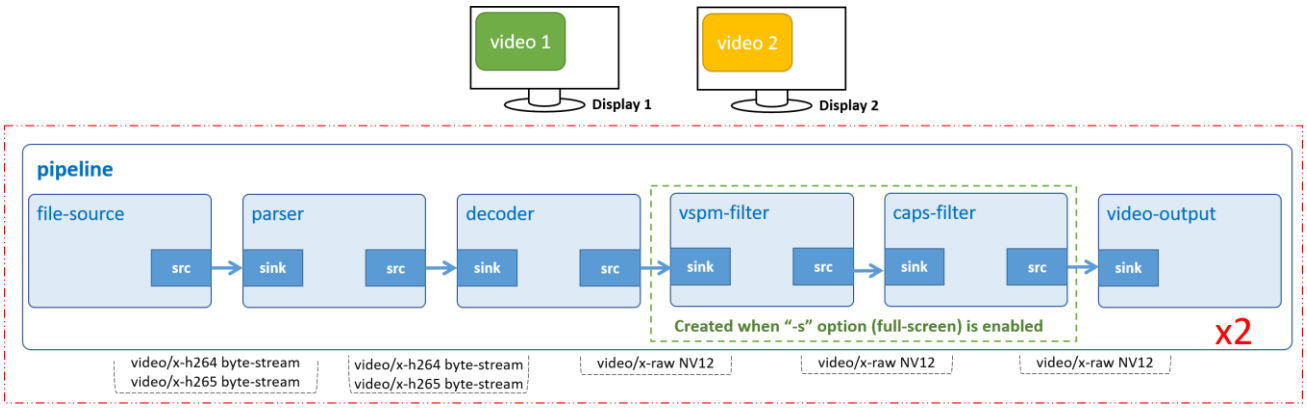


Figure 3.33 Multiple Display 2 pipeline

3.2.16.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <stdio.h>
#include <string.h>
#include <gst/gst.h>
#include <stdlib.h>
#include <stdbool.h>
#include <wayland-client.h>
#include <strings.h>
#include <libgen.h>

#define ARG_PROGRAM_NAME     0
#define ARG_INPUT1           1
#define ARG_INPUT2           2
#define ARG_SCALE            3
#define ARG_COUNT            4
#define REQUIRED_SCREEN_NUMBERS 2
#define PRIMARY_SCREEN_INDEX 0
#define SECONDARY_SCREEN_INDEX 1

typedef struct _CustomData
{
  GMainLoop *loop;
  int loop_reference;
  GMutex mutex;
  const char *video_ext;
  bool fullscreen;
} CustomData;

static void
try_to_quit_loop (CustomData * p_shared_data)
{
  g_mutex_lock (&p_shared_data->mutex);
  --(p_shared_data->loop_reference);
  if (0 == p_shared_data->loop_reference) {
    g_main_loop_quit ((p_shared_data->loop));
  }
  g_mutex_unlock (&p_shared_data->mutex);
}
```

```
static gboolean
bus_call (GstBus * bus, GstMessage * msg, gpointer data)
{
  GstElement *pipeline = (GstElement *) GST_MESSAGE_SRC (msg);
  gchar *pipeline_name = GST_MESSAGE_SRC_NAME (msg);

  switch (GST_MESSAGE_TYPE (msg)) {

    case GST_MESSAGE_EOS:
      g_print ("End-Of-Stream reached.\n");

      g_print ("Stoping %s...\n", pipeline_name);
      gst_element_set_state (pipeline, GST_STATE_NULL);
      g_print ("Deleting %s...\n", pipeline_name);
      gst_object_unref (GST_OBJECT (pipeline));
      try_to_quit_loop ((CustomData *) data);
      break;

    case GST_MESSAGE_ERROR:{
      gchar *debug_info;
      GError *err;
      gst_message_parse_error (msg, &err, &debug_info);
      g_printerr ("Error received from element %s: %s.\n",
          pipeline_name, err->message);
      g_printerr ("Debugging information: %s.\n",
          debug_info ? debug_info : "none");
      g_clear_error (&err);
      g_free (debug_info);

      g_print ("Stoping %s...\n", pipeline_name);
      gst_element_set_state (pipeline, GST_STATE_NULL);
      g_print ("Deleting %s...\n", pipeline_name);
      gst_object_unref (GST_OBJECT (pipeline));
      try_to_quit_loop ((CustomData *) data);
      break;
    }
    default:
      /* Don't care other message */
      break;
  }
  return TRUE;
}

guint
create_video_pipeline (GstElement ** p_video_pipeline, const gchar * input_file,
    struct screen_t * screen, CustomData * data)
{
  GstBus *bus;
  guint video_bus_watch_id;
  GstElement *video_source, *video_parser, *video_decoder, *video_sink;

  /* Create GStreamer elements for video play */
  *p_video_pipeline = gst_pipeline_new (NULL);
  video_source = gst_element_factory_make ("filesrc", NULL);
  video_sink = gst_element_factory_make ("waylandsink", NULL);

  if (strcasecmp ("h264", data->video_ext) == 0) {
    video_parser = gst_element_factory_make ("h264parse", "h264-parser");
    video_decoder = gst_element_factory_make ("omxh264dec", "h264-decoder");
  } else {
    video_parser = gst_element_factory_make ("h265parse", "h265-parser");
    video_decoder = gst_element_factory_make ("omxh265dec", "h265-decoder");
  }

  if (!*p_video_pipeline || !video_source || !video_parser || !video_decoder || !video_sink) {
    g_printerr ("One video element could not be created. Exiting.\n");
    return 0;
  }

  /* Adding a message handler for video pipeline */
  bus = gst_pipeline_get_bus (GST_PIPELINE (*p_video_pipeline));
  video_bus_watch_id = gst_bus_add_watch (bus, bus_call, data);
  gst_object_unref (bus);
  /* Add all elements into the video pipeline */
  /* file-source | parser | decoder | filter | capsfilter | video-output */
```

```
  gst_bin_add_many (GST_BIN (*p_video_pipeline), video_source, video_parser,
      video_decoder, video_sink, NULL);

  /* Set up for the video pipeline */
  /* Set the input file location of the file source element */
  g_object_set (G_OBJECT (video_source), "location", input_file, NULL);
  g_object_set (G_OBJECT (video_sink), "position-x", screen->x, "position-y",  screen->y, NULL);

  if (!data->fullscreen) {
    /* Link the elements together */
    /* file-source -> parser -> decoder -> video-output */
    if (!gst_element_link_many (video_source, video_parser, video_decoder,
          video_sink, NULL)) {
      g_printerr ("Video elements could not be linked.\n");
      gst_object_unref (*p_video_pipeline);
      return 0;
    }
  } else {
    GstElement *filter, *capsfilter;
    GstCaps *caps;

    /* Create vspm-filter and caps-filter */
    filter = gst_element_factory_make ("vspmfilter", NULL);
    capsfilter = gst_element_factory_make ("capsfilter", NULL);

    if (!filter || !capsfilter) {
      g_printerr ("One element could not be created. Exiting.\n");
      gst_object_unref (*p_video_pipeline);
      return 0;
    }

    /* Set property "dmabuf-use" of vspmfilter to true */
  /* Without it, waylandsink will display broken video */
  g_object_set (G_OBJECT (filter), "dmabuf-use", TRUE, NULL);

  /* Create simple cap which contains video's resolution */
  caps = gst_caps_new_simple ("video/x-raw",
      "width", G_TYPE_INT, screen->width,
      "height", G_TYPE_INT, screen->height, NULL);

  /* Add cap to capsfilter element */
  g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
  gst_caps_unref (caps);

  /* Link the elements together */
  /* file-source -> parser -> decoder -> filter -> capsfilter -> video-output */
  if (!gst_element_link_many (video_source, video_parser, video_decoder,
        filter, capsfilter, video_sink, NULL)) {
    g_printerr ("Video elements could not be linked.\n");
    gst_object_unref (*p_video_pipeline);
    return 0;
  }
  return video_bus_watch_id;
}

bool
play_pipeline (GstElement * pipeline, CustomData * p_shared_data)
{
  bool result = true;

  g_mutex_lock (&p_shared_data->mutex);
  ++(p_shared_data->loop_reference);
  if (gst_element_set_state (pipeline,
        GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
    g_printerr ("Unable to set the pipeline to the playing state.\n");
    --(p_shared_data->loop_reference);
    gst_object_unref (pipeline);

    result = false;
  }
  g_mutex_unlock (&p_shared_data->mutex);
  return result;
}

int
```

```
main (int argc, char *argv[])
{
  struct wayland_t *wayland_handler = NULL;
  struct screen_t *screens[REQUIRED_SCREEN_NUMBERS];
  int screen_numbers = 0;

  CustomData shared_data;
  GstElement *video_pipeline_1;
  GstElement *video_pipeline_2;
  guint video_bus_watch_id_1;
  guint video_bus_watch_id_2;

  const gchar *input_video_file_1 = argv[ARG_INPUT1];
  const gchar *input_video_file_2 = argv[ARG_INPUT2];
  const char* video1_ext;
  const char* video2_ext;
  char* file_name_1;
  char* file_name_2;

  if ((argc > ARG_COUNT) || (argc <= 2) || ((argc == ARG_COUNT) && (strcmp (argv[ARG_SCALE], "-s"))))
{
    g_print ("Error: Invalid arugments.\n");
    g_print ("Usage: %s <path to the first H264/H265 file> <path to the second H264/H265 file> [-s] \n",
argv[ARG_PROGRAM_NAME]);
    return -1;
  }

  /* Check full-screen option */
  if (argc == ARG_COUNT) {
    shared_data.fullscreen = true;
  }

  file_name_1 = basename ((char*) input_video_file_1);
  file_name_2 = basename ((char*) input_video_file_2);
  video1_ext = get_filename_ext (file_name_1);
  video2_ext = get_filename_ext (file_name_2);

  if ((strcasecmp ("h264", video1_ext) != 0) && (strcasecmp ("h265", video1_ext) != 0)) {
    g_print ("Unsupported video type. H264/H265 format is required\n");
    return -1;
  }

  if ((strcasecmp ("h264", video2_ext) != 0) && (strcasecmp ("h265", video2_ext) != 0)) {
    g_print ("Unsupported video type. H264/H265 format is required\n");
    return -1;
  }

  /* Get a list of available screen */
  wayland_handler = get_available_screens();
  if (wayland_handler == NULL)
  {
    g_printerr("Cannot detect monitors. Exiting.\n");
    return -1;
  }

  screen_numbers = wl_list_length(&(wayland_handler->screens));
  if (screen_numbers < REQUIRED_SCREEN_NUMBERS)
  {
    g_printerr("Detected %d monitors.\n", screen_numbers);
    g_printerr("Must have at least %d monitors to run the app. Exiting.\n", REQUIRED_SCREEN_NUMBERS);

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Check input file */
  if (!is_file_exist(input_video_file_1) || !is_file_exist(input_video_file_2))
  {
    g_printerr("Make sure the following (H264-encoded) files exist:\n");
    g_printerr("  %s\n", input_video_file_1);
    g_printerr("  %s\n", input_video_file_2);

    destroy_wayland(wayland_handler);
    return -1;
  }
```

```
  /* Extract required monitors */
  get_required_monitors(wayland_handler, screens, REQUIRED_SCREEN_NUMBERS);

  /* Initialization */
  gst_init (&argc, &argv);
  shared_data.loop = g_main_loop_new (NULL, FALSE);
  shared_data.loop_reference = 0; /* counter */
  shared_data.video_ext = video1_ext;
  g_mutex_init (&shared_data.mutex);

  /* The first display is screens[PRIMARY_SCREEN_INDEX] */
  /* The second display is screens[SECONDARY_SCREEN_INDEX] */
  video_bus_watch_id_1 =
     create_video_pipeline (&video_pipeline_1, input_video_file_1,
     screens[PRIMARY_SCREEN_INDEX], &shared_data);

  if (video_bus_watch_id_1 == 0)
  {
    destroy_wayland(wayland_handler);
    return -1;
  }

  shared_data.video_ext = video2_ext;

  video_bus_watch_id_2 =
     create_video_pipeline (&video_pipeline_2, input_video_file_2,
     screens[SECONDARY_SCREEN_INDEX], &shared_data);

  if (video_bus_watch_id_2 == 0)
  {
    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Set the video pipeline 1 to "playing" state */
  if (play_pipeline (video_pipeline_1, &shared_data) == true)
  {
    g_print ("Now playing video file: %s\n", input_video_file_1);
  }
  else
  {
    g_printerr("Unable to play video file: %s\n", input_video_file_1);

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Set the video pipeline 2 to "playing" state */
  if (play_pipeline (video_pipeline_2, &shared_data) == true)
  {
    g_print ("Now playing video file: %s\n", input_video_file_2);
  }
  else
  {
    g_printerr("Unable to play video file: %s\n", input_video_file_2);

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Iterate */
  g_print ("Running...\n");
  g_main_loop_run (shared_data.loop);

  /* Clean up "wayland_t" structure */
  destroy_wayland(wayland_handler);
  /* Out of loop. Clean up nicely */
  g_source_remove (video_bus_watch_id_1);
  g_source_remove (video_bus_watch_id_2);
  g_main_loop_unref (shared_data.loop);
  g_mutex_clear (&shared_data.mutex);
  g_print ("Program end!\n");

  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 3.2.13. Audio Video Play and 3.2.1. Audio Play.

Command-line argument

```
if ((argc > ARG_COUNT) || (argc <= 2) || ((argc == ARG_COUNT) && (strcmp (argv[ARG_SCALE], "-s"))))
{
    g_print ("Error: Invalid arugments.\n");
    g_print ("Usage: %s <path to the first H264/H265 file> <path to the second H264/H265 file> [-s] \n",
argv[ARG_PROGRAM_NAME]);
    return -1;
}
```

This application accepts 2 command-line arguments which points to 2 H.264/H.265 files.
If "-s" option is enabled, video will be scaled to full-screen.

Video pipeline

```
guint create_video_pipeline (GstElement ** p_video_pipeline, const gchar * input_file,
                             struct screen_t * screen, CustomData * data)
```

Basically, the pipeline is just like Video Play except it uses `gst_bus_add_watch()` instead of `gst_bus_timed_pop_filtered()` to receive messages (such as: error or `EOS` (`End-of-Stream`)) from `bus_call()` asynchronously.

Create pipelines

```
create_video_pipeline (&video_pipeline_1, input_video_file_1,
                       screens[PRIMARY_SCREEN_INDEX], &shared_data);

create_video_pipeline (&video_pipeline_2, input_video_file_2,
                       screens[SECONDARY_SCREEN_INDEX], &shared_data);
```

This code block creates 2 pipelines:

- Pipeline `video_pipeline_1` displays the first video in the main Wayland desktop. If "-s" option is enabled, video frames will be resized to the same size of main Wayland desktop.
- Pipeline `video_pipeline_2` displays the second video in the secondary Wayland desktop. If "-s" option is enabled video frames will be resized to the same size of secondary Wayland desktop.

3.2.16.2 Result

Can display 2 videos simultaneously on HDMI and LCD monitors.

Figure 3.34 Multiple Display 2 result

3.2.16.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.16.4 Special Instruction

- Download file vga1.h264 at:
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga1.h264
- Download file vga2.h264 at:
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga2.h264

Note:

RZ/G2E platform supports playing 2 1920x1080, 30 fps videos simultaneously
RZ/G2H, RZ/G2M and RZ/G2N platform support playing 2 2560x1440, 30 fps videos simultaneously

3.2.17 Overlapped Display

Display 3 overlapping H.264 videos.

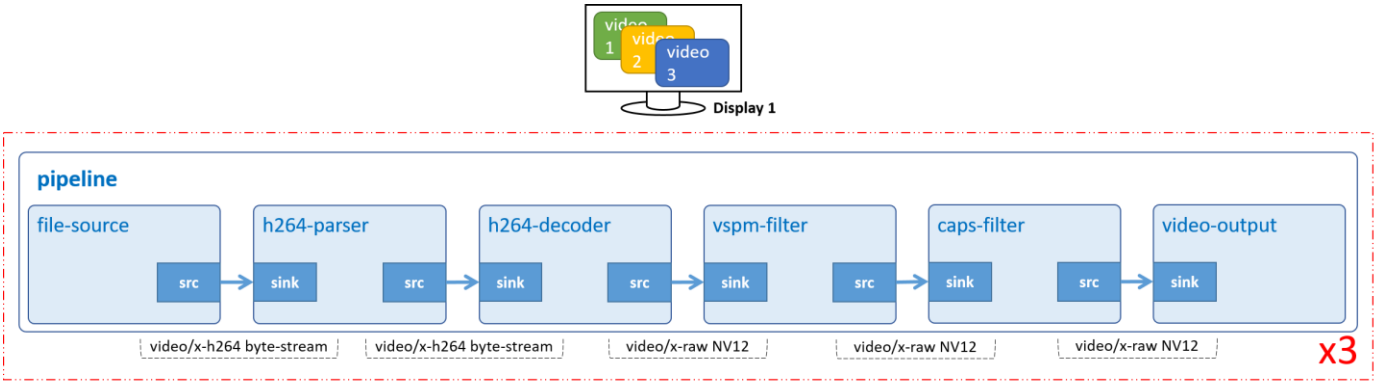


Figure 3.35 Overlapped Display pipeline

3.2.17.1 Source code

(1) Files

- main.c

(2) main.c

```
#include <stdio.h>
#include <string.h>
#include <gst/gst.h>
#include <stdlib.h>
#include <stdbool.h>
#include <wayland-client.h>

#define INPUT_VIDEO_FILE_1        "/home/media/videos/vga1.h264"
#define INPUT_VIDEO_FILE_2        "/home/media/videos/vga2.h264"
#define INPUT_VIDEO_FILE_3        "/home/media/videos/vga3.h264"

typedef struct _CustomData
{
  GMainLoop *loop;
  int loop_reference;
  GMutex mutex;
} CustomData;

static void
try_to_quit_loop (CustomData * p_shared_data)
{
  g_mutex_lock (&p_shared_data->mutex);
  --(p_shared_data->loop_reference);
  if (0 == p_shared_data->loop_reference) {
    g_main_loop_quit ((p_shared_data->loop));
  }
  g_mutex_unlock (&p_shared_data->mutex);
}

static gboolean
bus_call (GstBus * bus, GstMessage * msg, gpointer data)
{
  GstElement *pipeline = (GstElement *) GST_MESSAGE_SRC (msg);
  gchar *pipeline_name = GST_MESSAGE_SRC_NAME (msg);

  switch (GST_MESSAGE_TYPE (msg)) {
```

```
    case GST_MESSAGE_EOS:
      g_print ("End-Of-Stream reached.\n");

      g_print ("Stoping %s...\n", pipeline_name);
      gst_element_set_state (pipeline, GST_STATE_NULL);
      g_print ("Deleting %s...\n", pipeline_name);
      gst_object_unref (GST_OBJECT (pipeline));
      try_to_quit_loop ((CustomData *) data);
      break;

    case GST_MESSAGE_ERROR:{
      gchar *debug_info;
      GError *err;
      gst_message_parse_error (msg, &err, &debug_info);
      g_printerr ("Error received from element %s: %s.\n",
          pipeline_name, err->message);
      g_printerr ("Debugging information: %s.\n",
          debug_info ? debug_info : "none");
      g_clear_error (&err);
      g_free (debug_info);

      g_print ("Stoping %s...\n", pipeline_name);
      gst_element_set_state (pipeline, GST_STATE_NULL);
      g_print ("Deleting %s...\n", pipeline_name);
      gst_object_unref (GST_OBJECT (pipeline));
      try_to_quit_loop ((CustomData *) data);
      break;
    }
    default:
      /* Don't care other message */
      break;
  }
  return TRUE;
}

guint
create_video_pipeline (GstElement ** p_video_pipeline, const gchar * input_file,
    struct screen_t * screen, CustomData * data)
{
  GstBus *bus;
  guint video_bus_watch_id;
  GstElement *video_source, *video_parser, *video_decoder,
            *filter, *capsfilter, *video_sink;
  GstCaps *caps;

  /* Create GStreamer elements for video play */
  *p_video_pipeline = gst_pipeline_new (NULL);
  video_source = gst_element_factory_make ("filesrc", NULL);
  video_parser = gst_element_factory_make ("h264parse", NULL);
  video_decoder = gst_element_factory_make ("omxh264dec", NULL);
  filter = gst_element_factory_make ("vspmfilter", NULL);
  capsfilter = gst_element_factory_make ("capsfilter", NULL);
  video_sink = gst_element_factory_make ("waylandsink", NULL);

  if (!*p_video_pipeline || !video_source || !video_parser || !video_decoder
      || !filter || !capsfilter || !video_sink) {
    g_printerr ("One video element could not be created. Exiting.\n");
    return 0;
  }

  /* Adding a message handler for video pipeline */
  bus = gst_pipeline_get_bus (GST_PIPELINE (*p_video_pipeline));
  video_bus_watch_id = gst_bus_add_watch (bus, bus_call, data);
  gst_object_unref (bus);

  /* Add all elements into the video pipeline */
  /* file-source | h264-parser | h264-decoder | video-output */
  gst_bin_add_many (GST_BIN (*p_video_pipeline), video_source, video_parser,
      video_decoder, filter, capsfilter, video_sink, NULL);

  /* Set up for the video pipeline */
  /* Set the input file location of the file source element */
  g_object_set (G_OBJECT (video_source), "location", input_file, NULL);
  g_object_set (G_OBJECT (video_sink), "position-x", screen->x, "position-y",
```

```
      screen->y, NULL);

  /* Set property "dmabuf-use" of vspmfilter to true */
  /* Without it, waylandsink will display broken video */
  g_object_set (G_OBJECT (filter), "dmabuf-use", TRUE, NULL);

  /* Create simple cap which contains video's resolution */
  caps = gst_caps_new_simple ("video/x-raw",
      "width", G_TYPE_INT, screen->width,
      "height", G_TYPE_INT, screen->height, NULL);

  /* Add cap to capsfilter element */
  g_object_set (G_OBJECT (capsfilter), "caps", caps, NULL);
  gst_caps_unref (caps);

  /* Link the elements together */
  /* file-source -> h264-parser -> h264-decoder -> video-output */
  if (!gst_element_link_many (video_source, video_parser, video_decoder,
        filter, capsfilter, video_sink, NULL)) {
    g_printerr ("Video elements could not be linked.\n");
    gst_object_unref (*p_video_pipeline);
    return 0;
  }

  return video_bus_watch_id;
}

bool
play_pipeline (GstElement * pipeline, CustomData * p_shared_data)
{
  bool result = true;

  g_mutex_lock (&p_shared_data->mutex);
  ++(p_shared_data->loop_reference);
  if (gst_element_set_state (pipeline,
        GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
    g_printerr ("Unable to set the pipeline to the playing state.\n");
    --(p_shared_data->loop_reference);
    gst_object_unref (pipeline);

    result = false;
  } else {
    /* wait until the changing is complete */
    gst_element_get_state (pipeline, NULL, NULL, GST_CLOCK_TIME_NONE);
  }
  g_mutex_unlock (&p_shared_data->mutex);

  return result;
}
int
main (int argc, char *argv[])
{
  struct wayland_t *wayland_handler = NULL;
  struct screen_t *main_screen = NULL;
  struct screen_t temp;

  CustomData shared_data;
  GstElement *video_pipeline_1;
  GstElement *video_pipeline_2;
  GstElement *video_pipeline_3;

  guint video_bus_watch_id_1;
  guint video_bus_watch_id_2;
  guint video_bus_watch_id_3;

  const gchar *input_video_file_1 = INPUT_VIDEO_FILE_1;
  const gchar *input_video_file_2 = INPUT_VIDEO_FILE_2;
  const gchar *input_video_file_3 = INPUT_VIDEO_FILE_3;

  /* Get a list of available screen */
  wayland_handler = get_available_screens();

  /* Get main screen */
  main_screen = get_main_screen(wayland_handler);
```

```
if (main_screen == NULL)
{
  g_printerr("Cannot find any available screens. Exiting.\n");
  destroy_wayland(wayland_handler);
  return -1;
}

/* Check input file */
if (!is_file_exist(input_video_file_1) || !is_file_exist(input_video_file_2) ||
    !is_file_exist(input_video_file_3))
{
  g_printerr("Make sure the following (H264-encoded) files exist:\n");
  g_printerr("  %s\n", input_video_file_1);
  g_printerr("  %s\n", input_video_file_2);
  g_printerr("  %s\n", input_video_file_3);

  destroy_wayland(wayland_handler);
  return -1;
}

/* Initialization */
gst_init (&argc, &argv);
shared_data.loop = g_main_loop_new (NULL, FALSE);
shared_data.loop_reference = 0; /* Counter */
g_mutex_init (&shared_data.mutex);

/* The video will be half the size of the screen */
temp.width = main_screen->width / 2;
temp.height = main_screen->height / 2;

/* Create first pipeline */
temp.x = 0;
temp.y = 0;
video_bus_watch_id_1 =
    create_video_pipeline (&video_pipeline_1, input_video_file_1,
    &temp, &shared_data);

if (video_bus_watch_id_1 == 0)
{
  destroy_wayland(wayland_handler);
  return -1;
}

/* Create second pipeline */
temp.x = main_screen->width / 4;
temp.y = main_screen->height / 4;
video_bus_watch_id_2 =
    create_video_pipeline (&video_pipeline_2, input_video_file_2,
    &temp, &shared_data);

if (video_bus_watch_id_2 == 0)
{
  destroy_wayland(wayland_handler);
  return -1;
}

/* Create third pipeline */
temp.x = main_screen->width / 2;
temp.y = main_screen->height / 2;
video_bus_watch_id_3 =
    create_video_pipeline (&video_pipeline_3, input_video_file_3,
    &temp, &shared_data);

if (video_bus_watch_id_3 == 0)
{
  destroy_wayland(wayland_handler);
  return -1;
}

/* Set the video pipeline 1 to "playing" state */
if (play_pipeline (video_pipeline_1, &shared_data) == true)
{
  g_print ("Now playing video file: %s\n", input_video_file_1);
}
```

```
  else
  {
    g_printerr("Unable to play video file: %s\n", input_video_file_1);

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Set the video pipeline 2 to "playing" state */
  if (play_pipeline (video_pipeline_2, &shared_data) == true)
  {
    g_print ("Now playing video file: %s\n", input_video_file_2);
  }
  else
  {
    g_printerr("Unable to play video file: %s\n", input_video_file_2);

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Set the video pipeline 3 to "playing" state */
  if (play_pipeline (video_pipeline_3, &shared_data) == true)
  {
    g_print ("Now playing video file: %s\n", input_video_file_3);
  }
  else
  {
    g_printerr("Unable to play video file: %s\n", input_video_file_3);

    destroy_wayland(wayland_handler);
    return -1;
  }

  /* Iterate */
  g_print ("Running...\n");
  g_main_loop_run (shared_data.loop);

  /* Clean up "wayland_t" structure */
  destroy_wayland(wayland_handler);

  /* Out of loop. Clean up nicely */
  g_source_remove (video_bus_watch_id_1);
  g_source_remove (video_bus_watch_id_2);
  g_source_remove (video_bus_watch_id_3);
  g_main_loop_unref (shared_data.loop);
  g_mutex_clear (&shared_data.mutex);
  g_print ("Program end!\n");

  return 0;
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section: 3.2.15. Multiple Dipslay 1, 3.2.13. Audio-Video Play, and 3.2.1. Audio Play.

Input location

```
#define INPUT_VIDEO_FILE_1        "/home/media/videos/vga1.h264"
#define INPUT_VIDEO_FILE_2        "/home/media/videos/vga2.h264"
#define INPUT_VIDEO_FILE_3        "/home/media/videos/vga3.h264"
```

Video pipeline

```
guint create_video_pipeline (GstElement ** p_video_pipeline, const gchar * input_file,
                             struct screen_t * screen, CustomData * data)
```

Basically, the pipeline is just like Video Play except it uses `gst_bus_add_watch()` instead of

gst_bus_timed_pop_filtered() to receive messages (such as: error or EOS (End-of-Stream)) from bus_call() asynchronously.

Create pipelines

```
create_video_pipeline (&video_pipeline_1, input_video_file_1, &temp, &shared_data);
create_video_pipeline (&video_pipeline_2, input_video_file_2, &temp, &shared_data);
create_video_pipeline (&video_pipeline_3, input_video_file_3, &temp, &shared_data);
```

This code block creates 3 pipelines:

- Pipeline video_pipeline_1 resizes video frames of vga1.h264 to half size of main Wayland desktop and displays them at the origin coordinate. In this application, it will always be at (0, 0).
- Pipeline video_pipeline_2 resizes video frames of vga2.h264 to half size of the main desktop and displays them at coordinate (width / 4, height / 4).
- Pipeline video_pipeline_3 resizes video frames of vga3.h264 to half size of the main desktop and displays them at coordinate (width / 2, height / 2).

Play pipelines

```
int main (int argc, char *argv[])
{
  play_pipeline (video_pipeline_1, &shared_data)
  play_pipeline (video_pipeline_2, &shared_data)
  play_pipeline (video_pipeline_3, &shared_data)
}

void play_pipeline (GstElement * pipeline, CustomData * p_shared_data)
{
  ++(p_shared_data->loop_reference);
  if (gst_element_set_state (pipeline, GST_STATE_PLAYING) == GST_STATE_CHANGE_FAILURE) {
    g_printerr ("Unable to set the pipeline to the playing state.\n");
    --(p_shared_data->loop_reference);
    gst_object_unref (pipeline);
  }
}
```

Basically, this function sets the state of pipeline to PLAYING. If successful, it will increases loop_reference to indicate that there is 1 more running pipeline. Note that this variable must be 3 for this application to play 3 H.264 videos.

Stop pipelines

```
static void try_to_quit_loop (CustomData * p_shared_data)
{
  g_mutex_lock (&p_shared_data->mutex);
  --(p_shared_data->loop_reference);
  if (0 == p_shared_data->loop_reference) {
    g_main_loop_quit ((p_shared_data->loop));
  }
  g_mutex_unlock (&p_shared_data->mutex);
}
```

The main event loop will stop only if variable loop_reference reaches to 0. This means the application will exit when all 3 pipelines stopped. Also note that mutex is used to prevent GStreamer threads from reading incorrect value of loop_reference.

3.2.17.2 Result

Can display 3 overlapping videos on the main screen.

Figure 3.36 Overlapped Display result

3.2.17.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

3.2.17.4 Special Instruction

- Download file vga1.h264 at:
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga1.h264
- Download file vga2.h264 at:
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga2.h264
- Download file vga3.h264 at:
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga3.h264
- Please put these video files at location `/home/media/videos/` (on board).

4. Qt applications

Qt[1] is a cross-platform application development framework for desktop, embedded and mobile. Supported Platforms include Linux, OS X, Windows, VxWorks, QNX, Android, iOS, BlackBerry, Sailfish OS, and others.

Qt is not a programming language on its own. It is a framework written in C++. Thus the framework itself and applications/libraries using it can be compiled by any standard compliant C++ compiler like Clang, GCC, ICC, MinGW, and MSVC.

4.1 Hello world

In this section, we will introduce a basic knowledge about Qt Quick (items, properties, the meaning of main.cpp and main.qml) by creating a simple Qt HelloWorld application.

4.1.1 Development Environment

- Qt: 5.6.3 (edited by Renesas).

Please refer to section **2.1.1** to prepare equipment for GStreamer applications.

For more information about Qt knowledge, please refer Appendix:

1. Structure of Qt in RZ/G platform.
2. Qt Quick introduction.
3. Qt Creator IDE.

4.1.2 Application content

Display “Hello World” on the window.

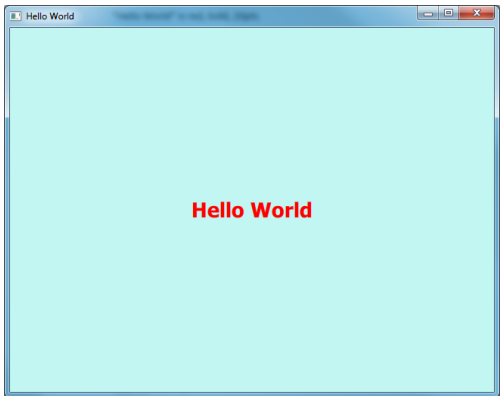


Figure 4.1 Hello World application

4.1.2.1 Source code

(1) Source code list

- main.c
- main.qml

(2) main.c

```
#include <QtGui/QGuiApplication>
#include <QtQml/QQmlApplicationEngine>
```

[1] https://wiki.qt.io/About_Qt

```
#include <QtGui/QScreen>
#include <QDebug>

int main(int argc, char *argv[])
{
    // Initializes the window system and constructs an application object with
    // argc command line arguments in argv
    QGuiApplication app(argc, argv);

    // This function load() will load the root QML file "main.qml"
    QQmlApplicationEngine engine;

    // Show information of screen (all monitors)
    // If 2 screen available, chose the larger as it is usually default
    QScreen *screen;
    if (QGuiApplication::screens().size() <= 1)
    {
        screen = QGuiApplication::primaryScreen();
    }
    else
    {
        QScreen *tmp_screen;
        if(QGuiApplication::screens().at(0) == NULL)
            screen = QGuiApplication::screens().at(1);
          else if(QGuiApplication::screens().at(1) == NULL)
            screen = QGuiApplication::screens().at(0);
          else {
            screen = QGuiApplication::screens().at(0);
            tmp_screen = QGuiApplication::screens().at(1);
            if((screen->availableSize().width() * screen->availableSize().height()) <
                (tmp_screen->availableSize().width() * tmp_screen->availableSize().height()))
                screen = QGuiApplication::screens().at(1);
          }
    }

    qDebug() << "Information for screen:" << screen->name();

    qDebug() << "  Available geometry:" << screen->availableGeometry().x()
                                        << screen->availableGeometry().y()
                                        << screen->availableGeometry().width() << "x"
                                        << screen->availableGeometry().height();

    qDebug() << "  Available size:" << screen->availableSize().width() << "x"
                                    << screen->availableSize().height();

    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    // Wait for Qt HelloWorld form to close, then returns a value indicating a run success or failure
    return app.exec();
}
```

Walkthrough:

```
QGuiApplication app(argc, argv);
```

The `QGuiApplication` class manages control flow and main settings of the application.

For any GUI application using Qt, there is precisely one `QGuiApplication` object no matter whether the application has 0, 1, 2 or more windows at any given time.

```
QQmlApplicationEngine engine;
```

This class provides a convenient way to load a single QML file.

```
engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
```

The load() function loads the root QML file `main.qml`.

```
return app.exec();
```

This code block waits for the window to close, then returns a value indicating the exit status of the application.

(3) main.qml

```
import QtQuick 2.5
import QtQuick.Window 2.2

//This Window component will create a new top-level window with title name "Hello World" for a Qt Quick
scene
Window {
    // Scale 1/4 full screen
    property int minimized_scale: 4
    //These properties are own of the window component
    visible: true
    width: Screen.desktopAvailableWidth/minimized_scale
    height: Screen.desktopAvailableHeight/minimized_scale
    title: "Hello World"
    color: "lightblue"
    visibility: "Minimized"
    id: mainWindow

    Component.onCompleted: {
            mainWindow.maximumWidth = Screen.desktopAvailableWidth
            mainWindow.maximumHeight = Screen.desktopAvailableHeight
            mainWindow.minimumWidth = Screen.desktopAvailableWidth/minimized_scale
            mainWindow.minimumHeight = Screen.desktopAvailableHeight/minimized_scale
            mainWindow.showMaximized();
            console.log("Width: ", Screen.desktopAvailableWidth,
                        "  Height: ", Screen.desktopAvailableHeight)
    }

    // This Text component will display text message named "Hello World" at the center of the application
    Text {
        anchors.centerIn: parent
        // Bold red "Hello World" text with size 20pts
        text: "Hello World"
        color: "red"
        font.bold: true
        font.pointSize: 20
    }
}
```

Walkthrough:

```
import QtQuick 2.5
```

The `QtQuick` module provides graphical primitive types (`Rectangle`, for example).

```
import QtQuick.Window 2.2
```

We must import `QtQuick.Window` namespace to use `Window` item.

```
Window {
```

The `Window` item represents a GUI form.

```
visible: true
```

We must set `visible` property of `Window` item to `true` to display the GUI form.

```
width: Screen.desktopAvailableWidth/minimized_scale
height: Screen.desktopAvailableHeight/minimized_scale
```

This code block sets the width and height of the GUI form.

```
title: "Hello World"
color: "lightblue"
```

As already explained by itself, the application will set its title to “Hello World” and adjust the background of `Window` to light blue color.

```
Text {
```

The `Text` item represents a simple text message.

```
anchors.centerIn: parent
```

The `anchors.centerIn` property will center the `Text` in the `Window`.

```
text: "Hello World"
color: "red"
font.bold: true
font.pointSize: 20
```

This code block assigns “Hello World” message to the `Text` item, then `bolds` the message and changes its size to `20 pt`. Note that, it also changes the color of the message to `red`.

Note:

For further information about Qt, please refer its official link[2].

[2] http://doc.qt.io

4.1.3 How to build and run Qt application

This section shows how to cross-compile and deploy Qt *Hello World* application.

4.1.3.1 How to extract SDK

Step 1. Install toolchain on a Host PC:

```
$ sudo sh ./poky-glibc-x86_64-core-image-qt-sdk-aarch64-toolchain-2.4.3.sh
```

Note:

sudo is optional in case user wants to extract SDK into a restricted directory (such as: /opt/).

If the installation is successful, the following messages will appear:

```
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
 $ . /opt/poky/ek874/environment-setup-aarch64-poky-linux
 $ . /opt/poky/ek874/environment-setup-armv7vehf-neon-pokymllib32-linux-gnueabi
```

Figure 4.2 SDK is successfully set up

Step 2. Set up cross-compile environment:

```
$ source /<Location in which SDK is extracted>/environment-setup-aarch64-poky-linux
```

Note:

User needs to run the above command once for each login session.

4.1.3.2 How to build and run Qt application

Step 1. Go to *qt-helloworld* directory:

```
$ cd $WORK/qt-helloworld
```

Step 2. Generate makefile:

```
$ qmake
```

Step 3. Cross-compile:

```
$ make
```

Step 4. Copy all files inside this directory to `/usr/share` directory on the target board:

```
$ scp -r $WORK/qt-helloworld/ <username>@<board IP>:/usr/share/
```

Step 5. Run the application:

```
/usr/share/qt-helloworld/qt-helloworld
```

4.2 Application samples

The following table shows multimedia applications in RZ/G2 platform ranging from playing, recording audio/video, to displaying multiple videos on multiple monitors.

Application name	Description
Audio Play	Play an Ogg/Vorbis audio file.
Audio Record	A simple GUI application for recording raw data from microphone, then store it in Ogg container.
Video Play	Play an MP4 file (no sound).
File Play	Play an MP4 file.
Multiple Displays 1	Display video simultaneously on HDMI and LCD monitors.
Multiple Displays 2	Display 2 videos simultaneously on HDMI and LCD monitors.
Overlapped Display	Display 3 overlapping videos.
3D Graphics	Display and automatically rotate 3D color shapes, such as: cube, sphere, and cone.
3D Graphics (with textures)	Display and automatically rotate 3D textured shapes, such as: cube, sphere, and cone.
Multimedia Player	A simple multimedia player.

Table 4.1 Qt sample applications

4.2.1 Audio Play

Play an Ogg/Vorbis audio file.

4.2.1.1 Source code

(1) Files

File name	Description
qt-audioplay.pro	For further information, please refer to Appendix, section 2 (Qt Quick introduction).
main.cpp	
main.qml	
qml.qrc	
audio.ogg	The hard-coded input file.

Table 4.2 Audio Play source code

(2) main.cpp

```
#include <QtGui/QGuiApplication>
#include <QtQml/QQmlApplicationEngine>
#include <QtQml/QQmlContext>
#include <QtGui/QScreen>
#include <QDebug>

#define AUDIOSDIRPATH "file:///home/media/audios"

int main(int argc, char *argv[])
{
    // Initializes the window system and constructs an application object
    // with argc command line arguments in argv
    QGuiApplication app(argc, argv);

    // This class provides a convenient way to load a single QML file
    QQmlApplicationEngine engine;

    // Show information of screen (all monitors)
    // If 2 screen available, chose the larger as it is usually default
    QScreen *screen;
    if (QGuiApplication::screens().size() <= 1)
    {
        screen = QGuiApplication::primaryScreen();
    }
    else
    {
        QScreen *tmp_screen;
        if(QGuiApplication::screens().at(0) == NULL)
            screen = QGuiApplication::screens().at(1);
          else if(QGuiApplication::screens().at(1) == NULL)
            screen = QGuiApplication::screens().at(0);
          else {
            screen = QGuiApplication::screens().at(0);
            tmp_screen = QGuiApplication::screens().at(1);
            if((screen->availableSize().width() * screen->availableSize().height()) <
               (tmp_screen->availableSize().width() * tmp_screen->availableSize().height()))
                screen = QGuiApplication::screens().at(1);
          }
    }
    qDebug() << "Information for screen:" << screen->name();

    qDebug() << "  Available geometry:" << screen->availableGeometry().x()
                                        << screen->availableGeometry().y()
                                        << screen->availableGeometry().width() << "x"
                                        << screen->availableGeometry().height();
```

```
    qDebug() << "  Available size:" << screen->availableSize().width() << "x"
             << screen->availableSize().height();

    // Set the value of the "audiosDirPath" variable which is path to audios
    QQmlContext *context = engine.rootContext();
    context->setContextProperty("audiosDirPath", AUDIOSDIRPATH);

    // This function load() will load the root QML file "main.qml"
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    // Wait for Qt HelloWorld form to close, then returns a value
    // indicating a run success or failure
    return app.exec();
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section: 4.1.2 Hello World.

```
#define AUDIOSDIRPATH "file:///home/media/audios"

QQmlApplicationEngine engine;

QQmlContext *context = engine.rootContext();
context->setContextProperty("audiosDirPath", AUDIOSDIRPATH);
```

The `QQmlApplicationEngine` class provides a convenient way to load `main.qml` file by using function `QQmlApplicationEngine::load()`. Beside this feature, it is also used to get a `QQmlContext` object which allows data to be exposed to the QML components.

In this case, the application binds the literal string `file://home/media/audios` to the `context`. Now, all QML items can access to this string via `audiosDirPath` variable.

(3) main.qml

```
import QtQuick 2.5
import QtQuick.Window 2.2
import QtMultimedia 5.5

// This Window component will create a new top-level window with
// title name "Audio Play" for a Qt Quick scene
Window {
    // Scale 1/4 full screen
    property int minimized_scale : 4
    // These properties are own of the Window component
    visible: true
    width: Screen.desktopAvailableWidth/minimized_scale
    height: Screen.desktopAvailableHeight/minimized_scale
    title: qsTr("Audio Play")
    color: "lightblue"
    visibility: "Minimized"
    id: mainWindow

    Component.onCompleted: {
            mainWindow.maximumWidth = Screen.desktopAvailableWidth
            mainWindow.maximumHeight = Screen.desktopAvailableHeight
            mainWindow.minimumWidth = Screen.desktopAvailableWidth/minimized_scale
            mainWindow.minimumHeight = Screen.desktopAvailableHeight/minimized_scale
            mainWindow.showMaximized();
            console.log("Width: ", Screen.desktopAvailableWidth, "  Height: ",
Screen.desktopAvailableHeight)
    }

    // This Text component will display text message named "Click to play audio"
    // at the center of the application.
    Text {
        id: textInfo
```

```
        text: qsTr("Click to play audio")
        anchors.centerIn: parent
        color: "red"
        font.bold: true
        font.pointSize: 20
    }

    // This Text component creates a text message and displays
    // audio information, such as: file name, size, bit rate, and codec.
    Text {
        text:
            "File name: audio.ogg\n" +
            "Bit rate: " + playMusic.metaData.audioBitRate + " (bps)\n" +
            "Codec: " + playMusic.metaData.audioCodec + "\n"
    }

    // This MediaPlayer component adds a media file and plays it
    MediaPlayer {
        id: playMusic
        source: audiosDirPath + "/audio.ogg"
        autoLoad: true
        autoPlay: false
        loops: MediaPlayer.Infinite
    }

    // This MouseArea component starts and pauses the audio.
    MouseArea {
        anchors.fill: parent
        onClicked: {
            // Toggle the audio when users click the application
            if (playMusic.playbackState == MediaPlayer.PlayingState) {
                textInfo.text = "Audio is paused";
                playMusic.pause();
            } else {
                textInfo.text = "Audio is playing";
                playMusic.play();
            }
        }
    }
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section: 4.1.2 Hello World.

```
import QtMultimedia 5.5
```

`QtMultimedia` provides a rich set of QML types and C++ classes to handle multimedia content.

```
MediaPlayer {
    id: playMusic
    source: audiosDirPath + "/audio.ogg"
    autoLoad: true
    autoPlay: false
    loops: MediaPlayer.Infinite
}
```

Simply speaking, the `MediaPlayer` item is used to play audio content. You can use it in conjunction with `VideoOutput` for rendering video.

In this application, this item will load `audio.ogg` at location `/home/media/audios` and play the file `repeatedly` when `play()` method is called.

Note that other QML items (inside `main.qml`) can refer to the `MediaPlayer` item as `playMusic`.

```
Text {
    text:
```

```
        "File name: audio.ogg\n" +
        "Bit rate: " + playMusic.metaData.audioBitRate + " (bps)\n" +
        "Codec: " + playMusic.metaData.audioCodec + "\n"
}
```

This `Text` item retrieves audio information (such as: bit rate, codec) from `metaData` group of `MediaPlayer`.

```
MouseArea {
    anchors.fill: parent
    onClicked: {
        // Toggle the audio when users click the application
        if (playMusic.playbackState == MediaPlayer.PlayingState) {
            textInfo.text = "Audio is paused";
            playMusic.pause();
        } else {
            textInfo.text = "Audio is playing";
            playMusic.play();
        }
    }
}
```

The `MouseArea` is an invisible item that is typically used in conjunction with a visible item in order to provide mouse handling for that item. In this case, this items covers entire `Window` to listen for mouse click event.

If user clicks on the `Window` when the audio is running, the application will pause the audio (`playMusic.pause`) and display "Audio is paused" message. Otherwise, it will start the audio (`playMusic.play`) and display "Audio is playing" message.

4.2.1.2 Result

Can play audio through speaker/headphone.

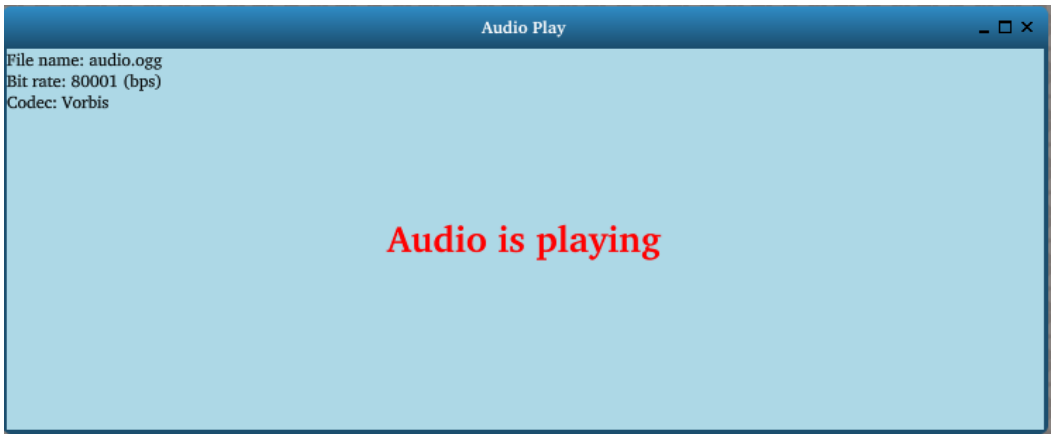


Figure 4.3 Audio Play result

4.2.1.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

4.2.1.4 Special instruction

- Download the input file at: https://upload.wikimedia.org/wikipedia/commons/b/bd/Rondo_Alla_Turka.ogg
- Please put it in this location `/home/media/audios/` and rename it `audio.ogg` (on board).
- To set the playback volume: please use the `alsamixer` or `amixer` tool. It depends on the audio system on the specific board. Reference https://en.wikipedia.org/wiki/Alsamixer

4.2.2 Audio Record

A simple GUI application for recording raw data from microphone, then store it in Ogg container.

4.2.2.1 Source code

(1) Files

File name	Description
qt-audiorecorder.pro	This is the main project file. It contains information of the project, such as: sources, headers, resource files...
main.cpp	The program's main entry point.
qaudiolevel.cpp	Define a status bar of audio level.
qaudiolevel.h	Declare `QAudioLevel` class to display audio level
qt-audiorecorder.cpp	Define an `AudioRecorder` class to record audio
qt-audiorecorder.h	Declare `AudioRecorder` class
qt-audiorecorder.ui	Define user interface of the application

Table 4.3 Audio Record source code

(2) main.cpp

```
#include "qt-audiorecorder.h"
#include <QtWidgets>
#include <QRect>
#include <QScreen>

#define MINIMIZED_SCALE4

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    AudioRecorder recorder;

    //Show information of screen (all monitors)
    // If 2 screen availabe, chose the larger as it is usually default
    QScreen *screen;
    if (QGuiApplication::screens().size() <= 1)
    {
        screen = QGuiApplication::primaryScreen();
    }
    else
    {
        QScreen *tmp_screen;
        if(QGuiApplication::screens().at(0) == NULL)
            screen = QGuiApplication::screens().at(1);
          else if(QGuiApplication::screens().at(1) == NULL)
            screen = QGuiApplication::screens().at(0);
          else {
            screen = QGuiApplication::screens().at(0);
            tmp_screen = QGuiApplication::screens().at(1);
            if((screen->availableSize().width() * screen->availableSize().height()) <
(tmp_screen->availableSize().width() * tmp_screen->availableSize().height()))
                screen = QGuiApplication::screens().at(1);
          }
    }
    qDebug() << "Information for screen:" << screen->name();
    qDebug() << "  Available geometry:" << screen->availableGeometry().x() <<
screen->availableGeometry().y() << screen->availableGeometry().width() << "x" <<
screen->availableGeometry().height();
    qDebug() << "  Available size:" << screen->availableSize().width() << "x" <<
screen->availableSize().height();
    int width = screen->availableSize().width();
    int height = screen->availableSize().height();
  recorder.setMaximumSize(width, height);
```

```
    // resize to set minimum window
    recorder.resize(width/MINIMIZED_SCALE, height/MINIMIZED_SCALE);
    recorder.showNormal();

    return app.exec();
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section: 4.1.2 Hello World.

```
AudioRecorder recorder;
recorder.showNormal();
```

An Audiorecorder object is created then is showed on the screen.

(3) qaudiolevel.cpp

```
#include <QtGui/QPainter>
#include "qaudiolevel.h"

// Default parent constructor to create the status bar
QAudioLevel::QAudioLevel(QWidget *parent)
  : QWidget(parent)
  , m_level(0.0)
{
    setMinimumHeight(15);
    setMaximumHeight(50);
}

// Update m_level value which is used to update status bar on Ui
void QAudioLevel::setLevel(qreal level)
{
    if (m_level != level) {
        m_level = level;
        update();
    }
}

// Catch ui paint event and draw element with specific request
// In this function catch event when m_level update then draw status bar with corresponding red & black
color
void QAudioLevel::paintEvent(QPaintEvent *event)
{
    Q_UNUSED(event);

    QPainter painter(this);
    // draw level
    qreal widthLevel = m_level * width();
    painter.fillRect(0, 0, widthLevel, height(), Qt::red);
    // clear the rest of the control
    painter.fillRect(widthLevel, 0, width(), height(), Qt::black);
}
```

Walkthrough:

```
void QAudioLevel::setLevel(qreal level)
{
    if (m_level != level) {
        m_level = level;
        update();
    }
}
```

This method is used to update level of audio whenever an audio buffer is processed in the media service.

```
void QAudioLevel::paintEvent(QPaintEvent *event)
```

```
{
    Q_UNUSED(event);

    QPainter painter(this);
    // draw level
    qreal widthLevel = m_level * width();
    painter.fillRect(0, 0, widthLevel, height(), Qt::red);
    // clear the rest of the control
    painter.fillRect(widthLevel, 0, width(), height(), Qt::black);
}
```

This method is used to draw a status bar of audio level. The part presenting for audio level is filled with red while the rest is filled with black.

(4) qt-audiorecorder.cpp

Walkthrough:

```
AudioRecorder::AudioRecorder(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::AudioRecorder),
    outputLocationSet(false)
```

These lines call the QMainWindow constructor which is the base class for the `AudioRecorder` class then create the UI class instance and assigns it to the `ui` member.

```
audioRecorder = new QAudioRecorder(this);
probe = new QAudioProbe;
probe->setSource(audioRecorder);
```

These line creates `QAudioRecorder` object and `QAudioProbe` object.

`QAudioRecorder` is used for recording of audio.

`QAudioProbe` class is used for monitoring audio being played or recorded. After setting the source to monitor with `setSource()`, the `audioBufferProbed()` signal will be emitted when audio buffers are flowing in the source media object.

```
for (QString &device: audioRecorder->audioInputs()) {
    ui->audioDeviceBox->addItem(device, QVariant(device));
}
```

Get a list of available audio inputs then add them to the combo box.

```
void AudioRecorder::updateProgress(qint64 duration);
```

This method updates total record duration.

```
void AudioRecorder::updateStatus(QMediaRecorder::Status status);
```

This method updates record state and output location.

```
void AudioRecorder::onStateChanged(QMediaRecorder::State state);
```

This method updates button's label.

```
void AudioRecorder::toggleRecord()
{
    if (audioRecorder->state() == QMediaRecorder::StoppedState) {
        audioRecorder->setAudioInput(boxValue(ui->audioDeviceBox).toString());
        QAudioEncoderSettings settings;
        settings.setBitRate(BIT_RATE);
        settings.setChannelCount(CHANEL_COUNT);
```

```
        settings.setCodec(RECORD_CODEC);
        QString container = RECORD_CONTAINER;
        audioRecorder->setEncodingSettings(settings, QVideoEncoderSettings(), container);
        audioRecorder->record();
    }
    else {
        audioRecorder->stop();
    }
}
```

When starting to record, the audio input selected in combo box is set as audio input of `audioRecorder`. A `QAudioEncoderSettings` object is created to specify desired properties then is passed to `audioRecorder`.

Method `record()` and `stop()` are used to start and stop recording.

```
QVector<qreal> getBufferLevels(const QAudioBuffer& buffer)
```

This method returns the audio level.

```
void AudioRecorder::processBuffer(const QAudioBuffer& buffer)
```

This method updates audio level whenever signal `audioBufferProbed` emits.

4.2.2.2 Result

Can record raw audio data and store it in OGG container.

Figure 4.4 Audio Record result

4.2.2.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

4.2.2.4 Special instruction

To check the output file: VLC media player (https://www.videolan.org/vlc/index.html).

4.2.3 Video Play

Play an MP4 file (no sound).

4.2.3.1 Source code

(1) Files

Application name	Description
qt-videoplay.pro	For further information, please refer to Appendix, section 2 (Qt Quick introduction).
main.cpp	
main.qml	
qml.qrc	
h264-hd-video.mp4	The hard-coded input file.

Table 4.4 Video Play source code

(2) main.cpp

```
#include <QtGui/QGuiApplication>
#include <QtQml/QQmlApplicationEngine>
#include <QtQml/QQmlContext>
#include <QtGui/QScreen>
#include <QDebug>

#define VIDEOSDIRPATH file:///home/media/videos
int main(int argc, char *argv[])
{
    // Initializes the window system and constructs an application object with
    // argc command line arguments in argv
    QGuiApplication app(argc, argv);

    // This class provides a convenient way to load a single QML file
    QQmlApplicationEngine engine;

    // Show information of screen (all monitors)
    // If 2 screen available, chose the larger as it is usually default
    QScreen *screen;
    if (QGuiApplication::screens().size() <= 1)
    {
        screen = QGuiApplication::primaryScreen();
    }
    else
    {
        QScreen *tmp_screen;
        if(QGuiApplication::screens().at(0) == NULL)
            screen = QGuiApplication::screens().at(1);
          else if(QGuiApplication::screens().at(1) == NULL)
            screen = QGuiApplication::screens().at(0);
          else {
            screen = QGuiApplication::screens().at(0);
            tmp_screen = QGuiApplication::screens().at(1);
            if((screen->availableSize().width() * screen->availableSize().height()) <
               (tmp_screen->availableSize().width() * tmp_screen->availableSize().height()))
                screen = QGuiApplication::screens().at(1);
          }
    }
    qDebug() << "Information for screen:" << screen->name();

    qDebug() << "  Available geometry:" << screen->availableGeometry().x()
                                        << screen->availableGeometry().y()
                                        << screen->availableGeometry().width() << "x"
                                        << screen->availableGeometry().height();
```

```
    qDebug() << "  Available size:" << screen->availableSize().width() << "x"
                                    << screen->availableSize().height();

    // Set the value of the "videosDirPath" variable which is path to videos
    QQmlContext *context = engine.rootContext();
    context->setContextProperty("videosDirPath", VIDEOSDIRPATH);

    // This function load() will load the root QML file "main.qml"
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    // Wait for Qt HelloWorld form to close, then returns a value
    // indicating a run success or failure
    return app.exec();
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section: 4.1.2 Hello World.

```
#define VIDEOSDIRPATH file:///home/media/videos

QQmlApplicationEngine engine;

QQmlContext *context = engine.rootContext();
context->setContextProperty("videosDirPath", VIDEOSDIRPATH);
```

The application retrieves the `context` from `QQmlApplicationEngine::rootContext()`, then use it to bind the literal string `file://home/media/videos` to the `context`. Now, all QML items can access to this string via `videosDirPath` variable.

(3) main.qml

```
import QtQuick 2.5
import QtQuick.Window 2.2
import QtMultimedia 5.5

// This Window component will create a new top-level window with
// title name "Video Play (no sound)" for a Qt Quick scene
Window {
    id: mainWindow
    // Scale 1/4 full screen
    property int minimized_scale: 4

    // These properties are own of the window component
    visible: true
    width: Screen.desktopAvailableWidth/minimized_scale
    height: Screen.desktopAvailableHeight/minimized_scale
    title: qsTr("Video Play (no sound)")
    color: "black"
    visibility: "Minimized"

    Component.onCompleted: {
            mainWindow.maximumWidth = Screen.desktopAvailableWidth
            mainWindow.maximumHeight = Screen.desktopAvailableHeight
            mainWindow.minimumWidth = Screen.desktopAvailableWidth/minimized_scale
            mainWindow.minimumHeight = Screen.desktopAvailableHeight/minimized_scale
            mainWindow.showMaximized();
            console.log("Width: ", Screen.desktopAvailableWidth, "  Height: ",
Screen.desktopAvailableHeight)
    }

    // This Text component will display text message named "Click to play video"
    // at the center of the application.
    Text {
        id: infoText
        text: "Click to play video"
        anchors.centerIn: parent
```

```
        color: "white"
        font.pointSize: 15
    }

    // This Text component will create a text message and display
    // video information, such as: file name, codec and resolution
    Text {
        color: "white"
        text:
            "File name: h264-hd-30.mp4\n" +
            "Codec: " + playVideo.metaData.videoCodec + "\n" +
            "Resolution: " + playVideo.metaData.resolution + "\n"
    }

    // This MediaPlayer component adds a media file and plays it
    MediaPlayer {
        id: playVideo
        source: videosDirPath + "/h264-hd-30.mp4"
        autoLoad: true
        autoPlay: false
        loops: MediaPlayer.Infinite
    }

    // This VideoOutput component is used for displaying video and camera on the monitor
    VideoOutput {
        anchors.fill: parent
        source: playVideo
    }

    // This MouseArea component starts and pauses the video.
    MouseArea {
        anchors.fill: parent
        onClicked: {
            // Disable "Click to play video" message
            // We need it to display once
            if (infoText.visible == true)
                infoText.visible = false;

            // Toggle the video when users click the application
            if (playVideo.playbackState == MediaPlayer.PlayingState) {
                playVideo.pause();
            } else {
                playVideo.play();
            }
        }
    }
}
```

Walkthrough:

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section: 4.1.2 Hello World and 4.2.1 Audio Play.

```
Text {
    color: "white"
    text:
        "File name: h264-hd-30.mp4\n" +
        "Codec: " + playVideo.metaData.videoCodec + "\n" +
        "Resolution: " + playVideo.metaData.resolution + "\n"
}
```

This `Text` item retrieves video information (such as: codec, resolution) from `metaData` group of `MediaPlayer` item (also known as `playVideo`).

```
MediaPlayer {
    id: playVideo
    source: videosDirPath + "/h264-hd-30.mp4"
    autoLoad: true
    autoPlay: false
```

```
    loops: MediaPlayer.Infinite
}

VideoOutput {
    anchors.fill: parent
    source: playVideo
}
```

In this application, the MediaPlayer is used with VideoOutput to render video.

```
MouseArea {
    anchors.fill: parent
    onClicked: {
        if (infoText.visible == true)
            infoText.visible = false;

        if (playVideo.playbackState == MediaPlayer.PlayingState) {
            playVideo.pause();
        } else {
            playVideo.play();
        }
    }
}
```

If user clicks on the `Window`, the application will hide video information (`infoText`), then pause the video (`playVideo.pause`) if it is running, otherwise, start the video (`playVideo.play`) if it is pausing.

4.2.3.2 Result

Can play MP4 file. Note that this application does not output audio.

Figure 4.5 Video Play result

4.2.3.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

4.2.3.4 Special instruction

- Download the input file at: https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/hmi-mmpoc-videos/h264-hd-30.mp4
- Please put it in this location `/home/media/videos/` (on board).

4.2.4 File Play

Play an MP4 file.

4.2.4.1 Source code

(1) Files

Application name	Description
qt-fileplay.pro	For further information, please refer to Appendix, section 2 (Qt Quick introduction).
main.cpp	
main.qml	
qml.qrc	
sintel_trailer-720p.mp4	The hard-coded input file.

Table 4.5 File Play source code

(2) main.cpp

```
#include <QtGui/QGuiApplication>
#include <QtQml/QQmlApplicationEngine>
#include <QtQml/QQmlContext>
#include <QtGui/QScreen>
#include <QDebug>

#define VIDEOSDIRPATH file:///home/media/videos


int main(int argc, char *argv[])
{
    // Initializes the window system and constructs an application object
    // with argc command line arguments in argv
    QGuiApplication app(argc, argv);

    // This class provides a convenient way to load a single QML file
    QQmlApplicationEngine engine;

    // Show information of screen (all monitors)
    // If 2 screen available, chose the larger as it is usually default
    QScreen *screen;
    if (QGuiApplication::screens().size() <= 1)
    {
        screen = QGuiApplication::primaryScreen();
    }
    else
    {
        QScreen *tmp_screen;
        if(QGuiApplication::screens().at(0) == NULL)
            screen = QGuiApplication::screens().at(1);
          else if(QGuiApplication::screens().at(1) == NULL)
            screen = QGuiApplication::screens().at(0);
          else {
            screen = QGuiApplication::screens().at(0);
            tmp_screen = QGuiApplication::screens().at(1);
            if((screen->availableSize().width() * screen->availableSize().height()) <
               (tmp_screen->availableSize().width() * tmp_screen->availableSize().height()))
                screen = QGuiApplication::screens().at(1);
          }
    }
    qDebug() << "Information for screen:" << screen->name();

    qDebug() << "  Available geometry:" << screen->availableGeometry().x()
                                        << screen->availableGeometry().y()
                                        << screen->availableGeometry().width() << "x"
                                        << screen->availableGeometry().height();
```

```
    qDebug() << "  Available size:" << screen->availableSize().width() << "x"
                                    << screen->availableSize().height();

    // Set the value of the "videosDirPath" variable which is path to videos
    QQmlContext *context = engine.rootContext();
    context->setContextProperty("videosDirPath", VIDEOSDIRPATH);

    // This function load() will load the root QML file "main.qml"
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    // Wait for Qt HelloWorld form to close, then returns a value
    // indicating a run success or failure
    return app.exec();
}
```

Walkthrough:

This file is the same as the `main.cpp` file of Video Play application

(3) main.qml

```
import QtQuick 2.5
import QtQuick.Window 2.2
import QtMultimedia 5.5

// This Window component will create a new top-level window with
// title name "Video Play (with sound)" for a Qt Quick scene
Window {
    // Scale 1/4 full screen
    property int minimized_scale: 4

    // These properties are own of the window component
    visible: true
    width: Screen.desktopAvailableWidth/minimized_scale
    height: Screen.desktopAvailableHeight/minimized_scale
    title: qsTr("Video Play (with sound)")
    color: "black"

    visibility: "Minimized"
    id: mainWindow

    Component.onCompleted: {
            mainWindow.maximumWidth = Screen.desktopAvailableWidth
            mainWindow.maximumHeight = Screen.desktopAvailableHeight
            mainWindow.minimumWidth = Screen.desktopAvailableWidth/minimized_scale
            mainWindow.minimumHeight = Screen.desktopAvailableHeight/minimized_scale
            mainWindow.showMaximized();
            console.log("Width: ", Screen.desktopAvailableWidth,
                        "  Height: ", Screen.desktopAvailableHeight)
    }


    // This Text component will display text message named "Click to play video"
    // at the center of the application.
    Text {
        id: infoText
        text: "Click to play video"
        anchors.centerIn: parent
        color: "white"
        font.pointSize: 15
    }

    // This Text component will create a text message and display
    // video information, such as: file name, codec and resolution
    Text {
        color: "white"
        text:
            "File name: sintel_trailer-720p.mp4\n" +
```

```
            "Codec: " + playVideo.metaData.videoCodec + "\n" +
            "Resolution: " + playVideo.metaData.resolution + "\n"
    }

    // This MediaPlayer component adds a media file and plays it
    MediaPlayer {
        id: playVideo
        source: videosDirPath + "/sintel_trailer-720p.mp4"
        autoLoad: true
        autoPlay: false
        loops: MediaPlayer.Infinite
    }

    // This VideoOutput component is used for displaying video on the monitor
    VideoOutput {
        anchors.fill: parent
        source: playVideo
    }

    // This MouseArea component starts and pauses the video.
    MouseArea {
        anchors.fill: parent
        onClicked: {
            // Disable "Click to play video" message
            // We need it to display once
            if (infoText.visible == true)
                infoText.visible = false;

            // Toggle the video when users click the application
            if (playVideo.playbackState == MediaPlayer.PlayingState) {
                playVideo.pause();
            } else {
                playVideo.play();
            }
        }
    }
}
```

Walkthrough:

This file is the same as the `main.qml` file of Video Play application

4.2.4.2 Result

Can play MP4 file.

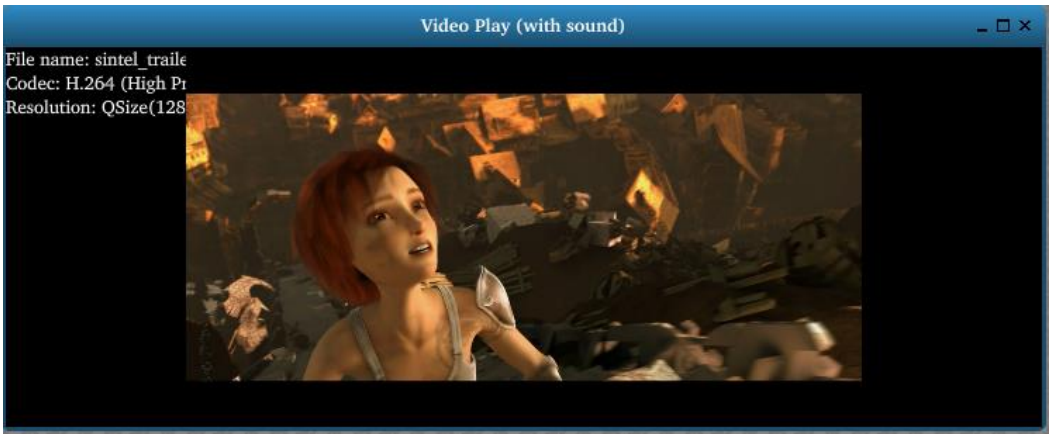


Figure 4.6 File Play result

4.2.4.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

4.2.4.4 Special instruction

- Download the input file at: https://download.blender.org/durian/trailer/sintel_trailer-720p.mp4
- Please put it in this location `/home/media/videos/` (on board).

4.2.5 Multiple Displays 1

Display 1 video simultaneously on HDMI and LCD monitors.

4.2.5.1 Source code

(1) Files

Application name	Description
main.cpp	Retrieve and pass the resolutions of HDMI and LCD monitors to `main.qml` file. Pass literal string of input file path to main.qml file.
main.qml	Instantiate `Content` item and pass it the input file and the monitor resolutions so that it can control video.
Content.qml	Instantiate `VideoItem` to pause and resume video.
VideoItem.qml	Define `VideoItem`.
Button.qml	Define UI button.

Table 4.6 Multiple Displays 1 source code

(2) Class Diagram

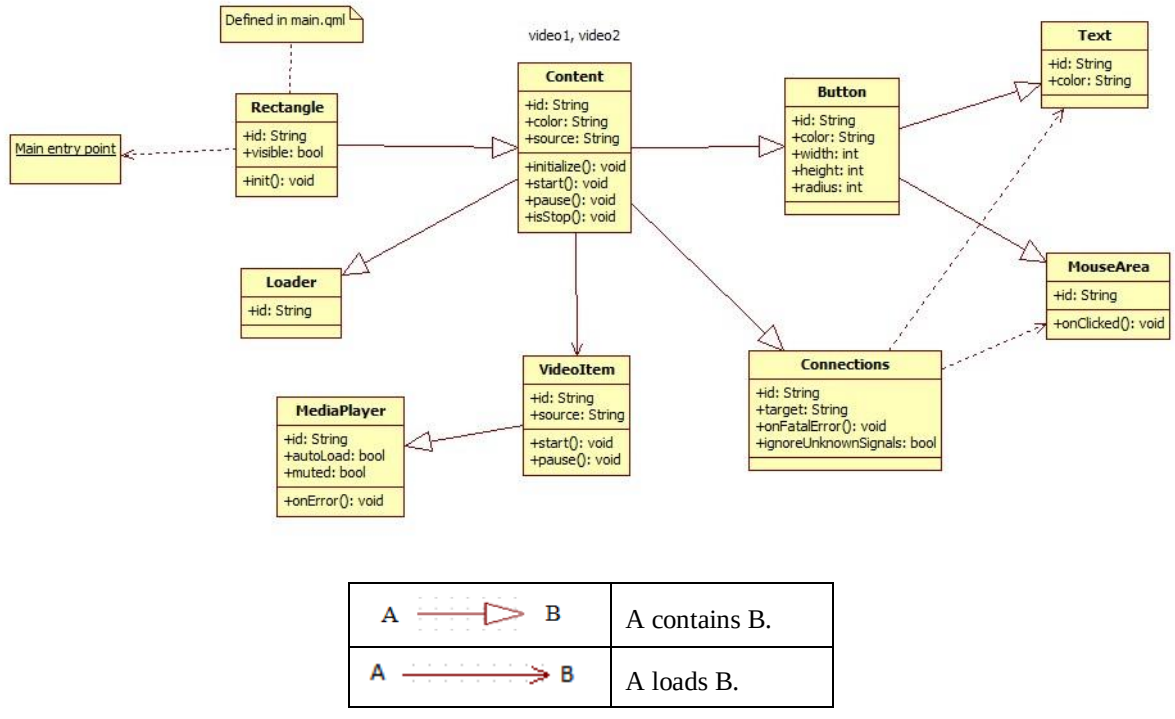


Figure 4.7 Multiple Displays 1 class diagram

(3) Flow chart

The following figure represents the workflow of *Multiple Display 1* application. When user clicks to the screen, if the video is playing, it will pause. If the video is paused, it will resume. Moreover, if the user clicks on `Exit` button, the application will exit.

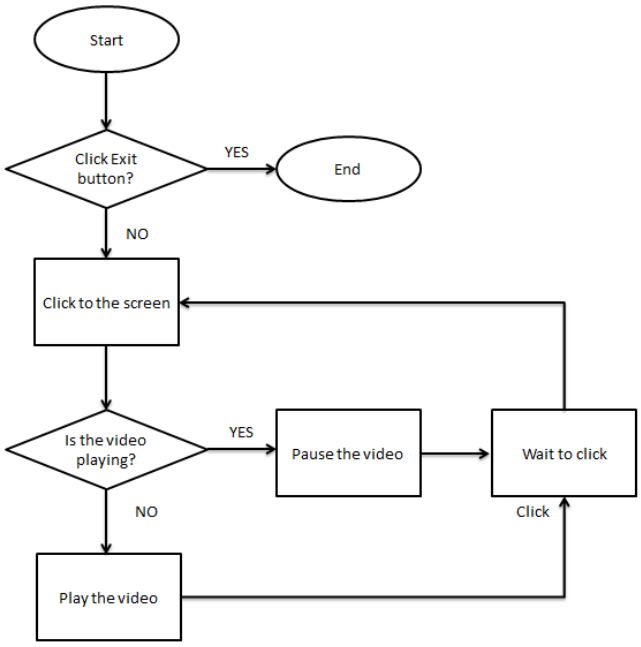


Figure 4.8 Multiple Displays 1 flow chart

(4) main.cpp

```
void dataScreens()
{
    // Classify width and height for screens
    foreach (QScreen *screen, QGuiApplication::screens()) {
        // Get screen has coordinates 0 and 0
        if ((screen->geometry().y() == 0) && (screen->geometry().x() == 0)) {
            // Get value width and height and set it for variable
             widthScreen1 = screen->geometry().width();
             heightScreen1 = screen->geometry().height();
             qDebug() << "  Geometry 1:" << screen->geometry().x() << screen->geometry().y()
                      << screen->geometry().width() << "x" << screen->geometry().height();
        } else {
            widthScreen2 = screen->geometry().width();
            heightScreen2 = screen->geometry().height();
            qDebug() << "  Geometry 2:" << screen->geometry().x() << screen->geometry().y()
                     << screen->geometry().width() << "x" << screen->geometry().height();
        }
    }
}
```

The `dataScreens()` function gets the resolutions of both HDMI and LCD monitors. The width and height of HDMI monitor will be put in `widthScreen1` and `heightScreen1` while the resolution of LCD monitor will be put in `widthScreen2` and `heightScreen2`.

```
QCommandLineParser parser;
parser.setApplicationDescription("Description: A simple application to play a video file");
parser.addHelpOption();
parser.addVersionOption();

QCommandLineOption input(QStringList() << "i" << "input",
```

```
        "Directory of input file. (Default: /home/media/videos/vga1.h264)",
        "path to video file", "/home/media/videos/vga1.h264");
parser.addOption(input);

// Process the actual command line arguments given by the user
parser.process(app);
```

The QCommandLineParser class and QCommandLineOption class provide a convenient way to define a set of options, parse the command-line arguments, and store which options have actually been used, as well as option values.

In this case, an optional argument is defined to set directory of input file. If user does not pass this optional argument, the default value /home/media/videos/vga1.h264 is used as directory of input file.

(5) main.qml

```
// Describe video on HDMI monitor
Content {
    id: video1
    color: "black"
    // Set location first screen
    x: 0
    y: 0
    source: parent.source
    // Set size first screen
    width: widthNumber1
    height: heightNumber1

    Loader {
        id: video1StartStopLoader
        onLoaded: {
            item.parent = video1
            item.anchors.fill = video1
        }
    }
    // Load startStopComponent
    onInitialized: video1StartStopLoader.sourceComponent = startStopComponent
}
```

The GUI uses Content.qml file to scale, display, pause, and resume the video.

The monitor resolution (widthNumber1 and heightNumber1, for example) is passed to the Content to scale video1. The application will display video1 at (0, 0) and video2 at (widthNumber1, 0) because the LCD monitor will always be to the right of the HDMI monitor.

The sourceComponent property holds and loads startStopComponent item which contains MouseArea::onClicked() method to pause and resume the video.

```
Component {
    id: startStopComponent

    Rectangle {
        id: root
        color: "transparent"
        // Function return the QML class is used
        function content() {
            return root.parent
        }

        Text {
            anchors {
                horizontalCenter: parent.horizontalCenter
                top: parent.top
                margins: 20
            }
            // Set content text based on status of video
            text: !(content().isStop()) ? " " : "Tap to start"
            color: "white"
        }
```

```
        MouseArea {
            anchors.fill: parent
            onClicked: {
                // Play application based on status of video
                if (content().isStop()) {
                    content().start()
                } else {
                    content().pause()
                }
            }
        }
    }
}
```

The `Component` object is used to control the video and display its status.

The `content()` function returns the `parent` item of `Rectangle` (`root`). In this application, its parents are `Content` item (which are `video1` and `video2`) because they are the only items that load the `startStopComponent` component.

The `isStop()` function checks if the video is playing or paused. When user clicks on the video, if it is playing, the application will call `Content::pause()` to pause the video, on the other hand, if it paused, the application will call `Content::play()` to resume it.

4.2.5.2 Result

Can display 1 video simultaneously on HDMI and LCD monitors.

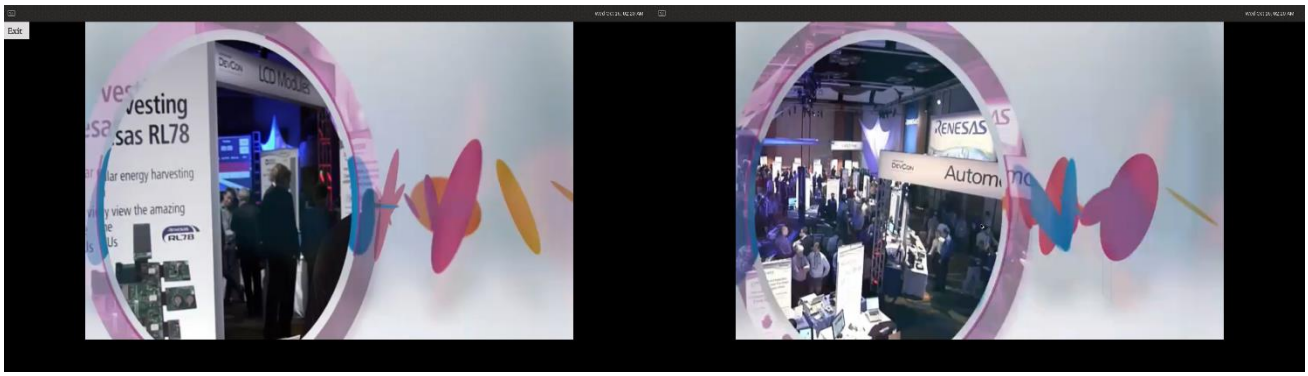


Figure 4.9 Multiple Displays 1 result

4.2.5.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

4.2.5.4 Special instruction

- Download the input file at:
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga1.h264

Note:

RZ/G2E platform supports playing 2 1920x1080, 30 fps videos simultaneously
RZ/G2H, RZ/G2M and RZ/G2N platform support playing 2 2560x1440, 30 fps videos simultaneously

4.2.6 Multiple Displays 2

Display 2 videos simultaneously on HDMI and LCD monitors.

4.2.6.1 Source code

(1) Files

Application name	Description
main.cpp	For further information, please refer to Multiple Displays 1 application
main.qml	
Content.qml	
VideoItem.qml	
Button.qml	

Table 4.7 Multiple Displays 2 source code

(2) Class diagram

This class diagram is the same as Multiple Displays 1's.

(3) Flow chart

This flow chart is the same as Multiple Displays 1's.

(4) main.qml

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 4.2.4 Multiple Displays 1 and 4.2.2 Video Play.

```
// Set value of source file
source1 = "file://" + videosDirPath1
source2 = "file://" + videosDirPath2
// Display video1 to HDMI monitor
Content {
    id: video1
    x: 0
    y: 0
    source: parent.source1
    width: widthNumber1
    height: heightNumber1
}

// Display video2 to LCD monitor
Content {
    id: video2
    x: widthNumber1
    y: 0
    source: parent.source1
    width: widthNumber1
    height: heightNumber1
}
```

Unlike Multiple Displays 1, this program will display 2 different video files (`source1` and `source2`), one on `video1` (HDMI monitor), the other on `video2` (LCD monitor).

4.2.6.2 Result

This application will play different videos on both HDMI screen and LVDS screen.

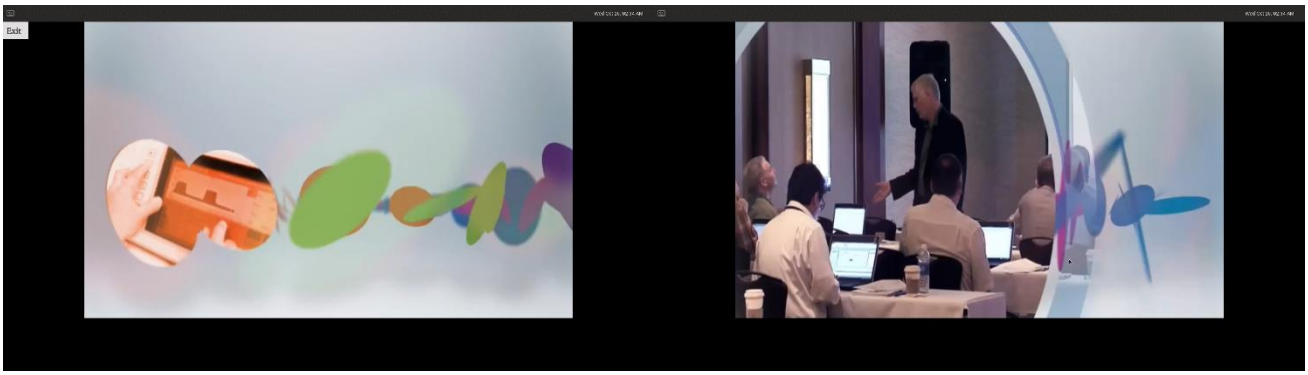


Figure 4.10 Multiple Displays 2 result

4.2.6.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

4.2.6.4 Special instruction

- Download file vga1.h264 at:
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga1.h264
- Download file vga2.h264 at:
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga2.h264

Note:

RZ/G2E platform supports playing 2 1920x1080, 30 fps videos simultaneously
RZ/G2H, RZ/G2M and RZ/G2N platform support playing 2 2560x1440, 30 fps videos simultaneously

4.2.7 Overlapped Display

Display 3 overlapping videos.

4.2.7.1 Source code

(1) Files

Application name	Description
main.cpp	Find and print out the largest monitor resolution. Maximize the application. Pass literal string of input file path to main.qml file.
main.qml	Instantiate `Content` item and pass it the input file so that it can control video.
Content.qml	Instantiate `VideoItem` to pause and resume video.
VideoItem.qml	Define `VideoItem`.

Table 4.8 Overlapped Display source code

(2) Class diagram

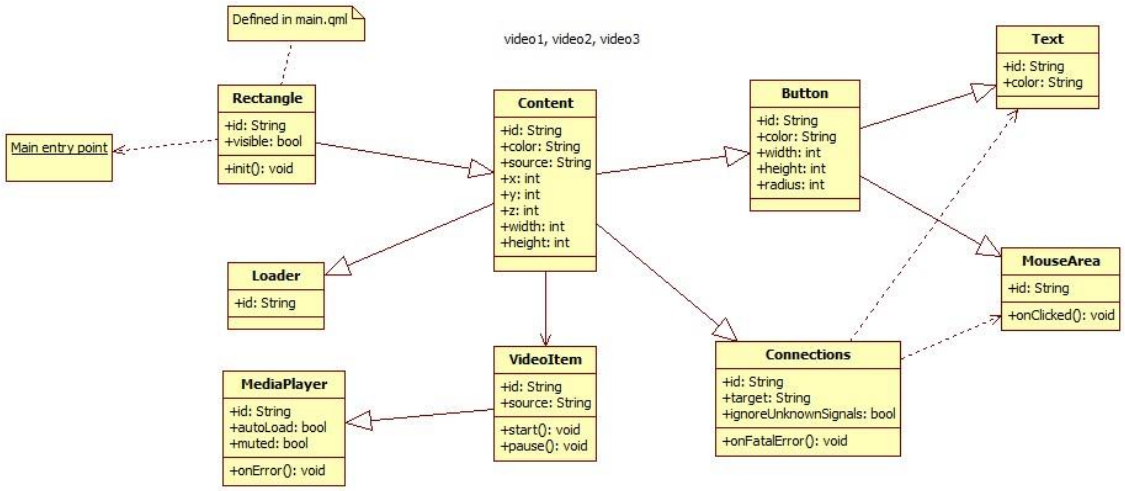


Figure 4.11 Overlapped Display class diagram

(3) Flow chart

The following figure represents the workflow of *Overlapped Display* application. When the user clicks to the screen, if the video is playing, it will be put on top. On the other hand, if the video is paused, it will resume and then be put on top.

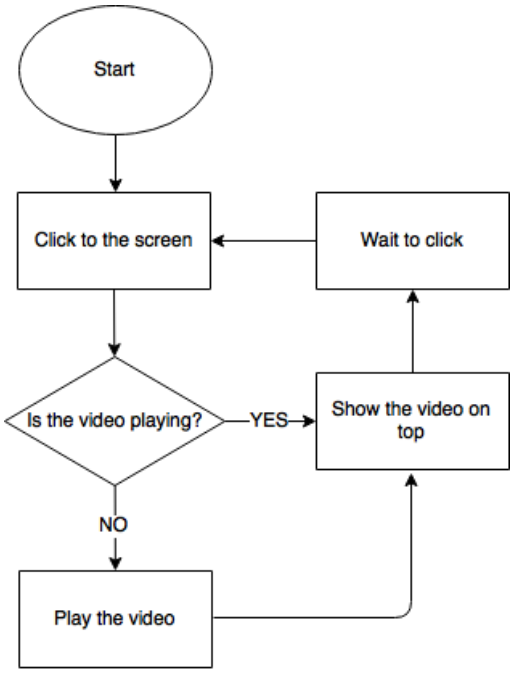


Figure 4.12 Overlapped Display flow chart

(4) main.qml

Note that this tutorial only discusses the important points of this application. For the rest of source code, please refer to section 4.2.4 Multiple Displays 1 and 4.2.5 Multiple Displays 2.

```
Content {
    id: video3
    color: "black"
    // Set location and order third screen
    x: position3
    y: position3
    z: low
    source:  parent.source2
    // Set size third screen
    width: widthVideo
    height: heightVideo

    Loader {
        id: video3StartStopLoader
        onLoaded: {
            item.parent = video3
            item.anchors.fill = video3
        }
    }
    // Load startStopComponent
    onInitialized: video3StartStopLoader.sourceComponent = startStopComponent
}
```

Unlike Multiple Displays 1 and Multiple Displays 2 application, the program includes new `Content` item known as `video3`. Note that `video1` and `video2` controls the same source (the first input), while `video3` controls the other source.

The `video3` will resize the video to resolution `500x300`, put it at location (`100`, `100`) at start up. It also uses `z` property to control its stacking order. By default, the `z` property is set to `0` (`low`) to display it under `video1` and `video2`.

```
drag.target: content()
```

User can move the videos using this statement.

```
onClicked: {
    // Change the video display
    switch(content()) {
        case video1: video1.z=high; video2.z=low; video3.z=low; break;
        case video2: video1.z=low; video2.z=high; video3.z=low; break;
        case video3: video1.z=low; video2.z=low; video3.z=high; break;
    }
    // Play video
    if (content().isStop())
        content().start()
}
```

When user clicks on the video, the application will display it on top while the other videos are displayed under it. Moreover, the video will resume, if it paused.

4.2.7.2 Result

Can display 3 overlapping videos on the main screen.

4.2.7.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

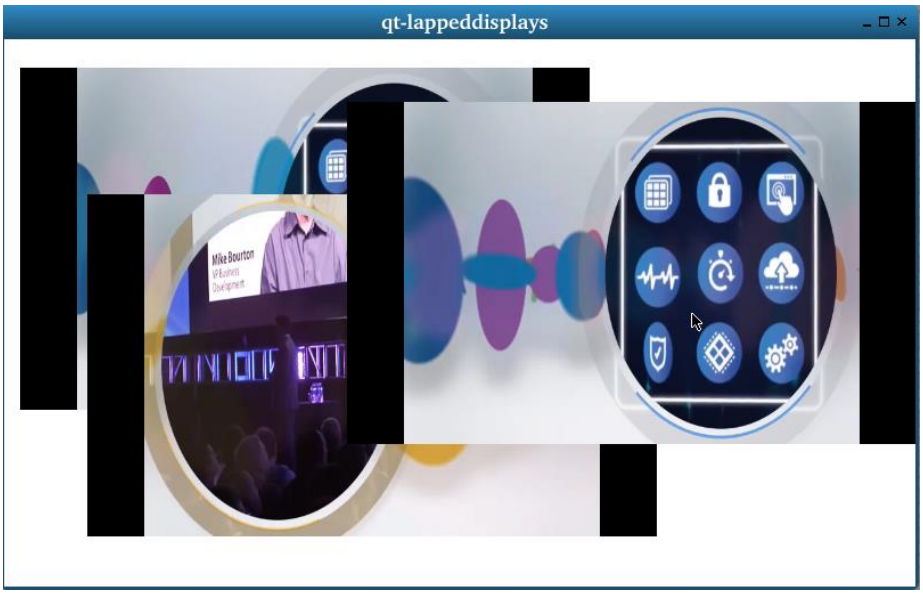


Figure 4.13 Overlapped Display result

4.2.7.4 Special instruction

- Download file vga1.h264 at:
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga1.h264
- Download file vga2.h264 at:
  https://www.renesas.com/jp/ja/img/products/media/auto-j/microcontrollers-microprocessors/rz/rzg/doorphone-videos/vga2.h264
- Please put it at location `/home/media/videos/` (on board).

Note:

RZ/G2E platform supports playing 3 1280x720, 30 fps videos simultaneously
RZ/G2H, RZ/G2M and RZ/G2N platform support playing 3 1920x1080, 30 fps videos simultaneously

4.2.8 3D Graphics

Display and automatically rotate 3D color shapes, such as: cube, sphere, and cone.

4.2.8.1 Source code

(1) Files

Application name	Description
cone.h	Declare specific behaviors of `Cone` class.
cone.cpp	Implement `Cone` class.
cube.h	Declare specific behaviors of `Cube` shape.
cube.cpp	Implement `Cube` class.
sphere.h	Declare specific behaviors of `Sphere` shape.
sphere.cpp	Implement `Sphere` class.
vertex.h	Declare `Vertex` class which contain a 3D position and an RBG color
vertex.cpp	Implement `Vertex` class.
fshader.glsl vshader.glsl	Written in GLSL language (a C-style language)[3]. OpenGL ES 2.0[4] uses the programmable pipeline which contains series of stages to render objects to the monitor. There are 2 stages that are programmable: Vertex Shader and Fragment Shader. File “vshader.glsl” will transform 3D vertices of objects and display them on 2D screen. File “fshader.glsl” will apply colors to vertices.
figure.h	Declare common behaviors of 3D figures in `Figure` class.
figure.cpp	Implement `Figure` abstract class.
main.cpp	The program’s main entry point.
program.h	Declare `Program` class.
program.cpp	Implement `Program` class.

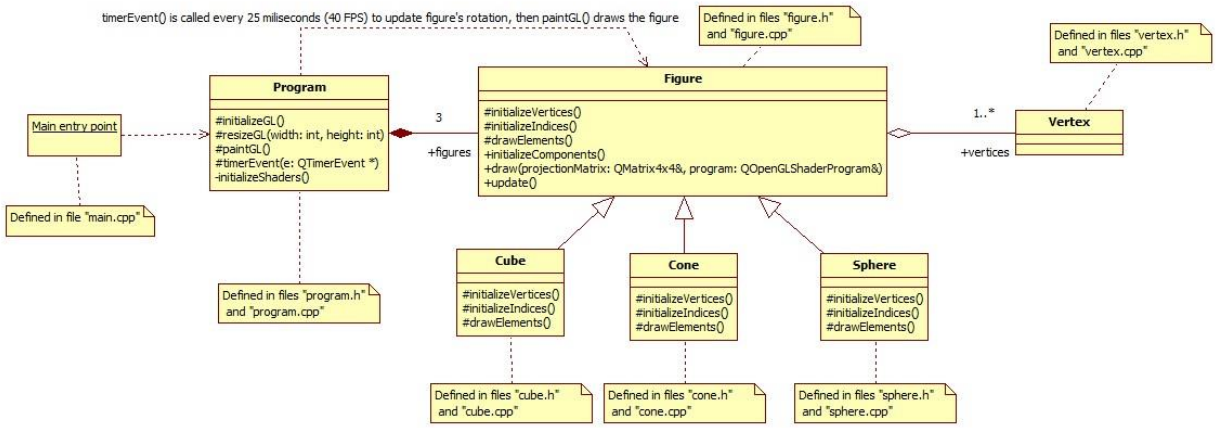
Table 4.9 3D Graphics source code

[3] https://en.wikipedia.org/wiki/OpenGL_Shading_Language

[4] https://en.wikibooks.org/wiki/GLSL_Programming/OpenGL_ES_2.0_Pipeline

(2) Class diagram

The `main()` function displays a GUI form (defined by `Program` class). Next, it creates 3 `Figure` instances and initializes OpenGL instances. Finally, it listens for resize event to zoom in, out, and asks the `Figures` to rotate and redraw them every `25 ms`.



Vertex	Class.
Defined in files "vertex.h" and "vertex.cpp"	Comments.
Main entry point	The main() function.
A 1..* +vertices B	A contains one or more B instances.
A 3 +figures B	A contains exact three B instances. When A destroys, B instances will be destroyed.
A B	A depends on B.
A B	A inherits from B.

Figure 4.14 3D Graphics class diagram

(3) Flow chart

The following figure represents the workflow of *3D Graphics* application. For simplicity, this diagram only shows the cube shape. Other shapes, such as: cone and sphere shape are similar.

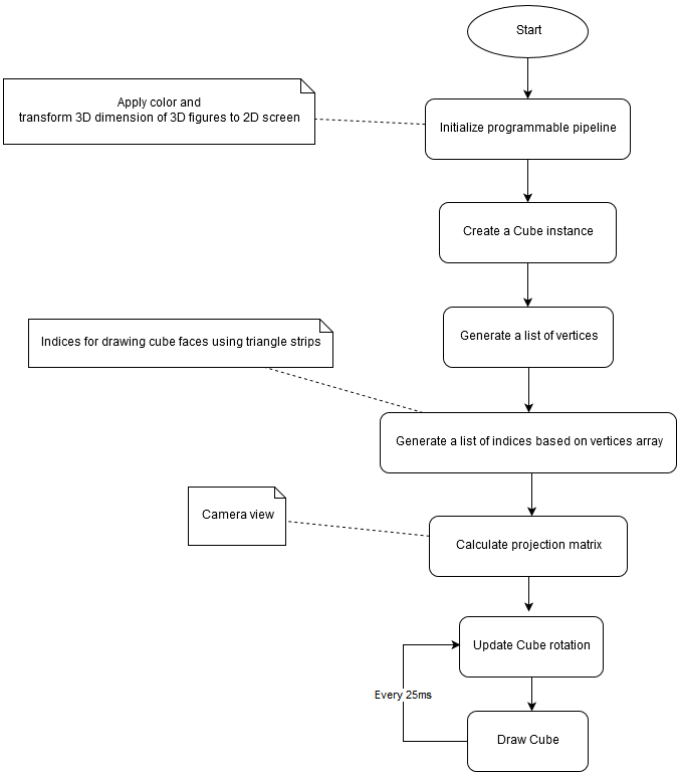


Figure 4.15 Cube shape workflow

The following table represents the application's workflow when user resizes its window.

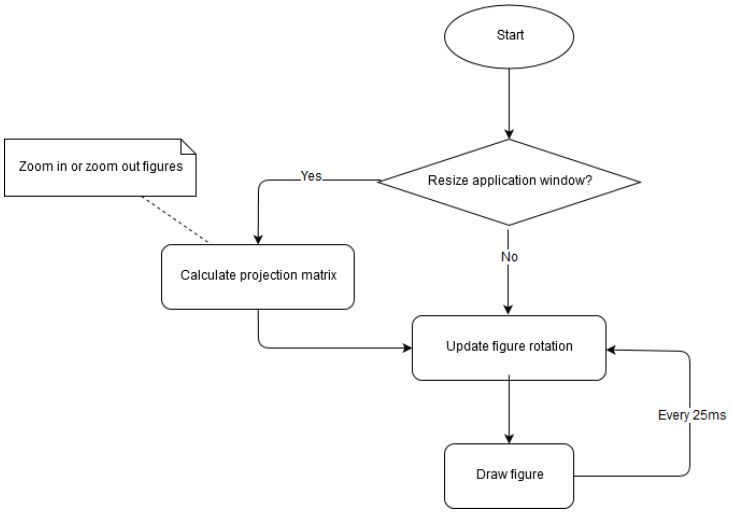


Figure 4.16 3D Graphics workflow

Note:

Please note that these figures do not have terminal symbols because this program will display and rotate 3D figures until user turns it off.

(4) program.cpp

```
void Program::resizeGL(int width, int height)
{
    // Calculate aspect ratio
    qreal aspect = qreal(width) / qreal(height ? height : 1);

    // Set near plane to 3.0, far plane to 20.0, field of view 45 degrees
    const qreal zNear = 2.0, zFar = 20.0, fov = 45.0;

    // Reset projection
    mProjection.setToIdentity();

    // Set perspective projection
    mProjection.perspective(fov, aspect, zNear, zFar);
}
```

The `resizeGL()` method calculates perspective projection matrix (camera view) based on:

- Resolution ratio of the window.
- Near and far plain. Note that outside of `zNear` and `zFar` range, object will not be visible.
- Field of view (`fov`) (the amount of "zoom"). The value is usually between 90 degree (extra wide) and 30 degree (quite zoomed in).

```
void Program::initializeShaders()
{
    // Compile vertex shader
    if (!mProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/vshader.glsl"))
        close();

    // Compile fragment shader
    if (!mProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/fshader.glsl"))
        close();

    // Link shader pipeline
    if (!mProgram.link())
        close();

    // Bind shader pipeline for use
    if (!mProgram.bind())
        close();
}
```

OpenGL ES 2.0 requires the application to create programmable pipeline by itself. There are two stages that it needs to re-implement, that are: Vertex Shader and Fragment Shader. Each stage is implemented in `vshader.glsl` and `fshader.glsl`. After that, they need to be compiled and linked so that the application can pass data to GPU.

(5) figure.cpp

```
void Figure::initializeComponents()
{
    // Initialize vertices
    initializeVertices();

    // Initialize indices
    initializeIndices();

    // Bind vertices and indices to VBOs
    mVerticesBuffer.bind();
    mVerticesBuffer.allocate(&mVertices[0], (int)mVertices.size() * sizeof(Vertex));

    mIndicesBuffer.bind();
    mIndicesBuffer.allocate(&mIndices[0], (int)mIndices.size() * sizeof(GLuint));
}
```

Both vertices and indices after being generated should be added to VBOs (Vertex buffer objects) for better performance when rendering. Before adding data or using VBOs, the application must inform OpenGL via `bind()`.

```
void Figure::draw(QMatrix4x4& projectionMatrix, QOpenGLShaderProgram& program)
{
    // Tell OpenGL which VBOs to use
    mVerticesBuffer.bind();
    mIndicesBuffer.bind();

    // Tell OpenGL programmable pipeline how to locate vertex position data
    int positionAddr = program.attributeLocation("a_position");
    program.enableAttributeArray(positionAddr);
    program.setAttributeBuffer(positionAddr, GL_FLOAT, Vertex::getPositionOffset(),
                               Vertex::POSITION_TYPLE_SIZE, Vertex::getVertexClassStride());

    // Tell OpenGL programmable pipeline how to locate color data
    int colorAddr = program.attributeLocation("a_color");
    program.enableAttributeArray(colorAddr);
    program.setAttributeBuffer(colorAddr, GL_FLOAT, Vertex::getColorOffset(),
                               Vertex::COLOR_TYPLE_SIZE, Vertex::getVertexClassStride());
```

Before drawing, the application must explicitly imply which vertices and indices VBOs that OpenGL needs to use.

Next, the data (such as: vertex position, vertex color) must assign to `a_position` and `a_color` variables defined in `vshader.glsl` for Vertex Shader stage.

```
    // Pass mvp matrix to OpenGL programmable pipeline
    QMatrix4x4 modelMatrix;
    modelMatrix.translate(mCenterVertex);
    modelMatrix.rotate(mRotation);
    modelMatrix.scale(mScaleAxis);

   program.setUniformValue("mvp_matrix", projectionMatrix * modelMatrix);
```

The `QMatrix4x4` class represents transformation matrix (which includes translation, rotation, and scale) in 3D space.

The information, such as: rotation, scale, transformation, and camera view will be calculated (via matrix multiplications) and inputted to `mvp_matrix` variable defined in `vshader.glsl` to correctly display objects.

```
void Figure::update()
{
    // Update transformation of figure
    mRotation = QQuaternion::fromAxisAndAngle(mRotationAxis, mAngularSpeed) * mRotation;
}
```

The `Quaternion` class represents rotation in 3D space. It consists of a 3D rotation axis specified by the x, y, and z coordinates, and a scalar representing the rotation angle.

The `update()` method will re-calculate figure's rotation based on rotation axis and rotation speed.

(6) cube.cpp

```
void Cube::initializeVertices()
{
    // Create vertices for cube
    Vertex verticesArr[] = {
        // Red face 1
        {QVector3D(-1.0f, -1.0f,  1.0f), QVector3D{ 1.0f, 0.0f, 0.0f }}, // v0
        {QVector3D( 1.0f, -1.0f,  1.0f), QVector3D{ 1.0f, 0.0f, 0.0f }}, // v1
        {QVector3D(-1.0f,  1.0f,  1.0f), QVector3D{ 1.0f, 0.0f, 0.0f }}, // v2
        {QVector3D( 1.0f,  1.0f,  1.0f), QVector3D{ 1.0f, 0.0f, 0.0f }}, // v3

        // Green face 2
        {QVector3D( 1.0f, -1.0f,  1.0f), QVector3D{ 0.0f, 1.0f, 0.0f }}, // v4
        {QVector3D( 1.0f, -1.0f, -1.0f), QVector3D{ 0.0f, 1.0f, 0.0f }}, // v5
        {QVector3D( 1.0f,  1.0f,  1.0f), QVector3D{ 0.0f, 1.0f, 0.0f }}, // v6
        {QVector3D( 1.0f,  1.0f, -1.0f), QVector3D{ 0.0f, 1.0f, 0.0f }}, // v7

        // Blue face 3
        {QVector3D( 1.0f, -1.0f, -1.0f), QVector3D{ 0.0f, 0.0f, 1.0f }}, // v8
```

```
        {QVector3D(-1.0f, -1.0f, -1.0f), QVector3D{ 0.0f, 0.0f, 1.0f }}, // v9
        {QVector3D( 1.0f,  1.0f, -1.0f), QVector3D{ 0.0f, 0.0f, 1.0f }}, // v10
        {QVector3D(-1.0f,  1.0f, -1.0f), QVector3D{ 0.0f, 0.0f, 1.0f }}, // v11

        // Yellow face 4
        {QVector3D(-1.0f, -1.0f, -1.0f), QVector3D{ 1.0f, 1.0f, 0.0f }}, // v12
        {QVector3D(-1.0f, -1.0f,  1.0f), QVector3D{ 1.0f, 1.0f, 0.0f }}, // v13
        {QVector3D(-1.0f,  1.0f, -1.0f), QVector3D{ 1.0f, 1.0f, 0.0f }}, // v14
        {QVector3D(-1.0f,  1.0f,  1.0f), QVector3D{ 1.0f, 1.0f, 0.0f }}, // v15

        // Pink face 5
        {QVector3D(-1.0f, -1.0f, -1.0f), QVector3D{ 1.0f, 0.0f, 1.0f }}, // v16
        {QVector3D( 1.0f, -1.0f, -1.0f), QVector3D{ 1.0f, 0.0f, 1.0f }}, // v17
        {QVector3D(-1.0f, -1.0f,  1.0f), QVector3D{ 1.0f, 0.0f, 1.0f }}, // v18
        {QVector3D( 1.0f, -1.0f,  1.0f), QVector3D{ 1.0f, 0.0f, 1.0f }}, // v19

        // Light blue face 6
        {QVector3D(-1.0f,  1.0f,  1.0f), QVector3D{ 0.0f, 1.0f, 1.0f }}, // v20
        {QVector3D( 1.0f,  1.0f,  1.0f), QVector3D{ 0.0f, 1.0f, 1.0f }}, // v21
        {QVector3D(-1.0f,  1.0f, -1.0f), QVector3D{ 0.0f, 1.0f, 1.0f }}, // v22
        {QVector3D( 1.0f,  1.0f, -1.0f), QVector3D{ 0.0f, 1.0f, 1.0f }}  // v23
    };

    for (size_t index = 0; index < sizeof(verticesArr) / sizeof(Vertex); index++) {
        mVertices.push_back(verticesArr[index]);
    }
}
```

This method generates normalized vertices (`verticesArr`). Every figure that inherits from `Figure` class must implement this method. For further information, please refer to Appendix section.

```
void Cube::initializeIndices()
{
    // Indices for drawing cube faces using triangle strips.
    GLuint indicesArr[] = {
        0,  1,  2,  3,  3,      // Red face 1
        4,  4,  5,  6,  7,  7,  // Green face 2
        8,  8,  9, 10, 11, 11,  // Blue face 3
        12, 12, 13, 14, 15, 15, // Yellow face 4
        16, 16, 17, 18, 19, 19, // Pink face 5
        20, 20, 21, 22, 23      // Light blue face 6
    };

    for (size_t index = 0; index < sizeof(indicesArr) / sizeof(GLuint); index++)
        mIndices.push_back(indicesArr[index]);
}
```

This method generates indices array (`indicesArr`) based on `verticesArr`. Every figure that inherits from `Figure` class must implement this method. For further information, please refer to Appendix section.

```
void Cube::drawElements()
{
    glDrawElements(GL_TRIANGLE_STRIP, (GLsizei)mIndices.size(), GL_UNSIGNED_INT, 0);
}
```

This method is used to draw `Cube` figure. Every figure that inherits from `Figure` class must implement this method.

Based on vertices array and indices array, the application can draw 3D figures using triangle strips (`GL_TRIANGLE_STRIP`), triangle fan (`GL_TRIANGLE_FAN`), or triangles (`GL_TRIANGLES`).

Note:

Other 3D figures, such as: `Sphere` and `Cone` have different mathematical properties than `Cube`, we must use different approaches to generate vertices and indices (from vertices array). For further information, please refer to Appendix section.

4.2.8.2 Result

Display and automatically rotate 3D color shapes, such as: cube, sphere, and cone.

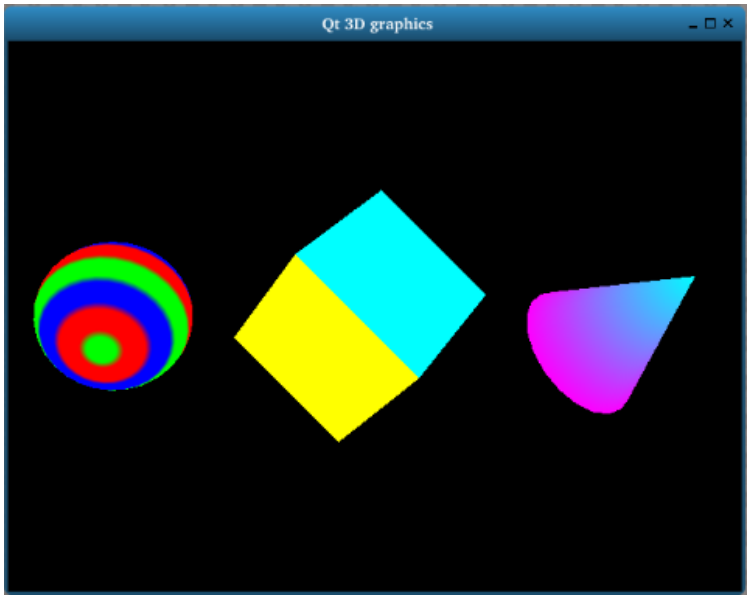


Figure 4.17 3D Graphics result

4.2.8.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

4.2.9 3D Graphics (with textures)

Display and automatically rotate 3D textured shapes, such as: cube, sphere, and cone.

4.2.9.1 Source code

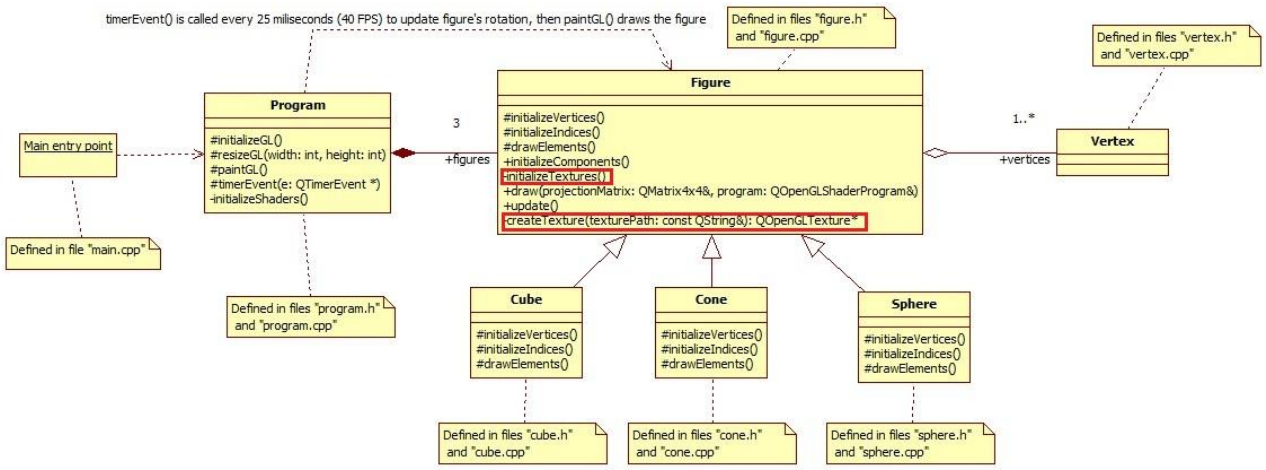
(1) Files

Application name	Description
cone.h	For further information, please refer to 3D Graphics application.
cone.cpp	
cube.h	
cube.cpp	
figure.h	
figure.cpp	
main.cpp	
program.h	
program.cpp	
sphere.h	
sphere.cpp	
fshader.glsl vshader.glsl	
vertex.h	Declare `Vertex` class which contains a 3D position and a texture 2D coordinate.
vertex.cpp	Implements `Vertex` class.
cube_face*.jpg	These are the textures of the cube shape.
sphere.jpg	These are the textures of the sphere shape.
cone_face*.jpg	These are the textures of the cone shape.

Table 4.10 3D Graphics (with textures) source code

(2) Class diagram

The `main()` function displays a GUI form (defined by `Program` class). Next, it creates 3 `Figure` instances and initializes OpenGL ES instances. Finally, it listens for resize event to zoom in, out, and asks the `Figures` to rotate and re-draw them every `25 ms`.



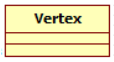
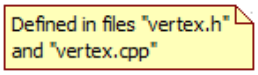
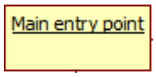
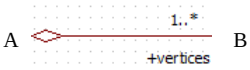
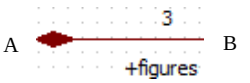
	Class.
	Comments.
	The main() function.
	A contains one or more B instances.
	A contains exact three B instances. When A destroys, B instances will be destroyed.
	A depends on B.
	A inherits from B.

Figure 4.18 3D Graphics (with textures) class diagram

(3) Workflow diagram

The following figure represents the workflow of *3D Graphics (with texture)* application. For simplicity, this diagram only shows the cube shape. Other shapes, such as: cone and sphere shape are similar.

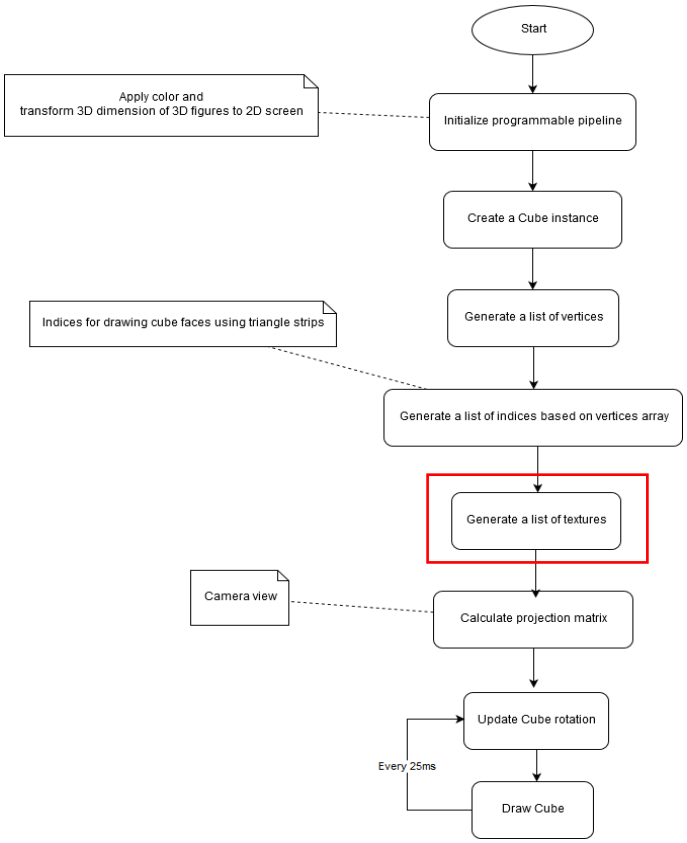


Figure 4.19 Cube shape workflow

Note:

For the application's workflow, please refer to 3D Graphics application.
Please note that the above figure does not have terminal symbols because this application will display, rotate and apply textures for 3D figures until the user turn it off.

(4) figure.cpp

```
// Initialize variable texture in Fragment Stage
program.setUniformValue("texture", 0);
```

The function sets value `0` to initialize `texture` variable in `fshader.glsl`.

```
QOpenGLTexture* Figure::createTexture(const QString& texturePath)
{
    // Instantiate object
    QOpenGLTexture* texture = NULL;

    // Load image
    texture = new QOpenGLTexture(QImage(texturePath).mirrored(true, true));

    // Set nearest filtering mode for texture minification
    texture->setMinificationFilter(QOpenGLTexture::Nearest);
```

```
    // Set bilinear filtering mode for texture magnification
    texture->setMagnificationFilter(QOpenGLTexture::Linear);

    // Wrap texture coordinates by repeating
    // f.ex. texture coordinate (1.1, 1.2) is same as (0.1, 0.2)
    texture->setWrapMode(QOpenGLTexture::Repeat);

    return texture;
}
```

This method generates a `QOpenGLTexture` object from a `texturePath`:

- The output contains an image mirrored in both horizontal and vertical direction.
- The image might lose details when it's minimized if we set minification filter to `nearest` but the application will gain the performance.
- In the normal use case, the image might be blocky and jagged when it's maximized too much. Set magnification filter to `bilinear` will solve this issue.

```
void Figure::initializeTextures()
{
    for (size_t index = 0; index < mTextures.size(); index++)
        mTextures[index] = Figure::createTexture(mTexturePaths[index]);
}
```

This method uses `Figure::createTexture()` to creates a list of `QOpenGLTexture` objects.

(5) cube.cpp

```
void Cube::drawElements()
{
    const unsigned int cubeFaces = 6U;
    const unsigned int verticesPerFace = 4U;

    for (unsigned int index = 0; index < cubeFaces; index++) {
        mTextures[index]->bind();
        glDrawElements(GL_TRIANGLE_STRIP, verticesPerFace, GL_UNSIGNED_INT,
                       (const GLvoid*)(index * verticesPerFace * sizeof(GLuint)));
    }
}
```

This `drawElements()` method is used to draw the cube shape. Every figure that inherits from `Figure` class must implement this method.

Note that the application uses `bind()` method to apply the texture to vertices that are going to be drawn.

4.2.9.2 Result

Can display and automatically rotate 3D textured shapes, such as: cube, sphere, and cone.

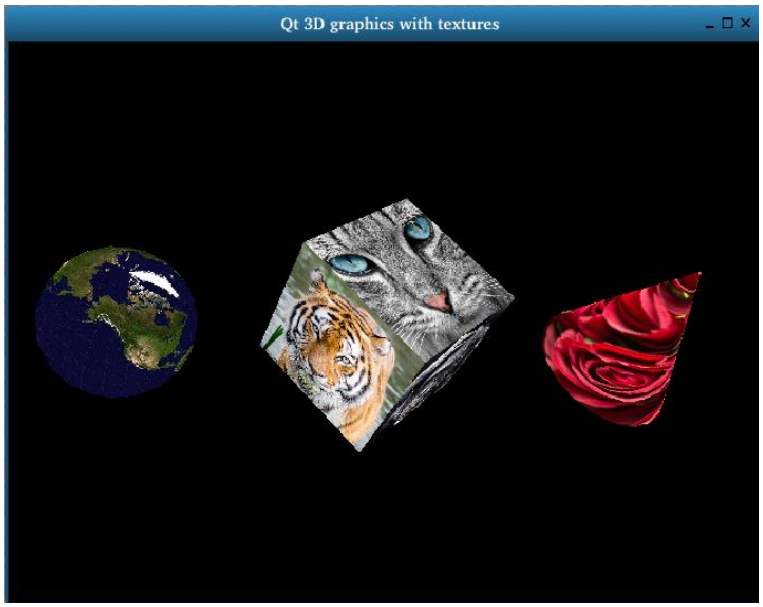


Figure 4.20 3D Graphics (with textures) result

4.2.9.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual uploaded.

4.2.10 Multimedia Player

A simple GUI multimedia player the can play videos (MP4), audios (WAV, Ogg/Vorbis, MP3), and display images (JPG, PNG, GIF, BMP)

It also shows how native code can be combined with QML (Qt Meta Language or Qt Modeling Language)[5] to implement advanced functions – in this case, C++ code is used to open media file.

4.2.11.1 Source code

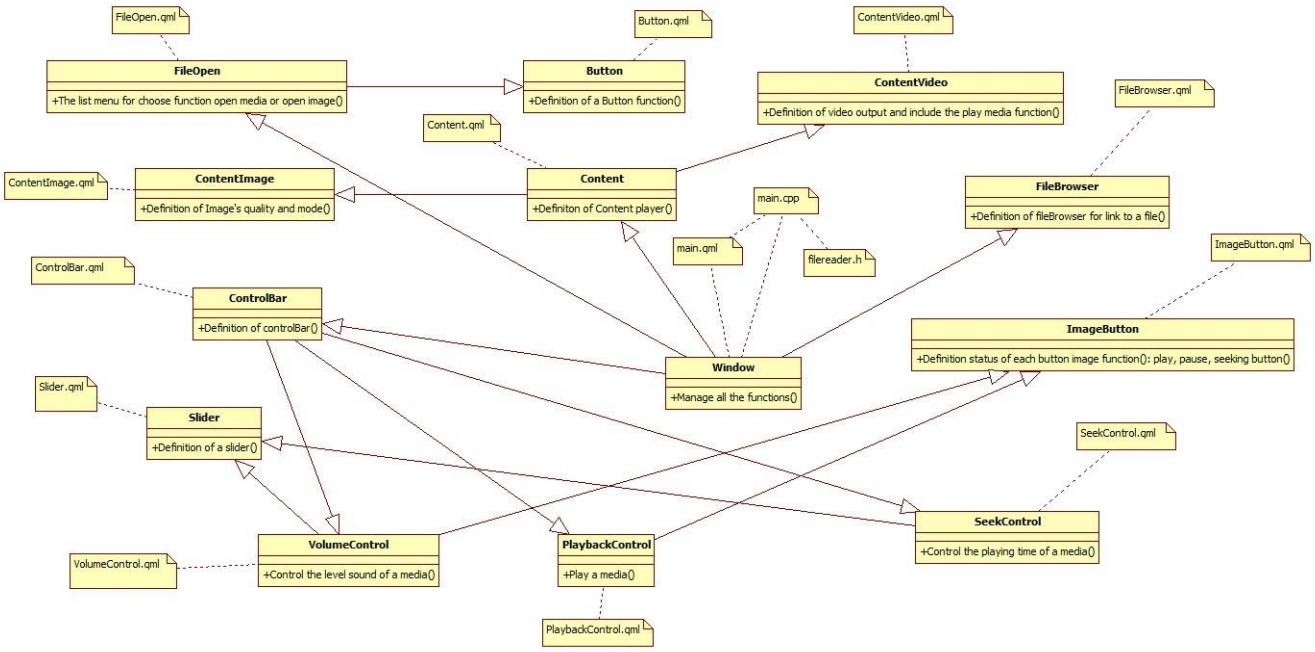
(1) Files

Application name	Description
qt-multimediaplayer.pro	This is the main project file. It contains information of the project, such as: sources, headers, resource files...
qml.qrc	Contain resources (such as: images, text files...) for this application.
*.qml	Contain source code and GUI: - Button.qml: Define text-only button. - Content.qml: Define `Content` item to control media files. - ContentImage:qml: Define `ContentImage` item to control input pictures. - ContentVideo.qml: Define `ContentVideo` item to control input audios/videos. - ControlBar.qml: Contain a play button and a seek button. - FileBrowser.qml: Represent a browser to view and select media files. - FileOpen.qml: Represent a dialog to open media files. - ImageButton.qml: Define image button. - main.qml: It is the main GUI of this application. - PlaybackControl.qml: Define play button. - SeekControl.qml: Define seek button. - Slider.qml: Define slider. - VolumeControl.qml: Control the volume.
filereader.h	Declare `FileReader` class.
main.cpp	The program's main entry point.
*.png	The application's resources.

[5] http://doc.qt.io/qt-4.8/qdeclarativeintroduction.html

Table 4.11 List of Qt source codes

(2) Class diagram



Class	Class	→	Inheritance arrow		File	-----	Link note

Figure 4.21 Multimedia Player class diagram

(3) Flow chart

The following figure represents the work flow of the *Multimedia Player* application.

First, the user clicks on the `Menu` icon to open media (MP4, WAV, Ogg/Vorbis) or image (JPEG).

- If the user clicks on the `Open image` button, the application will open a file browser. User can search for image files. Finally, they can press `Open` button to display the image.
- If the user clicks on the `Quit` button, the program will be closed.
- If the user clicks on the `Open media` button, the program will open a file browser. User can search for audio/video files. Finally, they can press `Open` button to play audio/video.

Next, when playing audio/video, the application will display information, such as: volume, playback state, playback position, and duration. User can use slider to adjust the volume and change playback position. They can also click on buttons to pause, stop, or resume the media file.

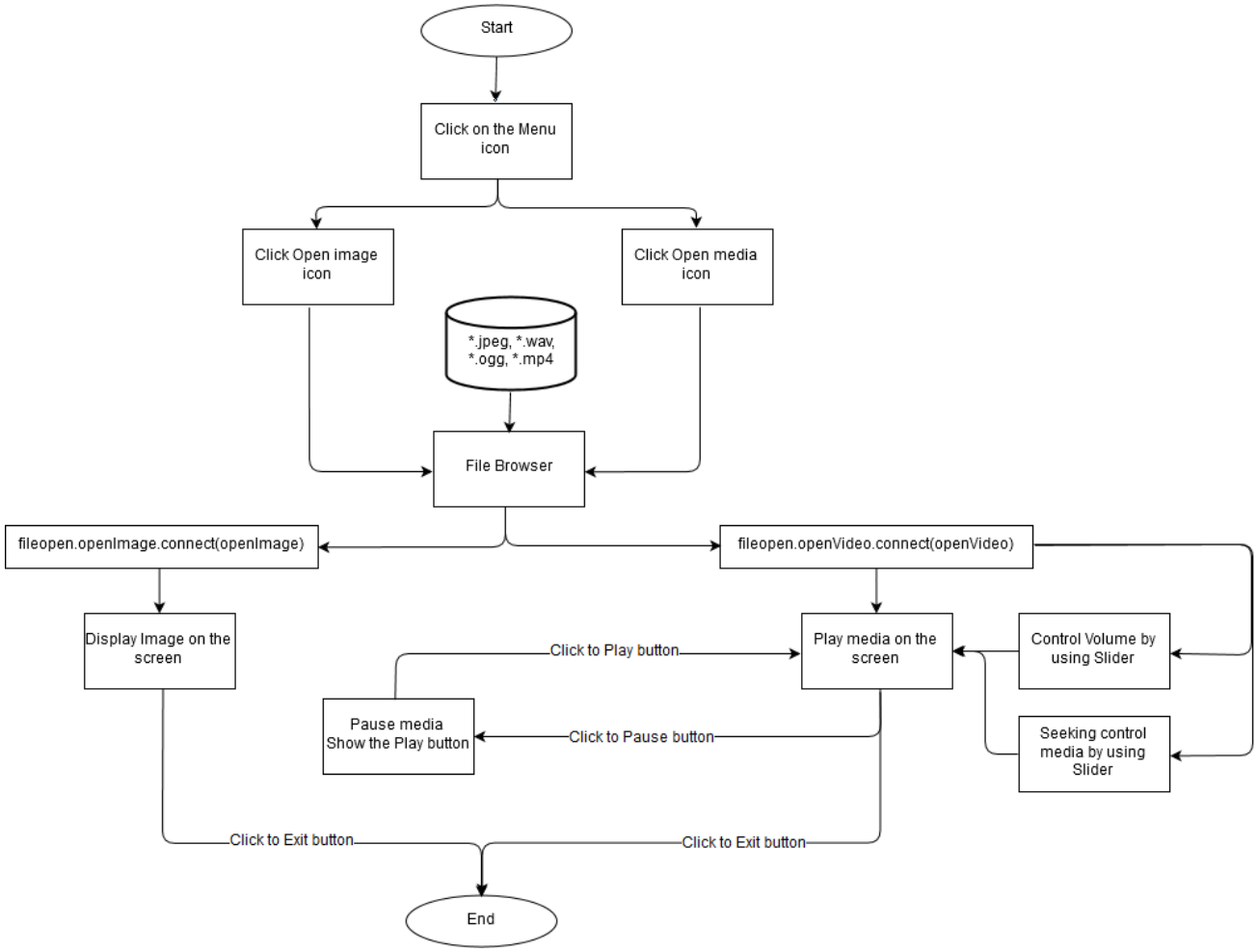


Figure 4.22 Multimedia Player flow chart

(4) FileOpen.qml

We define `Column` item as below to create `Menu` button:

```
// Create a column includes the buttons such as "Open media", "Open image", "Exit"
Column {
    anchors {
        top: menuField.bottom
        right: parent.right
        left: parent.left
        bottom: parent.bottom
        topMargin: (itemHeight / 6) * scaleh
    }

    spacing: 10 * scaleh
    visible: fileOpen.state == "expanded"

    Rectangle {
        width: itemWidth * scalew
        height: 1 * scaleh
        color: "#788ac5"
        anchors.left: parent.left
    }

    Button {
        Text {
            text: "Open media"
            font.pointSize: itemHeight * 0.4 * scaleh
            color: "#ffffff"
            anchors.verticalCenter: parent.verticalCenter
            anchors.horizontalCenter: parent.horizontalCenter
        }
        height: itemHeight  * scaleh
        width: 2 * itemWidth * scalew
        color: "#788ac5"
        onClicked: {
            fileOpen.state = "collapsed"
            root.openVideo()
        }
    }

   Rectangle {
        width: itemWidth * scalew
        height: 1 * scaleh
        color: "#788ac5"
        anchors.left: parent.left
    }

    Button {
        Text {
            text: "Open image"
            font.pointSize: itemHeight * 0.4 * scaleh
            color: "#ffffff"
            anchors.verticalCenter: parent.verticalCenter
            anchors.horizontalCenter: parent.horizontalCenter
        }
        height: itemHeight * scaleh
        width: 2 * itemWidth * scalew
        color: "#788ac5"
        onClicked: {
            fileOpen.state = "collapsed"
            root.openImage()
        }
    }

    Rectangle {
        width: itemWidth * scalew
        height: 1 * scaleh
        color: "#788ac5"
        anchors.left: parent.left
```

```
    }

    Button {
        Text {
            text: "Exit"
            font.pointSize: itemHeight * 0.4 * scaleh
            color: "#ffffff"
            anchors.verticalCenter: parent.verticalCenter
            anchors.horizontalCenter: parent.horizontalCenter
        }
        height: itemHeight * scaleh
        width: 2 * itemWidth * scalew
        color: "#788ac5"
        onClicked: {
           Qt.quit()
         }
    }

    Rectangle {
        width: 2 * itemWidth * scalew
        height: 1 * scaleh
        color: "#788ac5"
        anchors.left: parent.left
    }
}
```

In this application, we use the `MediaPlayer`[6] item to play audio. We can use it in conjunction with a `VideoOutput` to play video.

If we want to run this application on the board, we need to add some `GStreamer plugins` (such as: `libgstwavparse.so` or `libgstogg.so`) to play some specific media format, such as: WAV or Ogg/Vorbis.

(5) ContentVideo.qml

```
import QtQuick 2.2
import QtMultimedia 5.0

// This VideoOutput component is used for displaying video on the monitor
VideoOutput {
    id: videoOutput

    property alias mediaSource: mediaPlayer.source
    property alias mediaPlayer: mediaPlayer
    property bool isPlaying: false

    // Play video
    function play() {
        mediaPlayer.play()
    }
    // Stop video
    function stop() {
        mediaPlayer.stop()
    }
    // Pause video
    function pause() {
        mediaPlayer.pause()
    }

    source: mediaPlayer
    fillMode: VideoOutput.PreserveAspectFit

    // Add media playback to a scene
    MediaPlayer {
        id: mediaPlayer
        autoLoad: true
        autoPlay: true
        volume: 0.1
        loops: Audio.Infinite
```

[6] http://doc.qt.io/qt-5/qml-qtmultimedia-mediaplayer.html#details

```
        onPlaybackStateChanged: {
            if (playbackState === MediaPlayer.PlayingState) {
                videoOutput.isPlaying = true;
            } else {
                videoOutput.isPlaying = false;
            }
        }
        Component.onDestruction: {
            mediaPlayer.stop()
        }
    }
}
```

To open an image, we simply called the following function:

```
Component.onCompleted: fileSelected.connect(content.openImage)
```

4.2.10.2 Result

Can play audio, video, and display image. User can use some playback features (such as: start, stop, pause, resume...).

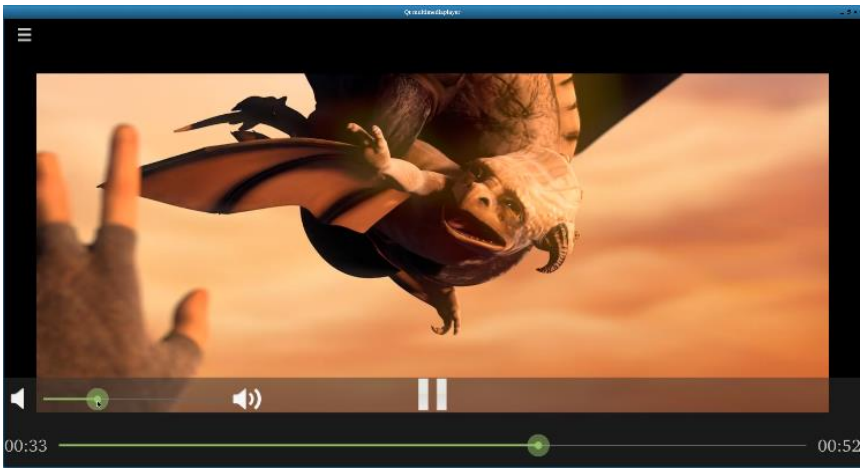


Figure 4.23 Multimedia Player result

4.2.10.3 How to get source code

Download source code from Renesas Marketplace website at the same location as this manual.

5. Appendix

5.1 Platform Architecture

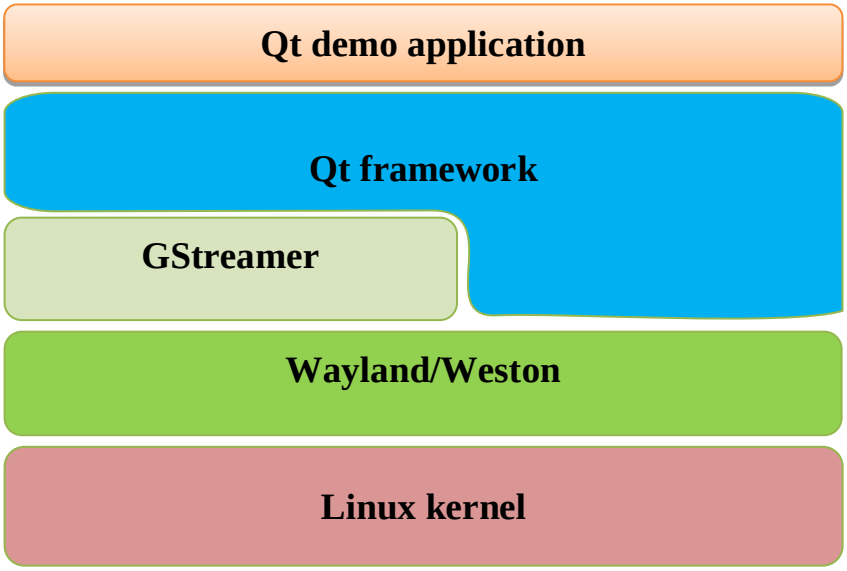


Figure 5.1 Platform Architecture

`Qt demo application`: Comprises our applications, such as: `Hello World`, `Audio Play`, `Video Play`...

`Qt framework`: Provides modules (`qtbase`, `qtmultimedia`, `qtdeclarative`, `qmake`, `qmlscene`) and tools (`qmake`) to develop Qt applications.

`GStreamer`: A framework for creating streaming media applications.

`Wayland/Weston`: A computer protocol that defines the communication between display server and its clients.

5.2 Qt Quick introduction

The Qt Quick module is a standard library for writing QML applications. While the Qt QML module provides the QML engine and language infrastructure, the Qt Quick module provides all the basic types which are necessary for creating user interfaces with QML. It provides a visual canvas and includes types for creating and animating visual components, receiving user input, creating data models and views, and delayed object instantiation.

The Qt Quick module also provides API for extending QML applications with C++ code.

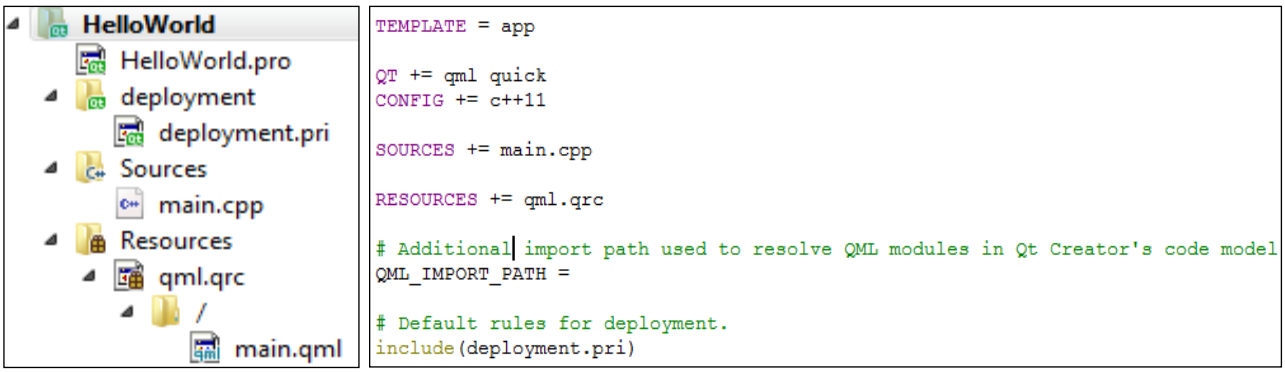


Figure 5.2 A simple Qt Quick project hierarchy (left) and the content of HelloWorld.pro (right)

File HelloWorld.pro: Contains all the information required by qmake tool to build application.

File deployment.pri: Contains common settings and code for .pro files.

File main.cpp: Initializes and loads GUI of Hello World application.

File main.qml: Describes the GUI of Hello World application.

5.3 Qt Creator IDE

Qt Creator provides a cross-platform, complete integrated development environment (IDE) for application developers to create applications for multiple desktop, embedded, and mobile device platforms, such as Android and iOS. It is available for Linux, macOS, and Windows operating systems.

5.3.1 How to install Qt Creator

Step 1. Visit https://www.qt.io/download-open-source website. Next, click Download the Qt Online Installer button. Finally, click Download button to get Qt Creator installer.

Step 2. Open the installer, click Next button.

Step 3. At Qt Account section, click Skip button to skip registration step.

Step 4. At Setup – Qt section, click Next button. The installer will fetch meta information from remote repository. This process will take a few minutes.

Step 5. At User Data Collection section, choose Disable sending usage statistic, then click Next button.

Step 6. At Installation Folder section, click Browse button to choose a location for Qt Creator.

Step 7. At Select Components section, expand Qt, then choose the highest Qt version. In this case, the highest version is Qt 5.13.2, but it might change over time.

Step 8. At Select Components section, expand Qt 5.13.2, then choose MinGW 7.x.x 32-bit. This is crucial; if you forget this step, your computer will not be able to compile and run C++ programs.

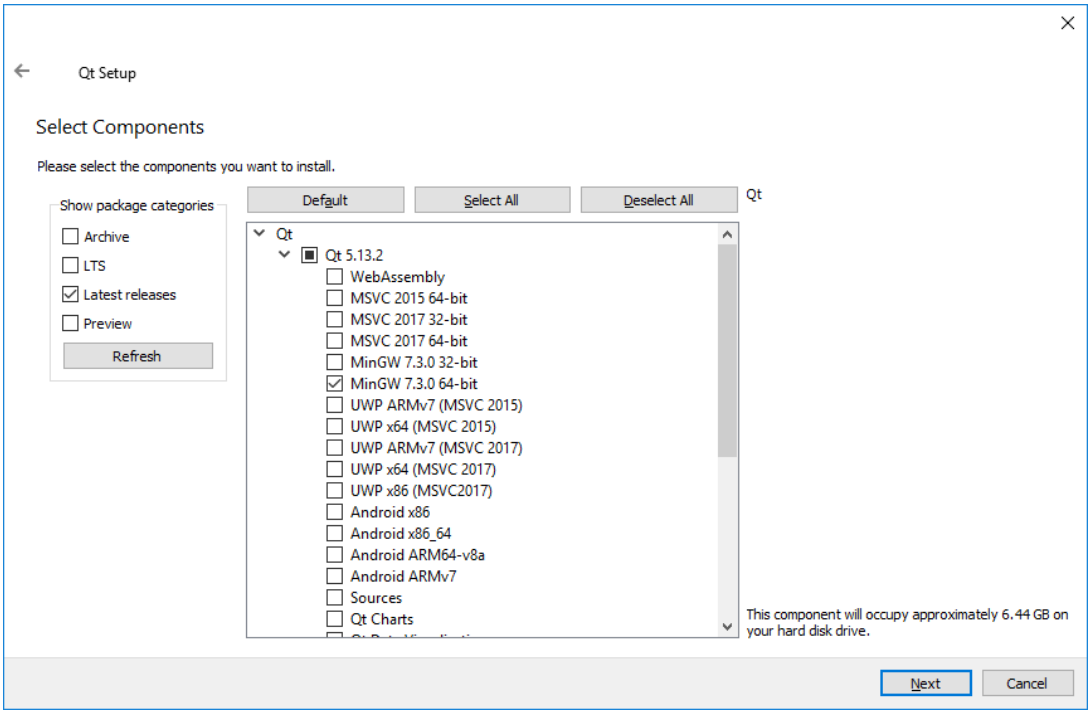


Figure 5.3 Qt Creator installer

Step 9. At `License Agreement` section, choose "`I have read and agree to the terms contained in the license agreements`", then click `Next` button multiple times. Finally, click `Install` button to install Qt Creator.

5.3.2 How to create Qt Quick Project

Step 1. Select `File` > `New File or Project` > `Application` > `Qt Quick Application – Empty` > `Choose`.

Step 2. At `Project Location` section, enter the name and choose the location of Qt Quick Application, then click `Next` button.

Step 3. At `Define Build System` section, select `qmake` as `Build system`, then click `Next` button.

Step 4. At `Define Project Details` section, leave everything untouched and click `Next` button.

Step 5. At `Kit Selection` section, choose `MinGW 7.x.x 32-bit`, then click `Next` button.

Step 6. At `Project Management` section, click `Finish` button to generate Qt Quick project.

5.4 How to draw shape

5.4.1 How to draw cube shape

5.4.1.1 How to generate vertices array?

According to the following figure, the application will list vertices for each face of the cube shape.

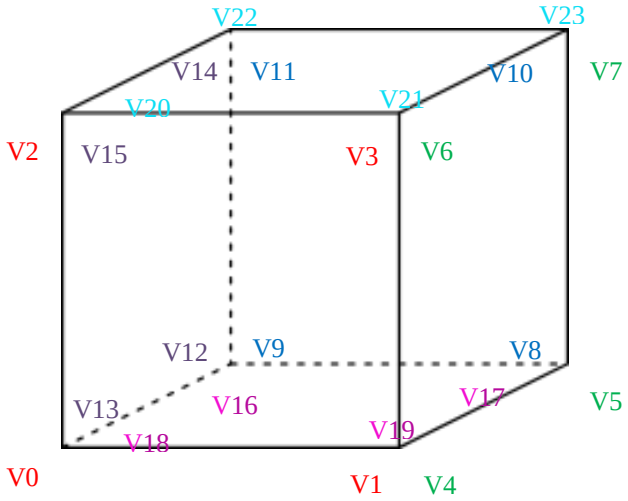


Figure 5.4 Vertices' locations of unit Cube shape

5.4.1.2 How to generate indices array?

This indices array is created by using technique triangles strips[1]. For further information about triangle strips, please refer to Cube example[2] from Qt creator.

In order to connect multiple triangles in cube's faces, we will arrange indices array as degenerate triangles[3]. Simply thinking, to connect two faces of cube, we have to double the last vertex of the last triangle of one face and double the first vertex of the first triangle of the other face. For example: to connect face v0v1v2v3 to face v4v5v6v7, we need to create indices array: 0, 1, 2, 3, 3, 4, 4, 5, 6, and 7.

5.4.2 How to draw sphere shape

5.4.2.1 How to find sphere's vertices?[4]

OpenGL ES 2.0 cannot draw sphere shape directly. It can only draw points, discrete lines, line strips, line loop, discrete triangles, triangle strips and triangle fan. So, we need to divide sphere into smaller shapes, such as: rectangles for easier managing.

From these rectangles, we simply divide them into smaller triangles and ask OpenGL ES 2.0 to draw these triangles to form a sphere shape.

There are multiple ways to do this, but in this case, the application will use longitude and latitude lines. Simply

[1] https://en.wikipedia.org/wiki/Triangle_strip

[2] http://doc.qt.io/qt-5/qtopengl-cube-example.html

[3] http://www.learnopengles.com/tag/degenerate-triangles/

[4] http://learningwebgl.com/blog/?p=1253

thinking, latitudes are the horizontal lines that are across the sphere from the top to the bottom. While longitude is the vertical line that separate the sphere into multiple segments.

The application will get vertices where longitudes and latitudes intersect. Then, the application will split each square form by two adjacent lines of longitude and two adjacent lines of latitude into two triangles, and draw them as the below figure.

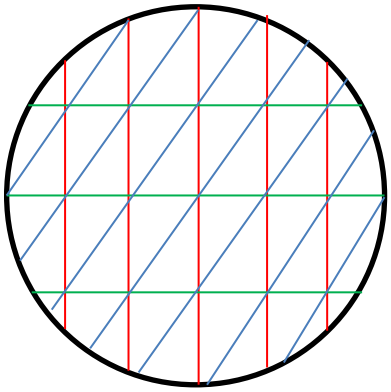


Figure 5.5 Drawing approach of the sphere shape

5.4.2.2 How to generate indices array?[5]

The application will loop through vertices array, and for each vertex, the application will call its index as `first`. Then it will find another vertex which is "below" the `first vertex` and call its index as `second`.

Next, the application will get the right next vertex of `first` and the right next vertex of `second` to form a rectangle. Finally, the application will split the rectangle into 2 triangles as the below figure.

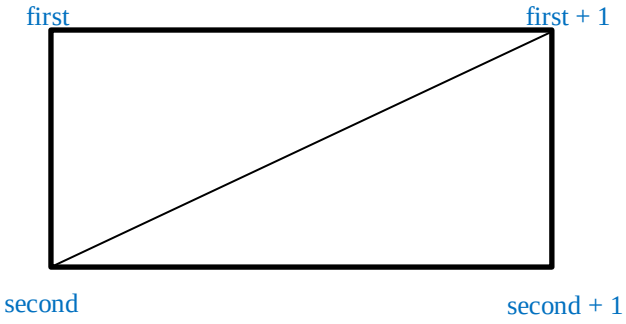


Figure 5.6 Generating indices approach of the sphere shape

5.4.3 How to draw cone shape

5.4.3.1 How to find cone's vertices?

OpenGL ES 2.0 cannot draw sphere shape directly. It can only draw points, discrete lines, line strips, line loop, discrete triangles, triangle strips and triangle fan. In order to draw cone shape, we must divide cone shape into multiple triangles. Then, draw all of the triangles to form a cone shape.

First, the application will add the top vertex at position (0, 1, 0). Then, based on number of chords, it will calculate

[5] http://learningwebgl.com/blog/?p=1253

vertices on base circle. Finally, the application also adds the center vertex of the base circle at location (0, -1, 0) for later drawing.

5.4.3.2 How to generate indices array?

It will loop through the second vertex to the second last vertex (to avoid first vertex which is `the top vertex` of cone shape and the last vertex which is `the center of base circle`). Basically, these vertices that the application is going to loop through are the vertices on the base circle.

For each vertex that the application loops through, it will add two triangles based on that vertex, the next vertex, `the top vertex` of cone shape and `the center vertex` of the base circle as the below figure.

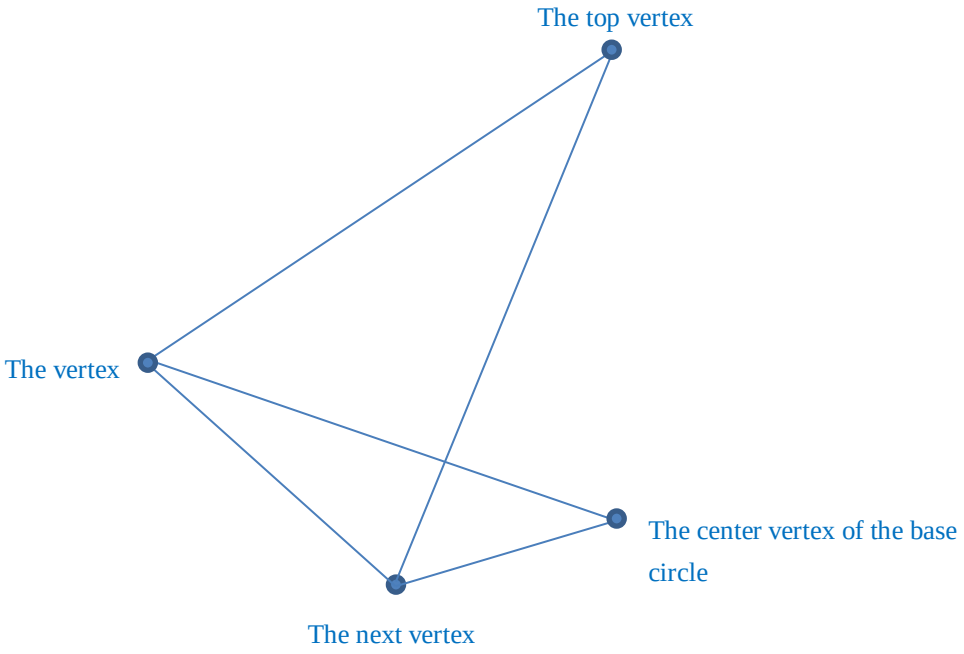


Figure 5.7 Generating indices approach of the cone shape

Revision History	Application Note for the RZ/G2 Group

Rev.	Date	Description	
		Page	Summary
1.00	Apr 2018	-	First Edition issued (Preliminary)
1.01	Oct 2019	-	Reviewed and modified instructions for RZ/G2 Group
1.02	Feb 2020	-	Added *gst-audiorecord*, *gst-audiovideorecord*, and *qt-audiorecorder*. Replaced hard-coded input file(s) with command line argument(s). Supported MIPI camera for *gst-videorecord* and *gst-audiovideorecord*. Supported USB microphone for *gst-audiorecord*, *gst-audiovideorecord*, and *qt-audiorecorder*. Supported H.265 video playback for the following applications: - gst-videoplay. - gst-videoplayer. - gst-audiovideoplay. - gst-fileplay. - gst-multipledisplays1. - gst-multipledisplays2.
1.03	Jul 2020	-	Support RZ/G2H platform

Application Note for the RZ/G2 Group

Publication Date: Rev.1.03 Jul. 24, 2020

Published by: Renesas Electronics Corporation

RZ/G2 Group

Renesas Electronics Corporation

R01US0429EJ0103