REFERENCE CODE - FULLY COMMENTED

```
/*
 * This file is developed based on the skeleton of base_task.c which is a basic
 * real-time task skeleton offered by liblitmus, user space library of Litmus-RT.
 * This file is useful to run a video processing application in real-time.

 * Libraries used: FFMPEG, SDL
 * Goal: To build a real-time Video player

 * References:
 * https://github.com/rambodrahmani/ffmpeg-video-player
 * http://dranger.com/ffmpeg/tutorial01.html
 * https://github.com/farhanr8/Litmus-RT_VideoApp

 * Code structure: This file contains several sections & sub-sections as listed.
 * The following are indexed & mentioned near respective code sections for ease.

 * 1. Header file inclusion section
 *       1a. Generic libraries that work as helpers
 *       1b. Libraries specific to FFMPEG
 *       1c. Libraries specific to SDL
 *       1d. Libraries specific to Litmus-RT
 * 2. Constant definitions section
 *       2a. Task specific constants
 *       2b. Buffer & Frame size constraints
 *       2c. Macros to handle errors
 * 3. Global declarations section
 *       3a. User-defined data structures
 *       3b. IO, FFMPEG & SDL related variables
 * 4. Function definitions section
 * 5. Main method
 * 6. Video processing job definition
 */
```

```c
/* *************************************************************** */
/* 1. HEADER FILE INCLUSION SECTION */

// 1a. Generic libraries that work as helpers
#include <math.h>
#include <time.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

// 1b. Libraries specific to FFMPEG
#include <libavutil/opt.h>
#include <libavutil/imgutils.h>
#include <libswscale/swscale.h>
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libswresample/swresample.h>

// 1c. Libraries specific to SDL
#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
#ifdef __MINGW32__
#undef main          // To prevents SDL from overriding main()
#endif

// 1d. Libraries specific to Litmus-RT
#include <litmus.h>

/* *************************************************************** */
/* 2. CONSTANT DEFINITIONS SECTION */

// 2a. Task specific constants
// Should be defined based on the task type & how often jobs must be released
// Helpful to use experimentation to decide upon these constants
#define PERIOD          17.5
#define RELATIVE_DEADLINE 100
#define EXEC_COST       10
```

```
// 2b. Buffer & Frame size constants
#define SDL_AUDIO_BUFFER_SIZE   1024
#define MAX_AUDIO_FRAME_SIZE   192000

// 2c. Macros to handle errors
#define CALL( exp ) do { \
        int ret; \
        ret = exp; \
        if (ret != 0) \
         fprintf(stderr, "%s failed: %m\n", #exp);\
        else \
         fprintf(stderr, "%s ok.\n", #exp); \
} while (0)

/* ************************************************************ */
/* 3. GLOBAL DECLARATIONS SECTION */

// 3a. User-defined data structures
// Handles PacketQueue structure & creates an alias for it
typedef struct PacketQueue {
        AVPacketList *first_pkt, *last_pkt;
        int nb_packets;
        int size;
        SDL_mutex *mutex;
        SDL_cond *cond;
} PacketQueue;

// 3b. IO, FFMPEG & SDL related variables
// Audio PacketQueue reference
PacketQueue audioq;

// Initializes FFMPEG variables
int     i, videoStream, audioStream, frameFinished;

// Global quit flag
int     quit = 0;

// Handles image scaling & is provided by FFMPEG
struct SwsContext    *sws_ctx = NULL;
```

```
// Helps to format IO Context
AVFormatContext         *pFormatCtx       = NULL;

// Handles Codec context for video stream
AVCodecContext    *pCodecCtxOrig    = NULL;
AVCodecContext    *pCodecCtx  = NULL;
AVCodec           *pCodec     = NULL;

// Handles Codec context for audio stream
AVCodecContext    *aCodecCtxOrig    = NULL;
AVCodecContext    *aCodecCtx  = NULL;
AVCodec           *aCodec     = NULL;

// Helps to store & handle compressed data
AVPacket    packet;

// Helps to describe raw/decoded audio or video data
AVFrame     *pFrame = NULL;

// SDL parameters
SDL_Overlay    *bmp;
SDL_Surface    *screen;
SDL_Rect       rect;
SDL_Event      event;
SDL_AudioSpec  wanted_spec, spec;

/* *************************************************************** */
/* 4. FUNCTION DEFINITIONS SECTION */

/* Declares the periodically invoked job.
 * Returns 1 -> task should exit.
 *        0 -> task should continue.
 */
int job(void);


/* Method to initialize the given PacketQueue
 * @param   q  is the PacketQueue to be initialized.
 */
void packet_queue_init(PacketQueue *q) {
```

```
        // Dynamically allocates memory for the audio queue with value 0
        memset(q, 0, sizeof(PacketQueue));

        // Returns the initialized and unlocked mutex or NULL on failure
        q->mutex = SDL_CreateMutex();

        if (!q->mutex) {
        // Could not create mutex
        printf("SDL_CreateMutex Error: %s.\n", SDL_GetError());
        return;
        }

        // Returns a new condition variable or NULL on failure
        q->cond = SDL_CreateCond();

        if (!q->cond) {
        // Could not create condition variable
        printf("SDL_CreateCond Error: %s.\n", SDL_GetError());
        return;
        }
}

/* ********************************************************************** */
/* Method to put the given AVPacket in the given PacketQueue
 * @param     q   is   the queue to be used for the insert
 * @param     pkt is the AVPacket to be inserted in the queue
 * Returns    0 if the AVPacket is correctly inserted in the given PacketQueue.
 */
int packet_queue_put(PacketQueue *q, AVPacket *pkt) {

        AVPacketList *pkt1;

        // Allocates the new AVPacketList to be inserted in the audio PacketQueue
        pkt1 = av_malloc(sizeof(AVPacketList));
        if (!pkt1)
                return -1;

        // Adds reference to given AVPacket which will be inserted at queue end
        pkt1->pkt = *pkt;
```

```
        pkt1->next = NULL;

        // Uses lock to ensure that only one process accesses resource at a time
        SDL_LockMutex(q->mutex);

        // Inserts new AVPacketList at the end of the queue
        // Checks if the queue is empty. If so, inserts at the start
        if (!q->last_pkt)
                q->first_pkt = pkt1;

        // If not, inserts at the end
        else
                q->last_pkt->next = pkt1;

        // Points the last AVPacketList in the queue to the newly created
        // AVPacketList & updates respective data associated
        q->last_pkt = pkt1;
        q->nb_packets++;
        q->size += pkt1->pkt.size;

        // Restarts a thread wait on a conditional variable
        SDL_CondSignal(q->cond);

        // Unlocks the muter lock posed
        SDL_UnlockMutex(q->mutex);

        return 0;
}

/* ****************************************************************** */
/* Method to get the first AVPacket from the given PacketQueue.
 * @param   q      is the PacketQueue to extract from
 * @param   pkt    is the first AVPacket extracted from the queue
 * @param   block = 0 to avoid waiting for an AVPacket to be inserted in the
 * given queue & != 0 otherwise.
 * Returns      < 0 if returning because the quit flag is set,
 *               0 if the queue is empty,
 *               1 if it is not empty and a packet was extract (pkt)
 */
```

```
static int packet_queue_get(PacketQueue *q, AVPacket *pkt, int block) {

        AVPacketList *pkt1;
        int ret;

        // Ensures that only one process will access the resource at a time
        SDL_LockMutex(q->mutex);

        for(;;) {
                // Exits if the global quit flag is set
                if(quit) {
                  ret = -1;
                  break;
                }

                // Points to the first AVPacketList in the queue
                pkt1 = q->first_pkt;

                // Packet is not NULL => queue is not empty
                if (pkt1) {

                        // Places second packet in the queue at first position
                        q->first_pkt = pkt1->next;

                        // Checks if queue is empty after removal
                        if (!q->first_pkt)
                                q->last_pkt = NULL;

                        // Updates respective data associated
                        q->nb_packets--;
                        q->size -= pkt1->pkt.size;

                        // Points pkt to the extracted packet
                        *pkt = pkt1->pkt;

                        // Frees up the dynamically allocated memory block
                        av_free(pkt1);
                        ret = 1;
                        break;
```

```
            } else if (!block) {

                // block = 0 avoids waiting for AVPacket ro be inserted
                ret = 0;
                break;

            } else
                SDL_CondWait(q->cond, q->mutex);
        }

        // Unlocks the mutex lock posed
        SDL_UnlockMutex(q->mutex);
        return ret;
}

/* ****************************************************************** */
/* Method to resample the audio data retrieved using FFMPEG before playing it.
 * @param   audio_decode_ctx    is the audio codec context retrieved from the original
AVFormatContext.
 * @param   decoded_audio_frame is the decoded audio frame.
 * @param   out_sample_fmt      is the audio output sample format
 * @param   out_channels        are audio output channels, retrieved from the original
audio codec context.
 * @param   out_sample_rate     is the audio output sample rate, retrieved from the
original audio codec context.
 * @param   out_buf             is the audio output buffer.
 * Returns the size of the resampled audio data.
 */

static int audio_resampling(
                    AVCodecContext * audio_decode_ctx,
                    AVFrame * decoded_audio_frame,
                    enum AVSampleFormat out_sample_fmt,
                    int out_channels,
                    int out_sample_rate,
                    uint8_t * out_buf)
{

   SwrContext * swr_ctx = NULL;
   int ret = 0;
```

```
int64_t in_channel_layout = audio_decode_ctx->channel_layout;
int64_t out_channel_layout = AV_CH_LAYOUT_STEREO;
int out_nb_channels = 0;
int out_linesize = 0;
int in_nb_samples = 0;
int out_nb_samples = 0;
int max_out_nb_samples = 0;
uint8_t ** resampled_data = NULL;
int resampled_data_size = 0;

// Quits if the global flag is set
if (quit)
    return -1;

// Allocates SwrContext
swr_ctx = swr_alloc();

if (!swr_ctx) {
    printf("Unable to allocate SqrContext !! \n");
    return -1;
}

// Get input audio channels
in_channel_layout = (audio_decode_ctx->channels ==

av_get_channel_layout_nb_channels(audio_decode_ctx->channel_layout)) ?
        audio_decode_ctx->channel_layout :
        av_get_default_channel_layout(audio_decode_ctx->channels);

// Checks if input audio channels correctly retrieved
if (in_channel_layout <= 0) {
    printf("Unable to retrieve input audio channels correctly !!\n");
    return -1;
}

// Sets output audio channels based on the input audio channels
if (out_channels == 1)
    out_channel_layout = AV_CH_LAYOUT_MONO;

else if (out_channels == 2)
```

```
    out_channel_layout = AV_CH_LAYOUT_STEREO;

else
    out_channel_layout = AV_CH_LAYOUT_SURROUND;

// Retrieves number of audio samples (per channel)
in_nb_samples = decoded_audio_frame->nb_samples;
if (in_nb_samples <= 0) {
    printf("Unable to retrieve audio samples from channel !!\n");
    return -1;
}

// Sets SwrContext parameters for resampling : In-channel layout
av_opt_set_int(
    swr_ctx,
    "in_channel_layout",
    in_channel_layout,
    0
);

// Sets SwrContext parameters for resampling : In-sample rate
av_opt_set_int(
    swr_ctx,
    "in_sample_rate",
    audio_decode_ctx->sample_rate,
    0
);

// Sets SwrContext parameters for resampling : In-sample format
av_opt_set_sample_fmt(
    swr_ctx,
    "in_sample_fmt",
    audio_decode_ctx->sample_fmt,
    0
);

// Sets SwrContext parameters for resampling : Out-channel layout
av_opt_set_int(
    swr_ctx,
    "out_channel_layout",
```

```
    out_channel_layout,
    0
);

// Sets SwrContext parameters for resampling : Out-sample rate
av_opt_set_int(
    swr_ctx,
    "out_sample_rate",
    out_sample_rate,
    0
);

// Sets SwrContext parameters for resampling : Out-sample format
av_opt_set_sample_fmt(
    swr_ctx,
    "out_sample_fmt",
    out_sample_fmt,
    0
);

// Initializes the SwrContext after setting all values
ret = swr_init(swr_ctx);

if (ret < 0) {
    printf("Failed to initialize the resampling context !!\n");
    return -1;
}

// Rescales the 64-bit integer with specified rounding
max_out_nb_samples = out_nb_samples = av_rescale_rnd(in_nb_samples,
out_sample_rate, audio_decode_ctx->sample_rate, AV_ROUND_UP);

// Checks if rescaling was successful
if (max_out_nb_samples <= 0) {
    printf("Rescaling the samples failed !!\n");
    return -1;
}

// Gets number of output audio channels
out_nb_channels = av_get_channel_layout_nb_channels(out_channel_layout);
```

```
// Allocates a data pointers array, samples buffer for out_nb_samples
ret = av_samples_alloc_array_and_samples(
        &resampled_data,
        &out_linesize,
        out_nb_channels,
        out_nb_samples,
        out_sample_fmt,
        0);

if (ret < 0) {
   printf("Unable to allocate destination samples !!\n");
   return -1;
}

// Retrieves output samples number taking into account the progressive delay
out_nb_samples = av_rescale_rnd(
      swr_get_delay(swr_ctx, audio_decode_ctx->sample_rate) + in_nb_samples,
         out_sample_rate, audio_decode_ctx->sample_rate,
         AV_ROUND_UP);

// Checks if output samples number was correctly retrieved
if (out_nb_samples <= 0) {
   printf("Failed to retrieve output samples number !!\n");
   return -1;
}

if (out_nb_samples > max_out_nb_samples) {

   // Frees memory block and set pointer to NULL
   av_free(resampled_data[0]);

   // Allocate a samples buffer for out_nb_samples samples
   ret = av_samples_alloc(resampled_data, &out_linesize,
           out_nb_channels, out_nb_samples, out_sample_fmt, 1);

   // Checks if samples buffer is correctly allocated
   if (ret < 0) {
      printf("Samples buffer is not correctly allocated !!\n");
      return -1;
   }
```

```
        max_out_nb_samples = out_nb_samples;
    }

    if (swr_ctx)
    {
        // Does the actual audio data resampling
        ret = swr_convert(swr_ctx, resampled_data, out_nb_samples,
                (const uint8_t **) decoded_audio_frame->data,
                decoded_audio_frame->nb_samples);

        // Checks audio conversion was successful
        if (ret < 0) {
            printf("Unable to convert data & resamples it !!\n");
            return -1;
        }

        // Gets the required buffer size for the given audio parameters
        resampled_data_size = av_samples_get_buffer_size(&out_linesize,
out_nb_channels,
                        ret, out_sample_fmt, 1);

        // Checks audio buffer size
        if (resampled_data_size < 0) {
      printf("Unable to assign required buffer size for the given audio parameters !! \n");
            return -1;
        }
    }
    else {
        printf("Null SWR Context !! \n");
        return -1;
    }

    // Copies the resampled data to the output buffer
    memcpy(out_buf, resampled_data[0], resampled_data_size);

    // Memory cleanup
    if (resampled_data)
        av_freep(&resampled_data[0]);
    av_freep(&resampled_data);
```

```
    resampled_data = NULL;

    if (swr_ctx)
        swr_free(&swr_ctx);

    return resampled_data_size;
}

/* ******************************************************************* */
/* Methods to get a packet from the queue if available.
 * Decode the extracted packet. Once we have the frame, resample it and simply
 * copy it to our audio buffer, while data_size is smaller than audio buffer.
 * @param   aCodecCtx   the audio AVCodecContext used for decoding
 * @param   audio_buf   the audio buffer to write into
 * @param   buf_size    the size of the audio buffer, 1.5 larger than the one
 *                      provided by FFmpeg
 * Returns 0 if everything goes well, -1 in case of error or quit
 */

int audio_decode_frame(AVCodecContext *aCodecCtx, uint8_t *audio_buf, int
buf_size) {

        int len1 = 0;
        int data_size = 0;

        // Allocates an AVPacket & sets its fields to default values
        AVPacket * avPacket = av_packet_alloc();

        static uint8_t *audio_pkt_data = NULL;
        static int audio_pkt_size = 0;

        // Allocates a new frame to decode audio packets
        static AVFrame * avFrame = NULL;
        avFrame = av_frame_alloc();

        if (!avFrame) {
                printf("Unable to allocate AVFrame !!\n");
                return -1;
        }
```

```
// As long as we don't get any error OR until the audio buffer
// is not smaller than data_size, we proceed with the below
for (;;) {
        if (quit)
                return -1;

        while(audio_pkt_size > 0) {

                int got_frame = 0;
                int ret = avcodec_receive_frame(aCodecCtx, avFrame);

                if (ret == 0)
                got_frame = 1;

                if (ret == AVERROR(EAGAIN))
                ret = 0;

                if (ret == 0)
                ret = avcodec_send_packet(aCodecCtx, avPacket);

                if (ret == AVERROR(EAGAIN))
                ret = 0;

                else if (ret < 0) {
                printf("Error while decoding audio.\n");
                return -1;
                }
                else
                len1 = avPacket->size;

                // Skip the frame if error occurs
                if (len1 < 0) {
                        audio_pkt_size = 0;
                        break;
                }

                audio_pkt_data += len1;
                audio_pkt_size -= len1;
                data_size = 0;
```

```
                    if (got_frame) {
                    // Audio resampling
                    data_size = audio_resampling(
                    aCodecCtx,
                    avFrame,
                    AV_SAMPLE_FMT_S16,
                    aCodecCtx->channels,
                    aCodecCtx->sample_rate,
                    audio_buf);
                        assert(data_size <= buf_size);
                    }

                    // No data yet => get more frames
                    if (data_size <= 0)
                        continue;

                    return data_size;
                }

                // Unreferences the buffer referenced by the packet
                if (avPacket->data)
                    av_packet_unref(avPacket);

                // Gets more audio AVPacket
                if (packet_queue_get(&audioq, avPacket, 1) < 0) {
                    printf("Unable to get more audio AVPacket !!\n");
                    return -1;
                }

                audio_pkt_data = avPacket->data;
                audio_pkt_size = avPacket->size;
        }
}

/* ************************************************************ */
/* Method to pull in data from audio_decode_frame()
 * Stores the result in an intermediary buffer
 * Attempts to write as many bytes as the amount defined by len to SDL
 * stream, and get more data if we don't have enough yet, or save it for later
 * if we have some left over.
```

```
 * @param   userdata    the pointer we gave to SDL.
 * @param   stream      the buffer we will be writing audio data to.
 * @param   len         the size of that buffer.
 */

void audio_callback(void *userdata, Uint8 *stream, int len) {
        int len1, audio_size;

        // Size of audio_buf = 1.5 x Size of the largest audio frame from FFMPEG
        static uint8_t audio_buf[(MAX_AUDIO_FRAME_SIZE * 3) / 2];
        static unsigned int audio_buf_size = 0;
        static unsigned int audio_buf_index = 0;

        // Retrieves the audio codec context
        aCodecCtx = (AVCodecContext *)userdata;

        // Runs as long as the SDL defined length > 0
        while (len > 0) {

                if (quit)
                        return;

                if (audio_buf_index >= audio_buf_size) {

                        // We have already sent all our data => get more
                        audio_size = audio_decode_frame(aCodecCtx, audio_buf,
sizeof(audio_buf));
                        if (audio_size < 0) {

                                // If error, we output silence
                                audio_buf_size = 1024;

                                // Clears memory
                                memset(audio_buf, 0, audio_buf_size);
                                printf("audio_decode_frame() failed !!\n");
                        }
                        else
                                audio_buf_size = audio_size;

                        audio_buf_index = 0;
```

```
        }

        len1 = audio_buf_size - audio_buf_index;
        if (len1 > len)
            len1 = len;

        // Copies data from audio buffer to the SDL stream
        memcpy(stream, (uint8_t *)audio_buf + audio_buf_index, len1);

        len -= len1;
        stream += len1;
        audio_buf_index += len1;
    }
}

/* ************************************************************ */
```

/* 5. MAIN METHOD */

```
int main(int argc, char** argv)
{
    int do_exit, ret;
    struct rt_task param;

    // Sets up task parameters
    init_rt_task_param(&param);
    param.exec_cost = ms2ns(EXEC_COST);
    param.period = ms2ns(PERIOD);
    param.relative_deadline = ms2ns(RELATIVE_DEADLINE);

    // Handling budget overruns
    param.budget_policy = NO_ENFORCEMENT;

    // Sets the real-time task's class to be Soft
    param.cls = RT_CLASS_SOFT;

    // Used by fixed priority plugins
    param.priority = LITMUS_LOWEST_PRIORITY;

    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER))
{
```

```
		fprintf(stderr, "Unable to initialize SDL - %s\n", SDL_GetError());
		exit(1);
	}

	// Opens the input video file
	if(avformat_open_input(&pFormatCtx, "/home/litmus/Videos/test-video.mp4",
NULL, NULL) != 0){
	printf("Unable to open the video file given !!\n");
		return -1;
	}

	// Retrieves stream information
	if(avformat_find_stream_info(pFormatCtx, NULL) < 0){
      printf("Unable to find stream information !!\n");
		return -1;
	}

	// Dumps information about file onto standard error
	av_dump_format(pFormatCtx, 0, "/home/litmus/Videos/test-video.mp4", 0);

	// Finds the first video stream
	videoStream = -1;
	audioStream = -1;

	for (int i = 0; i < pFormatCtx->nb_streams; i++) {
		if(pFormatCtx->streams[i]->codecpar->codec_type ==
AVMEDIA_TYPE_VIDEO && videoStream < 0) {
			videoStream = i;
		}

if(pFormatCtx->streams[i]->codecpar->codec_type==AVMEDIA_TYPE_AUDIO
&& audioStream < 0) {
			audioStream = i;
		}
	}

	if(videoStream==-1){
		printf("No Video Stream found !!");
		return -1; // Didn't find a video stream
	}
```

```
	if(audioStream==-1){
		printf("No Audio Stream found !!");
		return -1;		// Didn't find a audio stream
	}

	// Retrieves audio codec
	aCodec =
avcodec_find_decoder(pFormatCtx->streams[audioStream]->codecpar->codec_id);
	if(aCodec == NULL) {
		fprintf(stderr, "Unsupported audio codec !!\n");
		return -1;
	}

	// Copies the obtained audio codec context
	aCodecCtxOrig = avcodec_alloc_context3(aCodec);
	ret = avcodec_parameters_to_context(aCodecCtxOrig,
pFormatCtx->streams[videoStream]->codecpar);
	if (ret != 0) {
		printf("Unable to copy audio codec context !!\n");
		return -1;
	}

	aCodecCtx = avcodec_alloc_context3(aCodec);
	ret = avcodec_parameters_to_context(aCodecCtx,
pFormatCtx->streams[audioStream]->codecpar);

	if (ret != 0) {
	printf("Unable to copy audio codec context !!\n");
	return -1;
	}

	// Set audio settings from codec info for desired specifications
	wanted_spec.freq = aCodecCtx->sample_rate;
	wanted_spec.format = AUDIO_S16SYS;
	wanted_spec.channels = aCodecCtx->channels;
	wanted_spec.silence = 0;
	wanted_spec.samples = SDL_AUDIO_BUFFER_SIZE;
	wanted_spec.callback = audio_callback;
	wanted_spec.userdata = aCodecCtx;
```

```
	if(SDL_OpenAudio(&wanted_spec, &spec) < 0) {
		fprintf(stderr, "SDL_OpenAudio: %s\n", SDL_GetError());
		return -1;
	}

	// Initializes the audio AVCodecContext to use the given audio AVCodec
	if(avcodec_open2(aCodecCtx, aCodec, NULL)){
		printf("Unable to open audio codec !!\n");
		return -1;
	}

	packet_queue_init(&audioq);

	// Starts playing audio on the given audio device
	SDL_PauseAudio(0);

	// Retrieves video codec
	// Finds the decoder for the video stream
pCodec=avcodec_find_decoder(pFormatCtx->streams[videoStream]->codecpar->cod
ec_id);

	if(pCodec == NULL) {
		fprintf(stderr, "Unsupported video codec !!\n");
		return -1;			// Codec not found
	}

	// Copies video codec context for the audio context we retrieved earlier
	pCodecCtxOrig = avcodec_alloc_context3(pCodec);
	ret = avcodec_parameters_to_context(pCodecCtxOrig,
pFormatCtx->streams[videoStream]->codecpar);
	if (ret != 0) {
	printf("Unable to copy video codec context !!\n");
	return -1;
	}

	pCodecCtx = avcodec_alloc_context3(pCodec);
	ret = avcodec_parameters_to_context(pCodecCtx,
pFormatCtx->streams[videoStream]->codecpar);
```

```
if (ret != 0) {
printf("Unable to copy video codec context !!\n");
return -1;
}

// Initializes the video AVCodecContext to use the given video AVCodec
if(avcodec_open2(pCodecCtx, pCodec, NULL)<0)
        return -1;

// Allocates video frame
pFrame=av_frame_alloc();

// Makes a screen to put our video
#ifndef __DARWIN__
  screen = SDL_SetVideoMode(pCodecCtx->width, pCodecCtx->height, 0, 0);
#else
  screen = SDL_SetVideoMode(pCodecCtx->width, pCodecCtx->height, 24,
0);
#endif

if(!screen) {
        fprintf(stderr, "SDL: could not set video mode - exiting\n");
        exit(1);
}

// Allocates a place to put our YUV image on that screen
bmp = SDL_CreateYUVOverlay(pCodecCtx->width,
                           pCodecCtx->height,
                           SDL_YV12_OVERLAY,
                           screen);

// Initializes SWS context for software scaling
sws_ctx = sws_getContext(pCodecCtx->width,
                         pCodecCtx->height,
                         pCodecCtx->pix_fmt,
                         pCodecCtx->width,
                         pCodecCtx->height,
                         AV_PIX_FMT_YUV420P,
                         SWS_BILINEAR,
                         NULL,
```

```
                                        NULL,
                                        NULL
                                        );

// The task is in background mode upon startup
// Initializes real-time properties for the entire program
// Returns 0 on success
CALL( init_litmus() );

// Sets up real-time task params for given process
CALL( set_rt_task_param(gettid(), &param) );
fprintf(stderr, "%s\n", "Set to Real Time Task...");

// Transitions into real-time mode
CALL( task_mode(LITMUS_RT_TASK) );
fprintf(stderr,"%s\n","Running RT task...");

// Invokes real-time jobs
do {
        // Waits until next job is released
        sleep_next_period();

        // Invokes the job
        do_exit = job();
} while (!do_exit);

// Transitions into background mode
fprintf(stderr,"%s\n","Completed task successfully...");
fprintf(stderr,"%s\n","Changing to background task...");
CALL( task_mode(BACKGROUND_TASK) );

// Clean up, pint results & statistics then exit
fprintf(stderr,"%s\n","Cleaning...");
av_frame_free(&pFrame);

// Close the codecs
avcodec_close(pCodecCtxOrig);
avcodec_close(pCodecCtx);
avcodec_close(aCodecCtxOrig);
avcodec_close(aCodecCtx);
```

```
        // Close the video file
        avformat_close_input(&pFormatCtx);
        fprintf(stderr,"%s\n","Cleaning done...");
        fprintf(stderr,"%s\n","Program ending...");

        return 0;
}

/* ************************************************************** */
/* 6. VIDEO PROCESSING JOB DEFINITION */

int job(void)
{
        AVFrame pict;
        int ret;

        // Reads next frame of the stream from AVFormatContext
        // Splits it into packets by calling av_read_frame()
        if(av_read_frame(pFormatCtx, &packet) >= 0) {

           // Checks if video stream is found
           if(packet.stream_index == videoStream) {

                // Decodes video frame
                // Give the decoder raw compressed data in an AVPacket
                ret = avcodec_send_packet(pCodecCtx, &packet);

              if (ret < 0) {
                 printf("Unable to send packet for decoding !!\n");
                 return -1;
              }

              while (ret >= 0) {

                 // Gets decoded output data from decoder
                 ret = avcodec_receive_frame(pCodecCtx, pFrame);

                 // Checks if an entire frame was decoded
                 if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF)
```

```
            break;

        else if (ret < 0) {
            printf("Unable to decode video !!\n");
            return -1;
        }
        else
            frameFinished = 1;

        // Did we get a video frame?
          if(frameFinished) {
                SDL_LockYUVOverlay(bmp);

                pict.data[0] = bmp->pixels[0];
                pict.data[1] = bmp->pixels[2];
                pict.data[2] = bmp->pixels[1];

                pict.linesize[0] = bmp->pitches[0];
                pict.linesize[1] = bmp->pitches[2];
                pict.linesize[2] = bmp->pitches[1];

                // Convert the image into YUV format for SDL
                sws_scale(sws_ctx, (uint8_t const * const *)pFrame->data,
pFrame->linesize, 0, pCodecCtx->height, pict.data, pict.linesize);
                    SDL_UnlockYUVOverlay(bmp);
                    rect.x = 0;
                    rect.y = 0;
                    rect.w = pCodecCtx->width;
                    rect.h = pCodecCtx->height;
                    SDL_DisplayYUVOverlay(bmp, &rect);
                    av_packet_unref(&packet);
                }
            }
    }

    else if(packet.stream_index==audioStream)
            packet_queue_put(&audioq, &packet);
    else
        av_packet_unref(&packet);
```

```
            // Handles quit event (Ctrl + C, SDL Window closed)
            SDL_PollEvent(&event);

            switch(event.type) {
                case SDL_QUIT:
                        quit = 1;
                        SDL_Quit();
                        exit(0);
                        break;
                default:
                    break;
            }

            return 0;

        }
        else
            return 1;
}
```