i.MX 8M Immersiv3D Application Note

Document number I3DAN
Release For Production, 11/2021
Version 2 Release For Production 6.0.0

Table of Contents

Chapter 1. Introduction

This application note describes the i.MX 8M Audio Framework and explains procedures for integration, configuration, and usage of its features.

Chapter 2. Overview of i.MX 8M Audio Framework

The i.MX 8M Audio Framework (called Audio Framework in this document) aims to be an alternative to a Digital Signal Processor (DSP) on an audio system. For that, the Audio Framework can allocate up to two of the four application processors of the i.MX 8M to run different audio related use cases. To improve performance, the real time audio processing flow is separated into different stages. A simplified diagram is shown in Figure 1.

This application note describes major features of the Audio Framework, including how to control the audio pipeline, how to implement custom Post Processing Plugins, adaptation to custom boards.

Figure 1. Simplified Audio Framework diagram

Chapter 3. Post processing plugin

The Post Processing stage allows to apply different algorithms to an audio stream. Because every use case is different, the Audio Framework is modular and allows the integration of new algorithms as Post Processing Plugins (PPPs).

3.1. Architecture

Each Post Processing element is plugged into the Audio Framework and communicates with it through an Adaptation Layer (HAL), as shown in Figure 2. This communication allows the Audio Framework not only to control the Post Processing Plugin but also to pipeline and connect it with other Post Processing Plugins.

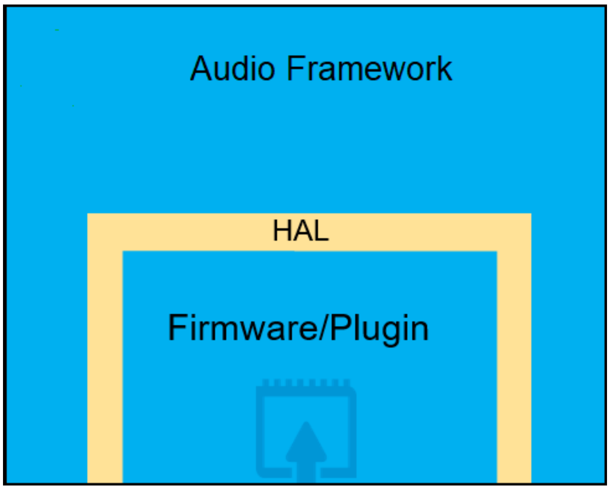

Figure 2. Post Processing Plugin

3.2. Post Processing API

The Audio Framework exposes two levels of API: Application level API and Post Processing level API.

On one hand, the application level API is a string-based REST API. This means that commands are sent as strings with a "parameter=value" format. This type of implementation allows users to create their own plugins in an OS and language-agnostic environment. Additionally, to the REST API, a binary API allows to share a C structure between Linux Application and Post Processing algorithm.

The application level API supports operations like getting capabilities of the platform, building/destroying a pipeline, adding/removing Post Processing elements to a pipeline, and interacting with a plugin. All these operations are used through 4 methods described in Table 1.

Table 1. Application level REST API

Command	Description
POST	Create a resource (create a post processing pipeline or element).
GET	Read information from a resource (get a property value).
PUT	Write information to a resource (set a property value).

Command	Description
DELETE	Delete a resource.

On the other hand, the Post Processing level API allows to add a Post Processing Algorithm in the system, expose its capabilities, initialize and terminate it, save/retrieve data, and expose the plugin private API to the applications. For this, the Post Processing Plugin can use the elements described in Table 2.

Table 2. Post Processing level API

Name	Type	Description
ppp_command_type	Enumeration	Lists possible REST Commands
cowbell_context	Structure	Post Processing Plugin context data
cowbell_driver	Structure	Structure allowing to register Post Processing Plugin callbacks
audio_metadata	Structure	Structure automatically filled after the decoder with information concerning the stream
register_ppp_driver	Function	Registers the Post Processing Plugin into Audio Framework
ppb_get_src	Function	Returns the pointer to “inplace” buffer where PPP can perform write accesses. For PPP with both source and sink pads, it returns the same pointer as ppb_get_sink.
ppb_get_sink	Function	Returns the pointer to “inplace” buffer where PPP can perform read accesses. For PPP with both source and sink pads, it returns the same pointer as ppb_get_src.
ppb_set_src_data_len	Function	Updates data length of every PPB associated to a source pad.
ppb_get_src_audio_metadata	Function	Returns the pointer to the metadata structure of the current audio chunk in the selected source pad. For PPP with both source and sink pads, it returns the same pointer as ppb_get_sink_audio_metadata.
ppb_get_sink_audio_metadata	Function	Returns the pointer to the metadata structure of the current audio chunk in the selected sink pad. For PPP with both source and sink pads, it returns the same pointer as ppb_get_src_audio_metadata.

Name	Type	Description
ppb_get_src_cust_metadata	Function	Returns the pointer to the customer metadata of the current audio chunk in the selected source pad. For PPP with both source and sink pads, it returns the same pointer as ppb_get_sink_cust_metadata.
ppb_get_sink_cust_metadata	Function	Returns the pointer to the customer metadata of the current audio chunk in the selected sink pad. For PPP with both source and sink pads, it returns the same pointer as ppb_get_src_cust_metadata.
cpu_to_pts_clock	Function	Converts CPU system clock to PTS timebase

Additionally, Audio Framework provides a set of parsing functions to interpret the commands received by the Post Processing Plugin:

Table 3. Post Processing parser API

Name	Type	Description
ppp_node_s	Structure	Node structure for Capabilities tree
ppp_type_and_size_t	Structure	Structure specifying the size and type of a property value
ppp_send_command	Function	Function used to send the REST API commands (POST, GET, PUT, and DELETE)
ppp_from_caps_to_case	Function	Provides the “key” properties from a Capabilities string
ppp_read_next_property_set	Function	Provides the “key” property and the “value” to set from a command string
ppp_insert	Function	Inserts a key property on a Capabilities tree
ppp_search	Function	Searches a key property on a Capabilities tree
ppp_delete_tree	Function	Deletes a key property on a Capabilities tree
ppp_set_string_to_type	Function	Converts a string to the defined type and size
ppp_get_string_from_type	Function	Converts key/value pair to "key=value"
ppp_get_type_and_size	Function	Parses type and size in a string
ppp_get_size	Function	Parses size of an array in a string
ppp_add_to_return_string	Function	Concatenates a string into the given one
ppp_read_next_property_to_get	Function	Provides the “key” property to return from a command string
ppp_read_next_property_to_set	Function	Provides the “key” property and its value to set from a command string
ppp_get_root	Function	Returns the root node of a PPP

Name	Type	Description
ppp_strdup	Function	Duplicates a string, returning an identical malloc'd string

Note that the `type_size` argument of functions `ppp_set_string_to_type` and `ppp_get_string_from_type` correspond to the following strings and match the corresponding types:

Table 4. Post Processing type_size string

String	C type
PPP_BOOL	"bool"
PPP_BOOL_ARRAY	"bool[N]"
PPP_CHAR	"char"
PPP_CHAR_ARRAY	"char[N]"
PPP_FLOAT	"float"
PPP_FLOAT_ARRAY	"float[N]"
PPP_DOUBLE	"double"
PPP_DOUBLE_ARRAY	"double[N]"
PPP_INT8	"int8_t"
PPP_INT8_ARRAY	"int8_t[N]"
PPP_INT16	"int16_t"
PPP_INT16_ARRAY	"int16_t[N]"
PPP_INT32	"int32_t"
PPP_INT32_ARRAY	"int32_t[N]"
PPP_INT64	"int64_t"
PPP_INT64_ARRAY	"int64_t[N]"
PPP_UINT8	"uint8_t"
PPP_UINT8_ARRAY	"uint8_t[N]"
PPP_UINT16	"uint16_t"
PPP_UINT16_ARRAY	"uint16_t[N]"
PPP_UINT32	"uint32_t"
PPP_UINT32_ARRAY	"uint32_t[N]"
PPP_UINT64	"uint64_t"
PPP_UINT64_ARRAY	"uint64_t[N]"

Audio samples and metadata are packed by the Input Manager and pushed to the rest of the pipeline. CPPs can retrieve this metadata for each chunk with the "ppb_get_XXX_metadata" API shown in Table 2. The Control Process also gets the same information from the decoder.

The metadata information contains: decoder id, number of channels, sampling rate, format_size, speaker id, timestamp, and PTS. Note that for the "audio_metadata" structure, the "decoder_id" value corresponds to the decoder types defined in `sdk/public/include/common/af_types.h`. Additionally, the "speaker_id" is an array with values corresponding to the channel names defined in `sdk/public/include/common/iec62574.h`, because they follow the IEC62574 norm.

The customer metadata provides a memory region for customers to fill with information to be transmitted along the pipeline. The "ppa_cust_md_size" REST API can be used to fix the size of this memory region in bytes: `PUT pipeline0/pipeline.elt/0/ppa_cust_md_size=3840`.

Both metadata and customer metadata are passed in a sequential order based on how elements are linked in the pipeline and synchronized with each audio chunk.

3.3. Implementing and integrating a custom plugin

3.3.1. Creating a custom post processing plugin

Every post processing plugin must register three functions:

- `static char *MyParser(struct cowbell_context *context, enum ppp_command_type cmd, char *command)`: This function interprets and executes the REST API commands passed as an argument.
- `static const char *MyGetCaps(void)`: This function returns the capabilities of the post processing plugin.
- `static const char *MyPostProcessing(struct cowbell_context *context, size_t len)`: This function performs the audio processing of the plugin to a chunk of bytes determined by `len`.

Additionally, there are two optional functions that can be used to make or log changes at the beginning/ending of the element:

- `static void *Start(struct cowbell_context *context)`: This function is called at the start of a new stream after the decoder has received the first audio sample. This function can be used to initialize/reset the CPP when receiving a new stream.
- `static void *Stop(struct cowbell_context *context)`: This function is called at the end of a stream, particularly when the pipeline is flushed.

These functions are needed by the Audio Framework to interact with the PPP through the cowbell_driver structure as follows:

```
static struct cowbell_driver ppp_driv = {
    .ops = {
        .start = volume_start, //optional
        .stop = volume_stop, //optional
        .parser = MyParser,
        .process = MyPostProcessing,
        .get_caps = MyGetCaps,
    },
    .compat = "ppp2af.elt"
};
```

Finally, the post processing plugin must be registered into the Audio Framework:

`register_ppp_driver(&ppp_driv)`;

3.3.2. Integrating a PPP to the pipeline

Once a PPP is developed, it must be included into the pipeline. The post processing pipeline (section of the pipeline where PPP can be added) start and end points are defined by “ppa_head” and “ppa_tail” properties. If the new element is not used as the starting or ending component of the pipeline, then it doesn’t need to be included in one of the previously mentioned properties. Please notice that this assumes other elements to be present and correctly linked between each other.

A new REST command file must be created to include new elements. Users can get inspiration from the default.rest file on the sdcard image (`/usr/local/share/rest/`). Be aware that <ppp_compat> corresponds to the compat field of `cowbell_driver`.

```
POST Element=pipeline0/<ppp_compat>/<ppp_name>
PUT pipeline0/pipeline.elt/0/ppa_head=<ppp_name>&ppa_tail=<ppp_name>
```

3.3.3. Compiling and running a new post processing plugin

Currently, the Audio Framework provides an SDK with libraries and public header files allowing to create a Little Kernel application linked to a customized plugin. This plugin is given as an example and allows to control the volume of the audio stream. Pipeline, capabilities, user structure, callbacks, and processing can be completely customized. Once post processing plugin modifications are done, export the build environment toolchain:

```
$ export ARMGCC_DIR=/<custom_path>/gcc-linaro-7.3.1-2018.05-x86_64_aarch64-elf
```

Run `/build_pp_imx8mm_release.sh` from `/path/to/sdk/build/cmake/` to generate binary, elf, and map files at `/path/to/sdk/build/cmake/pp_release`:

```
$ ./clean.sh
$ ./build_pp_imx8mm_release.sh
```

Finally, run this new binary application on the target in the same way as the pp_sample.

Similar steps can be done to build an SDK in the debug mode, using the `build_pp_imx8mm_debug.sh` script.

Please note: When an SDK is built in the debug mode, it still uses the prebuilt libraries from AF that are built in the release mode.

Audio Framework also provides a SHELL available on the second COM Port to communicate with the PPP. Commands should respect the following syntax:

```
ppp cmd "POST Element=pipeline0/volume.elt/volume0"
ppp cmd "POST Element=pipeline0/volume.elt/volume1"
ppp cmd "POST Link=pipeline0/volume.elt/volume0&pipeline0/volume.elt/volume1"
ppp cmd "GET pipeline0/<compat.elt>/<element>/<property1>&<property2>"
ppp cmd "PUT
pipeline0/<compat.elt>/<element>/<property1>=<value1>&<property2>=<value2>"
ppp cmd "DELETE Link=pipeline0/volume.elt/volume0&pipeline0/volume.elt/volume0"
ppp cmd "DELETE Element=pipeline0/volume.elt/volume0"
ppp cmd "DELETE Element=pipeline0/volume.elt/volume1"
```

A Linux interface has also been developed to access PPP. This interface is located under `/sys/devices/platform/bb800000.pci/pci0000:00/0000:00:00.0/0000:00:00.0.rpmsg_ppp.-1.-1/`. To send commands to Audio Framework, write the PPP file at that location. To retrieve information from Audio Framework, read that same file. Additionally, Audio Framework provides the `afrun.sh` script to send commands through this interface.

For example:

```
root@imx8mmevk:~# afrun.sh /dev/stdin
running: /dev/ttymxc1
PUT pipeline0/volume.elt/volume0/gain=1
> PUT pipeline0/volume.elt/volume0/gain=1
< OK
```

A particular command (from both Little Kernel and Linux shell) allows to create new elements that will share the same user data. This way, modifying a property of one element will impact all “connected” elements. Note that, only one new connected element can be created per command. Deleting connected elements is possible as long as the “child” element is deleted before the “parent" one. However, connecting an element to another element already sharing user data will actually connect it to the second one’s parent (see the example below).

In the following example, volume1, volume2, and volume3 share the same user data and volume1 is the “parent” element for both volume2 and volume3:

```
"POST Pipeline=pipeline1"
"POST Element=pipeline1/volume.elt/volume1"
"POST Element=pipeline1/volume.elt/volume2 pipeline1/volume.elt/volume1"
"POST Element=pipeline1/volume.elt/volume3 pipeline1/volume.elt/volume2"
```

3.4. Custom post processing example

Audio Framework provides a post processing plugin example: volume. This plugin allows to control the volume of the audio stream by adding gain to it.

The first step is to create a structure containing data specific to the post processing plugin. In this case, we only need the gain.

```
struct volume_data {
    float gain;/**< @brief Gain to be added to the audio stream */
};
```

Then, we need to implement the PPP callbacks. As for the capabilities for this example, we have a single property: gain. Be aware that each capability must be separated by the '&' character.

```
static const char *volume_get_caps(void)
{
    return "numsink=32&numsrc=32&gain=property";
}
```

The parser interfacing the REST API with the PPP implements each PPP_COMMAND to allocate the PPP structure, delete it, update the gain of the PPP, and return it. Audio Framework provides different helpers for this. Notice that the POST command is used to initialize the gain to a default value.

```
static char *volume_parser(struct cowbell_context *context,
                           enum ppp_command_type cmd, char *command)
{
    struct volume_data *data;
    int property_ret = 0, ret = 0;
    char *ptr_key = NULL;
    char *ptr_value = NULL;
    char *return_string = NULL;
    bool ppp_error = false;
    char *data_string = NULL;
    char *saveptr = NULL;
    cp_event_volume_t param;

    switch (cmd) {
    case PPP_COMMAND_POST:
        printlk(LK_DEBUG, "'%s' received POST command\n", context->name);
```

```
        data = osa_malloc(sizeof(struct volume_data));
        if (!data)
            return PPP_ALLOC_STRING_ERROR;

        context->user_data = data;
        /* Set default values */
        data->gain = 1.0f;
        break;
    case PPP_COMMAND_DELETE:
        osa_free(context->user_data);
        break;
    case PPP_COMMAND_PUT:
        data = (struct volume_data *) context->user_data;
        /* Proposed helper to parse command line */
        property_ret = ppp_read_next_property_to_set(command, &ptr_key, &ptr_value,
&saveptr);
        while (property_ret == ERRCODE_NO_ERROR && ppp_error == false) {
            PPP_SWITCH (ptr_key) {
            PPP_CASE ("gain"):
                /* Proposed helper to convert string to expected type */
                ret = ppp_set_string_to_type(ptr_value, &data->gain, "float");
                if (ERRCODE_NO_ERROR != ret) {
                    printlk(LK_ERR, "Error: Invalid command \"%s=%s\"\n", ptr_key,
ptr_value);
                    return PPP_ALLOC_STRING_ERROR;
                }
                param.volume = data->gain;
                /* Send Event of gain change to CP */
                ret = cp_send_event(0, CP_EVENT_CPP_VOLUME, (void *) &param,
sizeof(param));
                if (ERRCODE_NO_ERROR != ret) {
                    printlk(LK_ERR, "Error: Failed Send Event to CP\n");
                    return PPP_ALLOC_STRING_ERROR;
                }

                PPP_BREAK;

            PPP_DEFAULT:
                printlk(LK_ERR, "Error: Key \"%s=%s\" not found\n", ptr_key,
ptr_value);
                ppp_error = true;
                PPP_BREAK;
            }
            property_ret = ppp_read_next_property_to_set(NULL, &ptr_key, &ptr_value,
&saveptr);
        }

        return (ppp_error == false) ? PPP_ALLOC_STRING_SUCCESS :
PPP_ALLOC_STRING_ERROR;
    case PPP_COMMAND_GET:
        data = (struct volume_data *) context->user_data;
```

```
        /* Proposed helper to parse command line */
        property_ret = ppp_read_next_property_to_get(command, &ptr_key, &saveptr);
        while (property_ret == ERRCODE_NO_ERROR) {
            PPP_SWITCH (ptr_key) {

            PPP_CASE ("gain"):
                /* Proposed helper to convert type to expected string */
                data_string = ppp_get_string_from_type(ptr_key, &data->gain, "float");
                PPP_BREAK;

            PPP_DEFAULT:
                printlk(LK_ERR, "Error: Key \"%s\" not found\n", ptr_key);
                data_string = PPP_ALLOC_STRING_ERROR;
                PPP_BREAK;
            }

            /* Concatenate current string to return string */
            ppp_add_to_return_string(&return_string, data_string);
            /* Free memory allocated by ppp_get_string_from_type() */
            osa_free(data_string);
            property_ret = ppp_read_next_property_to_get(NULL, &ptr_key, &saveptr);
        }

        printlk(LK_DEBUG, "PPP_COMMAND_GET returns = %s\n", return_string);

        return return_string ? return_string : PPP_ALLOC_STRING_ERROR;
    default:
        return PPP_ALLOC_STRING_ERROR;
    }

    return PPP_ALLOC_STRING_SUCCESS;
}
```

The audio processing of the Volume plugin adds the specified gain to the audio stream.

```
static const char *volume_process(struct cowbell_context *context, size_t len)
{
    struct volume_data *data = (struct volume_data *) context->user_data;
    float *psink;
    size_t samples_count;
    size_t i, j;

    if (len % sizeof(float)) {
        printlk(LK_ERR, "Do not support this buffer len :%lu\n", len);
        return PPP_FIX_STRING_ERROR;
    }

    samples_count = len / sizeof(float);
    for (i = 0; i < AUDIO_CHANNELS_MAX; i++) {
        psink = (float *) ppb_get_sink(context, i);

        if (psink == NULL)
            continue;

        for (j = 0; j < samples_count; j++)
            *psink++ *= data->gain;
    }

    return PPP_FIX_STRING_SUCCESS;
}
```

There are two additional callbacks than can be included in the element: `start()` and `stop()` callbacks. Please note that these are optional. The following example shows how to include them:

```
static void volume_start(struct cowbell_context *context)
{
    struct volume_data *data = (struct volume_data *)context->user_data;

    printlk(LK_DEBUG, "volume start:%f\n", data->gain);
}

static void volume_stop(struct cowbell_context *context)
{
    struct volume_data *data = (struct volume_data *)context->user_data;

    printlk(LK_DEBUG, "volume stop:%f\n", data->gain);
}
```

Finally, the driver structure is created and the Volume plugin is registered.

```
static struct cowbell_driver ppp_volume = {
    .compat = "volume.elt",
    .ops = {
        .start = volume_start,
        .stop = volume_stop,
        .parser = volume_parser,
        .process = volume_process,
        .get_caps = volume_get_caps,
    },
};

static void __attribute__ ((constructor)) volume_init(void)
{
    register_ppp_driver(&ppp_volume);
}
```

Chapter 4. Little Kernel services

Little Kernel already provides several services that can be used by custom post processing plugins and the board adaptation files.

4.1. General purpose timer

The SDK provides a driver for the General Purpose Timer (GPT) of the i.MX 8M at `sdk/public/include/drivers/`. This driver allows PPPs to configure, start, stop, and get the counter of a selected GPT. Additionally, the CAPTURE feature can be enabled and configured with a callback. Please note that the COMPARE feature is not available. For more information on the GPT, see the i.MX 8M Reference Manual.

4.2. Custom IPC

Immersiv3D provides an interface for Linux and Custom Post Processing Plugins (CPP) to exchange up to 8 MB of binary data.

4.2.1. Provided CIPC endpoint

4.2.1.1. File interface

Immersiv3D provides a Linux daemon “ivshm_binary”, allowing to abstract the CIPC interface into a file-based exchange between Little Kernel and Linux. Indeed, the following API allows CPPs to directly read or write files in the Linux file system:

```
ssize_t cipc_size_file(unsigned id, char *name);
ssize_t cipc_read_file(unsigned id, void *buf, size_t len, char *name);
ssize_t cipc_write_file(unsigned id, void *buf, size_t len, char *name, unit32_t
oflags);
```

4.2.1.2. Direct device access

On the Linux side, a new device (`/dev/cipc`) is exposed for users to interact with this interface. To access it, Linux applications can do file operations like open, read, write, and close to send/receive data to/from the CIPC interface. For example:

```
write(/dev/cipc, "Hello from Linux", sizeof(char) * strlen("Hello from Linux"));
```

On the LK side, a set of API has been exported into the SDK so that CPPs can write and read the same CIPC interface:

```
size_t cipc_read_buf(unsigned id, void *buf, size_t len);
size_t cipc_write_buf(unsigned id, void *buf, size_t len);
```

Note that the "id" argument that corresponds to the Endpoint ID of the CIPC interface should be set to 0x203 to communicate with the endpoint under /dev/cipc.

Be aware that these functions on the LK side and the file operation on Linux only write and read to the CIPC buffer. They don't provide an event to notify the receiver of new data in the pipeline. However, this event can be simulated through the REST API, a timer, or any other signal depending on the use case.

For example, two new capabilities can be added to the CPP to act as a flag: LK_read_cipc and Linux_read_cipc. On one hand, Linux will send the LK_read_cipc REST command to notify that CPP can read the new data written by Linux. On the other hand, Linux can poll using the REST command to detect when Linux_read_cipc capability is set by the CPP. In this case, Linux can read the /dev/cipc to retrieve the new information. An example can be found in Appendix A.

4.2.2. Adding a new CIPC endpoint

Immersiv3D allows customers to create new CIPC endpoints to include new binary path into the system. If more than one CPP must communicate through the binary path with Linux, new CIPC endpoints must be created.

On the Linux side, the new endpoint must be declared as a node in the Linux device tree. For example, if users want to create a new “lpf_cipc”:

```
& ivshm_rpmsg {
    lpf_cipc {
        compatible = "fsl,rpmsg-binary";
        id = <0x400>;
        size = <8192>; /* Endpoint buffer size (B) */
        buffer = <8>; /* binary buffer size (MB) */
    };
};
```

Please note that all custom nodes must have an ID equal to or higher than 0x400. All lower IDs are reserved for internal Immersiv3D use.

This new node will automatically expose the “lpf_cipc” endpoint under /dev/lpf_cipc. Users can then do file operations as they are done for the /dev/cipc endpoint.

On the Little Kernel side, the new endpoint must be declared as a node in the LK device tree, using the same example as above:

```
lpf_cipc {
    compatible = "imx_ivshm_binary";
    size = <8192>; /* Endpoint buffer size (B) */
    buffer = <8>; /* binary buffer size (MB) */
    id = <0x400>;
    status = "ok";
};
```

Please note that id parameters of the node must be the same on both Linux and LK device trees. The remaining size, buffer, or buffer_bytes parameter can be tuned according to the use case:

- the size parameter aims to determine the maximum chunk per transfer (must be set to 8KB for a large file transfer).
- the buffer or buffer_bytes parameters aim to determine the receive buffer size available for an application.

Finally, the CPP can use the same CIPC API described in the previous chapter to communicate with the CIPC interface. For this example, the “id” argument of the functions should be 0x400.

Chapter 5. Hardware abstraction layer

Audio Framework provides a Hardware Abstraction Layer (HAL) to abstract the different source and sink devices from the pipeline. This allows Immersiv3D to have a single API to communicate with all types of inputs and output devices.

5.1. Input and output abstraction

The Hardware Abstraction Layer (HAL) is used to integrate Immersiv3D into an audio board different from the i.MX Audio Board (MCIMX8M-AUD) reference board. Currently, the HAL provides an input and output interface between Audio Framework pipeline and the LK source/sink drivers.

HAL can be used only for the RPC interface. The example `imx8mm-ab2-rpc.dts`, `imx8mn-ab2-rpc.dts`, and `imx8mnul-ab2-rpc.dts` device trees specify the `hal-input` and `hal-output` to use the `hdmi_lnx` and `dac_lnx` nodes, which correspond to the RPC interface. For more information on this interface, see section Section 7.1.2.

5.2. HAL API

Inside HAL, the available IO devices are initialized through input and output streams. Available streams are identified through hdmi, spdif, dac, dac2, adc, adc2, alsa, alsa-voice, and alsa-cpp. These are defined inside `hal-input` and `hal-output` nodes from the device tree files.

After a stream is registered inside HAL, it can be obtained with the following function:

```
struct audio_hal_stream * audio_hal_get_stream_by_name(const char *name)
```

This returns the corresponding HAL stream, which was registered at HAL initialization. The structure of HAL stream is explained in Table 5.

Table 5. Audio HAL stream parameters

Name	Type	Description
hw_device	Structure	Opaque type holding the hardware device drivers details
list_node node	Structure	Node to hook the element to a list within stream manager
direction	Structure	Stream direction
stream_type	Structure	Stream type currently in use
capabilities	Unsigned integer	Stream capabilities
get_name	Function	Returns the stream name as reported by the hardware device driver
get_stream_capabilities	Function	Returns a mask holding the capabilities of the stream

Name	Type	Description
set_stream_type	Function	Sets the stream type, returning 0 in case of success and a negative value otherwise
get_sample_rate	Function	Returns the stream sampling rate in Hz
set_sample_rate	Function	Sets the sample rate, returning 0 in case of success and a negative value otherwise
get_pcm_format	Function	Returns the PCM data format, determining bitwidth and endianness
set_pcm_format	Function	Sets the PCM data format, returning 0 in case of success and a negative value otherwise
get_period_size	Function	Returns the period size in frame, or a negative value in case of error
set_period_size	Function	Sets the period size, returning 0 in case of success, or a negative value in case of error
get_channels	Function	Gets the number of channels, returning 0 in case of success, or a negative value in case of error
set_channels	Function	Sets the number of channels, returning 0 in case of success, or a negative value in case of error
set_callback	Function	Sets the callback function for notifying stream changes and non-blocking actions completion
get_latency	Function	Returns audio hardware estimated latency in microseconds
get_timestamp	Function	Gets the timestamp of a specific HAL event, returning 0 if call is successful or a negative value otherwise
set_timestamp	Function	Sets the timestamp of a specific HAL event, usually to defer HAL actions, returning 0 if a call is successful or a negative value otherwise
get_buffer_size	Function	Gets the buffer size of the interface, returning 0 if a call is successful or a negative value otherwise.
get_bitrate	Function	Gets the estimated bit rate of the interface, returning 0 if a call is successful or a negative value otherwise
get_custom_format_layout	Function	Gets a custom format layout, returning 0 if a call is successful or ERR_NOT_SUPPORTED if a custom layout is not supported
open	Function	Opens the stream, returning status 0 for success, or a negative status otherwise

Name	Type	Description
close	Function	Closes the stream, returning status 0 for success, or a negative status otherwise
set_parameters	Function	Sets the audio stream parameters, returning status 0 for success, or a negative status otherwise
start	Function	Starts the stream, returning status 0 for success, or a negative status otherwise
stop	Function	Stops the stream, returning status 0 for success, or a negative status otherwise

The detailed description for the HAL interface can be found in the SDK release archive under the `public/include/hardware/audio.h` header file.

5.3. Audio data from LK

Audio Framework provides a method for sending audio data from Little Kernel to Linux. For this, a PPP element can be created, which extracts the channels data and sends it to Linux ALSA.

The ALSA stream object is obtained from HAL through `alsa-cpp-output` naming using the `audio_hal_get_stream_by_name` API. This can be done in a POST function of the new PPP element. After obtaining the corresponding HAL stream structure, it can be managed through the parameters mentioned in Table 5.

Configuration of the stream parameters can be done in the PUT function, assuming the stream has been initialized and obtained when PPP was created. Here are some examples of stream parameters that must be considered:

- Stream flow (open/start/stop/close)
- Data format (set_pcm_format)
- Number of channels (set_channels)

The HAL API offers the possibility of changing stream parameters, but the caller is in charge of formatting and writing the stream to HAL accordingly. This audio processing must be done in the '.process' function of the PPP element. An ALSA-specific scenario will firstly require a conversion of the data stream from float to integer and interleaving the input channels to match PPP format. Then, the converted data can be written with the following function:

```
ssize_t (*write)(struct audio_hal_stream_out *stream, void *buffer, size_t length);
```

On the Linux side, the ALSA capture parameters must match the pipeline stream configuration. A new ALSA device “AFppp” is available to interact with this interface.

Chapter 6. Control process

Please note that not all release packages provide access to the control process.

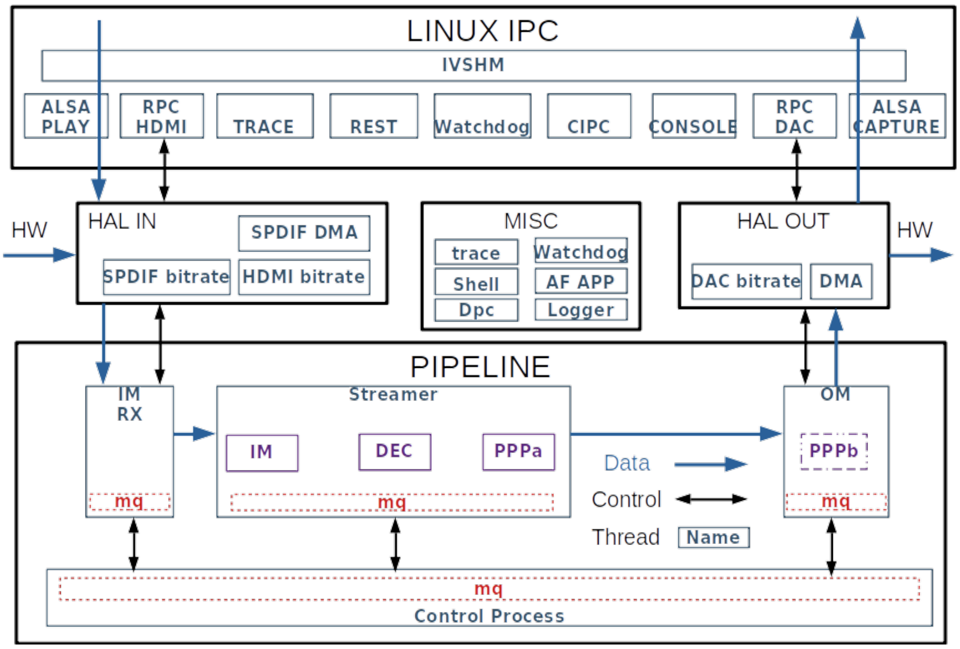

Figure 3. Audio Framework threads diagram

Audio flow control is handled by the control process thread which communicates with audio threads through message queues. Message queues involved in the control process message communication are blocking on incoming messages and running within their own thread.

The Immersiv3D SDK provides the source code for the main tasks performed by the control process under `sdk/public/source/cp/public/`.

6.1. Control process main

The “cp_main” file implements the main thread of the control process, where it initializes the message interface with all the other threads (input manager, decoder, post processing, and output manager). Additionally, it initializes the other tasks of the control process: pipe, ping, and mute. Finally, the main thread of control process will redirect all incoming messages to the correct handler.

6.2. Control process pipe

The pipe of the control process handles the sequencing of the pipeline, depending on the state of the control process and the messages received by the other elements. The control process integrates the following finite state machine:

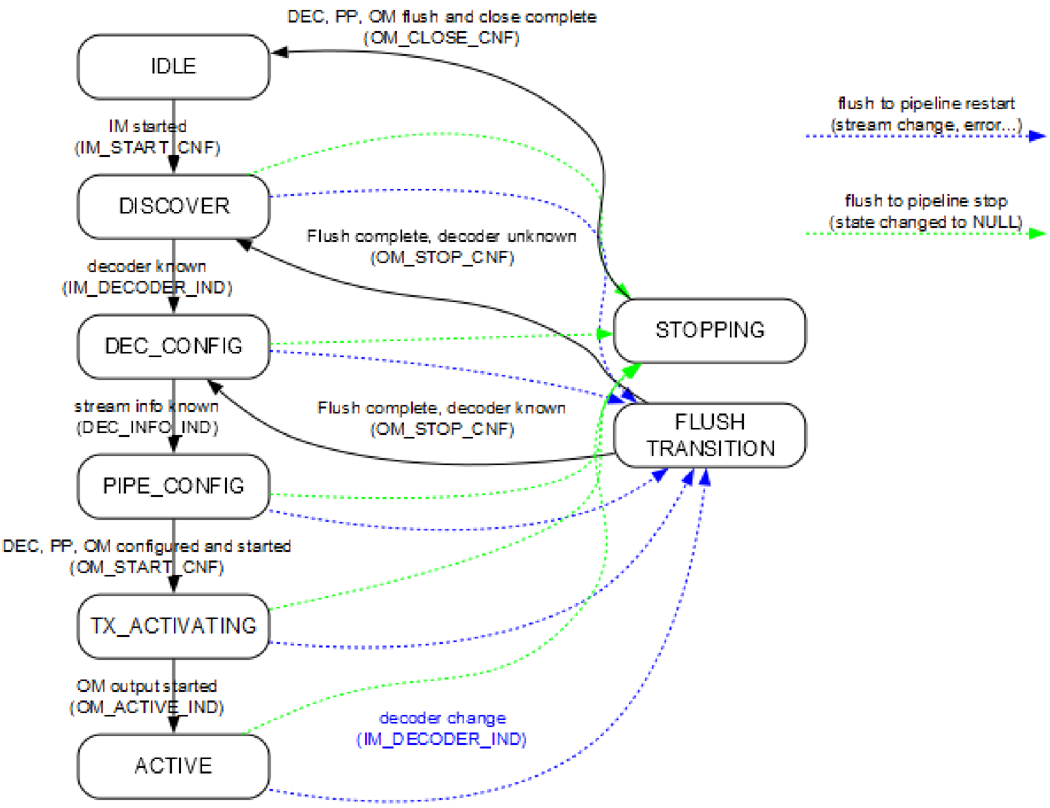

Figure 4. Control process finite state machine

The `cp_pipe` file specifies the handler for each state (`cp_pipe_state_table`). For each state, the message determines the action to be sent to the correct element.

6.2.1. IDLE state

This is the state of the control process once it has been initialized. According to each message received on this state, the control process will notify the correct element to start its configuration. Once the `IM_START_CONF` message is received, the control process will pass to the discover decoder state. Figure 5 shows a sequence diagram of the pipeline setup while the control process is in the idle state.

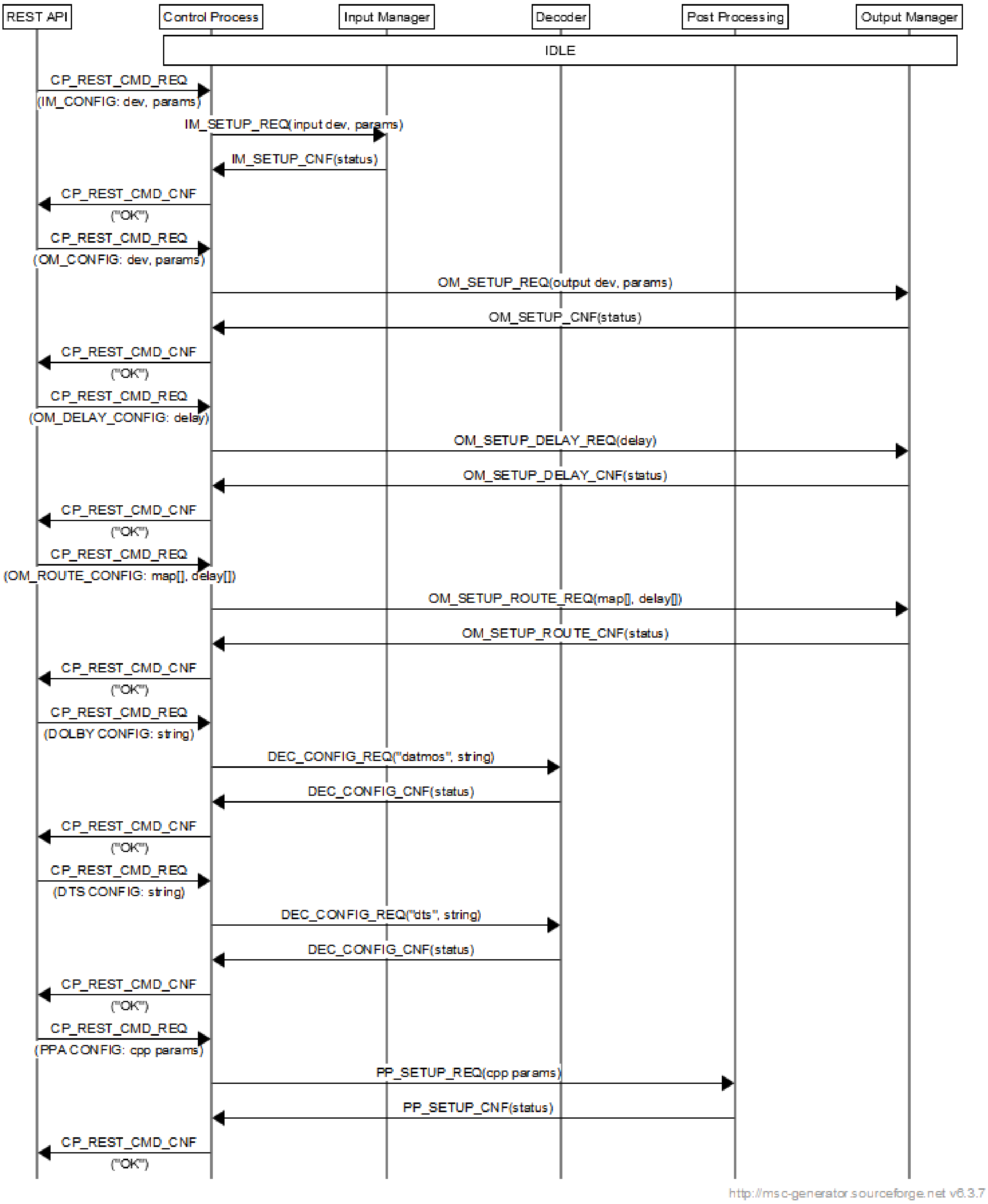

Figure 5. Pipeline setup sequence diagram

6.2.2. Discover decoder state

During this state, the input manager will drop the audio input until it detects the type of decoder needed for that stream. Once the type of decoder is discovered, the `IM_DECODER_IND` message is passed to the control process to switch to the decoder configuration state. Figure 6 illustrates this sequence.

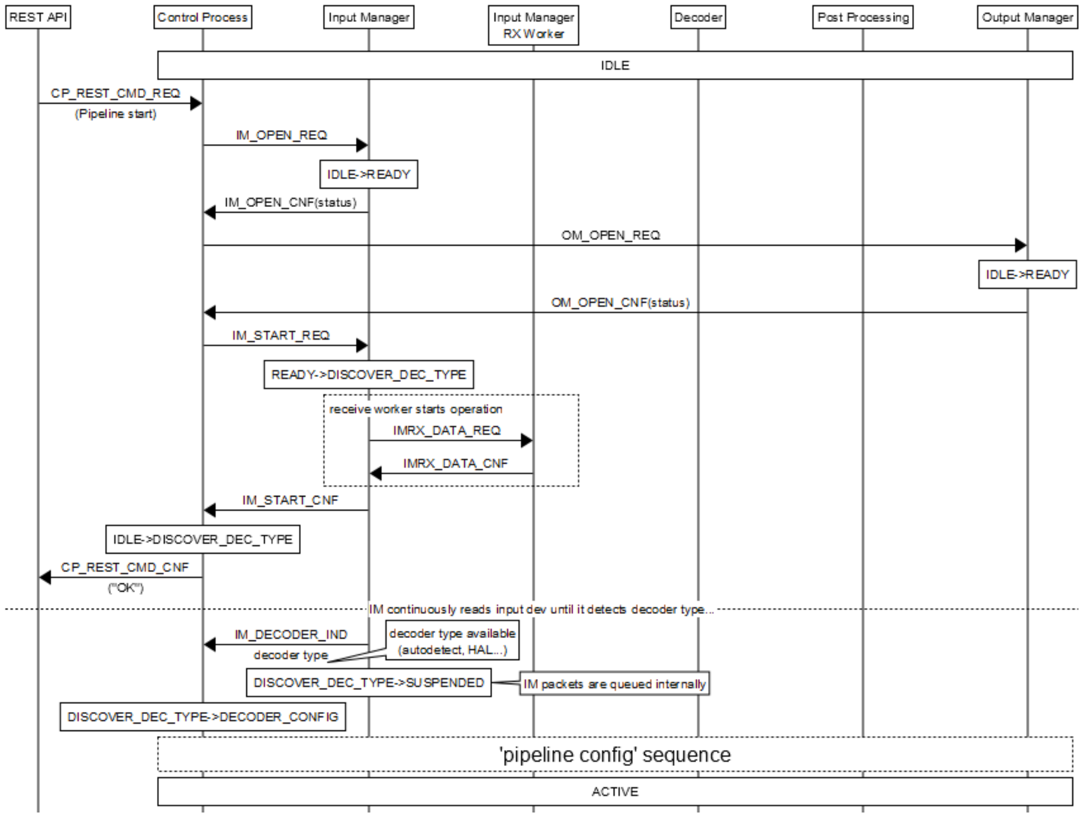

Figure 6. Pipeline start sequence diagram

6.2.3. Decoder configuration state

The decoder configuration state notifies the decoder to start decoding the first frame to detect the format of the audio stream and its configuration. Once this is done, the `DEC_INFO_IND` message is passed to the control process to switch to the pipeline configuration state. This sequence is shown in Figure 7.

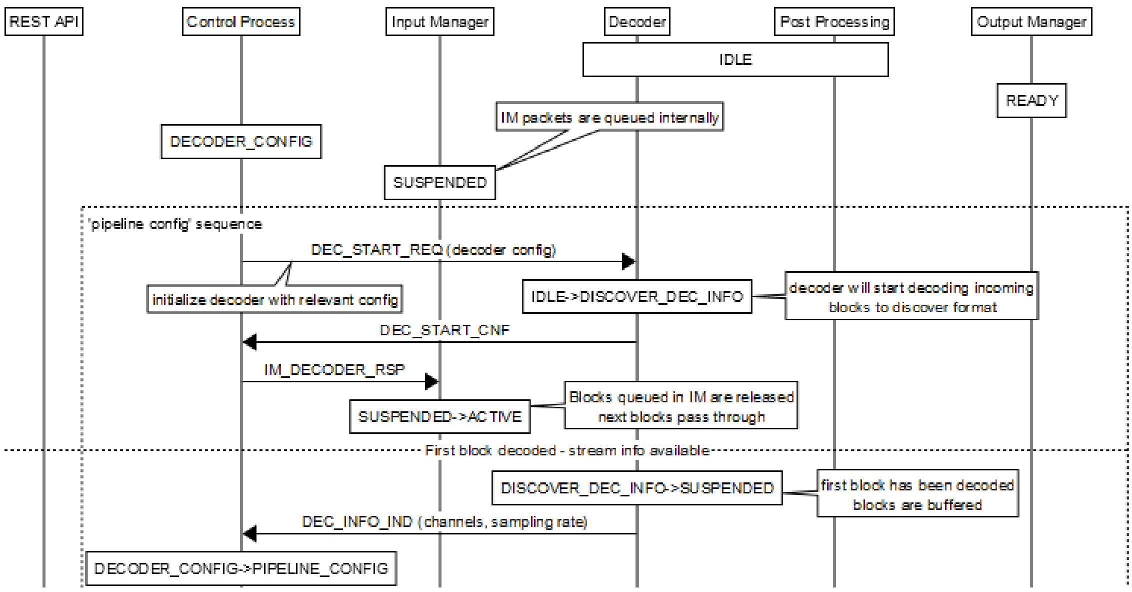

Figure 7. Decoder configuration sequence diagram

6.2.4. Pipeline configuration state

During this state, the control process activates the rest of the elements of the pipeline. Notice that the decoder is stall during this state. Figure 8 shows the sequence of this state.

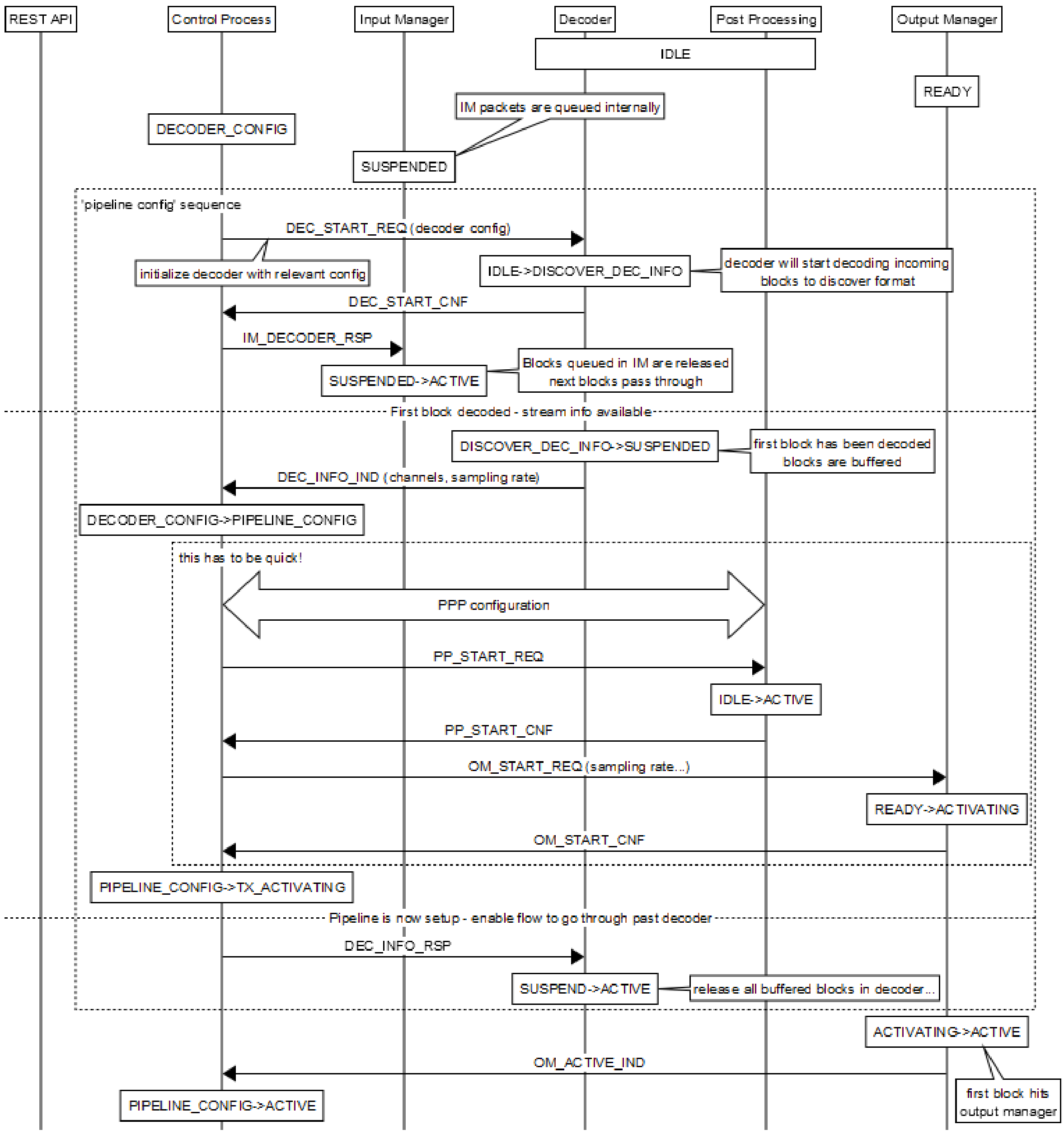

Figure 8. Pipeline configuration sequence diagram

6.2.5. TX activating state

The TX activating state will unblock the decoder and once the first decoded block arrives into the output manager, the `OM_ACTIVE_IND` message will make the control process pass to the active state. Details on this are shown in Figure 9.

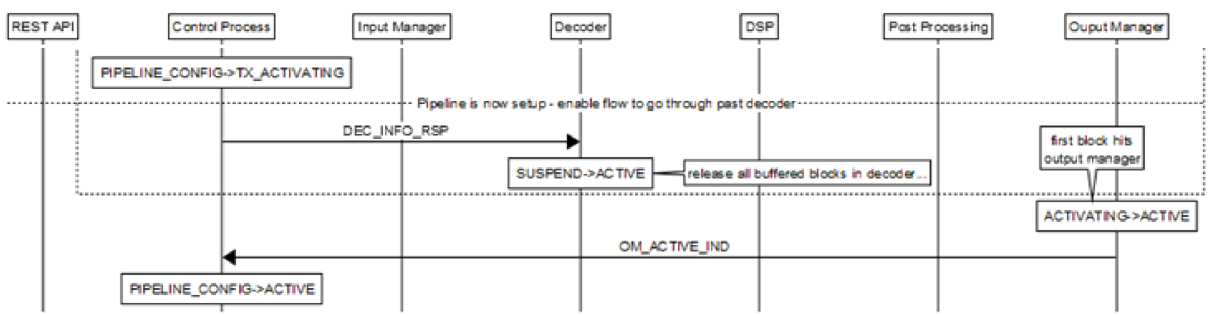

Figure 9. Active sequence diagram

6.2.6. Active state

This state corresponds to the systems state when audio is being streamed to the pipeline.

6.2.7. Flush transition state

The flush transition state is entered on a stream transition or when the pipeline is being stopped. The objective of this state is to inform every element that they must flush their buffers. Figure 10 provides the sequence diagram of this state.

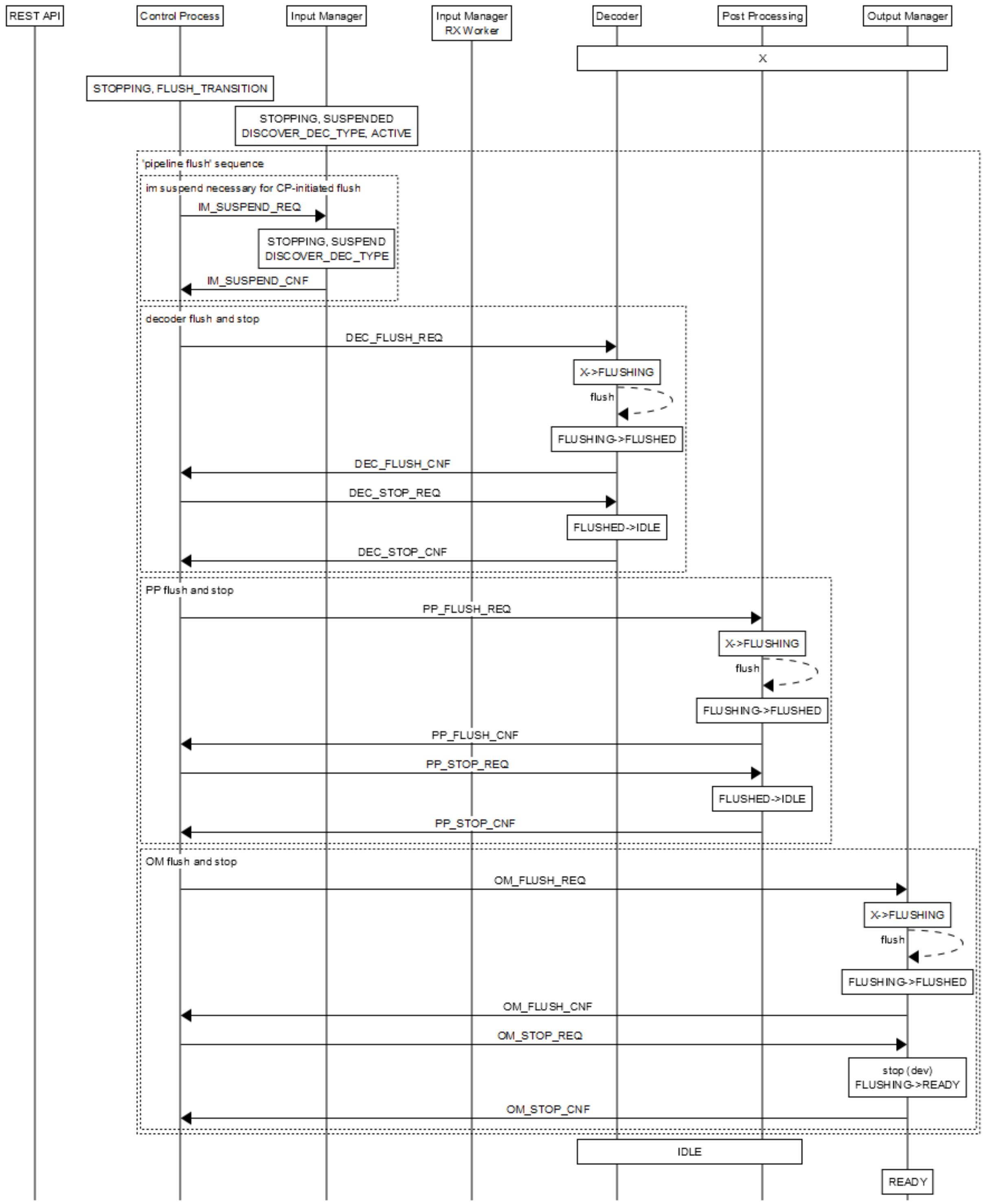

Figure 10. Flush transition sequence diagram

6.2.8. Stopping state

This state is entered when the pipeline must be closed. All elements are being set to the idle state and a flush transition sequence is also called. This sequence is described in Figure 11.

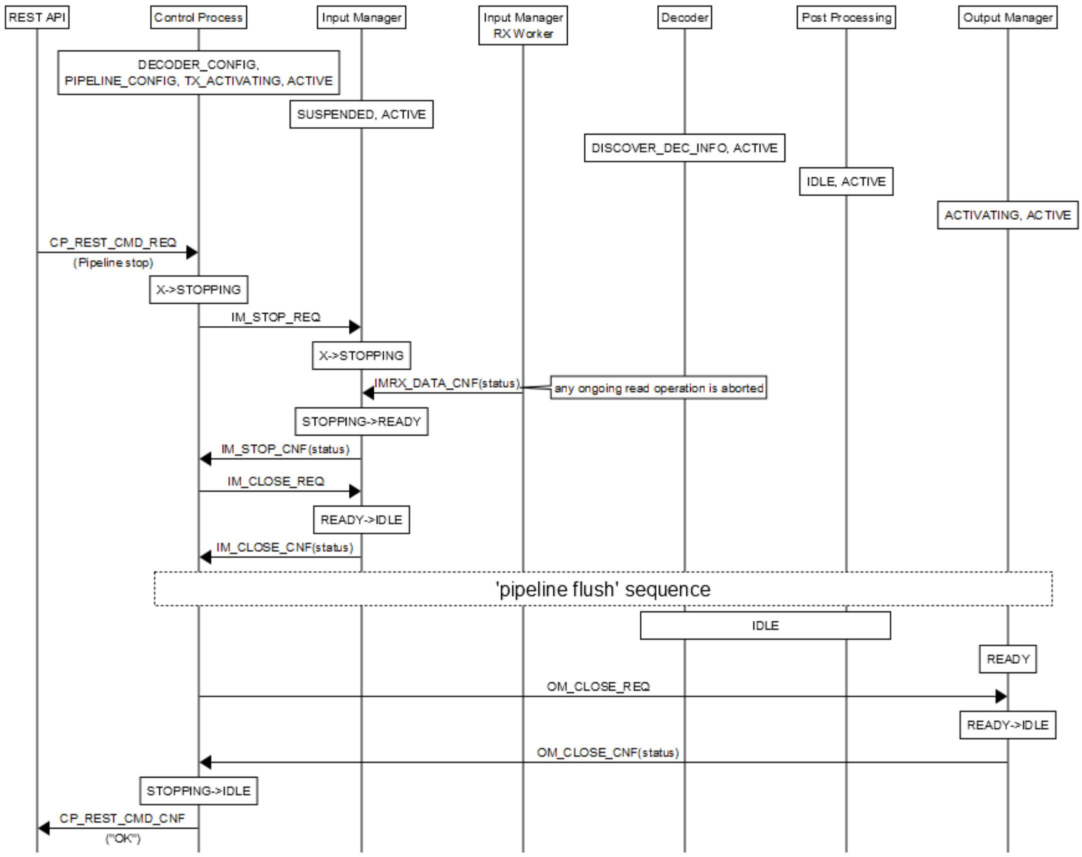

Figure 11. Stopping sequence diagram

6.3. Control process ping

The ping feature of the control process allows to ping the communication interface of every element handled by the control process. This is mainly used to make sure that the interface is correctly configured and active.

6.4. Control process mute

The mute feature allows to send events to the output manager to notify it about the pipeline and the decoder setup and the need to mute or unmute the output.

6.5. Frame configuration

The frame configuration details the different threads used by Immersiv3D, as well as their properties and their entry point functions.

6.6. Control process event notifier

The purpose of this feature is to allow any entity to register a callback for events in the control process to be notified about system states and updates.

This interface can be used as follows:

- Each entity/element will register the required callbacks for particular events.
- Callbacks will be called on a particular event.
- Entities can have information of CP state changes or other required information from any other elements.

The API involved here is shown in the next table. Details of the structures used in these functions are in the `cp_notify.h` header file.

cp_event_register	Function	Registers callback for getting notification from CP
cp_event_unregister	Function	Unregisters callback (delete entry) for getting notification from CP
cp_notify	Function	Calls entity registered callback

This is an example of usage from Little Kernel console:

```
/* Rgister callbacks */
] cp_event register
/* Unregister callbacks */
] cp_event unregister
```

6.7. Control process new event management

Immersiv3D offers an API that allows elements to send arbitrary notifications to the control process.

Following is the API that must be used:

cp_send_event	Function	Sends event notification to CP from external entities
cp_handle_event	Function	CP handler for received events

An example implementation is in Appendix B.

The corresponding header file (`cp_api.h`) must be included in the element when using this feature.

Here is an example of how to send an event when the gain value is changed in a volume element:

- Declare the event structure in the parser function:

  ```
  cp_event_volume_t param;
  ```

- Call the event function when the gain is updated:

```
                param.volume = data->gain;
                /* Send Event of gain change to CP */
                ret = cp_send_event(0, CP_EVENT_CPP_VOLUME, (void *) &param,
sizeof(param));
                if (ERRCODE_NO_ERROR != ret) {
                    printlk(LK_ERR, "Error: Failed Send Event to CP\n");
                    return PPP_ALLOC_STRING_ERROR;
                }
```

In this example, `CP_EVENT_CPP_VOLUME` is the example event created for the volume element.

Chapter 7. Board adaptation

Additionally to the custom post processing, Audio Framework provides a way to customize it and adapt it to different boards based on the i.MX 8M SOCs. This must consider the full architecture of the system: Linux, Little Kernel, and Jailhouse.

7.1. Linux configuration

On the Linux side, adapting Immersiv3D to a custom board implies both modifying or creating a device tree and developing the HDMI switch and DAC drivers using the RPC interface provided by Immersiv3D to communicate with the audio pipeline through the hardware abstraction layer.

7.1.1. Linux device tree

The Linux device tree provides the entire description of the hardware that will be configured and used by Linux. Each hardware module is represented by a node containing different properties. A list of current available properties specific to Immersiv3D is in Table 6. For the full list of available properties, see the Linux BSP documentation.

NXP's Linux BSP provides a device tree for the i.MX 8M SOCs (`imx8mm.dtsi` and `imx8mn.dtsi`) and several other device trees for specific boards using these SOCs. Immersiv3D uses 8M platform-specific dts (like `imx8mm-evk.dts` or `imx8mn-evk.dts`), Audio Board-specific dts (`*-ab2.dts`), dts specific for jailhouse implementation (`*-root.dts`), and dts specific for I3D configuration (like `i3d-base.dts`, `*-af.dts`, and the ones using RPC interface `*-rpc.dts`).

To adapt Immersiv3D to your board, use the `imx8mm.dtsi` or `imx8mn.dtsi` device trees and create your own `imx8mm-<user>.dts` or `imx8mn-<user>.dts` device tree that will enable/disable and configure the hardware resources of the board used by Linux. Finally, an audio framework device tree `imx8mm-<user>-af-rpc.dts` or `imx8mn-<user>-af-rpc.dts` can be used to add Immersiv3D specific nodes using the RPC interface. Please note that this implies that `imx8mm-<user>-af-rpc.dts` includes `imx8mm-<user>.dts`, which shall include `imx8mm.dtsi` (the same logic can be applied for imx8mn or imx8mnul).

Table 6. Linux device tree node properties

Immersiv3D Linux device tree properties	
compatible	The compatible property of a device node describes the specific binding or bindings, to which the node complies.
id	Determines the ID of the device. The interpretation of this ID might differ depending on the compatible driver. For CIPC and RPMSG-BIN nodes, the ID corresponds to the ID of the interface. Please notice that for these two nodes, the ID should be the same in Linux and LK device tree.
size	Determines the buffer of an IPC or RPMSG endpoint in bytes.
buffer, buffer_bytes	Determines the buffer size of the binary interface. The buffer parameter has the granularity of megabytes, whereas buffer_bytes has the granularity of bytes.

In the release package, the `imx8mm-evk-root.dts` file specifies the memory reserved for Jailhouse and its services. Please note that this must be aligned with the Jailhouse cell configuration.

```
&{/reserved-memory} {

    ivshmem_reserved: ivshmem@bbb00000 {
        no-map;
        reg = <0 0xbbb00000 0x0 0x00100000>;
    };

    ivshmem2_reserved: ivshmem2@bba00000 {
        no-map;
        reg = <0 0xbba00000 0x0 0x00100000>;
    };

    pci_reserved: pci@bb800000 {
        no-map;
        reg = <0 0xbb800000 0x0 0x00200000>;
    };

    loader_reserved: loader@bb700000 {
        no-map;
        reg = <0 0xbb700000 0x0 0x00100000>;
    };

    jh_reserved: jh@b7c00000 {
        no-map;
        reg = <0 0xb7c00000 0x0 0x00400000>;
    };

    /* 512MB */
    inmate_reserved: inmate@93c00000 {
        no-map;
        reg = <0 0x93c00000 0x0 0x24000000>;
    };
};

&{/reserved-memory/linux,cma} {
    alloc-ranges = <0 0x40000000 0 0x60000000>;
};
```

The `imx8mm-ab2-af.dts` file redefines the reserved memory for the inmate to assign it a specific value for Immersiv3D and it disables the resources that are going to be used by Little Kernel.

Please note that the node `ir_recv` has been disabled even though it is not used by Little Kernel. The IR receiver is using GPIO1 registers to handle interrupts (also used by Little Kernel) generating conflicts in the interrupt management. This is a limitation for the current architecture which does not allow to share the same GPIO controller between Linux and Little Kernel.

```
&{/} {
    reserved-memory {
        linux,cma {
            size = < 0x0 0x8000000 >;
        };
        rpmsg_reserved: rpmsg@0xb8000000 {
            no-map;
            reg = <0 0xb8000000 0 0x400000>;
        };
    };
};

&uart4 {
    status = "disabled";
};

&sdma2 {
    status = "disabled";
};

&sdma3 {
    status = "disabled";
};

&spdif1 {
    status = "disabled";
};

&micfil {
    status = "disabled";
};

&sai1 {
    status = "disabled";
};

&sai2 {
    status = "disabled";
};

&sai3 {
    status = "disabled";
};

&sai5 {
    status = "disabled";
};

&sai6 {
    status = "disabled";
};
```

```
&{/sound-spdif} {
    status = "disabled";
};

&{/sound-ak4458} {
    status = "disabled";
};

&{/sound-ak5552} {
    status = "disabled";
};
&ecspi2 {
    status = "disabled";
};

#ifndef RPC
&i2c3 {
    status = "disabled";
};

&i2c4 {
    status = "disabled";
};
#else

&i2c3 {
    status = "okay";

    pca6416: gpio@20 {
        status = "ok";
    };

    ak4458_1: ak4458@10 {
        compatible = "nxp,af,ak4458";
        reg = <0x10>;

        ak4458,pdn-gpio = < &pca6416 4 0>;

        rpmsg_rpc = <&rpmsg_rpc_dac>;
        status = "ok";
    };

    ak4458_2: ak4458@12 {
        status = "disabled";
        reg = <0x12>;
    };

    ak4458_3: ak4458@11 {
        status = "disabled";
        reg = <0x11>;
```

```
    };

    ak5552: ak5552@13 {
        compatible = "nxp,af,ak5558";
        reg = <0x13>;
        reset-gpios = <&pca6416 3 GPIO_ACTIVE_HIGH>;
        ak5558,pdn-gpio = <&pca6416 3 GPIO_ACTIVE_HIGH>;
        rpmsg_rpc = <&rpmsg_rpc_adc>;
        status = "ok";
    };
};

&i2c4 {
    clock-frequency = <100000>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c4>;
    status = "ok";

    ep9x: ep9x@61 {
        compatible = "nxp,af,ep92a7e";
        reg = < 0x61 >;
        status = "ok";

        ep92a7e,pw_en-gpio      = <&pca6416 6 0>;
        ep92a7e,reset-gpio      = <&pca6416 7 0>;
        ep92a7e,gpio0-gpio      = <&pca6416 1 0>;
        ep92a7e,gpio1-gpio      = <&pca6416 3 0>;
        ep92a7e,gpio2-gpio      = <&pca6416 5 0>;
        ep92a7e,irq-gpio        = <&pca6416 8 0>;
        ep92a7e,tx_mute-gpio    = <&pca6416 9 0>;
        rpmsg_rpc = <&rpmsg_rpc_hdmi>;
    };
};
```

Finally, the Linux device tree creates the nodes corresponding to the different services provided by Immersiv3D to interact between Linux and LK. Please note that these nodes shouldn't be modified, because they are necessary to the internal work of Immersiv3D. These are defined in i3d_base.dts:

```
/ {
#ifndef RPC
    i3d_options = "no-rpc";
#else
    i3d_options = "rpc";
#endif
    ivshm_rpmsg {
        compatible = "fsl,ivshm-rpmsg";
        prio = <0x62010300 0x62010301 0x62010302 0x62010303>; /* prio 98, SCHED_FIFO,
alsa ep ids */

        rpmsg_ppp {
```

```
            compatible = "fsl,rpmsg-ppp";
            id = <1>;
            size = <8192>;
        };
        rpmsg_console {
            compatible = "fsl,rpmsg-console";
            id = <2>;
            size = <16384>;
        };
        rpmsg_alsa {
            compatible = "fsl,rpmsg-alsa";
            id = <0x300 0x301 0x302 0x303>; /* main alsa + ppp + voice + microphones
*/
            size = <131072 131072 131072 131072>;
        };
        cipc {
            compatible = "fsl,rpmsg-binary";
            id = <0x203>;
            size = <1024>; /* Endpoint buffer size (B) */
            buffer_bytes = <1024>; /* binary buffer size (B) */
            status = "disabled";
        };
        rpmsg-bin {
            compatible = "fsl,rpmsg-binary";
            id = <0x201>;
            size = <8192>; /* Endpoint buffer size (B) */
            buffer_bytes = <32768>; /* binary buffer size (B) */
        };
        audio-weaver-rpmsg {
            compatible = "fsl,rpmsg-binary";
            id = <0x204>;
            size = <8192>; /* Endpoint buffer size (B) */
            buffer_bytes = <8192>; /* binary buffer size (B) */
            status = "disabled";
        };
        rpmsg-wd {
            compatible = "fsl,rpmsg-binary";
            id = <0x202>;
            size = <512>; /* Endpoint buffer size (B) */
            buffer_bytes = <512>; /* binary buffer size (B) */
        };
        lktraces {
            compatible = "fsl,rpmsg-binary";
            id = <0x200>;
            size = <8192>; /* Endpoint buffer size (B) */
            buffer = <8>; /* binary buffer size (MB) */
            no-overwrite; /* Do not overwrite buffer when full */
            status = "disabled";
        };
        af-event {
            compatible = "fsl,rpmsg-binary";
```

```
            id = <0x205>;
            size = <1024>; /* Endpoint buffer size (B) */
            buffer_bytes = <1024>; /* binary buffer size (B) */
            ascii-mode; /* read event as strings */
        };
        rpmsg_rpc_hdmi: rpmsg-rpc-hdmi {
            compatible = "fsl,rpmsg-rpc";
            id = <0x100>;
            size = <8192>;
        };
        rpmsg_rpc_dac: rpmsg-rpc-dac {
            compatible = "fsl,rpmsg-rpc";
            id = <0x101>;
            size = <8192>;
        };
        rpmsg_rpc_adc: rpmsg-rpc-adc {
            compatible = "fsl,rpmsg-rpc";
            id = <0x102>;
            size = <8192>;
        };
    };

    sound-rpmsg-main {
        compatible = "nxp,snd-af-ivshmem-pcm";
        nxp,name = "AF-main";
        nxp,card-id = <1>;
        nxp,compr;
        nxp,pcm-out;
        nxp,pcm-in;
        nxp,nb_chans = <16>;
        nxp,ep_id = <0x300>;
    };

    sound-rpmsg-ppp {
        compatible = "nxp,snd-af-ivshmem-pcm";
        nxp,name = "AF-ppp";
        nxp,card-id = <3>;
        nxp,pcm-in;
        nxp,nb_chans = <16>;
        nxp,ep_id = <0x301>;
    };

    sound-rpmsg-voice {
        compatible = "nxp,snd-af-ivshmem-pcm";
        nxp,name = "AF-voice";
        nxp,card-id = <2>;
        nxp,pcm-out;
        nxp,nb_chans = <2>;
        nxp,ep_id = <0x302>;
        nxp,period_time_min = <10666>;
        nxp,period_time_max = <21332>;
```

```
    };

    sound-rpmsg-mic {
        compatible = "nxp,snd-af-ivshmem-pcm";
        nxp,name = "AF-mic";
        nxp,card-id = <4>;
        nxp,pcm-in;
        nxp,nb_chans = <8>;
        nxp,ep_id = <0x303>;
    };
};
```

7.1.2. RPC interface

Immersiv3D provides a Hardware Abstraction Layer (HAL) to isolate the audio pipeline from the different types of input and output sources. However, input and output source drivers still need to communicate and configure the audio pipeline. For this, the RPC interface provides a set of callbacks that allows to align the configuration of those modules and the configuration of the audio pipeline.

Immersiv3D exposes an RPC API to allow the communication between Linux drivers and the HAL on LK side. This API is shown in Table 7.

Table 7. RPC Linux API

RPC API	
rpmsg_rpc_register_client (unsigned id, struct rpc_client_callback **cb, void *cookie)	This function registers the RPC client on Linux side. Please note that the ID must match the one defined in Linux and LK device tree (0x100 for HDMI, 0x101 for DAC, and 0x102 for ADC). Additionally, the cookie argument should be “linux-hdmi” for HDMI, “linux-dac” for DAC, and “linux-adc” for ADC.
rpmsg_rpc_unregister_client (struct rpmsg_rpc_dev *rpcdev)	This function unregisters the RPC client on Linux side.
rpmsg_rpc_get_cookie (struct rpmsg_rpc_dev *rpcdev)	This function retrieves the cookie associated to an RPC client.
is_rpmsg_rpc_ready (unsigned id)	This functions signals if the RPC interface is ready to send and receive data.
rpmsg_rpc_call (struct rpmsg_rpc_dev *rpcdev, unsigned rpc_id, void *in, size_t len, void *out, size_t *out_len)	This function allows to initiate an RPC transfer by sending a pointer with information or to be filled by the receiver.
rpmsg_rpc_reply (struct rpmsg_rpc_dev *rpcdev, struct rpc_client_callback *cb, void *d, size_t len)	This function allows to reply to an RPC call.
RPMSG_RPC_CALLBACK (_id, _fn)	This macro allows to register the callback functions of the driver in the RPC interface.

To use the RPC interface for the HDMI/DAC/ADC drivers, the correct Little Kernel device tree must be used. More information on this device tree are in Section 7.2.

7.1.2.1. HDMI switch driver

The HDMI switch driver must register some callbacks that allow the HAL to correctly configure the audio pipeline. These callbacks are registered with the “RPMSG_RPC_CALLBACK” macro and the correct callback ID. In addition to the callbacks, a set of events must be sent from Linux to LK to notify changes on the stream.

An example code is available in `jailhouse_all/linux-kernel/src/jailhouse-services/rpmsg-rpc-linux.c`.

7.1.2.1.1. HDMI RPC callbacks

- RPC_HDMI_INIT_ID

This callback is called when LK is initializing the platform’s hardware resources. The main objective of this function is to initialize the HDMI switch and all related modules. In the scenario where the HDMI switch driver is a module different from the RPC, this callback can be used to initialize the communication between the two modules.

Please note that if there is no initialization to be done, this callback should only return the RPC reply message.

- RPC_HDMI_OPEN_ID

This callback is called when the HDMI device is opened by LK. This function can be used to set a default configuration or to get the current configuration of the HDMI switch, to unmask HDMI related interruptions, or provide handlers for the different events that must be sent to Immersiv3D during streaming. Be aware that the HDMI device is opened at initialization and when changing the source device to `hdmi-input`.

Please note that if there is no configuration to be done here, this callback should only return the RPC reply message.

- RPC_HDMI_CLOSE_ID

This callback is called when the HDMI device is closed by LK. This function can be used to properly handle all configurations done when opening the device.

Please note that if there is no configuration to be done here, this callback should only return the RPC reply message.

- RPC_HDMI_G_CAP_ID

This callback is called after the HDMI switch initialization. The objective of this function is to provide the capabilities of the HDMI switch to Immersiv3D.

The Audio Framework must know how the HDMI switch is sending the audio data through the I2S lines. Immersiv3D currently supports 3 protocols:

- IEC 60958 (HDMI_CAP_AUDIO_FMT_60958): standard for linear PCM digital audio interfaces.
- IEC 61937 (HDMI_CAP_AUDIO_FMT_61937): Based on IEC 60958 for non-linear PCM encoded audio bitstreams.
- Custom (HDMI_CAP_AUDIO_FMT_CUSTOM): This specifies that the HDMI driver supports a custom protocol based on IEC 60958. The provided structure iec60958_custom_fmt_layout_t in the RPC_HDMI_G_CUSTOM_FMT_ID callback specifies the bit position of each field in the sub-frame. Please note that if the custom protocol doesn't have a particular field, its bit position should be set to -1.

In addition to the format of the audio bitstreams, the HDMI switch driver can inform Immersiv3D of its capabilities to detect changes or specific information on the stream:

- Sampling rate (HDMI_CAP_AUDIO_SAMPLE_RATE_CHANGE): The HDMI driver can detect a change on the sampling rate and inform the system.
- Stream type (HDMI_CAP_AUDIO_STREAM_TYPE_CHANGE): The HDMI driver can detect a change on the stream type:
    - HDMI_AUDIO_PKT_STD for Standard audio packets LPCM or 60958/61937
    - HDMI_AUDIO_PKT_HBR for HBR packets LCPM or 60958/61937
    - HDMI_AUDIO_PKT_DSD for Direct Stream Digital packets. Note that this is not currently supported.
    - HDMI_AUDIO_PKT_DST for Direct Stream Transfer packets. Note that this is not currently supported.
- Channel status (HDMI_CAP_AUDIO_CHANNEL_STATUS): The HDMI driver can extract the channel status and provide it to the system.
- Link (HDMI_CAP_AUDIO_LINK_CHANGE): The HDMI driver can detect a link change and inform the system.
- InfoFrame (HDMI_CAP_AUDIO_INFOFRAME): The HDMI driver can extract the InfoFrame and provide it to the system.
- Layout (HDMI_CAP_AUDIO_LAYOUT_CHANGE): The HDMI driver can detect a change on the layout:
    - HDMI_AUDIO_PKT_LAYOUT_0_2CH for up to 2 channels
    - HDMI_AUDIO_PKT_LAYOUT_1_8CH for up to 8 channels
- RPC_HDMI_G_CUSTOM_FMT_ID

This callback is called when the HDMI driver only supports a custom sub-frame format based on IEC60958 (HDMI_CAP_AUDIO_FMT_CUSTOM). This function will allow Immersiv3D to know the details of the frame structure to correctly extract the needed elements. Therefore, the RPC HDMI driver must provide a iec60958_custom_fmt_layout_t structure indicating the bit position of each field. If the custom protocol doesn't have a particular field, its bit position should be set to -1.

Please note that if the HDMI driver doesn't support any custom format, this callback should only return the RPC reply message.

- RPC_HDMI_G_PKT_LAYOUT_ID

This callback is mainly called just after the HDMI driver is opened. The objective of this function is to provide Immersiv3D the current layout configuration: `HDMI_AUDIO_PKT_LAYOUT_0_2CH` for up to 2 channels and `HDMI_AUDIO_PKT_LAYOUT_1_8CH` for up to 8 channels. Some streams (like PCM and Dolby Digital) use the `HDMI_AUDIO_PKT_LAYOUT_0_2CH` layout and others (like Dolby TrueHD) use the `HDMI_AUDIO_PKT_LAYOUT_1_8CH` layout. The HDMI driver must send this information to Immersiv3D to listen to the correct I2S RX lines.

- RPC_HDMI_G_PKT_TYPE_ID

This callback is mainly called just after the HDMI driver is opened. The objective of this function is to provide Immersiv3D with the current packet type: `HDMI_AUDIO_PKT_STD` for Standard audio packets LPCM or 60958/61937, `HDMI_AUDIO_PKT_HBR` for HBR packets LCPM or 60958/61937, `HDMI_AUDIO_PKT_DSD` for Direct Stream Digital packets (this is not currently supported), and `HDMI_AUDIO_PKT_DST` (this is not currently supported). The HDMI switch can send audio data as standard LPCM (for PCM) or HBR (for encoded streams).

- RPC_HDMI_G_INFOFRAME_ID

This callback is mainly called just after the HDMI driver is opened. The objective of this function is to provide Immersiv3D with the current InfoFrame. Some information needed by Immersiv3D (like the number of channels) is in the InfoFrame. Note that even if the HDMI driver can send an event to provide the InfoFrame, this callback must be correctly implemented, because it is used to configure the pipeline at initialization and when switching the source device.

- RPC_HDMI_G_CS_ID

This callback is mainly called just after the HDMI driver is opened. The objective of this function is to provide Immersiv3D with the current channel status. Some information needed by Immersiv3D (like the sample rate) is contained in the channel status. Note that even if the HDMI driver can send an event to provide the channel status (or the information contained in the channel status), this callback must be correctly implemented, because it is used to configure the pipeline at initialization and when switching the source device.

- RPC_HDMI_S_FORMAT_ID

This callback is called just after the `RPC_HDMI_G_CAP_ID` callback to select a format to be used by the HDMI switch. Once Immersiv3D has collected the supported formats from the HDMI driver, it will notify the HDMI driver to configure the HDMI switch to use a particular format. Note that the order of preference for the supported protocols is as follows:

1. IEC 60958
2. IEC 61937
3. Custom format
    - RPC_HDMI_S_IF_ID

This callback allows Immersiv3D to specify the audio interface to be used by the HDMI driver.

- RPC_HDMI_S_PKT_ID

This callback allows Immersiv3D to specify the packet type to be used by the HDMI driver.

7.1.2.1.2. HDMI RPC Events

Besides the callbacks, the HDMI driver must provide a set of events to notify Immersiv3D of changes during playback. This events must also use the RPC API.

- HDMI_EVENT_AUDIO_SAMPLE_RATE

This event must be sent to Immersiv3D to notify a change on the current sampling rate.

- HDMI_EVENT_AUDIO_STREAM_TYPE

This event must be sent to Immersiv3D when the stream type has changed.

- HDMI_EVENT_AUDIO_LINK

This event must be sent to Immersiv3D to notify a change on the current physical connection.

- HDMI_EVENT_AUDIO_MCLK

This event must be sent to Immersiv3D to notify a change on the status of the master clock.

- HDMI_EVENT_AUDIO_INFOFRAME

This event must be sent to Immersiv3D when a new InfoFrame is available.

- HDMI_EVENT_AUDIO_CHANNEL_STATUS

This event must be sent to Immersiv3D when a new channel status is available.

- HDMI_EVENT_AUDIO_LAYOUT_CHANGE

This event must be sent to Immersiv3D to notify a change on the layout used by the HDMI switch.

- HDMI_EVENT_ERROR

This event must be sent to Immersiv3D when the HDMI switch encounters an error.

7.1.2.1.3. HDMI RPC sequence

There are two main ways to implement the HDMI driver with the RPC interface. The first one is integrating the RPC APIs into the HDMI driver. The second one is to have the HDMI driver communicate with the HDMI Linux Control driver (or HDMI RPC driver) and have only this last driver implementing the RPC APIs. Please note that for the second type of implementation, the HDMI driver can be on Linux or it can even be on an external microcontroller. Please be aware that having the HDMI driver in an external microcontroller will increase the latency of the control interface between the driver and Little Kernel. The requirement for Immersive3D is that Little Kernel must be notified of any change in the physical interface 10 ms before the change is effective.

- HDMI and RPC in a single driver

Figure 12 shows an example of the sequence diagram at initialization.

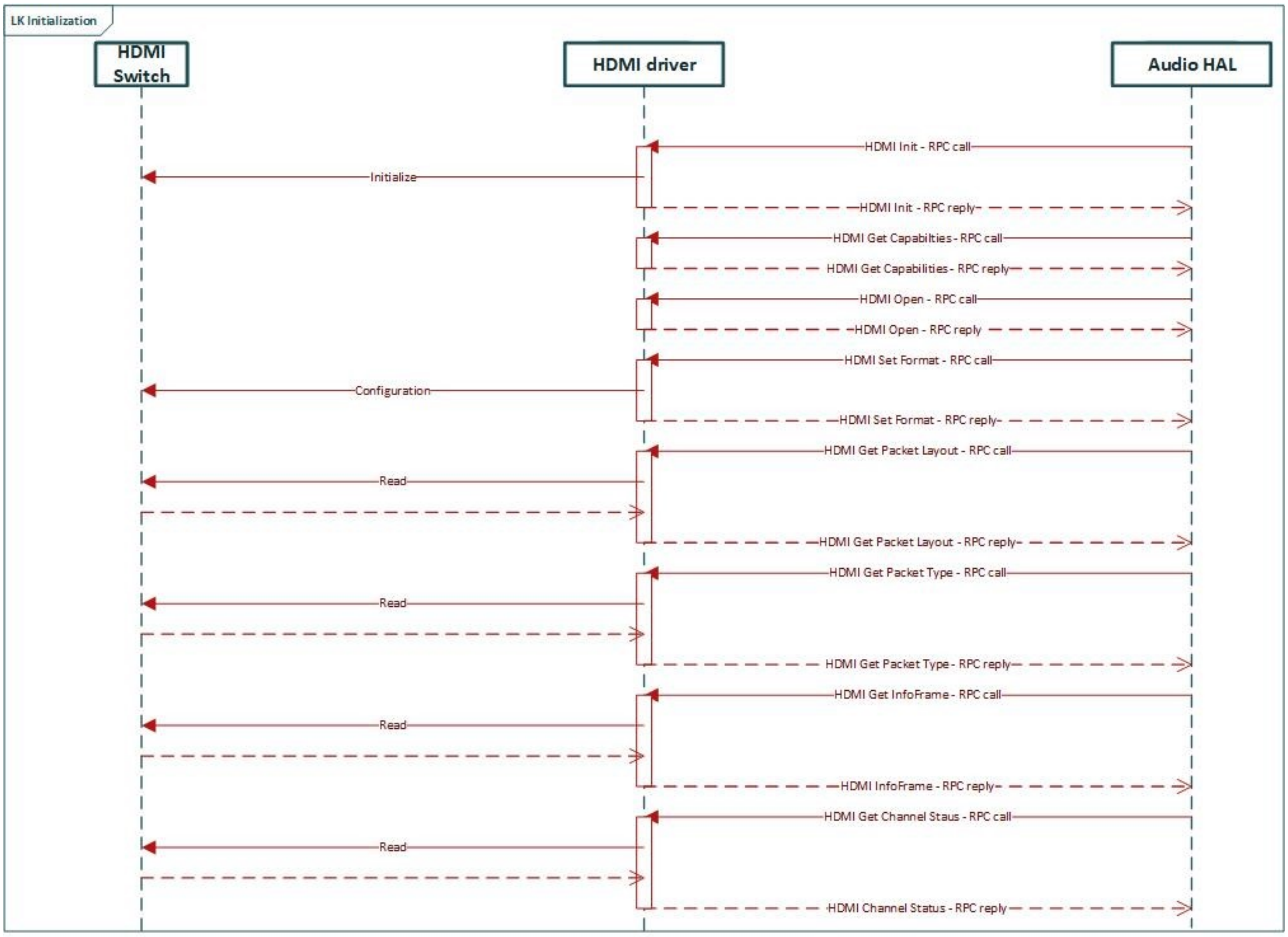

Figure 12. HDMI driver – initialization

Figure 13 shows an example of the sequence diagram at playback. Note that the IRQs sent by the HDMI switch notify changes on the stream configuration.

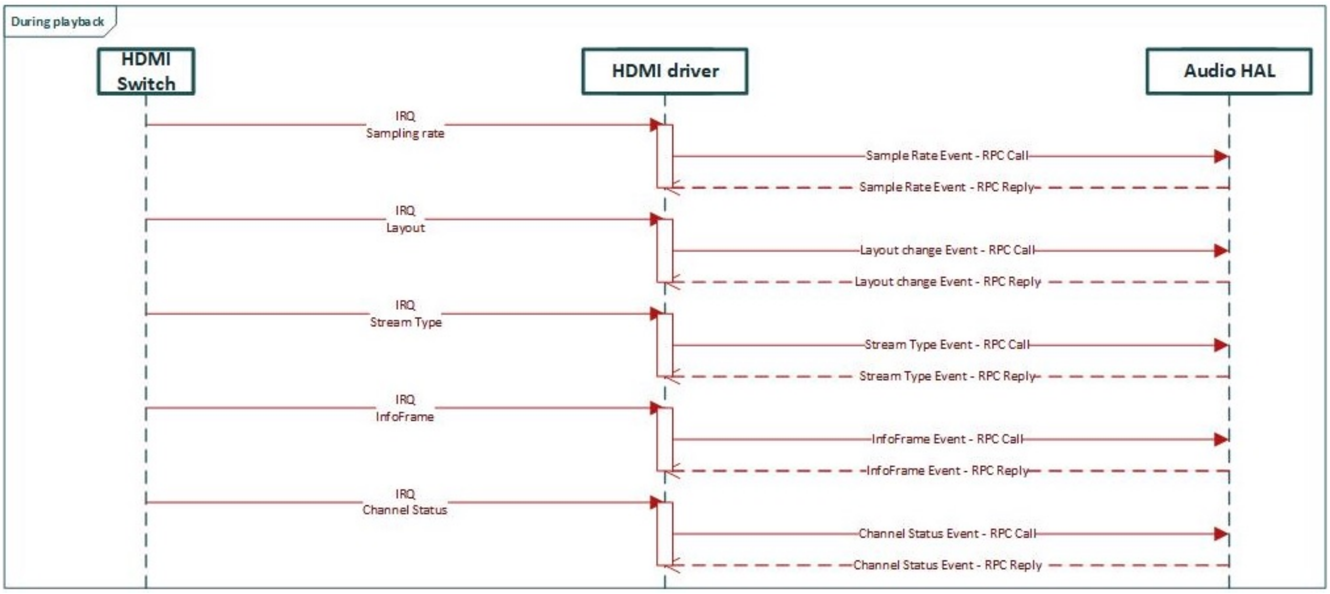

Figure 13. HDMI driver – playback

Finally, Figure 14 shows an example of the sequence diagram when changing the source device to HDMI. Note that the GET callbacks are needed to correctly configure the pipeline until a new stream configuration is detected.

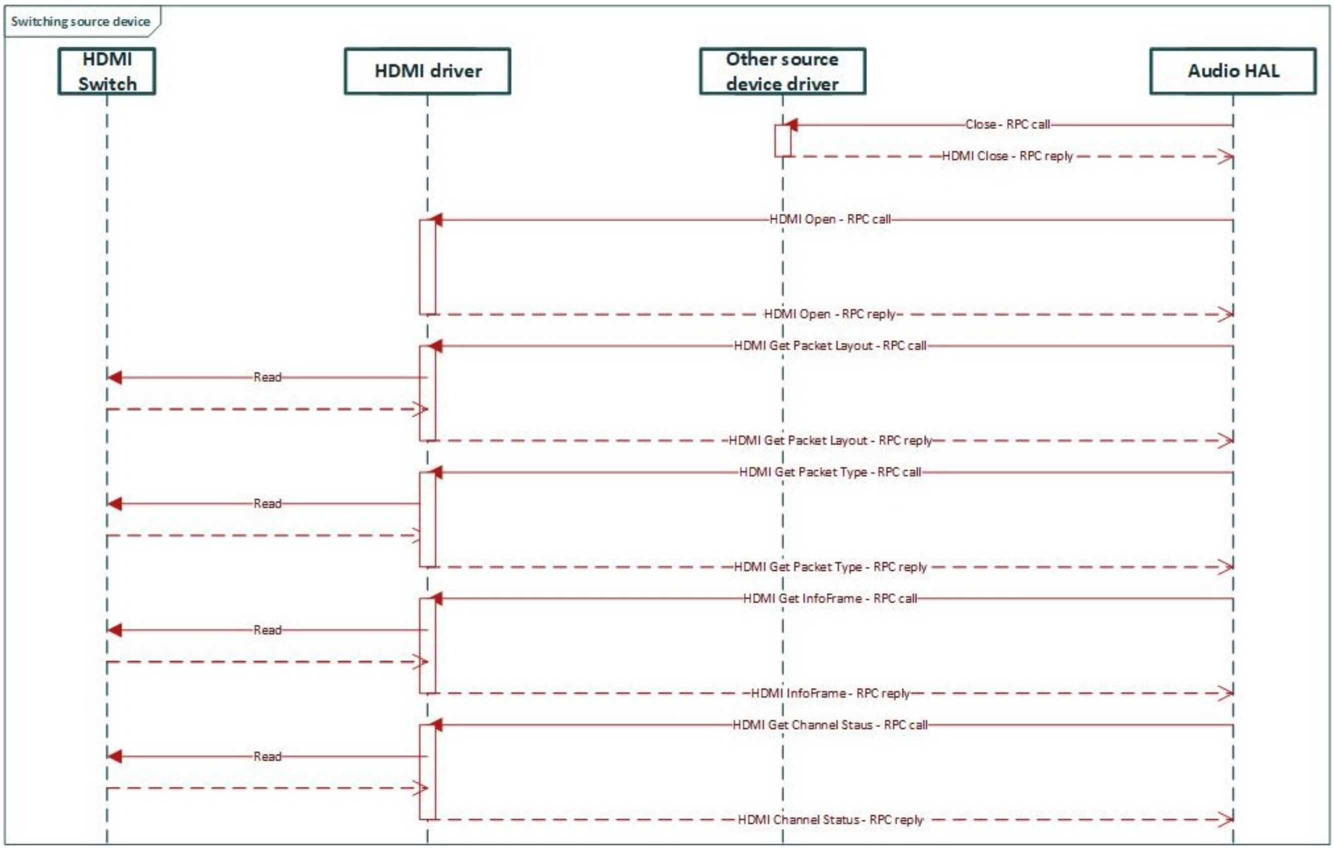

Figure 14. HDMI driver – switching to HDMI source device

7.1.2.1.4. External HDMI driver and HDMI Linux Control driver

Figure 15 shows an example of the sequence diagram at initialization.

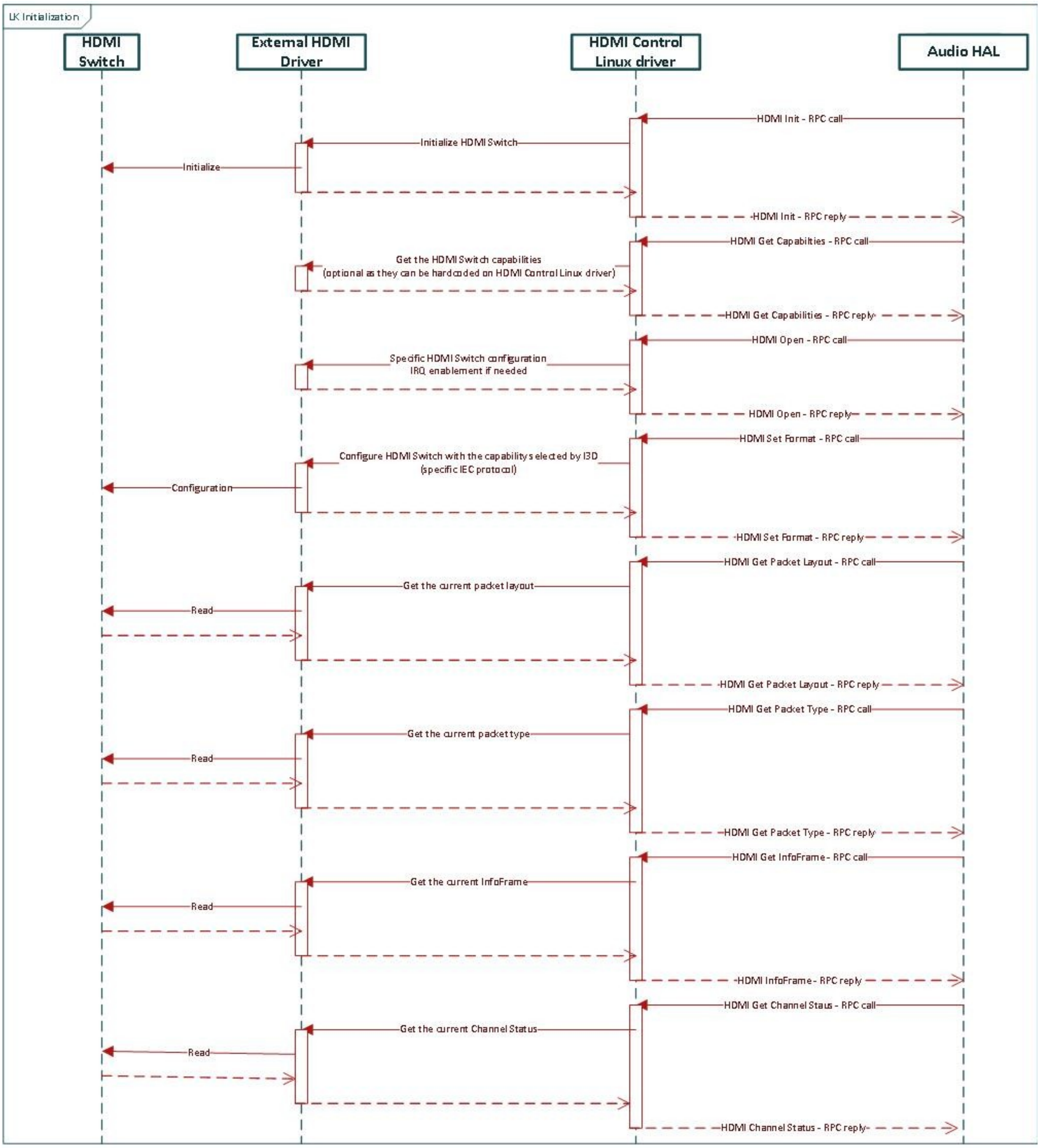

Figure 15. External HDMI driver - initialization

Figure 16 shows an example of the sequence diagram at playback. Note that the IRQs sent by the HDMI switch notify changes on the stream configuration.

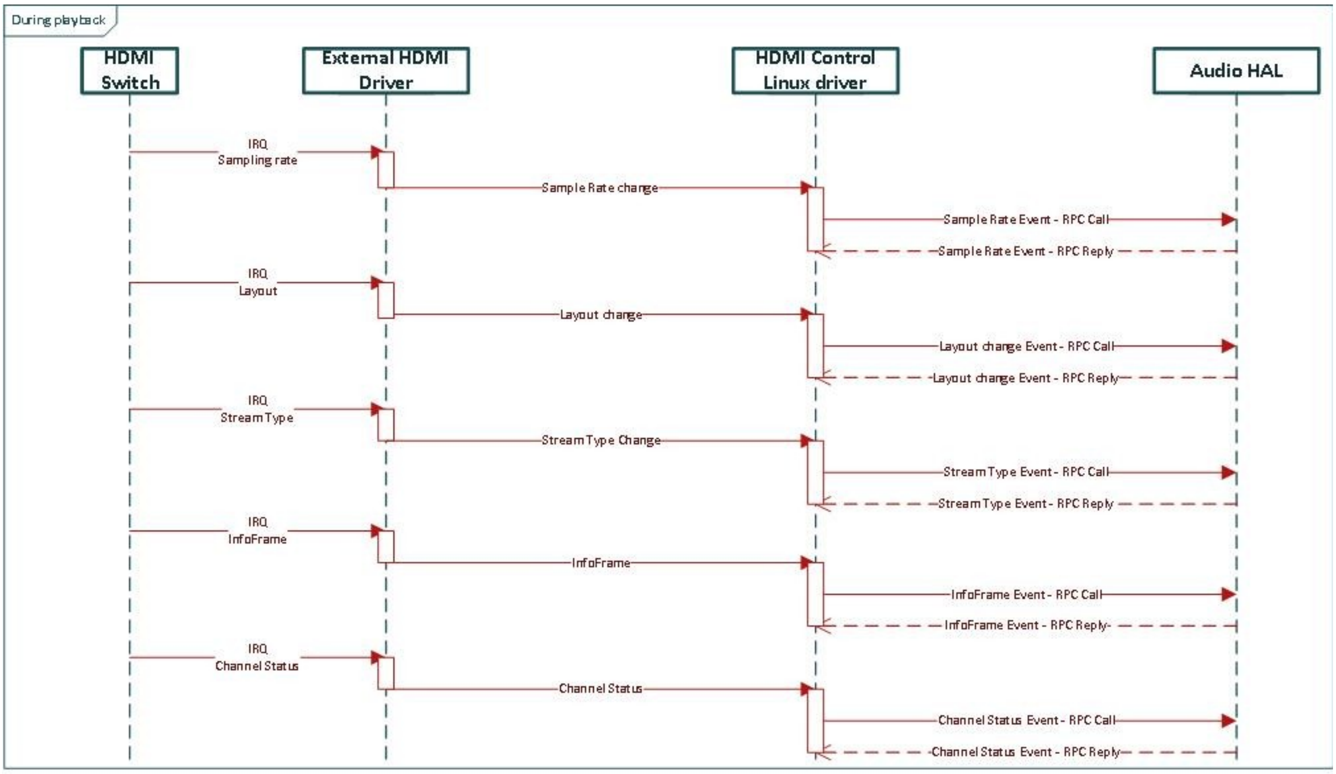

Figure 16. External HDMI driver - playback

Figure 17 shows an example of the sequence diagram when changing the source device to HDMI. Note that the GET callbacks are needed to correctly configure the pipeline until a new stream configuration is detected.

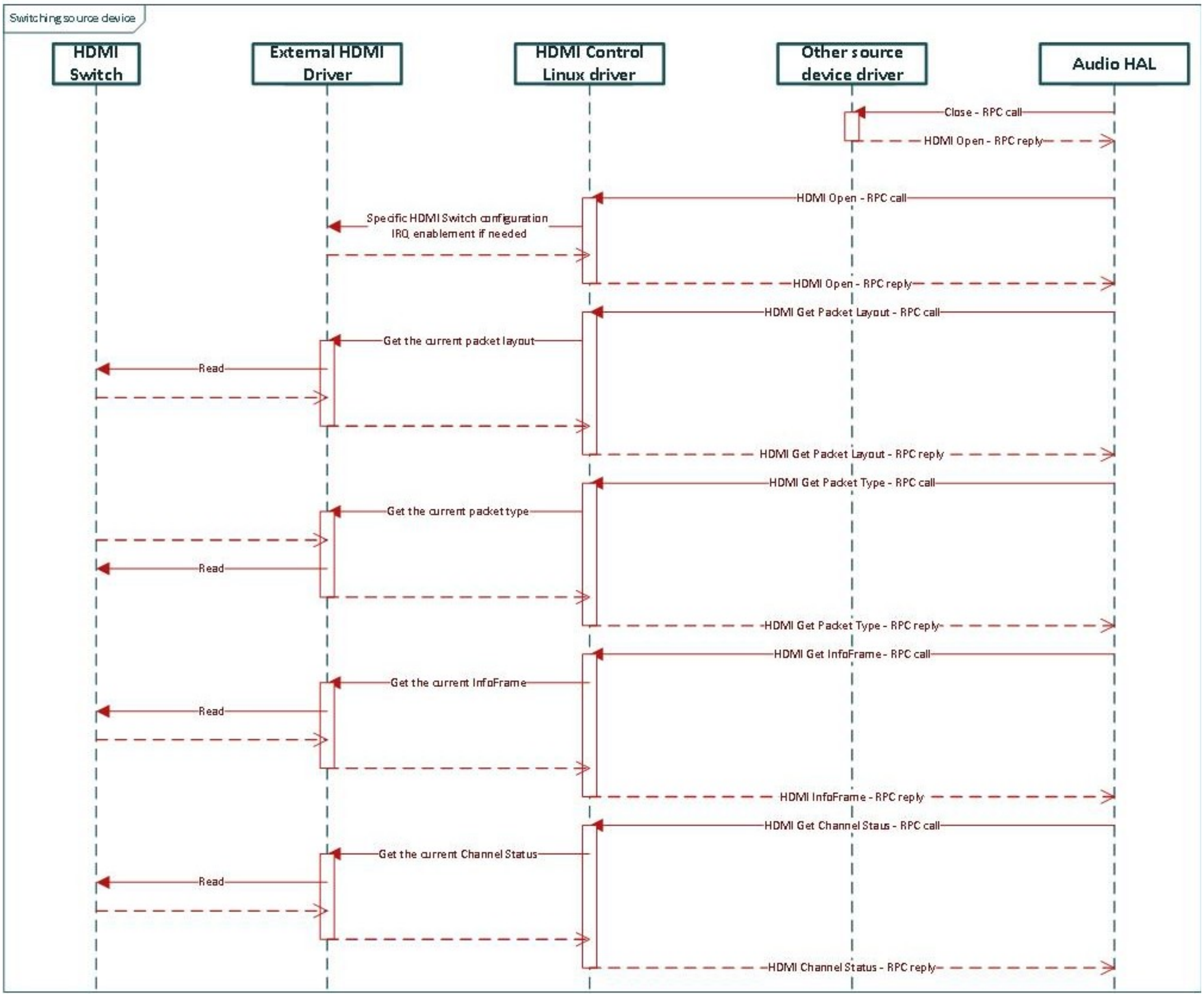

Figure 17. External HDMI driver – switching to HDMI source device

7.1.2.2. DAC driver

As for the HDMI switch driver, the DAC driver must register some callbacks that allow the HAL to correctly configure the audio pipeline. These callbacks are registered with the `RPMSG_RPC_CALLBACK` macro and the correct callback ID.

7.1.2.2.1. RPC_DAC_INIT_ID

This callback is called when LK is initializing the platform's hardware resources. The main objective of this function is to initialize the DAC and all related modules. In the scenario where the DAC driver is a module different from the RPC, this callback can be used to initialize the communication between the two modules.

Please note that if there is no initialization to be done, this callback should only return the RPC reply message.

7.1.2.2.2. RPC_DAC_OPEN_ID

This callback is called when the DAC device is opened by LK. This function can be used to set a default or get the current configuration of the DAC or to unmask DAC-related IRQs. Be aware that DAC device is opened at initialization and when changing the sink device to `dac-output`.

Please note that if there is no configuration to be done, this callback should only return the RPC reply message.

7.1.2.2.3. RPC_DAC_CLOSE_ID

This callback is called when the DAC device is closed by LK. This function can be used to properly handle all configurations done when opening the device.

Please note that if there is no configuration to be done, this callback should only return the RPC reply message.

7.1.2.2.4. RPC_DAC_G_CAP_ID

This callback is called after the DAC initialization. The objective of this function is to provide the capabilities of the DAC to Immersiv3D.

Audio Framework must know how to send audio data to the DAC through the I2S lines. Please note that the only supported capability for outputting PCM data is `DAC_CAP_PKT_PCM`.

7.1.2.2.5. RPC_DAC_S_FORMAT_ID

This callback allows Immersiv3D to notify the DAC driver to configure the DAC to use a particular format. Note that the `rpc_dac_s_format_s` structure is composed of:

- PCM format (`dac_audio_pcm_format_t`) specifying if the outputted data is in 16, 24, or 32 bits
- Audio format (`dac_audio_fmt_t`) specifying if the audio format is I2S, Left J, Right J, DSP A, DSP B, AC97, or PDM
- Audio packet (`dac_audio_pkt_t`) specifying if the packet type is standard PCM or DSD

7.1.2.2.6. DAC RPC sequence

Figure 18 shows an example of the sequence diagram at initialization.

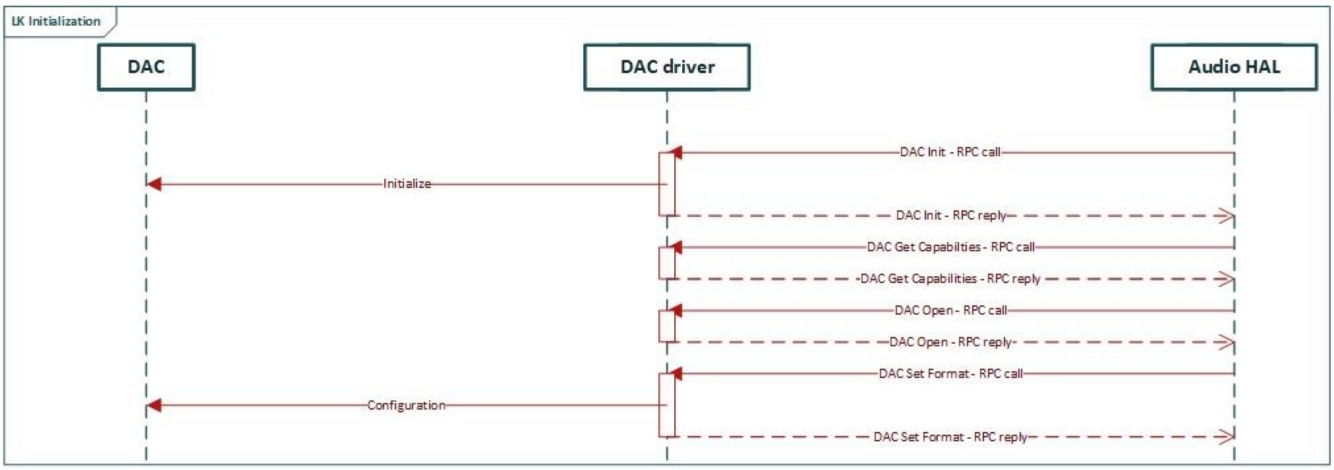

Figure 18. DAC driver – initialization

Figure 19 shows an example of the sequence diagram when changing the source device to DAC.

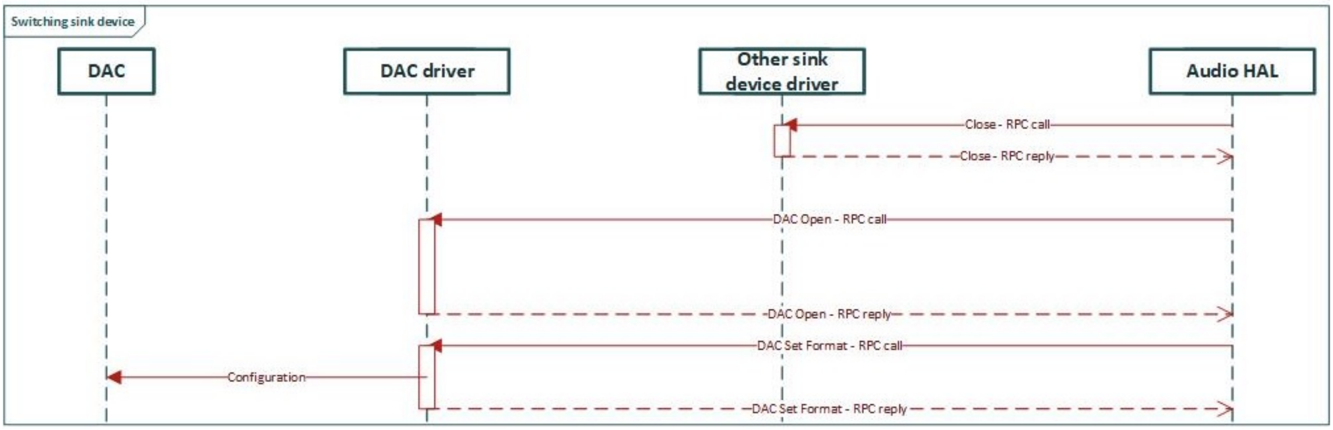

Figure 19. DAC driver – switching sink device to DAC

7.1.2.3. ADC driver

Similar to HDMI and DAC, the ADC driver must register some callbacks that allow the HAL to correctly configure the audio pipeline. These callbacks are registered with the `RPMSG_RPC_CALLBACK` macro and the correct callback ID.

7.1.2.3.1. RPC_ADC_INIT_ID

This callback is called when LK is initializing the platform's hardware resources. The main objective of this function is to initialize the ADC and all related modules. In the scenario where the ADC driver is a module different from the RPC, this callback can be used to initialize the communication between the two modules.

Please note that if there is no initialization to be done, this callback should only return the RPC reply message.

7.1.2.3.2. RPC_ADC_OPEN_ID

This callback is called when the ADC device is opened by LK. This function can be used to set a default or get the current configuration of the ADC or to unmask ADC related IRQs. Be aware that the ADC device is opened at initialization and when changing the source device to `adc-input`.

Please note that if there is no configuration to be done, this callback should only return the RPC reply message.

7.1.2.3.3. RPC_ADC_CLOSE_ID

This callback is called when the ADC device is closed by LK. This function can be used to properly handle all configurations done when opening the device.

Please note that if there is no configuration to be done, this callback should only return the RPC reply message.

7.1.2.3.4. RPC_ADC_G_CAP_ID

This callback is called after the ADC initialization. The objective of this function is to provide the capabilities of the ADC to Immersiv3D.

Audio Framework must know how to receive audio data from the ADC through the I2S lines. Please note that the only supported capability is `ADC_CAP_PKT_PCM` for receiving PCM data.

7.1.2.3.5. RPC_ADC_S_FORMAT_ID

This callback allows Immersiv3D to notify the ADC driver to configure the ADC to use a particular format. Note that the `rpc_adc_s_format_s` structure is composed of:

- PCM format (`adc_audio_pcm_format_t`) specifying if the outputted data is in 16, 24, or 32 bits
- Audio format (`adc_audio_fmt_t`) specifying if the audio format is I2S, Left J, Right J, DSP A, DSP B, AC97, or PDM
- Audio packet (`adc_audio_pkt_t`) specifying if the packet type is standard PCM or DSD

7.1.2.3.6. ADC RPC sequence

Figure 20 shows an example of the sequence diagram at initialization.

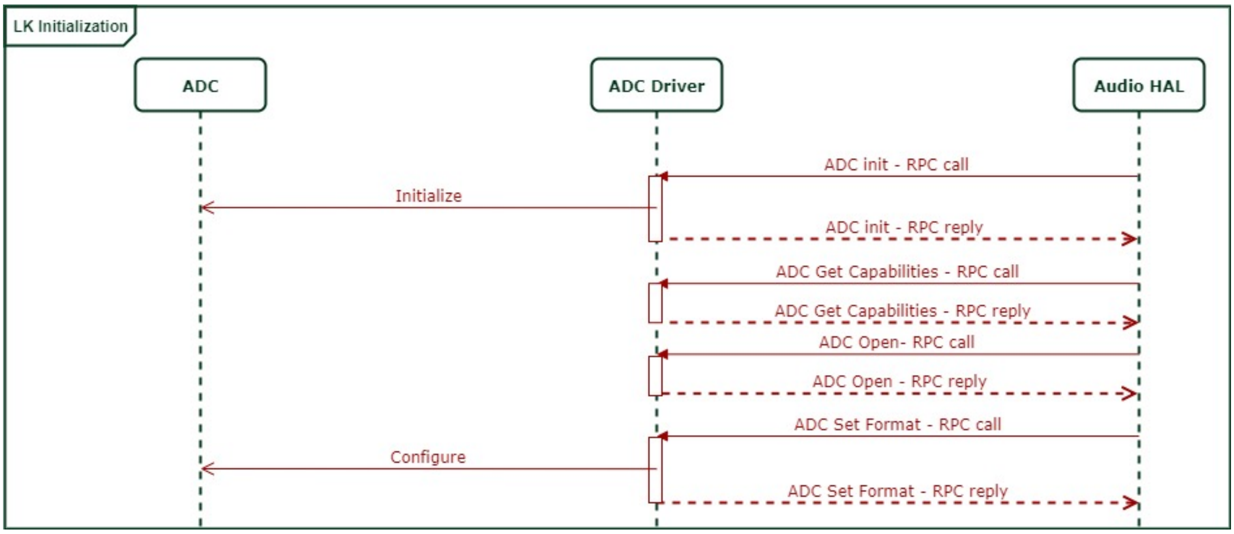

Figure 20. ADC driver – initialization

Figure 21 shows an example of the sequence diagram when changing the source device to ADC.

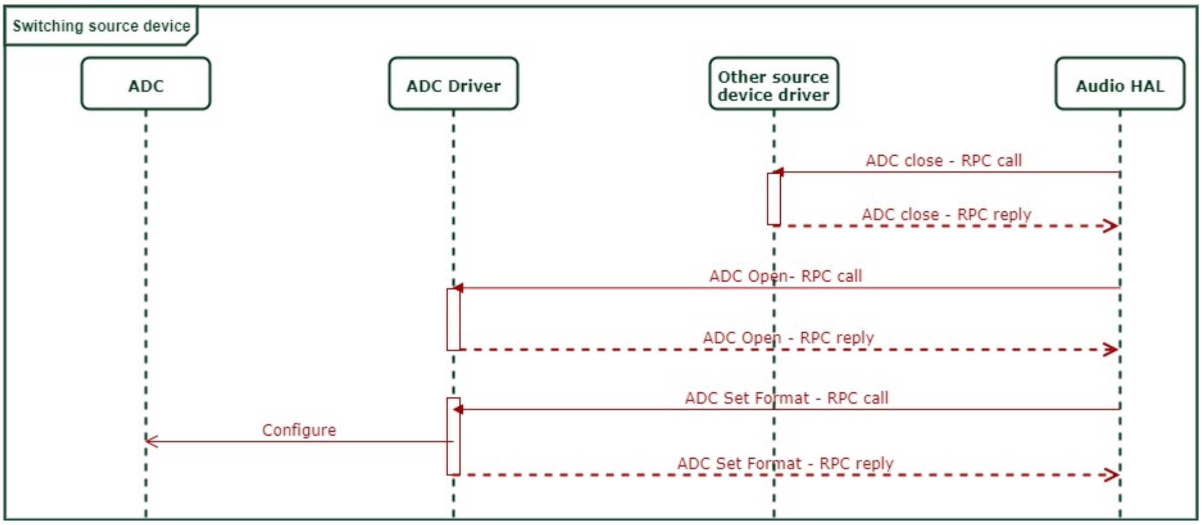

Figure 21. ADC driver – switching source device to ADC

7.2. Little Kernel configuration

On the LK side, adapting Immersiv3D to a custom board implies both modifying or creating a device tree and implementing a callback to initialize the custom board.

7.2.1. Little Kernel device tree

As for Linux, LK provides a device tree that specifies the entire description of the hardware that will be configured and used by LK. Each hardware module is represented by a node containing different properties. A list of currently available properties is in Table 8.

Immersiv3D provides a device tree for i.MX 8M SOCs (`imx8mm.dtsi` or `imx8mn.dtsi`), another for the i.MX 8M platforms (`imx8mm-cm.dts` or `imx8mn-cm.dts`), another specific to the i.MX Audio Board (`imx8mm-ab2.dts`, `imx8mn-ab2.dts`, or `imx8mnul-ab2.dts`), and another using the RPC interface (`imx8mm-ab2-rpc.dts`, `imx8mn-ab2-rpc.dts`, or `imx8mnul-ab2-rpc.dts`).

Users willing to adapt Immersiv3D to their boards should use the `imx8mm.dtsi` device tree and create their own `imx8mm-<user>.dts` device tree that will enable/disable and configure the hardware resources of the board used by LK. Finally, an audio framework device tree `imx8mm-<user>-rpc.dts`, including `af.dtsi`, can be used to add Immersiv3D specific nodes and the RPC interface. Please note that this implies that `imx8mm-<user>-rpc.dts` includes `imx8mm-<user>.dts`, which shall include `imx8mm.dtsi` (the same logic can be applied for imx8mn or imx8mnul).

Table 8. LK device tree node properties

LK device tree properties	
address-cells	Determines the number of cells for addresses used for addressable devices.
size-cells	Determines the number of cells for the length of the addressable device.
reg	Lists the address ranges used by the device through one or more cells.
reg-names	Provides a name to each reg cell. Please note that the order of each element indicates the association between them (reg 1 will get reg-names 1).
compatible	The compatible property of a device node describes the specific binding or bindings to which the node complies.
status	Determines if the device is enabled (status = “okay”) or disabled (status = “disabled”).
id	Determines the ID of the device. The interpretation of this ID may differ, depending on the compatible driver. For CIPC and RPMSG-BIN nodes, the ID corresponds to the ID of the interface. Please note that for these two nodes, the ID should be the same in Linux and LK device tree.
settings	Determines the PLL clock configuration.
interrupt-controller	Defines the device as interrupt controller (a device that receives interrupt signals).
interrupt-cells	This is a property of the interrupt controller node. It is used to define how many cells are in an interrupt specifier for the interrupt controller.

LK device tree properties	
interrupt-parent	This is a property of a device node containing a handle to the interrupt controller to which it is attached. Nodes without an interrupt-parent property can inherit the property from their parent node.
interrupts	This is a property of a device node containing a list of interrupt specifiers; one for each interrupt output signal.
interrupt-names	Provides a name to each interrupt listed on the interrupts property.
bus-id	Determines the bus ID to be used by the device.
bus-id-spdif	Determines the bus ID to be used by the SPDIF device.
bus-id-sai	Determines the bus ID to be used by the SAI device.
bus-id-i2c	Determines the bus ID to be used by the I2C device.
clock-cfg	Determines the configuration for each clock of the device. Please note that the configuration is as follows: <[clock source] [PLL divider] [pre-divider] [post-divider] [clock gating] [rate in Hz]>.
clock -names	Provides a name to each clock being configured by the clock-cfg property.
dma-cells	Determines the number of cells for the DMA device.
dmas	Determines DMA value for each associated name.
dma-names	Provides a name to each DMA device.
dma-period-length	Determines DMA period length for the SPDIF device.
dma-nr-period	Determines the number of DMA periods for the SPDIF device.
disable-dma	Disables the DMA device.
gpio-controller	Defines the device as the GPIO controller.
gpio-cells	This is a property of the GPIO controller node. It is used to define how many cells are in an GPIO specifier for the GPIO controller.
ngpios	Determines the number of GPIO instances.
enable-gpio	Enables the GPIO. The syntax is as follows: <[gpio controller] [gpio number] [initial configuration].
pinctrl-<id>	Determines the IO muxing configuration for a specific pin. The syntax is as follows: <[pin mux register] [mux mode] [input register] [input daisy] [configuration register] [Input on field] [configuration value]>.
pinctrl-names	Provides a name to each pin ID.
init	Determines the GPR initial configuration.
event-ids	Determines the IDs of latency events.
event-names	Provides a name to each event ID.
push-gpio	Determines the GPIO used for the “push” event for latency measurements (Obsolete).

LK device tree properties	
autodetect-gpio	Determines the GPIO used for the “autodetect” event for latency measurements (Obsolete).
hdmi	Determines the HDMI device to be used as the HDMI input.
adc	Pointer to ADC IP node (ADC stream device).
alsa	Determines the ALSA device to be used by the HAL node.
alsa-cpp	Determines the ALSA device to be used for sending audio data from CPP to Linux.
hdmi-sai	Determines the SAI lines to be used for the HDMI device.
hdmi-i2s-fmt	Determines the I2S format for each supported protocol. Please note that the configuration is as follows: <[I2S format for IEC60958] [I2S format for custom format] [I2S format for IEC61937]>. Please note that a value of 0xFF means that the protocol is not supported. The available I2S formats are: - 0: Left justified - 1: Right justified - 2: I2S - 3: PCM A - 4: PCM B - 5: AES3
hdmi-polarity	Determines the polarity of the HDMI device.
hdmi-latency	Determines the hardware latency of the HDMI device.
hdmi-settling-time	Determines the settling time of the HDMI device.
hdmi-bitrate-period	Determines the bitrate period of the HDMI device.
hdmi-bitrate-mode	Determines the bitrate mode of the HDMI device. The supported values are: - 1: Async mode - 2: Sync mode - 4: Cumulative mode
spdif	Determines the SPDIF device to be used as the SPDIF input.
spdif-latency	Determines the hardware latency of the SPDIF device. The default value is 5000 us.
spdif-sai	Determines the SAI lines to be used for the SPDIF device.

LK device tree properties	
spdif-bitrate-period	Determines the bitrate period of the SPDIF device.
spdif-bitrate-mode	Determines the bitrate mode of the SPDIF device.
dac	Determines the DAC device to be used as the DAC output. This property is optional.
dac-sai	Determines the SAI lines to be used for the DAC device.
dac-i2s-fmt	Determines the I2S format for each supported protocol. Please note that the configuration is as follows: <[I2S format for IEC60958] [I2S format for custom format] [I2S format for IEC61937]>. Please note that a value of 0xFF means that the protocol is not supported. The available I2S formats are: - 0: Left justified - 1: Right justified - 2: I2S - 3: PCM A - 4: PCM B - 5: AES3
dac-polarity	Determines the polarity of the DAC device.
dac-nch	Determines the maximum number of channels supported for the DAC device.
dac-latency	Determines the hardware latency of the DAC device.
dac-bitrate-period	Determines the bitrate period of the DAC device.
dac-bitrate-mode	Determines the bitrate mode of the DAC device.
dac-bitrate-sai-dev	Determines the DAC to use a different SAI for the DAC bitrate computation.
dac-disable-sai-counters	Disables the SAI counters for DAC device.
dac-slave	Configures the SAI port connected to the DAC as a slave. The default SAI mode is master.
dac2	Determines the secondary DAC device to be used as the DAC2 output. All DAC properties must be assigned in a similar way as for the main DAC.
sai	Determines the SAI to be used by the input and output Hardware Abstraction Layer.
rx,bcp	Determines the Rx Bit Clock polarity of the SAI node. 0 is active high (sampled on falling edge) and 1 is active low (sampled on rising edge).
tx,bcp	Determines the Tx Bit Clock polarity of the SAI node. 0 is active high (sampled on falling edge) and 1 is active low (sampled on rising edge).

LK device tree properties	
size	Determines the buffer of IPC or RPMSG endpoints in bytes.
buffer, buffer_bytes	Determines the buffer size of the binary interface. The buffer parameter has granularity of megabytes and buffer_bytes has granularity of bytes.
cs-gpio	Determines the specific GPIO for CS pin of ADV7627 HDMI switch.
reset-gpio	Determines the specific GPIO for Reset pin of ADV7627 HDMI switch.
int1-gpio	Determines the specific GPIO for Interrupt 1 pin of ADV7627 HDMI switch.
int2-gpio	Determines the specific GPIO for Interrupt 2 pin of ADV7627 HDMI switch.
pdn-gpio	Determines the specific GPIO for PDN pin of AK4458 DAC.
rpc,service-id	Determines the ID of the RPC interface used by the device.
i2s-fmt	Determines the I2S format for each supported protocol. Please note that the configuration is as follows: <[I2S format for IEC60958] [I2S format for custom format] [I2S format for IEC61937]>. Please note that a value of 0xFF means that the protocol is not supported. The available I2S formats are as follows: - 0: Left justified - 1: Right justified - 2: I2S - 3: PCM A - 4: PCM B - 5: AES3
ch_max	Determines the maximum channel number for the node.
sai-hdmi	Determines the clock mode. Available options: 0 (Default) and 1 (external clock). Please note that mode 1 cannot be configured by default on the EVK < - > Audio Board hardware. It requires the CPLDv2.4 update. For more information, please contact the I3D support team.
gpr-<component>	Determines the gpr register setting for each component (hdmi, spdif, alsa, adc) in this format: offset, mask, value.
pci_cfg	Determines the PCI start address and size in the following format: < <addr>, <size> >.
om-gpio	Determines the GPIO used for the “om” event for latency measurements.
fade-gpio	Determines the GPIO used for the “fade” event for latency measurements.
voice	Determines the voice device to be used as the voice input (voice path).

LK device tree properties	
pdm	Determines the PDM device to be used as the PDM input.
pdm-latency	Determines the hardware latency of the PDM device.
adc-sai	Determines the SAI lines to be used for the ADC input.
adc-nch	Determines the maximum number of channels supported for ADC.
adc-polarity	Determines the polarity of the ADC device.
adc-i2s-fmt	Determines the I2S format for each supported protocol. Please note that the configuration is as follows: <[I2S format for IEC60958] [I2S format for custom format] [I2S format for IEC61937]>. Please note that a value of 0xFF means that the protocol is not supported. The available I2S formats are: - 0: Left justified - 1: Right justified - 2: I2S - 3: PCM A - 4: PCM B - 5: AES3
adc-latency	Determines the hardware latency of the ADC input device.
adc-sampling-rate	Determines the sampling rate of the ADC input device.
adc-slave	Configures the SAI port connected to the ADC as slave. This is an optional property. The default SAI mode is master.
adc-enable-sai-counters	When the SAI is configured as slave, this option allows monitoring the input frequency and detect any changes to force the pipeline flush.
adc-start-lane	This property allows using a different RXDi start lane, instead of the default RXD0. The example for 4 channels RX using 2 slots is as follows: - sai-start-lane missing or set to 0: will use RXD0 + RXD1 - sai-start-lane = < 1 > : will use RXD1 + RXD2
adc2-*	Same options as above, for second ADC input.
pll-cfg	Determines the configuration for each PLL clock. Please note that the configuration is as follows: <[rate] [main divider] [p-divider] [s-divider] [k-divider] [reference clock] [pll control id]>.
pll-names	Provides a name to each PLL clock being configured by the pll-cfg property.
pll-names-<component>	Determines which PLLs can be assigned for each component (hdmi, spdif, alsa, adc).

LK device tree properties	
pll-mask-<component>	Determines the mask for the pll-names-<component> list (hdmi, spdif, alsa, adc).
polling-rate-ms	Determines the polling rate for the SPDIF channel status.
mclk-tx-config-names	Determines the PLL names used on TX for 44k, 48k, and 32k multipliers (in this specific order) for the SAI node.
mclk-rx-config-names	Determines the PLL names used on RX for 44k, 48k, and 32k multipliers (in this specific order) for the SAI node.
rx,mclk-select	Determines the RX MCLK of the SAI node: 0: bus clock, 1: MCLK1, 2: MCLK2, 3: MCLK3.
tx,mclk-select	Determines the TX MCLK of the SAI node: 0: bus clock, 1: MCLK1, 2: MCLK2, 3: MCLK3.
tx, sync-mode	Enables the sync-with-other-direction mode for SAI TX so that both TX and RX paths of the same SAI use the same RX clocks.
mclk,is-output	Determines if MCLK is set as output for the SAI node.
clock-audio	Determines the audio clock management component to be used.
pll-id	Determines the PLL ID for the PLL loop configuration node.
pll-polling-rate	Determines the PLL polling rate value for the PLL loop configuration node.
status-spdif	Enables the PLL loop configuration for the SPDIF input.
status-hdmi	Enables the PLL loop configuration for the HDMI input.
pll-loop-config	Determines the PLL configuration node to be used in the audio clock manager.
spdif-cs	Determines the SPDIF channel status node to be used for channel status extraction.
output,dly,size	Determines the maximum buffer size allocated used for delay adjustment per output paths.
common, channels	Determines the maximum concurrent channels in a pipeline [2-32].
input,settling-time	Determines the settling time in ms before returning PCM as fallback.
input,zero-detection-time	Determines the zeros duration to be considered as pause (zero is infinite).
input,ade-dtscd-detection-disable	Disables the DTS-CD detection while running PCM streams.
output,voice,hal	When specified, this option allows changing the HAL stream used as a voice source for voice mixing. This is an optional property. The default voice stream is the ALSA voice input. Other stream (such as adc-input or adc2-input) can be used as well.
<ep_name>-stack-kb	This optional device tree property can be used to configure the endpoint thread stack size to a user-defined value. This must be configured under the `ivshmem` node from the device tree. For example, to allocate 48 KB stack size to the `ppp` ivshmem enpoint thread, you can define: `ppp-stack-kb = <48>;`.

LK device tree properties	
uart-disabled	This optional device tree property disables UART configuration for I3D.
afe*, ref-buf-size	Specifies the buffer size (in bytes) used by AFE to store its input reference data, both before and after SSRC downsampling. This must be large enough with respect to the AFE period size and the number of input channels.
afe*, ref-mic-latency-us	Specifies the latency control (in microseconds) for the AFE path.
afe*, sample-rate	Specifies the AFE processing sample rate.
afe*, period-size	Specifies the AFE processing period size (in number of samples per channel).
afe*, sample-size	Specifies the AFE sample size (in bytes) for AFE processing. Only 4 is supported for the time being.
afe*, design-in-mic-channels	Specifies how many input microphone channels are supported by the loaded DSPC design.
afe*, design-in-ref-channels	Specifies how many input reference channels are supported by the loaded DSPC design.
afe*, design-out-channels	Specifies how many output channels are supported by the loaded DSPC design.
afe*,profile-msec	Defines the profiling period for AFE (in milliseconds). 0 means disabled.
afe*, ref-ssrc-quality	Determines the SSRC quality used for reference data downsampling.
afe*, mic-input-name	Determines which input stream is used as the microphone (instead of compiled-time fixed value, typically "mic-swpdm-input"). For example: "adc-input".
asrc**: resampler-taps	Determines the number of taps considered for the asrc resampler step. This is an optional parameter. It uses the default value (128), if it’s not defined. The allowed values are 64 and 128 (default).
asrc**: dma-buf-length	Determines the size of the DMA output buffer size in bytes. The default value is 128 KB.
asrc**: dma-rx-nr-period	Determines the number of buffers descriptors used by the DMA on rx transfers. The default value is 8.
asrc**: dma-tx-nr-period	Determines the number of buffers descriptors used by the DMA on tx transfers. The default value is 8.
asrc**: disable-dma	Disables the DMA transfer and uses the CPU copy to/from the hardware ASRC module. It can be either defined or not. By default, it is not defined.
asrc**: inout-wait	Adds some active wait time (in microsecond) at the end of the start process. This is an optional property. The default value is 0.
tx-channel-status	Transmits the channel status as part of the SPDIF TX signal.

Please note that AFE related properties are used on release packages supporting the audio front end feature.

Please note that ASRC related properties are used only on i.MX 8M Nano and Nano UL platforms, because these are used to configure the hardware ASRC module present on Nano/NanoUL only.

7.2.2. Board callback

The device tree also allows to register a callback to initialize the audio board by placing a “compatible” property on the board root node. The "board.c" file shows an example on how to export this callback so that Little Kernel can call it during boot. Particularly, this is done with the “BOARD_EXPORT” macro. The syntax is as follows:

```
BOARD_EXPORT(#board_name, compatible, callback)
```

7.2.3. SAI configuration

Immersiv3D provides specific types of configuration for the SAI to correctly transport the I2S audio stream. For this, the LK device tree provides two main properties:

- The SAI node should contain the “rx,bcp” and the “tx,bcp” properties that indicate the Bit Clock Polarity. When configured to 0, the bit clock is active high with drive outputs on the rising edge and sample inputs on the falling edge. When configured to 1, the bit clock is active low with drive outputs on the falling edge and sample inputs on the rising edge.
- The HDMI node should contain the “i2s-fmt” property for the supported protocols. This property is an array specifying the SAI configuration as follows: i2s-fmt = < [SAI config for IEC60958] [SAI config for Custom Protocol] [SAI config for IEC61937] >. Note that if a protocol is not supported, the corresponding “i2s-fmt” value should be 0xFF.

The available SAI configurations and values for “i2s-fmt” are as follows:

- Left justified (Figure 22) with a value of 0

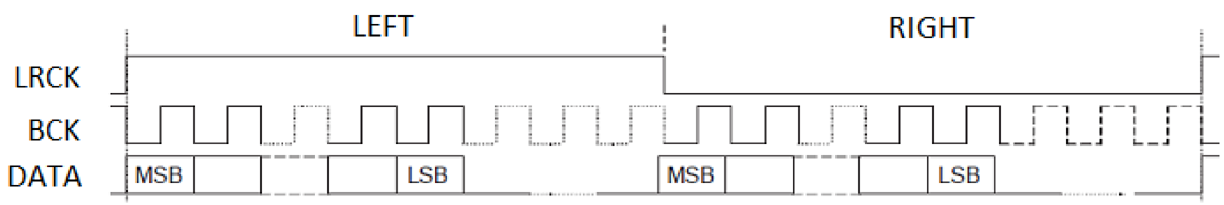

Figure 22. SAI configuration – Left Justified

- Right justified with a value of 1

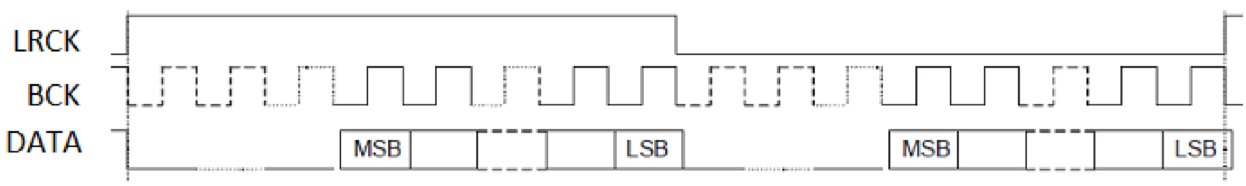

Figure 23. SAI configuration – Right Justified

- Standard I2S with a value of 2

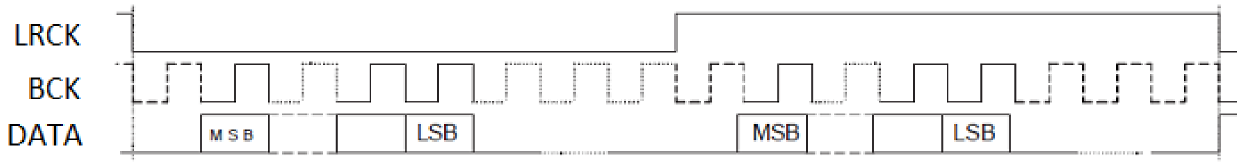

Figure 24. SAI configuration – Standard I2s

- PCM A with a value of 3

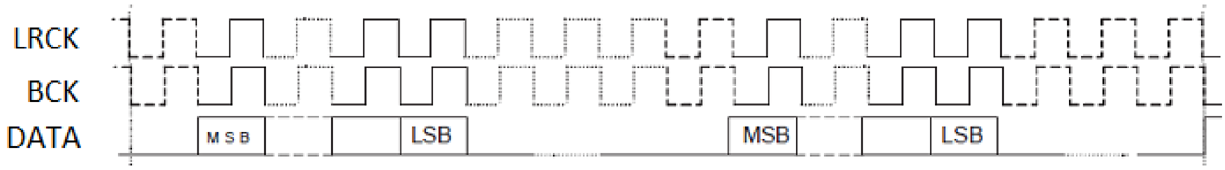

Figure 25. SAI configuration –PCM A

- PCM B with a value of 4

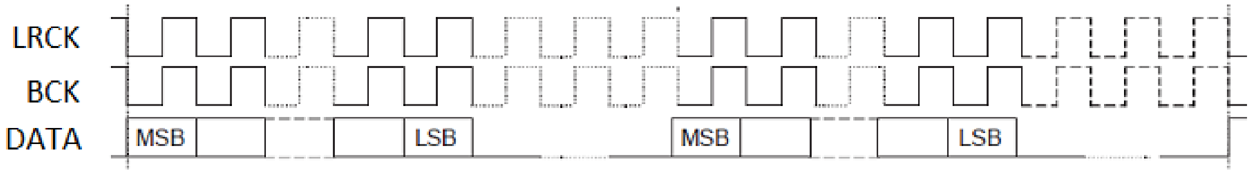

Figure 26. SAI configuration –PCM B

- AES3 with a value of 5

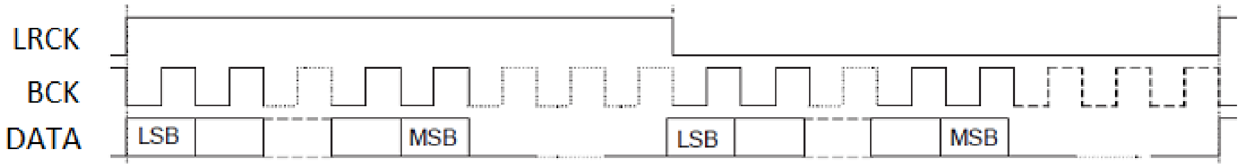

Figure 27. SAI configuration – AES 3

7.2.4. Multiple SAI TX configuration

Multiple-SAI allows transmitting data using several SAIs, thus allowing to use more lanes than a single SAI instance would allow. As of today, this feature is enabled only on the TX side of the SAI driver.

This feature has been introduced mainly for i.MX 8M Nano, because this SoC does not instantiate any 8-lane SAI (what i.MX 8M Mini did on SAI1). Because it is available as part of Audio Framework, it can be used also on i.MX 8M Mini.

As an example, Audio Framework can output 8 channels in I2S format using 4 lanes: 2 lanes on SAI3, 1 lane on SAI6, 1 lane on SAI7.

To enable multiple-SAI, you must declare a master SAI and one or more (up to 3) SAI as slaves. Be aware that BCLK and LRCK signals must be sent from a master SAI to slave SAIs, either internally to the SoC (when allowed, see `Reference Manual`, Chapter `Multiple SAI Synchronous mode` for reference) or externally (via board layout).

In the Audio Framework context, the multi-SAI enablement is done by describing a SAI chain, from the first (master) down to the other slave SAIs. This is achieved by adding the `tx,sai_chained` property into each related SAI node.

For example, if you want to declare a SAI3 → SAI6 → SAI7 chain:

```
&sai3{
    tx,sai_chained = <&sai6>;
};

&sai6{
    tx,sai_chained = <&sai7>;
    tx,slave_mode;
};

&sai7{
    /* no SAI chain, as SAI7 is the last one on the SAI chain */
    tx,slave_mode;
};
```

Please note that in addition to the SAI chain property, you must consequently adapt the master/slave property of the related SAIs, to force all secondary SAIs to slaves (master is the default configuration when not specified). This is achieved by adding the `tx,slave_mode` property to the related nodes.

Because the multi-SAI feature requires using DMA to feed data across the various SAI IPs, you must:

- enable DMA
- make sure that the 0-copy cached mode is used:
  - tx,dma-mode = < SDMA_MODE_ZEROCOPY_CACHED_BUF >;
- use the `SDMA_PERIPHERAL_TYPE_MULTI_SAI_TX` SDMA script on the associated channel

For example:

```
&sai3{
    tx,sai_chained = <&sai6>;
    tx,dma-mode = < SDMA_MODE_ZEROCOPY_CACHED_BUF >;
    dmas = <&sdma3 SDMA_REQ_SAI3_RX SDMA_PERIPHERAL_TYPE_MULTI_FIFO_SAI_RX 2>,
           <&sdma3 SDMA_REQ_SAI3_TX SDMA_PERIPHERAL_TYPE_MULTI_SAI_TX 2>;
};
```

current implementation does not support more than 4 SAIs in a chain.

The multi-SAI feature can be combined with the TDM feature. This allows support for high number of channels spread across several SAIs. To do so, see the TDM feature description in the `User Guide` to enable the TDM for DAC output. The same number of slots will be used for all SAIs involved in the DAC SAI chain.

Check the CPLD mode settings when configuring this feature.

7.2.5. Audio clock configuration

Immersiv3D uses a specific node in the device tree to generate the assignments of audio clock for better management: `audio_mclk`.

Each needed clock configuration is assigned with a clock name that reflects its usage: audio component – sai number. This is then attributed to the component that needs that specific clock configuration.

The following is an example for HDMI. In this scenario, `hdmi-mclk-sai1` and `spdif-cs-sai3` are configured when using HDMI.

```
clock-names =
    "spdif-mclk-sai1",
    "alsa-mclk-sai1",
    "hdmi-mclk-sai1",
    "adc-mclk-sai3",
    "spdif-cs-sai3";
clock-cfg = <
    kCLOCK_Idx_RootSai1 kCLOCK_SaiRootmuxAudioPll2 1 16 kCLOCK_Sai1
24576000
    kCLOCK_Idx_RootSai1 kCLOCK_SaiRootmuxAudioPll2 1 16 kCLOCK_Sai1
24576000
    kCLOCK_Idx_RootSai1 kCLOCK_SaiRootmuxAudioPll2 1 16 kCLOCK_Sai1
24576000
    kCLOCK_Idx_RootSai3 kCLOCK_SaiRootmuxAudioPll1 1 8 kCLOCK_Sai3
49152000
    kCLOCK_Idx_RootSai3 kCLOCK_SaiRootmuxAudioPll1 1 8 kCLOCK_Sai3
49152000
>;
config-names = "spdif", "hdmi", "alsa", "adc";
config-spdif = "spdif-mclk-sai1","spdif-cs-sai3";
config-hdmi = "hdmi-mclk-sai1","spdif-cs-sai3";
config-alsa = "alsa-mclk-sai1","spdif-cs-sai3";
config-adc = "adc-mclk-sai3";
```

The audio clock configuration node also contains the definitions of the PLLs that locks them to the expected sample rate. The representative names are placed inside the `pll-names` property and the actual values are in `pll-cfg`. These values are actually written in PLL registers to obtain the corresponding frequency. Below is an example with the associated correspondence:

```
pll-names =
    "mclk-48k-pll2",
    "mclk-44k-pll2",
    "mclk-32k-pll2",
    "mclk-48k-pll1",
    "mclk-44k-pll1",
    "mclk-32k-pll1";
pll-cfg = <
    393215995U 262 2 3 9437 kANALOG_PllRefOsc24M kCLOCK_AudioPll2Ctrl
    361267196U 361 3 3 17511 kANALOG_PllRefOsc24M kCLOCK_AudioPll2Ctrl
    262143997U 262 3 3 9437 kANALOG_PllRefOsc24M kCLOCK_AudioPll2Ctrl
    393215995U 262 2 3 9437 kANALOG_PllRefOsc24M kCLOCK_AudioPll1Ctrl
    361267196U 361 3 3 17511 kANALOG_PllRefOsc24M kCLOCK_AudioPll1Ctrl
    262143997U 262 3 3 9437 kANALOG_PllRefOsc24M kCLOCK_AudioPll1Ctrl
>;
```

Having these definitions simplifies the PLL rate assignments for each component. For example, if a 48k rate is required for the HDMI clock using PLL2, then ‘pll-mask-hdmi' must be set according to the ‘pll-names’ property.

```
pll-names-hdmi =
    "mclk-48k-pll1",
    "mclk-44k-pll1",
    "mclk-32k-pll1",
    "mclk-48k-pll2",
    "mclk-44k-pll2",
    "mclk-32k-pll2";
pll-mask-hdmi = <0x4>;
```

PLL names are also used in the SAI nodes for the `mclk-tx-config-names` and `mclk-rx-config names` properties to retrieve the corresponding PLL multiplier. There are tree values that must be filled for each `mclk-*` property: the first one is for the 44k rate, the second one is for 48k, and the last one is for 32k. Please note that this is a fixed order from the driver.

The following is an example for setting PLL1 rates on both RX and TX and making multiplies of 32k using `mclk-48k-pll`:

```
/* mclk config names, in order 44k, 48k, 32k configurations */
mclk-tx-config-names = "mclk-44k-pll1", "mclk-48k-pll1", "mclk-48k-pll1";
mclk-rx-config-names = "mclk-44k-pll1", "mclk-48k-pll1", "mclk-48k-pll1";
```

7.2.6. Channel status support

The channel status can be transmitted as part of the SPDIF TX signal. Its value is specified via the device tree using the `tx-channel-status` property.

The following is an example:

```
&spdif1 {
    clock-names = "bus", "clk-tx";
    clock-cfg = <
        kCLOCK_Idx_RootNone 0 kCLOCK_Divider_None kCLOCK_Divider_None kCLOCK_None
200000000
        kCLOCK_Idx_RootSpdif1 kCLOCK_SpdifRootmuxAudioPll2 1 8 kCLOCK_None 49152000
        >;

    /* PCM, copyright, 48kHZ (dynamically updated), 24 bits samples */
    tx-channel-status = < 0x04 0x0 0x0 0x2 0xdb 0x0 0x40 0x0 >;
};
```

Please note that the transmitted channel status value, specifically the bits containing the sampling rate, will be updated dynamically to reflect the actual pipeline output sampling rate. As such, the sampling rate bits specified in device tree property must be considered only as a default value, which may be overwritten internally as soon as the pipeline output starts.

7.3. Jailhouse configuration

In Immersiv3D, Jailhouse is used to separate and isolate the i.MX 8M SOC hardware resources between the Linux and LK worlds. For this, two main configuration files are needed: the Root cell configuration and the Little Kernel configuration.

7.3.1. Root cell

The root cell (`imx8mm.c` - `imx8mm.cell` for i.MX 8M Mini, `imx8mn.c` - `imx8mn.cell` for i.MX 8M Nano, and `imx8mnul.c` - `imx8mnul.cell` for i.MX 8M Nano UL) specifies the hardware resources accessible by Jailhouse. This hypervisor needs access to all the hardware to properly isolate it for LK. Therefore, all resources of the i.MX 8M SOC should be added in this file. Particularly, in the `mem_regions` field to access them and the `irqchips` field to receive the corresponding interruptions.

7.3.2. Little Kernel cell

The LK cell (`imx8mm-lk-rpc.c` – `imx8mm-lk-rpc.cell`, as well as `imx8mn-lk-rpc.c` – `imx8mn-lk-rpc.cell`, and `imx8mnul-lk-rpc.c` – `imx8mnul-lk-rpc.cell`) specifies the hardware sources that Jailhouse will allow LK to access. Please note that this must be aligned with LK and the Linux device tree. Giving access to the same modules from Linux and LK can cause unexpected behavior. The `mem_regions` and `irqchips` fields must be correctly modified and adapted to the user's board.

7.4. Memory configuration

The memory configuration provided in the release is made for the reference i.MX 8M Mini SOC (X-8MMINILPD4), respectively i.MX 8M Nano SOC (X-8MNANOD4) and i.MX 8M Nano UltraLite SOC (X-8MNANOD3L) with the i.MX Audio Board (MCIMX8M-AUD). For any custom hardware configuration, Linux kernel, Little Kernel, and Jailhouse related memory regions can be reconfigured.

7.4.1. Memory usage overview

The global Little Kernel memory footprint required for I3D is approximately the sum of:

- **Static memory** from LK binary (lk.elf / lk.bin files): includes code (.text section), constants (.rodata section), and variables (.data and .bss section);
- **Dynamic memory** (heap): the consumption depends on the actual need for dynamic allocations via malloc(), calloc(), realloc(), and so on. Therefore, it varies according to the actual use case.

The static memory footprint depends on the build configuration. A consolidated value for all sections can be obtained by multiple ways, for example using the 'size' GNU binary utility. In the example below, the total static footprint is 21378208 bytes (20.4 MB).

```
$ /opt/toolchains/gcc-linaro-7.3.1-2018.05-x86_64_aarch64-elf/bin/aarch64-elf-size
build-imx8mm-af-virt/lk.elf
text            data             bss            dec             hex
filename
11061224        8710032 1606952 21378208        14634a0 build-imx8mm-af-virt/lk.elf
```

The dynamic memory is allocated from the heap, which is the space mapped in the LK memory after the image. The heap size grows as memory allocations add up and require more total memory. It does not shrink, even if a part of the dynamic memory is later freed, so that it is available for reuse. The heap size and freed memory segments can be listed from the LK kernel console using 'heap info' from the shell. The example below shows a heap size of 0x2aa5000 (42.6 MB) with multiple free memory segments.

```
] heap info
[79341.894127] shell > Heap dump (using miniheap):
[79341.894132] shell >       base 0xffff0000014de000, len 0x2aa5000
[79341.894136] shell >  free list:
[79341.894138] shell >         base 0xffff000002e68448, end 0xffff000002e68468, len
0x20
[79341.894139] shell >         base 0xffff000002e68488, end 0xffff000002e684b8, len
0x30
[79341.894142] shell >         base 0xffff000002ec9228, end 0xffff000002ec9288, len
0x60
[...]
```

The heap can grow as long as there are pages (4 KB) still available in the physical allocator and the physical memory manager (pmm). The physical allocators keep track of memory areas (arenas) consisting in a number of contiguous pages. At runtime, the remaining pages are available for the heap to grow to serve additional dynamic allocation. Remaining number of pages can be monitored using the 'pmm arenas' command from the LK shell. The following example has 1899 pages (7.4 MB) left and available to the heap out of a single arena of 0x4000000 (64 MB).

```
] pmm arenas
[80256.454344] shell > arena 0xffff000000a9d038: name 'ram' base 0x80000000 size
0x4000000 priority 0 flags 0x1
[80256.454348] shell >  page_array 0xffff00000147e000, free_count 1899
[80256.454348] shell >  free ranges:
[80256.454532] shell >                 0x83895000 - 0x84000000
```

The PMM memory arena size can be changed through the LK device tree using the meminfo node.

7.4.2. Jailhouse memory configuration

This section provides information about the memory configuration for both Linux and Little Kernel corresponding changes and Jailhouse cells.

Each memory region is represented by physical and virtual start addresses, size, and flags. Below is an example for the IO node.

```
/* IO */ {
    .phys_start = 0x00000000,
    .virt_start = 0x00000000,
    .size =
    0x40000000,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
             JAILHOUSE_MEM_IO,
},
```

7.4.2.1. Linux root cell

On the Jailhouse side, the layout of the memory regions for Linux is defined in `imx8mm.c`, `imx8mn.c`, or `imx8mnul.c` under `configs/arm64/` from Jailhouse sources. This must be clearly delimited with no overlapping regions.

The following is an example of the memory configuration for the Linux root cell on 8M Mini:

Table 9. Memory layout in 8M mini Linux root cell

Resource	Memory range
IO	0x00000000 - 0x40000000
RAM 00	0x40000000 – 0xb3c00000
RAM 01	0xb8000000 – 0xbb700000
RAM 02	0xbbc00000 - 0xbe000000
Inmate memory	0xb3c00000 - 0xb7c00000
Ivshmem	0xbba00000 - 0xbbc00000
Hypervisor memory	0xb7c00000 – 0xb8000000
Loader	0xbb700000 - 0xbb800000

Resource	Memory range
PCI	0xbb800000 - 0xbba00000
OP-TEE memory	0xbe000000 – 0xc0000000

When memory regions are reconfigured, please consider the memory alignment constraints from the reserved-memory node from Linux dts files.

Please note that each element must be consistent with the corresponding child node from the reserved memory.

The memory resources in the Jailhouse configuration cell must be in the Linux memory range defined in the memory node in Linux dts files.

Specifically for I3D, the Linux Root Cell's memory regions "inmate" and "Loader" aren't needed. Therefore, these regions can be removed.

7.4.2.2. Little Kernel cell

The Little Kernel Jailhouse configuration is found in `imx8mm-lk.c`, `imx8mn-lk.c`, or `imx8mnul-lk.c` cell file under `configs/arm64/` from jailhouse sources. This cell file configures the hardware resources used by LK and also memory regions specific to Jailhouse, like IVSHMEM and communication regions.

The RAM size of the LK must be consistent with the meminfo node from the Little Kernel dts.

The following is an example of the changes that must be done when resizing the RAM for the Little Kernel cell:

- Little Kernel cell configuration file (`imx8mm-lk.c`):

```
/* RAM: 256MB */ {
    .phys_start = 0x93c00000,
    .virt_start = 0x80000000,
    .size = 0x10000000 ,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
       JAILHOUSE_MEM_EXECUTE | JAILHOUSE_MEM_LOADABLE,
},
```

- Little Kernel dts file (`imx8mm-ab2.dts`):

```
meminfo {
    phys_start = < 0x93c00000 >;
    size = < 0xfc00000 >; /* 252 MiB */

    phys_db_offset = < 0xfc00000 >; /* 252 MiB */
};
```

- Linux dts file (imx8mm-ab2-af.dts):

```
&inmate_reserved {
    no-map;
    reg = <0 0x93c00000 0x0 0x10000000 ;
}
```

Please note that after applying these custom changes, the corresponding files must be rebuilt (imx8mm-lk-rpc.cell, imx8mm-ab2-rpc.dtb, and imx8mm-ab2-af-rpc.dtb) and updated on the target. A similar file hierarchy must be updated for i.MX 8M Nano and i.MX 8M Nano UltraLite SOC.

The starting address for the Little Kernel can be reconfigured to map Little Kernel to a different physical address.

The following is an example for the changes that must be done when changing the RAM physical address for the Little Kernel cell:

- Little Kernel cell configuration file (imx8mm-lk.c):

```
/* RAM: 512MB */ {
    .phys_start = 0x70000000,
    .virt_start = 0x80000000,
    .size = 0x20000000,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
       JAILHOUSE_MEM_EXECUTE | JAILHOUSE_MEM_LOADABLE,
},
```

- Little Kernel dts file (imx8mm-ab2.dts):

```
meminfo {
    phys_start = < 0x70000000 >;
    size = < 0x1fc00000 >; /* 508 MiB */

    phys_db_offset = < 0x1fc00000 >; /* 508 MiB */
};
```

- Linux dts file (imx8mm-ab2-af.dts):

```
&inmate_reserved {
    no-map;
    reg = <0 0x70000000 0x0 0x24000000>;
};
```

Please note that after applying these custom changes, the corresponding files must be rebuilt (imx8mm-lk-rpc.cell, imx8mm-ab2-rpc.dtb, and imx8mm-ab2-af-rpc.dtb) and updated on the target. A similar file hierarchy must be updated for i.MX 8M Nano and i.MX 8M Nano UltraLite SOC.

7.4.3. Little Kernel memory configuration

In addition to the memory regions, memory usage by Little Kernel can be reduced depending on the use cases that must be supported. The following nodes are not mandatory and specific to certain use cases:

Table 10. LK I3D node description

Node	Use case
af_event:bin-device205	Endpoint for event manager.
rpmsg-bin	Endpoint for Linux filesystem read/write. The application buffer must be large enough to hold a complete file read.
rpmsg-wd	Endpoint for watchdog events.
cipc	Use I3D's CIPC interface to exchange data from Linux to LK through Linux's filesystem.
alsa_main:alsa-device300	Use I3D's ALSA main interface to playback audio from Linux or to capture audio from LK.
alsa_cpp:alsa-device301	Use I3D's CPP ALSA interface to capture audio from LK. This implies implementing a CPP that will send the audio content to the CPP ALSA interface.
alsa_voice:alsa-device302	Use I3D's voice interface to playback voice from Linux to LK.
alsa_mic:alsa-device303	Mic interface to capture microphone from LK.
audio-weaver-rpmsg	Default I3D doesn't include audio weaver. Therefore, this node can be removed.
spdif1	I3D SPDIF device.

Please note that some of these endpoints are also present at the Linux side, so they must be disabled there as well.

Besides Little Kernel/Linux configurable nodes, there are also pipeline configurable properties which are listed in the af.dtsi device tree:

Table 11. Pipeline memory configuration

Node	Comment

common0:af_common0	Defines the maximum concurrent channels in the pipeline [2-32]. Impact on lipsync buffer size (192 kHz max, 500 ms maximum): - 16 MB for 32 channels; - 8 MB for 16 channels. Default value is 16.
om0:af_om0	Defines the output/channel buffer delay per channel, definition per output path. The default value is 1 MB (64 KB x 16 channels).

Chapter 8. Revision history

Revision number	Date	Substantive changes
Rev 2.5	06/24/2019	Update HAL API and binary path.
Rev 2.6	06/26/2019	Update Board Adaptation chapter with more detail.
Rev 2.7	06/27/2019	Add Control Process Chapter. Added missing fields in LK device tree.
Rev 2.8	07/22/2019	Add SAI configuration section.
Rev 2.9	07/29/2019	Add Memory Configuration chapter.
Rev 2.10	09/02/2019	Update Memory Configuration chapter.
Rev 2.11	09/19/2019	Add new event management in CP. Add event notifier in CP.
Rev 2.12	09/29/2019	Remove af_sink/af_source references. Add start() and stop() callbacks for PP element. Update CP event handling information.
Rev 2.13	12/17/2019	Add ADC RPC information.
Rev 2.14	01/09/2020	Update device tree examples. Add audio clock configuration node details.
Rev V2 2.0.0	10/08/2020	Add i.MX 8M Nano and i.MX Audio Board. Update I3D PP Level and Parser API. Update memory configuration section. Update RPC interface support. Update CIPC endpoint API.
Rev V2 3.0.0	15/12/2020	Add Multiple SAI TX Configuration chapter. Update LK device tree options. Update Control process Event Notifier. Merge Stream/Control endpoints for ALSA path.

Revision number	Date	Substantive changes
Rev V2 4.0.0	24/03/2021	Update New event handling example. Update Jailhouse Memory Configuration. Update LK device tree properties.
Rev V2 5.0.0	26/05/2021	Update LK device tree properties from Little Kernel Device Tree chapter.
Rev V2 5.1.0	13/07/2021	Update LK device tree properties from Little Kernel configuration chapter. Update Flush transition sequence diagram from Control process chapter.
Rev V2 6.0.0	3/11/2021	Add support for i.MX 8M Nano UltraLite platform. Update LK device tree properties from Little Kernel configuration chapter. Add Channel status support chapter.

Annex A: CIPC custom post processing example

```
/*
 * Copyright (c) 2011-2013 Freescale Semiconductor, Inc. All Rights Reserved
 * Copyright 2021 NXP
 *
 * NXP Confidential. This software is owned or controlled by NXP and may only
 * be used strictly in accordance with the applicable license terms found in
 * file LICENSE.txt
 *
 */

/*!
 * @file        volume.c
 * @brief       This file contains an example of Post Processing
 *              Plugin (PPP). It controls the volume of an audio stream
 */
#include <string.h>

/*
 * Include Audioframwork header files
 */
#include "ppp_provider.h"
#include "ppp_api_parser.h"
#include "osa_common.h"
#include "cp_api.h"
#include "debug.h"

/**
 * @defgroup DBG_MACROS Debug Macros
 *
 * @{
 */

/**
 * Maximum Audio channels Macro
 */
#define AUDIO_CHANNELS_MAX CASCFG_PP_AUDIO_CHANNELS_MAX

/**
 * @brief User Data structure
 */
struct volume_data {
    float gain;/**< @brief Gain to be added to the audio stream */
};
```

```
/**
 * @brief This function will do the link between key-value from the REST API and the C
structure.
 *        It can create/delete a Volume element and get/put the gain of the PPP
 *
 * @param context Pointer to the structure representing the context of the Volume PPP
 * @param cmd Rest Command to be handled
 * @param command Pointer to user's string command
 *
 * return PPP_ALLOC_STRING_SUCCESS
 * return PPP_ALLOC_STRING_ERROR
 */
static char *volume_parser(struct cowbell_context *context,
                           enum ppp_command_type cmd, char *command)
{
    struct volume_data *data;
    int property_ret = 0, ret = 0;
    char *ptr_key = NULL;
    char *ptr_value = NULL;
    char *return_string = NULL;
    bool ppp_error = false;
    char *data_string = NULL;
    char *saveptr = NULL;
    cp_event_volume_t param;

    switch (cmd) {
    case PPP_COMMAND_POST:
        printlk(LK_DEBUG, "'%s' received POST command\n", context->name);
        data = osa_malloc(sizeof(struct volume_data));
        if (!data)
            return PPP_ALLOC_STRING_ERROR;

        context->user_data = data;
        /* Set default values */
        data->gain = 1.0f;
        break;
    case PPP_COMMAND_DELETE:
        osa_free(context->user_data);
        break;
    case PPP_COMMAND_PUT:
        data = (struct volume_data *) context->user_data;
        /* Proposed helper to parse command line */
        property_ret = ppp_read_next_property_to_set(command, &ptr_key, &ptr_value,
&saveptr);
        while (property_ret == ERRCODE_NO_ERROR && ppp_error == false) {
            PPP_SWITCH (ptr_key) {
            PPP_CASE ("gain"):
                /* Proposed helper to convert string to expected type */
                ret = ppp_set_string_to_type(ptr_value, &data->gain, "float");
                if (ERRCODE_NO_ERROR != ret) {
                    printlk(LK_ERR, "Error: Invalid command \"%s=%s\"\n", ptr_key,
```

```
ptr_value);
                    return PPP_ALLOC_STRING_ERROR;
                }
                param.volume = data->gain;
                /* Send Event of gain change to CP */
                ret = cp_send_event(0, CP_EVENT_CPP_VOLUME, (void *) &param,
sizeof(param));
                if (ERRCODE_NO_ERROR != ret) {
                    printlk(LK_ERR, "Error: Failed Send Event to CP\n");
                    return PPP_ALLOC_STRING_ERROR;
                }

                PPP_BREAK;

            PPP_DEFAULT:
                printlk(LK_ERR, "Error: Key \"%s=%s\" not found\n", ptr_key,
ptr_value);
                ppp_error = true;
                PPP_BREAK;
            }
            property_ret = ppp_read_next_property_to_set(NULL, &ptr_key, &ptr_value,
&saveptr);
        }

        return (ppp_error == false) ? PPP_ALLOC_STRING_SUCCESS :
PPP_ALLOC_STRING_ERROR;
    case PPP_COMMAND_GET:
        data = (struct volume_data *) context->user_data;
        /* Proposed helper to parse command line */
        property_ret = ppp_read_next_property_to_get(command, &ptr_key, &saveptr);
        while (property_ret == ERRCODE_NO_ERROR) {
            PPP_SWITCH (ptr_key) {

            PPP_CASE ("gain"):
                /* Proposed helper to convert type to expected string */
                data_string = ppp_get_string_from_type(ptr_key, &data->gain, "float");
                PPP_BREAK;

            PPP_DEFAULT:
                printlk(LK_ERR, "Error: Key \"%s\" not found\n", ptr_key);
                data_string = PPP_ALLOC_STRING_ERROR;
                PPP_BREAK;
            }

            /* Concatenate current string to return string */
            ppp_add_to_return_string(&return_string, data_string);
            /* Free memory allocated by ppp_get_string_from_type() */
            osa_free(data_string);
            property_ret = ppp_read_next_property_to_get(NULL, &ptr_key, &saveptr);
        }
```

```
        println(LK_DEBUG, "PPP_COMMAND_GET returns = %s\n", return_string);

        return return_string ? return_string : PPP_ALLOC_STRING_ERROR;
    default:
        return PPP_ALLOC_STRING_ERROR;
    }

    return PPP_ALLOC_STRING_SUCCESS;
}


/**
 * @brief This function will do the post processing.
 *        It adds a gain to the audio stream.
 *
 * @param context Pointer to the structure representing the context of the Volume PPP
 *
 * return "OK"
 */
static const char *volume_process(struct cowbell_context *context, size_t len)
{
    struct volume_data *data = (struct volume_data *) context->user_data;
    float *psink;
    size_t samples_count;
    size_t i, j;

    if (len % sizeof(float)) {
        printlk(LK_ERR, "Do not support this buffer len :%lu\n", len);
        return PPP_FIX_STRING_ERROR;
    }

    samples_count = len / sizeof(float);
    for (i = 0; i < AUDIO_CHANNELS_MAX; i++) {
        psink = (float *) ppb_get_sink(context, i);

        if (psink == NULL)
            continue;

        for (j = 0; j < samples_count; j++)
            *psink++ *= data->gain;
    }

    return PPP_FIX_STRING_SUCCESS;
}


/**
 * @brief This function is called before starting stream
 *
 * @param context Pointer to the structure representing the context of the Volume PPP
 */
```

```
static void volume_start(struct cowbell_context *context)
{
    struct volume_data *data = (struct volume_data *)context->user_data;

    printlk(LK_DEBUG, "volume start:%f\n", data->gain);
}


/**
 * @brief This function is called after stream has stopped
 *
 * @param context Pointer to the structure representing the context of the Volume PPP
 */
static void volume_stop(struct cowbell_context *context)
{
    struct volume_data *data = (struct volume_data *)context->user_data;

    printlk(LK_DEBUG, "volume stop:%f\n", data->gain);
}


/**
 * @brief This function will return the PPP capabilities.
 *
 *
 * return PPP capabilities
 */
static const char *volume_get_caps(void)
{
    return "numsink=32&numsrc=32&gain=property";
}


/**
 * Driver structure containing PPP name and callbacks
 */
static struct cowbell_driver ppp_volume = {
    .compat = "volume.elt",
    .ops = {
        .start = volume_start,
        .stop = volume_stop,
        .parser = volume_parser,
        .process = volume_process,
        .get_caps = volume_get_caps,
    },
};


/**
 * Initilization function to register Volume PPP
 */
```

```
static void __attribute__ ((constructor)) volume_init(void)
{
    register_ppp_driver(&ppp_volume);
}
```

Annex B: New event handling example

SDK has implemented an example event CP_EVENT_CPP_VOLUME, which requires modification in two files:

- Modification in the sap_cp.h:

```
/*
 * Event identifiers
 */
enum CP_EVENT_ID {
    CP_EVENT_CPP_VOLUME, // event example
};

typedef struct {
    float volume;
} cp_event_volume_t;

typedef struct {
    frm_msg_header_t hdr;

    unsigned id;   /* command identifier */
    cp_cmd_sync_t sync; /* sync object for blocking call */
    union {
        cp_event_volume_t volume_config; /* Volume param */
    } param;
} cp_event_ind_t;
```

CP_EVENT_CPP_VOLUME is the ID used for the example volume event and cp_event_volume_t is the data structure associated with the event.

- Modification in the cp_event.c:

```
    case CP_EVENT_CPP_VOLUME:
        printlk(LK_NOTICE, "CP_EVENT_CPP_VOLUME [DATA] volume = %f \r\n", event_req-
>param.volume_config.volume);
        break;
```

The event CP_EVENT_CPP_VOLUME is handled by adding a switch case in cp_handle_event API.

How to Reach Us:

Home Page:

nxp.com

Web Support:

nxp.com/support