ALSA Compress-Offload API

Pierre-Louis.Bossart <pierre-louis.bossart@linux.intel.com>

Vinod Koul <vinod.koul@linux.intel.com>

Overview

Since its early days, the ALSA API was defined with PCM support or constant bitrates payloads such as IEC61937 in mind. Arguments and returned values in frames are the norm, making it a challenge to extend the existing API to compressed data streams.

In recent years, audio digital signal processors (DSP) were integrated in system-on-chip designs, and DSPs are also integrated in audio codecs. Processing compressed data on such DSPs results in a dramatic reduction of power consumption compared to host-based processing. Support for such hardware has not been very good in Linux, mostly because of a lack of a generic API available in the mainline kernel.

Rather than requiring a compatibility break with an API change of the ALSA PCM interface, a new 'Compressed Data' API is introduced to provide a control and data-streaming interface for audio DSPs.

The design of this API was inspired by the 2-year experience with the Intel Moorestown SOC, with many corrections required to upstream the API in the mainline kernel instead of the staging tree and make it usable by others.

Requirements

The main requirements are:

- separation between byte counts and time. Compressed formats may have a header per file, per frame, or no header at all. The payload size may vary from frame-to-frame. As a result, it is not possible to estimate reliably the duration of audio buffers when handling compressed data. Dedicated mechanisms are required to allow for reliable audio-video synchronization, which requires precise reporting of the number of samples rendered at any given time.
- Handling of multiple formats. PCM data only requires a specification of the sampling rate, number of channels and bits per sample. In contrast, compressed data comes in a variety of formats. Audio DSPs may also provide support for a limited number of audio encoders and decoders embedded in firmware, or may support more choices through dynamic download of libraries.
- Focus on main formats. This API provides support for the most popular formats used for audio and video capture and playback. It is likely that as audio compression technology advances, new formats will be added.
- Handling of multiple configurations. Even for a given format like AAC, some implementations may support AAC multichannel but HE-AAC stereo. Likewise WMA10 level M3 may require too much memory and cpu cycles. The new API needs to provide a generic way of listing these formats.
- Rendering/Grabbing only. This API does not provide any means of hardware acceleration, where PCM samples are provided back to user-space for additional processing. This API focuses instead on streaming compressed data to a DSP, with the assumption that the decoded samples are routed to a physical output or logical back-end.
- Complexity hiding. Existing user-space multimedia frameworks all have existing enums/structures for each compressed format. This new API assumes the existence of a platform-specific compatibility layer to expose, translate and make use of the capabilities of the audio DSP, eg. Android HAL or PulseAudio sinks. By construction, regular applications are not supposed to make use of this API.

Design

The new API shares a number of concepts with the PCM API for flow control. Start, pause, resume, drain and stop commands have the same semantics no matter what the content is.

The concept of memory ring buffer divided in a set of fragments is borrowed from the ALSA PCM API. However, only sizes in bytes can be specified.

Seeks/trick modes are assumed to be handled by the host.

The notion of rewinds/forwards is not supported. Data committed to the ring buffer cannot be invalidated, except when dropping all buffers.

The Compressed Data API does not make any assumptions on how the data is transmitted to the audio DSP. DMA transfers from main memory to an embedded audio cluster or to a SPI interface for external DSPs are possible. As in the ALSA PCM case, a core set of routines is exposed; each driver implementer will have to write support for a set of mandatory routines and possibly make use of optional ones.

The main additions are

get_caps
: This routine returns the list of audio formats supported. Querying the codecs on a capture stream will return encoders,

decoders will be listed for playback streams.

get_codec_caps
: For each codec, this routine returns a list of capabilities. The intent is to make sure all the capabilities correspond to valid settings, and to minimize the risks of configuration failures. For example, for a complex codec such as AAC, the number of channels supported may depend on a specific profile. If the capabilities were exposed with a single descriptor, it may happen that a specific combination of profiles/channels/formats may not be supported. Likewise, embedded DSPs have limited memory and cpu cycles, it is likely that some implementations make the list of capabilities dynamic and dependent on existing workloads. In addition to codec settings, this routine returns the minimum buffer size handled by the implementation. This information can be a function of the DMA buffer sizes, the number of bytes required to synchronize, etc, and can be used by userspace to define how much needs to be written in the ring buffer before playback can start.

set_params
: This routine sets the configuration chosen for a specific codec. The most important field in the parameters is the codec type; in most cases decoders will ignore other fields, while encoders will strictly comply to the settings

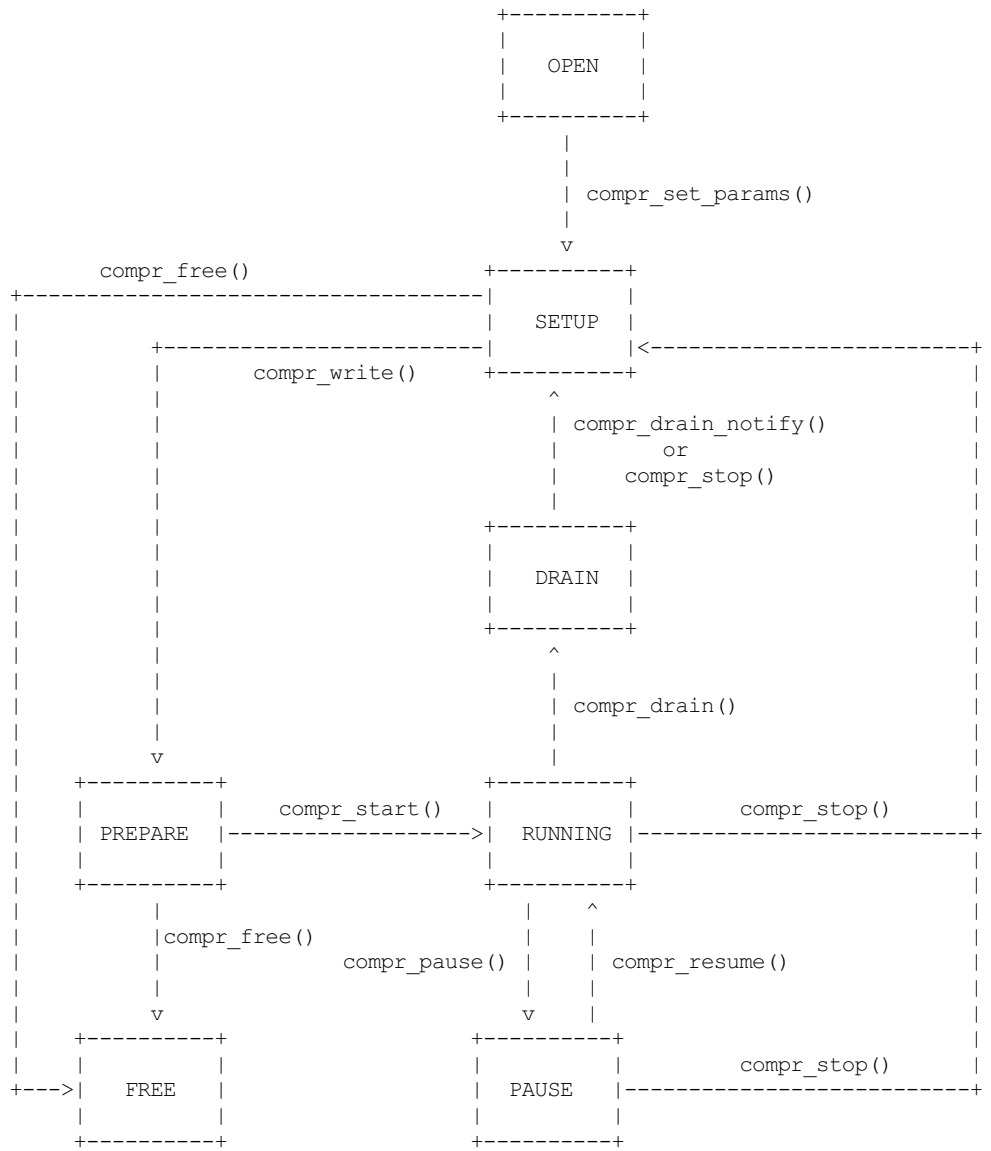
get_params
: This routines returns the actual settings used by the DSP. Changes to the settings should remain the exception.

get_timestamp
: The timestamp becomes a multiple field structure. It lists the number of bytes transferred, the number of samples processed and the number of samples rendered/grabbed. All these values can be used to determine the average bitrate, figure out if the ring buffer needs to be refilled or the delay due to decoding/encoding/io on the DSP.

Note that the list of codecs/profiles/modes was derived from the OpenMAX AL specification instead of reinventing the wheel. Modifications include: - Addition of FLAC and IEC formats - Merge of encoder/decoder capabilities - Profiles/modes listed as bitmasks to make descriptors more compact - Addition of set_params for decoders (missing in OpenMAX AL) - Addition of AMR/AMR-WB encoding modes (missing in OpenMAX AL) - Addition of format information for WMA - Addition of encoding options when required (derived from OpenMAX IL) - Addition of rateControlSupported (missing in OpenMAX AL)

State Machine

The compressed audio stream state machine is described below



Gapless Playback

When playing thru an album, the decoders have the ability to skip the encoder delay and padding and directly move from one track content to another. The end user can perceive this as gapless playback as we don't have silence while switching from one track to another

Also, there might be low-intensity noises due to encoding. Perfect gapless is difficult to reach with all types of compressed data, but works fine with most music content. The decoder needs to know the encoder delay and encoder padding. So we need to pass this to DSP. This metadata is extracted from ID3/MP4 headers and are not present by default in the bitstream, hence the need for a new interface to pass this information to the DSP. Also DSP and userspace needs to switch from one track to another and start using data for second track.

The main additions are:

set_metadata
: This routine sets the encoder delay and encoder padding. This can be used by decoder to strip the silence. This needs to be set before the data in the track is written.

set_next_track
: This routine tells DSP that metadata and write operation sent after this would correspond to subsequent track

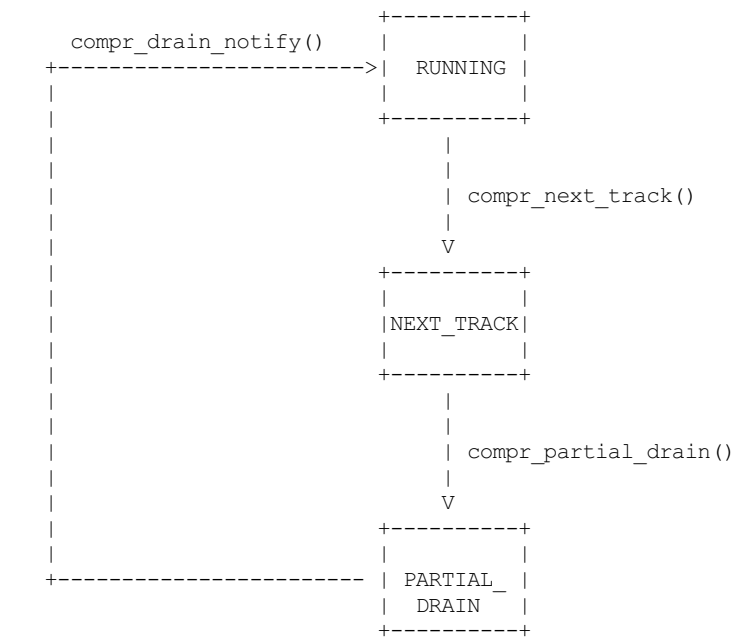
partial drain
: This is called when end of file is reached. The userspace can inform DSP that EOF is reached and now DSP can start skipping padding delay. Also next write data would belong to next track

Sequence flow for gapless would be: - Open - Get caps / codec caps - Set params - Set metadata of the first track - Fill data of the first track - Trigger start - User-space finished sending all, - Indicate next track data by sending set_next_track - Set metadata of the next track - then call partial_drain to flush most of buffer in DSP - Fill data of the next track - DSP switches to second track

(note: order for partial_drain and write for next track can be reversed as well)

Gapless Playback SM

For Gapless, we move from running state to partial drain and back, along with setting of meta_data and signalling for next track



Not supported

- Support for VoIP/circuit-switched calls is not the target of this API. Support for dynamic bit-rate changes would require a tight coupling between the DSP and the host stack, limiting power savings.
- Packet-loss concealment is not supported. This would require an additional interface to let the decoder synthesize data when frames are lost during transmission. This may be added in the future.
- Volume control/routing is not handled by this API. Devices exposing a compressed data interface will be considered as regular ALSA devices; volume changes and routing information will be provided with regular ALSA kcontrols.
- Embedded audio effects. Such effects should be enabled in the same manner, no matter if the input was PCM or compressed.
- multichannel IEC encoding. Unclear if this is required.
- Encoding/decoding acceleration is not supported as mentioned above. It is possible to route the output of a decoder to a capture stream, or even implement transcoding capabilities. This routing would be enabled with ALSA kcontrols.
- Audio policy/resource management. This API does not provide any hooks to query the utilization of the audio DSP, nor any preemption mechanisms.

- No notion of underrun/overrun. Since the bytes written are compressed in nature and data written/read doesn't translate directly to rendered output in time, this does not deal with underrun/overrun and maybe dealt in user-library

Credits

- Mark Brown and Liam Girdwood for discussions on the need for this API
- Harsha Priya for her work on intel_sst compressed API
- Rakesh Ughreja for valuable feedback
- Sing Nallasellan, Sikkandar Madar and Prasanna Samaga for demonstrating and quantifying the benefits of audio offload on a real platform.