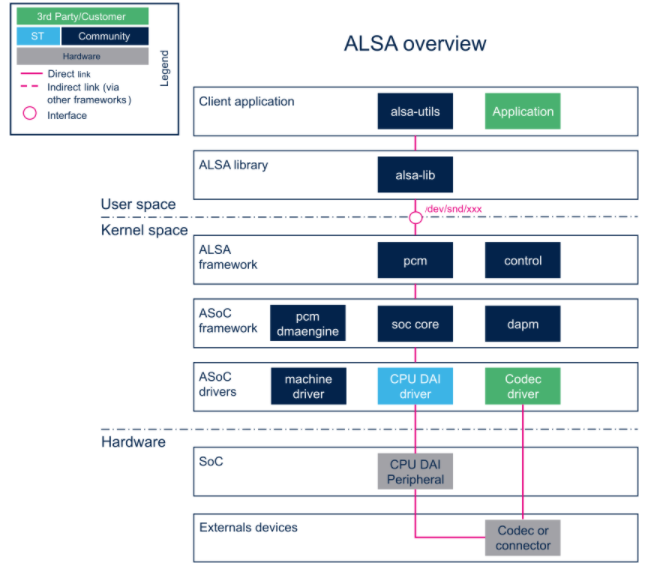
**ALSA OVERVIEW**

1. **OVERVIEW**

ALSA (Advanced Linux Sound Architecture) is a software framework and part of the Linux kernel, it provides audio functionality to the Linuxoperating system. ALSA can divide in to 2 parts: User space and Kernel space.

* User space consists application, utilities and libraries
* Kernel space consists frameworks and drivers
* **alsa-utils** (User space)

Contains the command line utilities for the ALSA project (aplay, arecord, amixer, alsamixer ...)

* **alsa-lib** (User space)

Contains the ALSA library used by programs requiring an access to the ALSA sound interface

* **ALSA framework** (Kernel space)

The ALSA core provides an API to implement audio drivers and PCM/control interfaces to expose audio devices on the userland

* **ASoC framework (ALSA System On Chip)** (Kernel space)

The aim of the ALSA System on Chip (ASoC) layer is to improve ALSA support for embedded system-on-chip processors and audio codecs.

* **ASoC drivers** (Kernel space)

ASoC drivers allow the implementation of hardware dependent code for ASoC driver classes

Prior to the ASoC subsystem there was some support in the kernel for SoC audio, however it had some limitations

* Codec drivers were often tightly coupled to the underlying SoC CPU. This is not ideal and leads to code duplication
* There was no standard method to signal user initiated audio events (e.g. Headphone/Mic insertion, Headphone/Mic detection after an insertion event).
* Drivers tended to power up the entire codec when playing (or recording) audio. There was also no support for saving power via changing codec oversampling rates, bias currents, etc.

The ASoC layer is designed to address these issues and provide the following features

* ***Codec independence***. Allows reuse of codec drivers on other platforms and machines.
* ***Easy I2S/PCM audio interface setup between codec and SoC***. Each SoC interface and codec registers its audio interface capabilities with the core and are subsequently matched and configured when the application hardware parameters are known.
* ***Dynamic Audio Power Management (DAPM)***. DAPM automatically sets the codec to its minimum power state at all times. This includes powering up/down internal power blocks depending on the internal codec audio routing and any active streams.
* ***Pop and click reduction***. Pops and clicks can be reduced by powering the codec up/down in the correct sequence (including using digital mute). ASoC signals the codec when to change power states.
* ***Machine specific controls***: Allow machines to add controls to the sound card (e.g. volume control for speaker amplifier).

To achieve all this, ASoC basically splits an embedded audio system into multiple re-usable component drivers:

* ***Codec class drivers***: The codec class driver is platform independent and contains audio controls, audio interface capabilities, codec DAPM definition and codec IO functions. Codec class drivers should be generic code that can run on any architecture and machine.
* ***Platform class drivers***: The platform class driver includes the audio DMA engine driver, digital audio interface (DAI) drivers (e.g. I2S, AC97, PCM) and any audio DSP drivers for that platform.
* ***Machine class driver***: The machine driver class acts as the glue that describes and binds the other component drivers together to form an ALSA “sound card device”. It handles any machine specific

ASoC Core

Machine driver

CPU DAI

Codec DAI

Codec driver

Platform driver

*ALSA driver architecture*

1. **ASoC layer**
2. **ASoC Codec class driver**

The codec class driver is generic and hardware independent code that configures the codec, FM, MODEM, BT or external DSP to provide audio capture and playback. It should contain no code that is specific to the target platform or machine. All platform and machine specific code should be added to the platform and machine drivers respectively. Each codec class driver must provide the following features:

* + Codec Digital Audio Interface (DAI) and PCM configuration: use struct *snd\_soc\_dai\_driver* to define DAI and PCM capabilities and operations
  + Codec control IO - using RegMap API: see *include/linux/regmap.h*
  + Mixers and audio controls: All the codec mixers and audio controls defined in *soc.h*
  + Codec audio operations
  + DAPM description: describes the codec power components and their relationships and registers to the ASoC core
  + DAPM event handler: put the codec to sleep when not in use

Optionally, codec drivers can also provide:

* + DAC Digital mute control: Most codecs have a digital mute before the DACs that can be used to minimise any system noise

1. **ASoC Platform driver**

An ASoC platform driver class can be divided into audio DMA drivers, SoC DAI drivers and DSP drivers. The platform drivers only target the SoC CPU and must have no board specific code.

* + Audio DMA drivers: support operations: startup, shutdown, free, prepare, trigger
  + SoC Digital Audio Interface (DAI) drivers: Each SoC DAI driver must provide the following features:
* Digital audio interface (DAI) description
* Digital audio interface configuration
* PCM’s description
* SYSCLK configuration
* Suspend and resume (optional)
  + DSP drivers: Each SoC DAI driver must provide the following features:
  + DAPM graph
  + Mixer controls
  + DMA IO to/from DSP buffers (if applicable)
  + Definition of DSP front end (FE) PCM devices

1. **ASoC Machine driver**

The ASoC machine (or board) driver is the code that glues together all the component drivers (e.g. codecs, platforms and DAIs). It also describes the relationships between each component which include audio paths, GPIOs, interrupts, clocking, jacks and voltage regulators.

1. **ASoC Digital Audio Interface**

ASoC supports the three main Digital Audio Interfaces (DAI): AC97, I2S and PCM

* + **AC97**: AC97 has 5 wires: SDATA\_IN, SDATA\_OUT, BCLK (bit clock). AC97 commonly found on many PC sound cards and portable devices. Each AC97 frame is 21uS long and is divided into 13 time slots
  + **I2S**: I2S has 4 wires: SDATA\_IN (data in), SDATA\_OUT (data out), LRCLK (left-right clock), BCLK (bit clock).

Bit clock usually varies depending on the sample rate and the master system clock (SYSCLK). LRCLK is the same as the sample rate.

I2S has several different operating modes:

* + *I2S*: MSB is transmitted on the falling edge of the first BCLK after LRC transition.
  + *Left Justified*: MSB is transmitted on transition of LRCLK.
  + *Right Justified*: MSB is transmitted sample size BCLKs before LRCLK transition.
  + **PCM**: PCM is a 4 wire interface, similar to I2S, which can support a more flexible protocol. It has BCLK (bit clock), SYNC (sync), SDATA\_IN (data in), SDATA\_OUT (data out). BCLK usually varies depending on sample rate while SYNC runs at the sample rate.

Common PCM operating modes:

* + *Mode A*: MSB is transmitted on falling edge of first BCLK after FRAME/SYNC.
  + *Mode B*: MSB is transmitted on rising edge of FRAME/SYNC

1. **Dynamic Audio Power Management (DAPM) for portable devices**

Dynamic Audio Power Management (DAPM) is designed to allow portable Linux devices to use the minimum amount of power within the audio subsystem at all times.

DAPM is also completely transparent to all user space applications as all power switching is done within the ASoC core. No code changes or recompiling are required for user space applications. DAPM makes power switching decisions based upon any audio stream (capture/playback) activity and audio mixer settings within the device. There are 4 power domains within DAPM:

* + Codec bias domain: Usually controlled at codec probe/remove and suspend/resume
  + Platform/Machine domain: physically connected inputs and output
  + Path domain: audio subsystem signal paths. Automatically set when mixer and mux settings are changed by the user
  + Stream domain: DACs and ADCs.

All audio components that effect power are called widgets, it can fall into a number of types: Mixer, Mux, ADC, DAC, Switch, Input, Output, Headphone, Speaker, Mic, … Widgets are defined in *include/sound/soc-dapm.h*.

1. **Audio Pops and clicks**

Pops and clicks are unwanted audio artifacts caused by the powering up and down of components within the audio subsystem. In PC, when an audio module is either loaded or unloaded. In portable system with DAPM, when power state is changed depending on the audio usage, it can cause pops and clicks.

* + Minimising Playback Pops and Clicks: Playback pops in portable audio subsystems cannot be completely eliminated currently, Pops can be reduced within playback by powering the audio components in a specific order
  + Minimising Capture Pops and Clicks: delay activating the ADC until all the pops have occurred. This follows similar power rules to playback.
  + Zipper noise: can occur within the audio playback or capture stream when a volume control is changed near its maximum gain value. The zipper noise is heard when the gain increase or decrease changes the mean audio signal amplitude too quickly. It can be minimised by enabling the zero cross setting for each volume control (**changed the gain to occur when the signal crosses the zero amplitude line**)

1. **Audio clocking**
   * Master clock

This audio master clock can be derived from a number of sources (e.g. crystal, PLL, CPU clock) and is responsible for producing the correct audio playback and capture sample rates.

* + DAI clocks

This clock is usually driven by a BCLK (bit clock) and used to drive the digital audio data across the link between the codec and CPU.

The DAI also has a frame clock to signal the start of each audio frame. This clock is sometimes referred to as LRCLK (left right clock). The relationship between BCLK and LRCLK depends on the codec or SoC CPU in particular. In general it is best to configure BCLK to the lowest possible speed (depending on your rate, number of channels and word size) to save on power.

It is also desirable to use the codec (if possible) to drive (or master) the audio clocks as it usually gives more accurate sample rates than the CPU.

1. **ASoC jack detection (include/sound/jack.h)**

ASoC provides a version of this API adding two additional features:

* + It allows more than one jack detection method to work together on one user visible jack. In embedded systems it is common for multiple to be present on a single jack but handled by separate bits of hardware.
  + Integration with DAPM, allowing DAPM endpoints to be updated automatically based on the detected jack status (eg, turning off the headphone outputs if no headphones are present).

This is done by splitting the jacks up into three things working together:

* + The jack - struct *snd\_soc\_jack*: represents a physical jack on the system
  + *snd\_soc\_jack\_pin*: These represent a DAPM pin to update depending on some of the status bits supported by the jack.
  + Jack detection methods: Blocks of code which is able to monitor some input to the system and update a jack by calling *snd\_soc\_jack\_report()*. Often this is done based on the status of a GPIO - a handler for this is provided by the *snd\_soc\_jack\_add\_gpio()* function

These are all hooked together by the machine driver depending on the system hardware. The machine driver will set up the snd\_soc\_jack and the list of pins to update then set up one or more jack detection mechanisms to update that jack based on their current status.

1. **Dynamic PCM**

Dynamic PCM allows an ALSA PCM device to digitally route its PCM audio to various digital endpoints during the PCM stream runtime. e.g. PCM0 can route digital audio to I2S DAI0, I2S DAI1 or PDM DAI2. This is useful for on SoC DSP drivers that expose several ALSA PCMs and can route to multiple DAIs.