Audio with embedded Linux training

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Latest update: November 05, 2025.

Document updates and training details:
https://bootlin.com/training/audio

Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com

Audio with embedded Linux training

- These slides are the training materials for Bootlin's *Audio with embedded Linux* training course.
- If you are interested in following this course with an experienced Bootlin trainer, we offer:
  - **Public online sessions**, opened to individual registration. Dates announced on our site, registration directly online.
  - **Dedicated online sessions**, organized for a team of engineers from the same company at a date/time chosen by our customer.
  - **Dedicated on-site sessions**, organized for a team of engineers from the same company, we send a Bootlin trainer on-site to deliver the training.
- Details and registrations: https://bootlin.com/training/audio
- Contact: training@bootlin.com

Icon by Eucalyp, Flaticon

About Bootlin

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Bootlin introduction

- Engineering company
  - In business since 2004
  - Before 2018: *Free Electrons*
- Team based in France and Italy
- Serving **customers worldwide**
- **Highly focused and recognized expertise**
  - Embedded Linux
  - Linux kernel
  - Embedded Linux build systems
- **Strong open-source** contributor
- Activities
  - **Engineering** services
  - **Training** courses
- https://bootlin.com

Bootlin engineering services

Bootloader / firmware development

U-Boot, Barebox, OP-TEE, TF-A, .../

Linux kernel porting and driver development

Linux BSP development, maintenance and upgrade

Embedded Linux build systems

Yocto, OpenEmbedded, Buildroot, ...

Embedded Linux integration

Boot time, real-time, security, multimedia, networking

Open-source upstreaming

Get code integrated in upstream Linux, U-Boot, Yocto, Buildroot, ...

Bootlin training courses

Embedded Linux system development On-site: 4 or 5 days Online: 7 * 4 hours	Linux kernel driver development On-site: 5 days Online: 7 * 4 hours	Yocto Project system development On-site: 3 days Online: 4 * 4 hours	Buildroot system development On-site: 3 days Online: 5 * 4 hours	Embedded Linux networking On-site: 3 days Online: 4 * 4 hours
Understanding the Linux graphics stack On-site: 2 days Online: 4 * 4 hours	Embedded Linux audio On-site: 2 days Online: 4 * 4 hours	Real-Time Linux with PREEMPT_RT On-site: 2 days Online: 3 * 4 hours	Linux debugging, tracing, profiling and performance analysis On-site: 3 days Online: 4 * 4 hours	

All our training materials are freely available
under a free documentation license (CC-BY-SA 3.0)
See https://bootlin.com/training/

Bootlin, an open-source contributor

- Strong contributor to the **Linux** kernel
  - In the top 30 of companies contributing to Linux worldwide
  - Contributions in most areas related to hardware support
  - Several engineers maintainers of subsystems/platforms
  - 9000 patches contributed
  - https://bootlin.com/community/contributions/kernel-contributions/
- Contributor to **Yocto Project**
  - Maintainer of the official documentation
  - Core participant to the QA effort
- Contributor to **Buildroot**
  - Co-maintainer
  - 6000 patches contributed
- Significant contributions to U-Boot, OP-TEE, Barebox, etc.
- Fully **open-source training materials**

Bootlin on-line resources

- Website with a technical blog:
  https://bootlin.com
- Engineering services:
  https://bootlin.com/engineering
- Training services:
  https://bootlin.com/training
- LinkedIn:
  https://www.linkedin.com/company/bootlin
- Elixir - browse Linux kernel sources on-line:
  https://elixir.bootlin.com

Icon by Freepik, Flaticon

Training quiz and certificate

- You have been given a quiz to test your knowledge on the topics covered by the course. That's not too late to take it if you haven't done it yet!
- At the end of the course, we will submit this quiz to you again. That time, you will see the correct answers.
- It allows Bootlin to assess your progress thanks to the course. That's also a kind of challenge, to look for clues throughout the lectures and labs / demos, as all the answers are in the course!
- Another reason is that we only give training certificates to people who achieve at least a 50% score in the final quiz **and** who attended all the sessions.

Participate!

During the lectures...

- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- Don't hesitate to share your experience too, for example to compare Linux with other operating systems you know.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- In on-line sessions
  - Please always keep your camera on!
  - Also make sure your name is properly filled.
  - You can also use the "Raise your hand" button when you wish to ask a question but don't want to interrupt.
- All this helps the trainer to engage with participants, see when something needs clarifying and make the session more interactive, enjoyable and useful for everyone.

Collaborate!

As in the Free Software and Open Source community, collaboration between participants is valuable in this training session:

- Use the dedicated Matrix channel for this session to add questions.
- If your session offers practical labs, you can also report issues, share screenshots and command output there.
- Don't hesitate to share your own answers and to help others especially when the trainer is unavailable.
- The Matrix channel is also a good place to ask questions outside of training hours, and after the course is over.

Sound and its representation

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

What is sound?

- Sound is caused by vibrations
- Vibrations create waves, travelling through a medium
- Humans perceive acoustic waves with their ears, as eardrum are vibrating, converting the signal for the brain
- It is usually represented as a sine wave, however, it is a longitudinal wave (compression/rarefaction) in air and water and a transversal wave in solids.

Sound characteristics

- Sound waves have a frequency, measured in Hertz (Hz), this is the pitch of the sound.
- They also have an amplitude, measured in decibels (dB), this is the loudness of the sound.
- Multiple waves of different frequencies and amplitude combine to create the actual sound with different qualities and timbre.

Sound digitization - samples

- Sound waves are continuous curves composed of a infinite number of points.
- For any point on the curve, it is possible to measure the audio level of this point.
- This is a sample. We can then take samples at regular interval to have a digital representation of the curve.

Sound digitization - sample rate

- The sample rate, or sampling frequency is the number of samples taken per seconds.
- If the sampling frequency is too slow, we may have aliasing issues were the sampled signal doesn't match the analog signal.
- The **Shannon-Nyquist theorem** states that the sampling frequency needs to be at least twice the maximum signal frequency to accurately digitize a signal.
- The Human ear can hear sound frequencies between approximately 20 Hz and 20 kHz.

Aliasing example, the sampled signal is in blue

Sound digitization - sample size

- The sample value varies from 0 to the maximum amplitude value.
- If the amplitude is 1.0, then it varies from -1.0 and 1.0
- The sample size, in bits, then defines the resolution.
- Common sample sizes are 16 and 24 bits.
- 8 bits is getting very rare due to the poor audio quality and 32 bits samples can be used when specific alignment is required.

Sound digitization - sample format

There are multiple ways to store samples in memory or on disk:

- as signed integers
- as unsigned integers
- as floating points

Also, they can be stored in little-endian or big-endian order. For 24bit samples, packing can also differ: either they are packed on 3 bytes or they can be packed in a 32bit integer with the most significant byte being ignored.

Sound digitization - conclusions

- We can then store sound as a sequence of samples and the specific sample rate that was used.
- This method is called Linear Pulse-code modulation or LPCM.
- A sampling rate of about 40kHz is needed.

Sound digitization - example WAV

WAV is a format based on RIFF and has the following header:

Position	Value	Description
1 - 4	“RIFF”	RIFF FOURCC code
5 - 8		File size in bytes, minus 8 (32-bit integer).
9 -12	“WAVE”	WAVE FOURCC code
13-16	“fmt ”	Format chunk marker (includes trailing space)
17-20	16	Length of format data, 16 for PCM
21-22	1	Audio format, 1 for PCM
23-24	2	Number of channels
25-28	48000	Sample rate
29-32	176400	Byte rate = (Sample rate * BitsPerSample * channels) / 8.
33-34	4	BlockAlign = (BitsPerSample * Channels) / 8
35-36	16	Bits per sample
37-40	“data”	Data chunk header
41-44		Size of the data section in bytes

Sound digitization - example WAV

```
0000000  52 49 46 46  24 EE 02 00  57 41 56 45  66 6D 74 20   RIFF$...WAVEfmt.
0000010  10 00 00 00  01 00 02 00  80 BB 00 00  00 EE 02 00   ................
0000020  04 00 10 00  64 61 74 61  00 EE 02 00  FF FF 00 00   ....data........
0000030  F5 02 00 00  E0 05 01 00  D0 08 FF FF  AE 0B 00 00   ................
0000040  8E 0E 01 00  55 11 FE FF  19 14 04 00  C1 16 FC FF   ....U...........
0000050  62 19 03 00  E0 1B FD FF  50 1E 03 00  A2 20 FE FF   b.......P.......
```

Embedded audio Hardware

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Anatomy

Example of an embedded system sound card

CODECs

CODECs

- A CODEC is a device that COdes and DECodes audio samples.
- It integrates an analog-to-digital converter (ADC) and a digital-to-analog converter (DAC) into a single chip.
- It converts a voltage signal from an analog input (e.g. microphone) to a sequence of samples or converts a stream of samples to a voltage for an analog output (e.g. speaker driver).
- It also has one or multiple digital audio interfaces (DAI) to transfer samples to or from a microcontroller or microprocessor.
- Usually an extra digital bus is used for configuration

Digital audio interface - signals

The CODEC DAI is a synchronous serial bus. A common PCM interface is represented here:

Digital audio interface - signals

- The PCM DAI uses two clocks: the bit clock and the frame clock.
  - The bit clock is usually referred to as BCK or BCLK
  - The frame clock is often called FCLK/FSCK/FSCLK, LRCK/LRCLK (Left Right clock) or WCLK (word clock). Its rate is the sample rate also called Fs.
  - The relationship between BCK and FSCK is: $bck = fsck * Nchannels * BitDepth$
- It also has one or multiple data lines.

Digital audio interface - Data

- Codecs may have multiple data in or data out lines, one line per channel pair.
- Codecs may also have multiple DAI, one full interface for data in and one for data out.

e.g. AD1937 has:

- 8 DACs in 4 pairs, 4 ADCs in 2 pairs
- clocks for data-in: DBCLK, DLRCLK
- 4 data-in lines (DSDATA[1-4])
- clocks for data-out: ABCLK, ALRCLK
- 2 data-out lines (ASDATA[1-2])

MCLK

- MCLK is the codec clock. It is sometimes referred as the system clock. The IC needs it to be working.
- Some codecs will also require it to be able to use the control interface.
- Can be provided by the SoC when it has suitable clocks or a crystal.
- Some codecs are able to use BCLK or LRCLK as their clock, making MCLK optional.
- Usually the codecs will expect MCLK to be a multiple of BCLK. Usually specified as a multiple of Fs.

SoC Digital Audio Interface

SoC

- The SoC also has a dedicated synchronous serial interface.
- Some are generic serial interfaces others are dedicated to audio formats.
- It has a DMA controller or a peripheral DMA controller (PDC) able to copy samples from memory to the serial interface registers or FIFO.
- It quite often also has dedicated multimedia (audio/video) clocks.
- Examples: Atmel SSC, NXP SSI, NXP SAI, TI McASP
- Some SoCs have a separate SPDIF controller
- Some SoCs (Allwinner A33, Atmel SAMA5D2) have the codec and the amplifier on the SoC itself so the sound card is completely on the SoC.

Digital formats

Digital formats - Left Justified

Digital formats - Right Justified

Digital formats - I2S

Digital formats - DSP A

Digital formats - DSP B

Digital formats - TDM

Digital formats - AC-link

AC97 uses TDM slots. Slot 0 is 16bit wide and is the tag. Then twelve 20bit wide slots are used to transmit data.

Digital formats - PDM

There is another, less common interface, using Pulse Density Modulation. It has two signals per channels, clock and data. Data has only one bit.

Digital formats - S/PDIF or IEC 60958

S/PDIF uses only one wire. Data is encoded using BMC (Biphase Mark Code), also known as differential Manchester encoding. Its clock is then twice the bitrate.

Blocks of 192 frames are transmitted, each frame consisting of two subframes (32bit words). There are three different preambles, one for start of block and channel 0, one for channel 0 and one for channel 1.

Auxiliary devices

Auxiliary devices

- Some devices may be on the analog path of the audio signal.
- They can be amplifiers, potentiometers or multiplexers.
- Some can be controlled and should be exposed as controls of the sound card.

Clocks

Clocks: producer/consumer

- One of the DAI is responsible to generate the bit clock, it is the bit clock producer (previously: master).
- One of the DAI is responsible to generate the frame clock, it is the frame producer.
- Some CODECs have a great set of PLLs and dividers, allowing to get a precise BCLK from many different MCLK rates.
- Quite often, it is better to use the CODEC as producer. However, some SoCs have specialized audio PLLs.

ASoC

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

ASoC

ASoC, ALSA System on Chip: is a Linux kernel subsystem created to provide better ALSA support for system-on-chip and portable audio codecs. It allows to reuse codec drivers across multiple architectures and provides an API to integrate them with the SoC audio interface.

- created for that use case
- designed for codec drivers reuse
- has an API to write codec drivers
- has an API to write SoC interface drivers

ASoC components

- Codec class drivers: define the codec capabilities (audio interface, audio controls, analog inputs and outputs).
- Platform class drivers: defines the SoC audio interface (also referred as CPU DAI), sets up DMA when applicable.
- Codec to platform integration: nowadays, usually done through device tree, previously required writing a machine driver in C.

Note: The codec can be part of another IC (PMIC, Bluetooth or MODEM chips).

simple-audio-card

simple-card

Most sound cards, can now be described using device tree. This is done using a sound node with a `simple-audio-card` compatible string.

- The DT bindings are documented in `Documentation/devicetree/bindings/sound/simple-card.yaml`
- The driver handling it is `sound/soc/generic/simple-card.c`

Since 2017, OF-graph based bindings are available.

- They are documented in `Documentation/devicetree/bindings/sound/audio-graph-card.yaml`
- The driver handling it is `sound/soc/generic/audio-graph-card.c`

Both required a few changes in the SoC DAI drivers to be usable for example to select the audio mode for the SSC on Microchip SoCs or configure properly the i.MX audmux.

simple-card - example 1

Let's say we have an ADAU1372 codec connected to an i.Mx6UL SAI. First, enable the SAI and the codec:

```
&sai2 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_sai2>;
        status = "okay";
};

&i2c1 {
        adau1372: codec@3c {
                #sound-dai-cells = <0>;
                compatible = "adi,adau1372";
                reg = <0x3c>;
                clock-names = "mclk";
                clocks = <&adau1372z_xtal>;
        };
};

/ {
        adau1372z_xtal: adau1372z_xtal {
                compatible = "fixed-clock";
                #clock-cells = <0>;
                clock-frequency = <12288000>;
        };
};
```

simple-card - example 1

Now, describe the sound card:

```
sound {
        compatible = "simple-audio-card";
        simple-audio-card,name = "imx6ul-adau1372";

        simple-audio-card,dai-link@0 {
                format = "i2s";
                bitclock-master = <&adau1372_dai>;
                frame-master = <&adau1372_dai>;

                sai2_dai: cpu {
                        sound-dai = <&sai2>;
                };

                adau1372_dai: codec {
                        sound-dai = <&adau1372>;
                };
        };
};
```

For convenience, the codec is the producer, it generates both BCLK and FSCLK.

simple-card - example 2

The ADAU1372 has actually 4 channels and can do TDM:

```
sound {
        compatible = "simple-audio-card";
        simple-audio-card,name = "imx6ul-adau1372";

        simple-audio-card,dai-link@0 {
                format = "i2s";
                bitclock-master = <&adau1372_dai>;
                frame-master = <&adau1372_dai>;

                sai2_dai: cpu {
                        sound-dai = <&sai2>;
                        dai-tdm-slot-num = <4>;
                        dai-tdm-slot-width = <32>;
                };

                adau1372_dai: codec {
                        sound-dai = <&adau1372>;
                        dai-tdm-slot-num = <4>;
                        dai-tdm-slot-width = <32>;
                };
        };
};
```

simple-card - example 3

However, the ADAU1372 has an hardware issue and doesn't generate the proper BCLK when doing TDM4 with a 32kHz sample rate. The SAI has to be master:

```
sound {
        compatible = "simple-audio-card";
        simple-audio-card,name = "imx6ul-adau1372";

        simple-audio-card,dai-link@0 {
                format = "i2s";
                bitclock-master = <&sai2_dai>;
                frame-master = <&sai2_dai>;

                sai2_dai: cpu {
                        sound-dai = <&sai2>;
                        dai-tdm-slot-num = <4>;
                        dai-tdm-slot-width = <32>;
                };

                adau1372_dai: codec {
                        sound-dai = <&adau1372>;
                        dai-tdm-slot-num = <4>;
                        dai-tdm-slot-width = <32>;
                };
        };
};
```

simple-card - example 3

The result is not what is expected:

```
# aplay test.wav
Playing WAVE 'test.wav' :
Signed 16 bit Little Endian, Rate  32000 Hz, Stereo
aplay: set_params:1403: Unable to install hw params:
[...]
# dmesg
[...]
fsl-sai 202c000.sai: failed to derive required Tx rate: 4096000
fsl-sai 202c000.sai: ASoC: can't set 202c000.sai hw params: -22
# cat /sys/kernel/debug/clk/clk_summary
pll3                          1        1        0   480000000          0     0  50000
   pll3_bypass                1        1        0   480000000          0     0  50000
      pll3_usb_otg            2        3        0   480000000          0     0  50000
         pll3_pfd2_508m       0        0        0   508235294          0     0  50000
            sai2_sel          0        0        0   508235294          0     0  50000
               sai2_pred      0        0        0   127058824          0     0  50000
                  sai2_podf   0        0        0    63529412          0     0  50000
                     sai2     0        0        0    63529412          0     0  50000
```

Indeed, there is no way for the SAI to divide 63529412 to get the proper BCLK!

device tree - clocks

It is possible to reparent clocks using `assigned-clock-parents` and set the clock rate using `assigned-clock-rates`.

```
&sai2 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_sai2>;
        assigned-clocks = <&clks IMX6UL_CLK_SAI2_SEL>, <&clks IMX6UL_CLK_SAI2>;
        assigned-clock-parents = <&clks IMX6UL_CLK_PLL4_AUDIO_DIV>;
        assigned-clock-rates = <196608000>, <24576000>;
        status = "okay";
};
```

Notice that 24.576MHz was selected for the sai input clock as it is not able to divide by 3 to obtain the 4.096MHz BCLK.

simple-card - example 4

There is a possible cost reduction, the SAI is able to output its clock to feed to the codec MCLK instead of the crystal:

```
&sai2 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_sai2>;
        fsl,sai-mclk-direction-output;
        status = "okay";
};

&i2c1 {
        adau1372: codec@3c {
                #sound-dai-cells = <0>;
                compatible = "adi,adau1372";
                reg = <0x3c>;
                clock-names = "mclk";
                clocks = <&clks IMX6UL_CLK_SAI2>;
                assigned-clocks = <&clks IMX6UL_CLK_SAI2_SEL>, <&clks IMX6UL_CLK_SAI2>;
                assigned-clock-parents = <&clks IMX6UL_CLK_PLL4_AUDIO_DIV>;
                assigned-clock-rates = <196608000>, <24576000>;
        };
};
```

This replaces the 12.288MHz crystal by the 24.576 MCLK from the SAI. This works because the codec has a configurable divider for MCLK and can divide by 2. Also the clock parents and rates assignment has moved to the codec because of probing order.

simple-card - routing

It is possible but not mandatory to list the actual audio connections present on the board, this is called routing. The first step is to define the board connectors, in this case two stereo line input jack (Line0 and Line1) and a stereo jack output.

```
        simple-audio-card,widgets =
                "Line", "Line0",
                "Line", "Line1",
                "Headphone", "Headphone Jack",
```

simple-card - routing

Routing audio from the codec to the board connector is then done using `simple-audio-card,routing`

```
simple-audio-card,routing =
        "AIN0", "Line0",
        "AIN1", "Line0",
        "AIN2", "Line1",
        "AIN3", "Line1",
        "Headphone Jack", "HPOUTL",
        "Headphone Jack", "HPOUTR",
```

Look for `SND_SOC_DAPM_OUTPUT` and `SND_SOC_DAPM_INPUT` to know what the codec is providing.

Machine driver

Machine driver

The machine driver registers a `struct snd_soc_card`.

include/sound/soc.h

```
int snd_soc_register_card(struct snd_soc_card *card);
int snd_soc_unregister_card(struct snd_soc_card *card);
int devm_snd_soc_register_card(struct device *dev, struct snd_soc_card *card);
[...]
/* SoC card */
struct snd_soc_card {
        const char *name;
        const char *long_name;
        const char *driver_name;
        struct device *dev;
        struct snd_card *snd_card;
[...]
        /* CPU <--> Codec DAI links  */
        struct snd_soc_dai_link *dai_link;  /* predefined links only */
        int num_links;  /* predefined links only */
        struct list_head dai_link_list; /* all links */
        int num_dai_links;
[...]
};
```

struct snd_soc_dai_link

struct snd_soc_dai_link is used to create the link between the CPU DAI and the codec DAI.

include/sound/soc.h

```
struct snd_soc_dai_link {
        /* config - must be set by machine driver */
        const char *name;                        /* Codec name */
        const char *stream_name;                /* Stream name */

        /*
         * You MAY specify the link's CPU-side device, either by device name,
         * or by DT/OF node, but not both. If this information is omitted,
         * the CPU-side DAI is matched using .cpu_dai_name only, which hence
         * must be globally unique. These fields are currently typically used
         * only for codec to codec links, or systems using device tree.
         */
        /*
         * You MAY specify the DAI name of the CPU DAI. If this information is
         * omitted, the CPU-side DAI is matched using .cpu_name/.cpu_of_node
         * only, which only works well when that device exposes a single DAI.
         */
        struct snd_soc_dai_link_component *cpus;
        unsigned int num_cpus;
```

struct snd_soc_dai_link

```
        /*
         * You MUST specify the link's codec, either by device name, or by
         * DT/OF node, but not both.
         */
        /* You MUST specify the DAI name within the codec */
        struct snd_soc_dai_link_component *codecs;
        unsigned int num_codecs;
[...]
        unsigned int dai_fmt;           /* format to set on init */
[...]
}
```

Example 1

sound/soc/atmel/atmel_wm8904.c

```
SND_SOC_DAILINK_DEFS(pcm,
        DAILINK_COMP_ARRAY(COMP_EMPTY()),
        DAILINK_COMP_ARRAY(COMP_CODEC(NULL, "wm8904-hifi")),
        DAILINK_COMP_ARRAY(COMP_EMPTY()));

static struct snd_soc_dai_link atmel_asoc_wm8904_dailink = {
        .name = "WM8904",
        .stream_name = "WM8904 PCM",
        .dai_fmt = SND_SOC_DAIFMT_I2S
                | SND_SOC_DAIFMT_NB_NF
                | SND_SOC_DAIFMT_CBP_CFP,
        .ops = &atmel_asoc_wm8904_ops,
        SND_SOC_DAILINK_REG(pcm),
};

static struct snd_soc_card atmel_asoc_wm8904_card = {
        .name = "atmel_asoc_wm8904",
        .owner = THIS_MODULE,
        .dai_link = &atmel_asoc_wm8904_dailink,
        .num_links = 1,
        .dapm_widgets = atmel_asoc_wm8904_dapm_widgets,
        .num_dapm_widgets = ARRAY_SIZE(atmel_asoc_wm8904_dapm_widgets),
        .fully_routed = true,
};
```

Example 1

sound/soc/atmel/atmel_wm8904.c

```
static int atmel_asoc_wm8904_dt_init(struct platform_device *pdev)
{
        struct device_node *np = pdev->dev.of_node;
        struct device_node *codec_np, *cpu_np;
        struct snd_soc_card *card = &atmel_asoc_wm8904_card;
        struct snd_soc_dai_link *dailink = &atmel_asoc_wm8904_dailink;
[...]
        cpu_np = of_parse_phandle(np, "atmel,ssc-controller", 0);
        if (!cpu_np) {
                dev_err(&pdev->dev, "failed to get dai and pcm info\n");
                ret = -EINVAL;
                return ret;
        }
        dailink->cpus->of_node = cpu_np;
        dailink->platforms->of_node = cpu_np;
        of_node_put(cpu_np);

        codec_np = of_parse_phandle(np, "atmel,audio-codec", 0);
        if (!codec_np) {
                dev_err(&pdev->dev, "failed to get codec info\n");
                ret = -EINVAL;
                return ret;
        }
        dailink->codecs->of_node = codec_np;
        of_node_put(codec_np);
```

Example 1

sound/soc/atmel/atmel_wm8904.c

```
static int atmel_asoc_wm8904_probe(struct platform_device *pdev)
{
        struct snd_soc_card *card = &atmel_asoc_wm8904_card;
        struct snd_soc_dai_link *dailink = &atmel_asoc_wm8904_dailink;
        int id, ret;

        card->dev = &pdev->dev;
        ret = atmel_asoc_wm8904_dt_init(pdev);
        if (ret) {
                dev_err(&pdev->dev, "failed to init dt info\n");
                return ret;
        }

        id = of_alias_get_id((struct device_node *)dailink->cpus->of_node, "ssc");
        ret = atmel_ssc_set_audio(id);
        if (ret != 0) {
                dev_err(&pdev->dev, "failed to set SSC %d for audio\n", id);
                return ret;
        }

        ret = snd_soc_register_card(card);
        if (ret) {
                dev_err(&pdev->dev, "snd_soc_register_card failed\n");
                goto err_set_audio;
        }
```

Routing

After linking the codec driver with the SoC DAI driver, it is still necessary to define what are the codec outputs and inputs that are actually used on the board. This is called routing.

- statically: using the `.dapm_routes` and `.num_dapm_routes` members of `struct snd_soc_card`
- from device tree:

```
int snd_soc_of_parse_audio_routing(struct snd_soc_card *card,
                                   const char *propname);
```

Routing example: static

sound/soc/rockchip/rockchip_max98090.c

```
static const struct snd_soc_dapm_route rk_audio_map[] = {
        {"IN34", NULL, "Headset Mic"},
        {"IN34", NULL, "MICBIAS"},
        {"Headset Mic", NULL, "MICBIAS"},
        {"DMICL", NULL, "Int Mic"},
        {"Headphone", NULL, "HPL"},
        {"Headphone", NULL, "HPR"},
        {"Speaker", NULL, "SPKL"},
        {"Speaker", NULL, "SPKR"},
};
[...]
static struct snd_soc_card snd_soc_card_rk = {
        .name = "ROCKCHIP-I2S",
        .owner = THIS_MODULE,
        .dai_link = &rk_dailink,
        .num_links = 1,
[...]
        .dapm_widgets = rk_dapm_widgets,
        .num_dapm_widgets = ARRAY_SIZE(rk_dapm_widgets),
        .dapm_routes = rk_audio_map,
        .num_dapm_routes = ARRAY_SIZE(rk_audio_map),
        .controls = rk_mc_controls,
        .num_controls = ARRAY_SIZE(rk_mc_controls),
};
```

Routing example: DT

sound/soc/atmel/atmel_wm8904.c

```
static int atmel_asoc_wm8904_dt_init(struct platform_device *pdev)
{
[...]
        ret = snd_soc_of_parse_card_name(card, "atmel,model");
        if (ret) {
                dev_err(&pdev->dev, "failed to parse card name\n");
                return ret;
        }

        ret = snd_soc_of_parse_audio_routing(card, "atmel,audio-routing");
        if (ret) {
                dev_err(&pdev->dev, "failed to parse audio routing\n");
                return ret;
        }
[...]
}
```

Routing example: DT

Documentation/devicetree/bindings/sound/atmel-wm8904.txt

```
- atmel,audio-routing: A list of the connections between audio components.
  Each entry is a pair of strings, the first being the connection's sink,
  the second being the connection's source. Valid names for sources and
  sinks are the WM8904's pins, and the jacks on the board:

  WM8904 pins:

  * IN1L
  * IN1R
  * IN2L
  * IN2R
  * IN3L
  * IN3R
  * HPOUTL
  * HPOUTR
  * LINEOUTL
  * LINEOUTR
  * MICBIAS

  Board connectors:

  * Headphone Jack
  * Line In Jack
  * Mic
```

Routing example

Documentation/devicetree/bindings/sound/atmel-wm8904.txt

```
Example:
sound {
        compatible = "atmel,asoc-wm8904";
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_pck0_as_mck>;

        atmel,model = "wm8904 @ AT91SAM9N12EK";

        atmel,audio-routing =
                "Headphone Jack", "HPOUTL",
                "Headphone Jack", "HPOUTR",
                "IN2L", "Line In Jack",
                "IN2R", "Line In Jack",
                "Mic", "MICBIAS",
                "IN1L", "Mic";

        atmel,ssc-controller = <&ssc0>;
        atmel,audio-codec = <&wm8904>;
};
```

Routing: codec pins

The available codec pins are defined in the codec driver. Look for the `SND_SOC_DAPM_INPUT` and `SND_SOC_DAPM_OUTPUT` definitions.

sound/soc/codecs/wm8904.c

```
static const struct snd_soc_dapm_widget wm8904_adc_dapm_widgets[] = {
SND_SOC_DAPM_INPUT("IN1L"),
SND_SOC_DAPM_INPUT("IN1R"),
SND_SOC_DAPM_INPUT("IN2L"),
SND_SOC_DAPM_INPUT("IN2R"),
SND_SOC_DAPM_INPUT("IN3L"),
SND_SOC_DAPM_INPUT("IN3R"),
[...]
};

static const struct snd_soc_dapm_widget wm8904_dac_dapm_widgets[] = {
[...]
SND_SOC_DAPM_OUTPUT("HPOUTL"),
SND_SOC_DAPM_OUTPUT("HPOUTR"),
SND_SOC_DAPM_OUTPUT("LINEOUTL"),
SND_SOC_DAPM_OUTPUT("LINEOUTR"),
};
```

Routing: board connectors

The board connectors are defined in the machine driver, in the `struct snd_soc_dapm_widget` part of the registered `struct snd_soc_card`.

sound/soc/atmel/atmel_wm8904.c

```
static const struct snd_soc_dapm_widget atmel_asoc_wm8904_dapm_widgets[] = {
        SND_SOC_DAPM_HP("Headphone Jack", NULL),
        SND_SOC_DAPM_MIC("Mic", NULL),
        SND_SOC_DAPM_LINE("Line In Jack", NULL),
};
```

Clocking: producer/consumer

The producer/consumer relationship is declared part of the `.dai_fmt` field of `struct snd_soc_dai_link`.

include/sound/soc.h

```
/*
 * DAI hardware clock providers/consumers
 *
 * This is wrt the codec, the inverse is true for the interface
 * i.e. if the codec is clk and FRM provider then the interface is
 * clk and frame consumer.
 */
#define SND_SOC_DAIFMT_CBP_CFP                (1 << 12) /* codec clk provider & frame provider */
#define SND_SOC_DAIFMT_CBC_CFP                (2 << 12) /* codec clk consumer & frame provider */
#define SND_SOC_DAIFMT_CBP_CFC                (3 << 12) /* codec clk provider & frame consumer */
#define SND_SOC_DAIFMT_CBC_CFC                (4 << 12) /* codec clk consumer & frame consumer */

/* previous definitions kept for backwards-compatibility, do not use in new contributions */
#define SND_SOC_DAIFMT_CBM_CFM                SND_SOC_DAIFMT_CBP_CFP
#define SND_SOC_DAIFMT_CBS_CFM                SND_SOC_DAIFMT_CBC_CFP
#define SND_SOC_DAIFMT_CBM_CFS                SND_SOC_DAIFMT_CBP_CFC
#define SND_SOC_DAIFMT_CBS_CFS                SND_SOC_DAIFMT_CBC_CFC
```

sound/soc/atmel/atmel_wm8904.c

```
        .dai_fmt = SND_SOC_DAIFMT_I2S
                 | SND_SOC_DAIFMT_NB_NF
                 | SND_SOC_DAIFMT_CBM_CFM,
```

Clocking: dynamically changing clocks

The `.ops` member of `struct snd_soc_dai_link` contains useful callbacks.

include/sound/soc.h

```
/* SoC audio ops */
struct snd_soc_ops {
        int (*startup)(struct snd_pcm_substream *);
        void (*shutdown)(struct snd_pcm_substream *);
        int (*hw_params)(struct snd_pcm_substream *, struct snd_pcm_hw_params *);
        int (*hw_free)(struct snd_pcm_substream *);
        int (*prepare)(struct snd_pcm_substream *);
        int (*trigger)(struct snd_pcm_substream *, int);
};
```

`.hw_params` is called when setting up the audio stream. The `struct snd_pcm_hw_params` contains the audio characteristics. Use `params_rate()` to get the sample rate, `params_channels` for the number of channels and `params_format` to get the format (including the bit depth). Finally, `snd_soc_params_to_bclk` calculates the bit clock.

Clocking: hw_params

- params_rate gets the sample rate
- params_channels gets the number of channels
- params_format gets the format (including the bit depth)
- snd_soc_params_to_bclk calculates the bit clock.
- snd_soc_dai_set_sysclk sets the clock rate and direction for the DAI (SoC or codec)

```
int snd_soc_dai_set_sysclk(struct snd_soc_dai *dai, int clk_id,
        unsigned int freq, int dir);
```

- it is also possible to configure the PLLs and clock divisors if necessary

```
int snd_soc_dai_set_clkdiv(struct snd_soc_dai *dai,
        int div_id, int div);
int snd_soc_dai_set_pll(struct snd_soc_dai *dai,
        int pll_id, int source, unsigned int freq_in, unsigned int freq_out);
```

Clocking example

sound/soc/atmel/atmel_wm8904.c

```
static int atmel_asoc_wm8904_hw_params(struct snd_pcm_substream *substream,
                struct snd_pcm_hw_params *params)
{
        struct snd_soc_pcm_runtime *rtd = substream->private_data;
        struct snd_soc_dai *codec_dai = rtd->codec_dai;
        int ret;

        ret = snd_soc_dai_set_pll(codec_dai, WM8904_FLL_MCLK, WM8904_FLL_MCLK,
                32768, params_rate(params) * 256);
        if (ret < 0) {
                pr_err("%s - failed to set wm8904 codec PLL.", __func__);
                return ret;
        }
```

Clocking example

sound/soc/atmel/atmel_wm8904.c

```
        /*
         * As here wm8904 use FLL output as its system clock
         * so calling set_sysclk won't care freq parameter
         * then we pass 0
         */
        ret = snd_soc_dai_set_sysclk(codec_dai, WM8904_CLK_FLL,
                        0, SND_SOC_CLOCK_IN);
        if (ret < 0) {
                pr_err("%s -failed to set wm8904 SYSCLK\n", __func__);
                return ret;
        }

        return 0;
}

static struct snd_soc_ops atmel_asoc_wm8904_ops = {
        .hw_params = atmel_asoc_wm8904_hw_params,
};
```

CODEC driver

CODEC driver

The CODEC driver registers a `struct snd_soc_component_driver`. Before v4.17, it was `struct snd_soc_codec_driver`. Also registers a `struct snd_soc_dai_driver`

include/sound/soc.h

```
int snd_soc_register_component(struct device *dev,
                 const struct snd_soc_component_driver *component_driver,
                 struct snd_soc_dai_driver *dai_drv, int num_dai);
int devm_snd_soc_register_component(struct device *dev,
                 const struct snd_soc_component_driver *component_driver,
                 struct snd_soc_dai_driver *dai_drv, int num_dai);
```

snd_soc_component_driver

include/sound/soc-component.h

```
struct snd_soc_component_driver {
        const char *name;

        /* Default control and setup, added after probe() is run */
        const struct snd_kcontrol_new *controls;
        unsigned int num_controls;
        const struct snd_soc_dapm_widget *dapm_widgets;
        unsigned int num_dapm_widgets;
        const struct snd_soc_dapm_route *dapm_routes;
        unsigned int num_dapm_routes;

        int (*probe)(struct snd_soc_component *component);
        void (*remove)(struct snd_soc_component *component);
        int (*suspend)(struct snd_soc_component *component);
        int (*resume)(struct snd_soc_component *component);

    [...]
```

snd_soc_component_driver

- `struct snd_kcontrol_new *controls` is an array of controls (volume, mixing, muxing, switches) available on the CODEC.
- `struct snd_soc_dapm_widget *dapm_widgets` is an array of power management controls so ASoC can power down the routes that are not currently used.
- `struct snd_soc_dapm_route *dapm_routes` is an array describing those routes.

snd_soc_component_driver

include/sound/soc-component.h

```
        /* component wide operations */
        int (*set_sysclk)(struct snd_soc_component *component,
                          int clk_id, int source, unsigned int freq, int dir);
        int (*set_pll)(struct snd_soc_component *component, int pll_id,
                       int source, unsigned int freq_in, unsigned int freq_out);
        [...]
        int (*hw_params)(struct snd_soc_component *component,
                         struct snd_pcm_substream *substream,
                         struct snd_pcm_hw_params *params);
        [...]
}
```

snd_soc_component_driver

- `set_sysclk` allows setting the input clock of the component.
- `set_pll` allows setting the PLLs, this is mostly useful when the component is the clock producer.
- `hw_params` is a callback called on PCM stream setup. When called, all the parameters of the stream are known so it is possible to configure the component to handle the stream correctly.
- Those are mostly not used, the DAI specific callbacks are used instead.

snd_soc_dai_driver

include/sound/soc-dai.h

```
/*
 * Digital Audio Interface Driver.
 *
 * Describes the Digital Audio Interface in terms of its ALSA, DAI and AC97
 * operations and capabilities. Codec and platform drivers will register this
 * structure for every DAI they have.
 *
 * This structure covers the clocking, formating and ALSA operations for each
 * interface.
 */
struct snd_soc_dai_driver {
        /* DAI description */
        const char *name;
        [...]

        /* ops */
        const struct snd_soc_dai_ops *ops;
        const struct snd_soc_cdai_ops *cops;

        /* DAI capabilities */
        struct snd_soc_pcm_stream capture;
        struct snd_soc_pcm_stream playback;
        [...]
};
```

snd_soc_pcm_stream

include/sound/soc.h

```
/* SoC PCM stream information */
struct snd_soc_pcm_stream {
        const char *stream_name;
        u64 formats;                    /* SNDRV_PCM_FMTBIT_* */
        unsigned int rates;             /* SNDRV_PCM_RATE_* */
        unsigned int rate_min;          /* min rate */
        unsigned int rate_max;          /* max rate */
        unsigned int channels_min;      /* min channels */
        unsigned int channels_max;      /* max channels */
        unsigned int sig_bits;          /* number of bits of content */
};
```

PCM5102

DIN (i2s)
Audio Interface
8x Interpolation Filter
32bit ΔΣ Modulator
Current Segment DAC
Current Segment DAC
I/V
I/V
Analog Mute
Analog Mute
LINE OUT
Zero Data Detector
Advanced Mute Control
Clock Halt Detection
PCM510x
LRCK
BCK
MCK
PLL Clock
UVP/Reset
POR
Ch. Pump
Power Supply
CPVDD (3.3V)
AVDD (3.3V)
DVDD (3.3V)
GND
VNEG
CAPM
CAPP

pcm5102a.c

sound/soc/codecs/pcm5102a.c

```
static struct snd_soc_dai_driver pcm5102a_dai = {
        .name = "pcm5102a-hifi",
        .playback = {
                .channels_min = 2,
                .channels_max = 2,
                .rates = SNDRV_PCM_RATE_8000_384000,
                .formats = SNDRV_PCM_FMTBIT_S16_LE |
                           SNDRV_PCM_FMTBIT_S24_LE |
                           SNDRV_PCM_FMTBIT_S32_LE
        },
};

static struct snd_soc_component_driver soc_component_dev_pcm5102a = {
        .idle_bias_on                   = 1,
        .use_pmdown_time                = 1,
        .endianness                     = 1,
};

static int pcm5102a_probe(struct platform_device *pdev)
{
        return devm_snd_soc_register_component(&pdev->dev, &soc_component_dev_pcm5102a,
                        &pcm5102a_dai, 1);
}
```

PCM3008

VINL
VCOM
VINR
VOUTL
VOUTR
Analog Front-End Circuit
Reference
Analog Front-End Circuit
+
–
+
–
Delta-Sigma Modulator
Delta-Sigma Modulator
Decimation and High Pass Filter
Decimation and High Pass Filter
ADC
Analog Low-Pass Filter
Analog Low-Pass Filter
Delta-Sigma Modulator
Delta-Sigma Modulator
Interpolation Filter 8× Oversampling
Interpolation Filter 8× Oversampling
DAC
Serial Interface
De-Emphasis Control
Power Down Control
Power Supply
Clock
LRCK
BCK
DIN
DOUT
DEM0
DEM1
PDDA
PDAD

pcm3008.c

sound/soc/codecs/pcm3008.c

```
#define PCM3008_RATES (SNDRV_PCM_RATE_32000 | SNDRV_PCM_RATE_44100 |        \
                       SNDRV_PCM_RATE_48000)

static struct snd_soc_dai_driver pcm3008_dai = {
        .name = "pcm3008-hifi",
        .playback = {
                .stream_name = "PCM3008 Playback",
                .channels_min = 1,
                .channels_max = 2,
                .rates = PCM3008_RATES,
                .formats = SNDRV_PCM_FMTBIT_S16_LE,
        },
        .capture = {
                .stream_name = "PCM3008 Capture",
                .channels_min = 1,
                .channels_max = 2,
                .rates = PCM3008_RATES,
                .formats = SNDRV_PCM_FMTBIT_S16_LE,
        },
};
```

pcm3008.c

sound/soc/codecs/pcm3008.c

```
static const struct snd_soc_component_driver soc_component_dev_pcm3008 = {
        .dapm_widgets           = pcm3008_dapm_widgets,
        .num_dapm_widgets       = ARRAY_SIZE(pcm3008_dapm_widgets),
        .dapm_routes            = pcm3008_dapm_routes,
        .num_dapm_routes        = ARRAY_SIZE(pcm3008_dapm_routes),
        .idle_bias_on           = 1,
        .use_pmdown_time        = 1,
        .endianness             = 1,
};

static int pcm3008_codec_probe(struct platform_device *pdev)
{
        [...]

        return devm_snd_soc_register_component(&pdev->dev,
                        &soc_component_dev_pcm3008, &pcm3008_dai, 1);
}
```

pcm3008.c

sound/soc/codecs/pcm3008.c

```
static const struct snd_soc_dapm_widget pcm3008_dapm_widgets[] = {
SND_SOC_DAPM_INPUT("VINL"),
SND_SOC_DAPM_INPUT("VINR"),

SND_SOC_DAPM_DAC_E("DAC", NULL, SND_SOC_NOPM, 0, 0, pcm3008_dac_ev,
                   SND_SOC_DAPM_PRE_PMU | SND_SOC_DAPM_POST_PMD),
SND_SOC_DAPM_ADC_E("ADC", NULL, SND_SOC_NOPM, 0, 0, pcm3008_adc_ev,
                   SND_SOC_DAPM_PRE_PMU | SND_SOC_DAPM_POST_PMD),

SND_SOC_DAPM_OUTPUT("VOUTL"),
SND_SOC_DAPM_OUTPUT("VOUTR"),
};

static const struct snd_soc_dapm_route pcm3008_dapm_routes[] = {
        { "PCM3008 Capture", NULL, "ADC" },
        { "ADC", NULL, "VINL" },
        { "ADC", NULL, "VINR" },

        { "DAC", NULL, "PCM3008 Playback" },
        { "VOUTL", NULL, "DAC" },
        { "VOUTR", NULL, "DAC" },
};
```

ASoC component controls

snd_soc_component_driver

- Controls allow to export configuration knobs of the component to userspace.
- ASoC provides many helpers to define them instead of filling `struct snd_kcontrol_new`

snd_kcontrol_new

include/sound/control.h

```
struct snd_kcontrol_new {
        snd_ctl_elem_iface_t iface;      /* interface identifier */
        unsigned int device;             /* device/client number */
        unsigned int subdevice;          /* subdevice (substream) number */
        const char *name;                /* ASCII name of item */
        unsigned int index;              /* index of item */
        unsigned int access;             /* access rights */
        unsigned int count;              /* count of same elements */
        snd_kcontrol_info_t *info;
        snd_kcontrol_get_t *get;
        snd_kcontrol_put_t *put;
        union {
                snd_kcontrol_tlv_rw_t *c;
                const unsigned int *p;
        } tlv;
        unsigned long private_value;
};
```

kcontrol helpers

include/sound/soc.h

```
#define SOC_SINGLE(xname, reg, shift, max, invert) \
{       .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname, \
        .info = snd_soc_info_volsw, .get = snd_soc_get_volsw,\
        .put = snd_soc_put_volsw, \
        .private_value = SOC_SINGLE_VALUE(reg, shift, max, invert, 0) }
#define SOC_SINGLE_RANGE(xname, xreg, xshift, xmin, xmax, xinvert) \
{       .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = (xname),\
        .info = snd_soc_info_volsw_range, .get = snd_soc_get_volsw_range, \
        .put = snd_soc_put_volsw_range, \
        .private_value = (unsigned long)&(struct soc_mixer_control) \
                {.reg = xreg, .rreg = xreg, .shift = xshift, \
                 .rshift = xshift,  .min = xmin, .max = xmax, \
                 .invert = xinvert} }
#define SOC_SINGLE_TLV(xname, reg, shift, max, invert, tlv_array) \
{       .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname, \
        .access = SNDRV_CTL_ELEM_ACCESS_TLV_READ |\
                SNDRV_CTL_ELEM_ACCESS_READWRITE,\
        .tlv.p = (tlv_array), \
        .info = snd_soc_info_volsw, .get = snd_soc_get_volsw,\
        .put = snd_soc_put_volsw, \
        .private_value = SOC_SINGLE_VALUE(reg, shift, max, invert, 0) }
```

kcontrol examples

sound/soc/codecs/pcm3168a.h

```
#define PCM3168A_DAC_PWR_MST_FMT                0x41
#define PCM3168A_DAC_PSMDA_SHIFT                7
```

sound/soc/codecs/pcm3168a.c

```
        SOC_SINGLE("DAC Power-Save Switch", PCM3168A_DAC_PWR_MST_FMT,
                   PCM3168A_DAC_PSMDA_SHIFT, 1, 1),
```

- This exposes a simple on/off switch named "DAC Power-Save Switch" for bit 7 in register 0x41.

kcontrol examples

sound/soc/codecs/pcm3168a.h

```
#define PCM3168A_ADC_MUTE                        0x55
```

sound/soc/codecs/pcm3168a.c

```
        SOC_DOUBLE("ADC1 Mute Switch", PCM3168A_ADC_MUTE, 0, 1, 1, 0),
```

- This exposes a Left/Right switch named "ADC1 Mute Switch" for bit 0 (left) and 1 (right) in register 0x55.

kcontrol examples

sound/soc/codecs/sgtl5000.h

```
#define SGTL5000_DAP_MAIN_CHAN            0x0120
```

sound/soc/codecs/sgtl5000.c

```
/* tlv for DAP channels, 0% - 100% - 200% */
static const DECLARE_TLV_DB_SCALE(dap_volume, 0, 1, 0);
[...]
        SOC_SINGLE_TLV("DAP Main channel", SGTL5000_DAP_MAIN_CHAN,
        0, 0xffff, 0, dap_volume),
```

- This a single volume control named "DAP Main channel". It is controlled by register 0x120 and can take values up to 0xffff.

include/uapi/sound/tlv.h

```
#define SNDRV_CTL_TLVD_DECLARE_DB_SCALE(name, min, step, mute) \
```

kcontrol examples

sound/soc/codecs/pcm3168a.h

```
#define PCM3168A_DAC_VOL_MASTER                    0x47
```

sound/soc/codecs/pcm3168a.c

```
/* -100db to 0db, register values 0-54 cause mute */
static const DECLARE_TLV_DB_SCALE(pcm3168a_dac_tlv, -10050, 50, 1);
[...]
        SOC_SINGLE_RANGE_TLV("Master Playback Volume",
                        PCM3168A_DAC_VOL_MASTER, 0, 54, 255, 0,
                        pcm3168a_dac_tlv),
```

- This a single volume control named "Master Playback Volume". It is controlled by register 0x47 and can take values 54 to 255. The datasheet states that 0 to 54 is mute.

kcontrol examples

sound/soc/codecs/pcm3168a.h

```
#define PCM3168A_DAC_VOL_CHAN_START                0x48
```

sound/soc/codecs/pcm3168a.c

```
/* -100db to 0db, register values 0-54 cause mute */
static const DECLARE_TLV_DB_SCALE(pcm3168a_dac_tlv, -10050, 50, 1);
[...]
        SOC_DOUBLE_R_RANGE_TLV("DAC1 Playback Volume",
                        PCM3168A_DAC_VOL_CHAN_START,
                        PCM3168A_DAC_VOL_CHAN_START + 1,
                        0, 54, 255, 0, pcm3168a_dac_tlv),
```

- This a Left/Right volume control named "DAC1 Playback Volume". Left is controlled by register 0x48, right channel is in register 0x49 and both can take values 54 to 255. The datasheet states that 0 to 54 is mute.

kcontrol examples

sound/soc/codecs/pcm3168a.h

```
#define PCM3168A_DAC_ATT_DEMP_ZF                0x46
#define PCM3168A_DAC_DEMP_SHIFT                 4
```

sound/soc/codecs/pcm3168a.c

```
static const char *const pcm3168a_demp[] = {
                "Disabled", "48khz", "44.1khz", "32khz" };

static SOC_ENUM_SINGLE_DECL(pcm3168a_dac_demp, PCM3168A_DAC_ATT_DEMP_ZF,
                PCM3168A_DAC_DEMP_SHIFT, pcm3168a_demp);
[...]
        SOC_ENUM("DAC De-Emphasis", pcm3168a_dac_demp),
```

- This creates a control named "DAC De-Emphasis". Allowing to choose between four different values. This is controlled in register 0x46, bits 4 and 5.

kcontrol examples

sound/soc/codecs/sgtl5000.h

```
#define SGTL5000_DAP_AVC_THRESHOLD                    0x0126
```

sound/soc/codecs/sgtl5000.c

```
static int avc_get_threshold(struct snd_kcontrol *kcontrol,
                             struct snd_ctl_elem_value *ucontrol)

[...]
static const DECLARE_TLV_DB_MINMAX(avc_threshold, 0, 9600);
[...]
        SOC_SINGLE_EXT_TLV("AVC Threshold Volume", SGTL5000_DAP_AVC_THRESHOLD,
                           0, 96, 0, avc_get_threshold, avc_put_threshold,
                           avc_threshold),
```

- This a single volume control named "AVC Threshold Volume".

kcontrol names

- Naming actually matters, userspace tools use them to populate the user interface properly!
- Controls named similarly will be grouped together:
  - "Playback" and "Capture" controls may be exposed separately in the UI
  - "Mute Switch" and "Volume" for a similarly named controls can be shown as a single control

- Master Playback Switch
- Master Playback Volume
- Headphone Mic Boost Volume
- Capture Volume

regmap

regmap

- has its roots in ASoC (ALSA)
- can use I2C, SPI and MMIO (also SPMI)
- actually abstracts the underlying bus
- can handle locking when necessary
- can cache registers
- can handle endianness conversion
- can handle IRQ chips and IRQs
- can check register ranges
- handles read only, write only, volatile, precious registers
- handles register pages
- API is defined in `include/linux/regmap.h`
- implemented in `drivers/base/regmap/`

regmap: creation

```
#define regmap_init(dev, bus, bus_context, config)                \
        __regmap_lockdep_wrapper(__regmap_init, #config,          \
                                 dev, bus, bus_context, config)
```

```
#define regmap_init_i2c(i2c, config)                              \
        __regmap_lockdep_wrapper(__regmap_init_i2c, #config,      \
                                 i2c, config)
```

```
#define regmap_init_spi(dev, config)                              \
        __regmap_lockdep_wrapper(__regmap_init_spi, #config,      \
                                 dev, config)
```

- Also `devm_` versions
- and `_clk` versions, preparing, enabling and disabling clocks when necessary

regmap: config

include/linux/regmap.h

```
struct regmap_config {
[...]
        int reg_bits;
        int reg_stride;
[...]
        bool (*writeable_reg)(struct device *dev, unsigned int reg);
        bool (*readable_reg)(struct device *dev, unsigned int reg);
        bool (*volatile_reg)(struct device *dev, unsigned int reg);
        bool (*precious_reg)(struct device *dev, unsigned int reg);
[...]
        int (*reg_read)(void *context, unsigned int reg, unsigned int *val);
        int (*reg_write)(void *context, unsigned int reg, unsigned int val);
        int (*reg_update_bits)(void *context, unsigned int reg,
                               unsigned int mask, unsigned int val);

[...]
        const struct reg_default *reg_defaults;
        unsigned int num_reg_defaults;
[...]
};
```

regmap: config

- `reg_bits` Number of bits in a register address, mandatory.
- `reg_stride` The register address stride. Valid register addresses are a multiple of this value. If set to 0, a value of 1 will be used.
- `writeable_reg`, `readable_reg`, `volatile_reg`, `precious_reg`: Optional callbacks returning true if the register is writeable, readable, volatile or precious. volatile registers won't be cached. precious registers will not be read unless the driver explicitly calls a read function. There are also tables in the `struct regmap_config` for the same purpose.
- `reg_read`, `reg_write`, `reg_update_bits`: Optional callbacks that if filled will be used to perform accesses. `reg_update_bits` should only be provided if specific locking is required.
- `reg_defaults`: Power on reset values for registers (for use with register cache support).
- `num_reg_defaults`: Number of elements in `reg_defaults`.

regmap: access

- ```
  int regmap_read(struct regmap *map, unsigned int reg, unsigned int *val);
  ```
- ```
  int regmap_write(struct regmap *map, unsigned int reg, unsigned int val);
  ```
- ```
  static inline int regmap_update_bits(struct regmap *map, unsigned int reg,
                                       unsigned int mask, unsigned int val)
  ```
- ```
  #define regmap_read_poll_timeout(map, addr, val, cond, sleep_us, timeout_us)
  ```
- ```
  int regmap_test_bits(struct regmap *map, unsigned int reg, unsigned int bits);
  ```
- ```
  static inline int regmap_update_bits_check(struct regmap *map, unsigned int reg,
                                             unsigned int mask, unsigned int val,
                                             bool *change)
  ```

regmap: cache management

- ```
  int regcache_sync(struct regmap *map);
  ```
- ```
  int regcache_sync_region(struct regmap *map, unsigned int min,
                           unsigned int max);
  ```
- ```
  int regcache_drop_region(struct regmap *map, unsigned int min,
                           unsigned int max);
  ```
- ```
  void regcache_cache_only(struct regmap *map, bool enable);
  ```
- ```
  void regcache_cache_bypass(struct regmap *map, bool enable);
  ```
- ```
  void regcache_mark_dirty(struct regmap *map);
  ```

regmap: example

sound/soc/codecs/max9877.c

```
static const struct regmap_config max9877_regmap = {
        .reg_bits = 8,
        .val_bits = 8,

        .reg_defaults = max9877_regs,
        .num_reg_defaults = ARRAY_SIZE(max9877_regs),
        .cache_type = REGCACHE_RBTREE,
};

static int max9877_i2c_probe(struct i2c_client *client)
{
        struct regmap *regmap;
        int i;

        regmap = devm_regmap_init_i2c(client, &max9877_regmap);
        if (IS_ERR(regmap))
                return PTR_ERR(regmap);

        /* Ensure the device is in reset state */
        for (i = 0; i < ARRAY_SIZE(max9877_regs); i++)
                regmap_write(regmap, max9877_regs[i].reg, max9877_regs[i].def);

        return devm_snd_soc_register_component(&client->dev,
                        &max9877_component_driver, NULL, 0);
}
```

regmap: i2c and spi device example

sound/soc/codecs/adau1372.c

```
const struct regmap_config adau1372_regmap_config = {
        .val_bits = 8,
        .reg_bits = 16,
        .max_register = 0x4d,

        .reg_defaults = adau1372_reg_defaults,
        .num_reg_defaults = ARRAY_SIZE(adau1372_reg_defaults),
        .volatile_reg = adau1372_volatile_register,
        .cache_type = REGCACHE_RBTREE,
};
EXPORT_SYMBOL_GPL(adau1372_regmap_config);
```

sound/soc/codecs/adau1372-i2c.c

```
static int adau1372_i2c_probe(struct i2c_client *client)
{
        return adau1372_probe(&client->dev,
                devm_regmap_init_i2c(client, &adau1372_regmap_config), NULL);
}
```

regmap: i2c and spi device example

sound/soc/codecs/adau1372-spi.c

```
static int adau1372_spi_probe(struct spi_device *spi)
{
        struct regmap_config config;

        config = adau1372_regmap_config;
        config.read_flag_mask = 0x1;

        return adau1372_probe(&spi->dev,
                devm_regmap_init_spi(spi, &config), adau1372_spi_switch_mode);
}
```

regmap: ASoC components

- snd_soc_component regmap accessors also exist, they are available either implicitly as the component core calls dev_get_regmap(component->dev, NULL) to retrieve or create a regmap for the device or explicitly by calling snd_soc_component_init_regmap()

include/sound/soc-component.h

```
/* component IO */
unsigned int snd_soc_component_read(struct snd_soc_component *component,
                                    unsigned int reg);
int snd_soc_component_write(struct snd_soc_component *component,
                            unsigned int reg, unsigned int val);
int snd_soc_component_update_bits(struct snd_soc_component *component,
                                  unsigned int reg, unsigned int mask,
                                  unsigned int val);
[...]
int snd_soc_component_test_bits(struct snd_soc_component *component,
                                unsigned int reg, unsigned int mask,
                                unsigned int value);
```

ASoC component callbacks

snd_soc_dai_ops

include/sound/soc-dai.h

```
struct snd_soc_dai_ops {
        /*
         * DAI clocking configuration, all optional.
         * Called by soc_card drivers, normally in their hw_params.
         */
        int (*set_sysclk)(struct snd_soc_dai *dai,
                int clk_id, unsigned int freq, int dir);
        int (*set_pll)(struct snd_soc_dai *dai, int pll_id, int source,
                unsigned int freq_in, unsigned int freq_out);
        int (*set_clkdiv)(struct snd_soc_dai *dai, int div_id, int div);
        int (*set_bclk_ratio)(struct snd_soc_dai *dai, unsigned int ratio);

        /*
         * DAI format configuration
         * Called by soc_card drivers, normally in their hw_params.
         */
        int (*set_fmt)(struct snd_soc_dai *dai, unsigned int fmt);
        int (*xlate_tdm_slot_mask)(unsigned int slots,
                unsigned int *tx_mask, unsigned int *rx_mask);
        int (*set_tdm_slot)(struct snd_soc_dai *dai,
                unsigned int tx_mask, unsigned int rx_mask,
                int slots, int slot_width);
```

snd_soc_dai_ops

include/sound/soc-dai.h

```
        /*
         * DAI digital mute - optional.
         * Called by soc-core to minimise any pops.
         */
        int (*mute_stream)(struct snd_soc_dai *dai, int mute, int stream);

        /*
         * ALSA PCM audio operations - all optional.
         * Called by soc-core during audio PCM operations.
         */
        int (*startup)(struct snd_pcm_substream *,
                struct snd_soc_dai *);
        void (*shutdown)(struct snd_pcm_substream *,
                struct snd_soc_dai *);
        int (*hw_params)(struct snd_pcm_substream *,
                struct snd_pcm_hw_params *, struct snd_soc_dai *);
        int (*hw_free)(struct snd_pcm_substream *,
                struct snd_soc_dai *);
        int (*prepare)(struct snd_pcm_substream *,
                struct snd_soc_dai *);
[...]
};
```

hw_params

- The most useful callback
- It is used to configure the component to match the parameters of the audio stream.
- Called when the stream is ready to be played, before any data is transferred.
- If the requested parameters cannot be supported by the hardware, the `hw_params` callback can return an error code to indicate that the stream cannot be opened. Otherwise, the callback returns successfully, and the audio stream can be started.

snd_pcm_hw_params

- Holds the stream parameters
- Usually not accessed directly but through accessors:
  - params_channels: the number of channels
  - params_rate: the sample rate
  - params_width: the number of bits per sample

hw_params example

sound/soc/codecs/tlv320aic31xx.c

```
static int aic31xx_hw_params(struct snd_pcm_substream *substream,
                             struct snd_pcm_hw_params *params,
                             struct snd_soc_dai *dai)
{
        struct snd_soc_component *component = dai->component;
        struct aic31xx_priv *aic31xx = snd_soc_component_get_drvdata(component);
        u8 data = 0;

        switch (params_width(params)) {
        case 16:
                break;
        case 20:
                data = (AIC31XX_WORD_LEN_20BITS <<
                        AIC31XX_IFACE1_DATALEN_SHIFT);
                break;
        case 24:
                data = (AIC31XX_WORD_LEN_24BITS <<
                        AIC31XX_IFACE1_DATALEN_SHIFT);
                break;
        case 32:
                data = (AIC31XX_WORD_LEN_32BITS <<
                        AIC31XX_IFACE1_DATALEN_SHIFT);
                break;
        default:
                dev_err(component->dev, "%s: Unsupported width %d\n",
                        __func__, params_width(params));
                return -EINVAL;
        }
```

hw_params example

sound/soc/codecs/tlv320aic31xx.c

```
        snd_soc_component_update_bits(component, AIC31XX_IFACE1,
                                      AIC31XX_IFACE1_DATALEN_MASK,
                                      data);

        /*
         * If BCLK is used as PLL input, the sysclk is determined by the hw
         * params. So it must be updated here to match the input frequency.
         */
        if (aic31xx->sysclk_id == AIC31XX_PLL_CLKIN_BCLK) {
                aic31xx->sysclk = params_rate(params) * params_width(params) *
                                  params_channels(params);
                aic31xx->p_div = 1;
        }

        return aic31xx_setup_pll(component, params);
}
```

aic31xx_setup_pll() then uses the parameters to set the CODEC PLLs and clocks properly. The usual ways to achieve that are to either do the calculations or prepare an array matching parameters to register values.

set_sysclk

- This sets the system clock parameters of the component, in particular which one is selected, its frequency and the direction.
- This allows the component to set up PLLs and clocks.
- This is called from the machine driver, using `snd_soc_dai_set_sysclk()`
- It can return an error in case the clock is not available or the frequency is not in the supported range.
- A component wide version exists, called using `snd_soc_component_set_sysclk()`, very rarely used.

set_sysclk example

```
static int aic31xx_set_dai_sysclk(struct snd_soc_dai *codec_dai,
                                  int clk_id, unsigned int freq, int dir)
{
        struct snd_soc_component *component = codec_dai->component;
        struct aic31xx_priv *aic31xx = snd_soc_component_get_drvdata(component);
        int i;
[...]
        for (i = 1; i < 8; i++)
                if (freq / i <= 20000000)
                        break;
        if (freq/i > 20000000) {
                dev_err(aic31xx->dev, "%s: Too high mclk frequency %u\n",
                        __func__, freq);
                return -EINVAL;
        }
        aic31xx->p_div = i;

        for (i = 0; i < ARRAY_SIZE(aic31xx_divs); i++)
                if (aic31xx_divs[i].mclk_p == freq / aic31xx->p_div)
                        break;
        if (i == ARRAY_SIZE(aic31xx_divs)) {
                dev_err(aic31xx->dev, "%s: Unsupported frequency %d\n",
                        __func__, freq);
                return -EINVAL;
        }

        /* set clock on MCLK, BCLK, or GPIO1 as PLL input */
        snd_soc_component_update_bits(component, AIC31XX_CLKMUX, AIC31XX_PLL_CLKIN_MASK,
                              clk_id << AIC31XX_PLL_CLKIN_SHIFT);

        aic31xx->sysclk_id = clk_id;
        aic31xx->sysclk = freq;

        return 0;
}
```

set_fmt

- This sets the format of the PCM bus
- This is called from the machine driver, using `snd_soc_dai_set_fmt()`
- Available formats are:
  - `SND_SOC_DAIFMT_I2S`
  - `SND_SOC_DAIFMT_RIGHT_J`
  - `SND_SOC_DAIFMT_LEFT_J`
  - `SND_SOC_DAIFMT_DSP_A`
  - `SND_SOC_DAIFMT_DSP_B`
  - `SND_SOC_DAIFMT_AC97`
  - `SND_SOC_DAIFMT_PDM`
- Also the polarity can be changed:
  - `SND_SOC_DAIFMT_NB_NF`: normal bit clock + normal frame
  - `SND_SOC_DAIFMT_NB_IF`: normal bit clock + invert frame
  - `SND_SOC_DAIFMT_IB_NF`: invert bit clock + normal frame
  - `SND_SOC_DAIFMT_IB_IF`: invert bit clock + invert frame

set_fmt

- The clock directions can also be set:
  - SND_SOC_DAIFMT_CBP_CFP: codec clk provider and frame provider
  - SND_SOC_DAIFMT_CBC_CFP: codec clk consumer and frame provider
  - SND_SOC_DAIFMT_CBP_CFC: codec clk provider and frame consumer
  - SND_SOC_DAIFMT_CBC_CFC: codec clk consumer and frame consumer
- These used to have another name:

include/sound/soc-dai.h

```
/* previous definitions kept for backwards-compatibility, do not use in new contributions */
#define SND_SOC_DAIFMT_CBM_CFM                SND_SOC_DAIFMT_CBP_CFP
#define SND_SOC_DAIFMT_CBS_CFM                SND_SOC_DAIFMT_CBC_CFP
#define SND_SOC_DAIFMT_CBM_CFS                SND_SOC_DAIFMT_CBP_CFC
#define SND_SOC_DAIFMT_CBS_CFS                SND_SOC_DAIFMT_CBC_CFC
```

set_fmt example

```
static int aic31xx_set_dai_fmt(struct snd_soc_dai *codec_dai,
                               unsigned int fmt)
{
        struct snd_soc_component *component = codec_dai->component;
        u8 iface_reg1 = 0;
        u8 iface_reg2 = 0;
        u8 dsp_a_val = 0;
[...]
        switch (fmt & SND_SOC_DAIFMT_CLOCK_PROVIDER_MASK) {
        case SND_SOC_DAIFMT_CBP_CFP:
                iface_reg1 |= AIC31XX_BCLK_MASTER | AIC31XX_WCLK_MASTER;
                break;
[...]
        }

        /* signal polarity */
        switch (fmt & SND_SOC_DAIFMT_INV_MASK) {
        case SND_SOC_DAIFMT_NB_NF:
[...]
        }

        /* interface format */
        switch (fmt & SND_SOC_DAIFMT_FORMAT_MASK) {
[...]
        }

        snd_soc_component_update_bits(component, AIC31XX_IFACE1,
                            AIC31XX_IFACE1_DATATYPE_MASK |
                            AIC31XX_IFACE1_MASTER_MASK,
                            iface_reg1);
```

set_tdm_slot

- This callback configures the DAI for TDM operation.
- `slots` is the total number of slots of the TDM stream and `slot_width` the width of each slot in bit clock cycles.
- `tx_mask` and `rx_mask` are bitmasks specifying the active slots of the TDM stream for the specified DAI, i.e. which slots the DAI should write to or read from. A set bit means the channel is active.
- This is called from the machine driver, using `snd_soc_dai_set_tdm_slot()`
- This allows to explicitly configure mismatching stream and bus sample width.
- TDM mode must be disabled when `slots` is 0.

trigger

- This callback is called when the stream status is updated.
- It allows to listen for events.
- This is called from the Alsa core, in `soc_pcm_trigger()` using `snd_soc_pcm_dai_trigger()`
- A component version exists.
- Available states are:
  - `SNDRV_PCM_TRIGGER_STOP`
  - `SNDRV_PCM_TRIGGER_START`
  - `SNDRV_PCM_TRIGGER_PAUSE_PUSH`
  - `SNDRV_PCM_TRIGGER_PAUSE_RELEASE`
  - `SNDRV_PCM_TRIGGER_SUSPEND`
  - `SNDRV_PCM_TRIGGER_RESUME`
  - `SNDRV_PCM_TRIGGER_DRAIN`

trigger example

- The PCM1789 needs the system clock, bit clock and frame clock to be synchronized as soon as it gets out of reset.
- With DAPM, those clocks are disabled until a stream is ready to be played.
- A solution is to reset the device when a stream is played.

trigger example

sound/soc/codecs/pcm1789.c

```
static int pcm1789_trigger(struct snd_pcm_substream *substream, int cmd,
                           struct snd_soc_dai *dai)
{
        struct snd_soc_component *component = dai->component;
        struct pcm1789_private *priv = snd_soc_component_get_drvdata(component);
        int ret = 0;

        switch (cmd) {
        case SNDRV_PCM_TRIGGER_START:
        case SNDRV_PCM_TRIGGER_RESUME:
        case SNDRV_PCM_TRIGGER_PAUSE_RELEASE:
                schedule_work(&priv->work);
                break;
        case SNDRV_PCM_TRIGGER_STOP:
        case SNDRV_PCM_TRIGGER_SUSPEND:
        case SNDRV_PCM_TRIGGER_PAUSE_PUSH:
                break;
        default:
                ret = -EINVAL;
        }

        return ret;
}
```

trigger example

sound/soc/codecs/pcm1789.c

```
static void pcm1789_work_queue(struct work_struct *work)
{
        struct pcm1789_private *priv = container_of(work,
                                                    struct pcm1789_private,
                                                    work);

        /* Perform a software reset to remove codec from desynchronized state */
        if (regmap_update_bits(priv->regmap, PCM1789_MUTE_CONTROL,
                               0x3 << PCM1789_MUTE_SRET, 0) < 0)
                dev_err(priv->dev, "Error while setting SRET");
}
```

set_bias_level

- This callback is called by DAPM through `snd_soc_dapm_set_bias_level()` and `snd_soc_component_set_bias_level()` once the component gets activated.
- It allows to listen for power events.
- Available events are:
  - `SND_SOC_BIAS_ON`: Bias is fully on for audio playback and capture operations.
  - `SND_SOC_BIAS_PREPARE`: Prepare for audio operations. Called before DAPM switching for stream start and stop operations.
  - `SND_SOC_BIAS_STANDBY`: Low power standby state when no playback/capture operations are in progress. NOTE: The transition time between STANDBY and ON should be as fast as possible and no longer than 10ms.
  - `SND_SOC_BIAS_OFF`: Power Off. No restrictions on transition times.

set_bias_level example

- There are CODECs that won't even listen on the control bus until there are clocks on the PCM bus or that will stay powered off as much as possible.
- A solution is to use regcache.

set_bias_level example

sound/soc/codecs/ssm2518.c

```
static int ssm2518_set_bias_level(struct snd_soc_component *component,
        enum snd_soc_bias_level level)
{
        struct ssm2518 *ssm2518 = snd_soc_component_get_drvdata(component);
        int ret = 0;

        switch (level) {
        case SND_SOC_BIAS_ON:
                break;
        case SND_SOC_BIAS_PREPARE:
                break;
        case SND_SOC_BIAS_STANDBY:
                if (snd_soc_component_get_bias_level(component) == SND_SOC_BIAS_OFF)
                        ret = ssm2518_set_power(ssm2518, true);
                break;
        case SND_SOC_BIAS_OFF:
                ret = ssm2518_set_power(ssm2518, false);
                break;
        }

        return ret;
}
```

set_bias_level example

sound/soc/codecs/ssm2518.c

```
static int ssm2518_set_power(struct ssm2518 *ssm2518, bool enable)
{
        int ret = 0;

        if (!enable) {
                ret = regmap_update_bits(ssm2518->regmap, SSM2518_REG_POWER1,
                        SSM2518_POWER1_SPWDN, SSM2518_POWER1_SPWDN);
                regcache_mark_dirty(ssm2518->regmap);
        }

        if (ssm2518->enable_gpio)
                gpiod_set_value_cansleep(ssm2518->enable_gpio, enable);

        regcache_cache_only(ssm2518->regmap, !enable);

        if (enable) {
                ret = regmap_update_bits(ssm2518->regmap, SSM2518_REG_POWER1,
                        SSM2518_POWER1_SPWDN | SSM2518_POWER1_RESET, 0x00);
                regcache_sync(ssm2518->regmap);
        }

        return ret;
}
```

Auxiliary devices

Amplifier

What about the amplifier?

- Supported using *auxiliary devices*
- Register a `struct snd_soc_aux_dev` array using the `.aux_dev` and `.num_aux_devs` fields of the registered `struct snd_soc_card`
- This will expose the auxiliary devices control widgets as part of the sound card
- There is a driver for simple amplifiers driven by a single GPIO, `simple-amplifier`
  - `Documentation/devicetree/bindings/sound/simple-audio-amplifier.yaml`
  - `sound/soc/codecs/simple-amplifier.c`

Auxiliary devices

sound/soc/samsung/neo1973_wm8753.c

```
static struct snd_soc_aux_dev neo1973_aux_devs[] = {
        {
                .name = "dfbmcs320",
                .codec_name = "dfbmcs320.0",
        },
};

static struct snd_soc_card neo1973 = {
        .name = "neo1973",
        .owner = THIS_MODULE,
        .dai_link = neo1973_dai,
        .num_links = ARRAY_SIZE(neo1973_dai),
        .aux_dev = neo1973_aux_devs,
        .num_aux_devs = ARRAY_SIZE(neo1973_aux_devs),
```

simple-amplifier - example 1

arch/arm64/boot/dts/allwinner/sun50i-a64-pinebook.dts

```
        speaker_amp: audio-amplifier {
                compatible = "simple-audio-amplifier";
                VCC-supply = <&reg_ldo_io0>;
                enable-gpios = <&pio 7 7 GPIO_ACTIVE_HIGH>; /* PH7 */
                sound-name-prefix = "Speaker Amp";
        };

&sound {
        status = "okay";
        simple-audio-card,aux-devs = <&codec_analog>, <&speaker_amp>;
        simple-audio-card,widgets = "Microphone", "Internal Microphone Left",
                                    "Microphone", "Internal Microphone Right",
                                    "Headphone", "Headphone Jack",
                                    "Speaker", "Internal Speaker";
        simple-audio-card,routing =
                        "Left DAC", "AIF1 Slot 0 Left",
                        "Right DAC", "AIF1 Slot 0 Right",
                        "Speaker Amp INL", "LINEOUT",
                        "Speaker Amp INR", "LINEOUT",
                        "Internal Speaker", "Speaker Amp OUTL",
                        "Internal Speaker", "Speaker Amp OUTR",
                        "Headphone Jack", "HP",
```

simple-amplifier - example 2

```
dio2133: analog-amplifier {
        compatible = "simple-audio-amplifier";
        sound-name-prefix = "AU2";
        VCC-supply = <&hdmi_5v>;
        enable-gpios = <&gpio GPIOH_5 GPIO_ACTIVE_HIGH>;
};

sound {
        compatible = "amlogic,gx-sound-card";
        model = "GXL-LIBRETECH-S905X-CC";
        audio-aux-devs = <&dio2133>;
        audio-widgets = "Line", "Lineout";
        audio-routing = "AU2 INL", "ACODEC LOLN",
                        "AU2 INR", "ACODEC LORN",
                        "Lineout", "AU2 OUTL",
                        "Lineout", "AU2 OUTR";
```

Audio is routed through AU2, the amplifier.

Input Muxing

- There may be a muxer on the analog input lines.
- If controlled using a gpio, the `simple-mux` driver is available.
- It exposes two inputs: "IN1" and "IN2" and one output, "OUT".
- The device tree binding allows to provide a prefix to make the routes specific.
  - `Documentation/devicetree/bindings/sound/simple-audio-mux.yaml`
  - `sound/soc/codecs/simple-mux.c`

simple-mux example

```
mic_mux: mic-mux {
        compatible = "simple-audio-mux";
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_micsel>;
        mux-gpios = <&gpio5 5 GPIO_ACTIVE_LOW>;
        sound-name-prefix = "Mic Mux";
};
```

- This exposes routes between `Mic Mux IN1` and `Mic Mux IN2` to `Mic Mux OUT`.
- This route is controlled by `gpio5 5`.
- A control named `Mic Mux Muxer` will be exposed to userspace.

simple-mux example

arch/arm64/boot/dts/freescale/imx8mq-librem5-devkit.dts

```
sound {
        compatible = "simple-audio-card";
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_hpdet>;
        simple-audio-card,aux-devs = <&speaker_amp>, <&mic_mux>;
        simple-audio-card,name = "Librem 5 Devkit";
        simple-audio-card,format = "i2s";
        simple-audio-card,widgets =
                "Microphone", "Builtin Microphone",
                "Microphone", "Headset Microphone",
                "Headphone", "Headphones",
                "Speaker", "Builtin Speaker";
        simple-audio-card,routing =
                "MIC_IN", "Mic Mux OUT",
                "Mic Mux IN1", "Headset Microphone",
                "Mic Mux IN2", "Builtin Microphone",
                "Mic Mux OUT", "Mic Bias",
                "Headphones", "HP_OUT",
                "Builtin Speaker", "Speaker Amp OUTR",
                "Speaker Amp INR", "LINE_OUT";
        simple-audio-card,hp-det-gpio = <&gpio3 20 GPIO_ACTIVE_HIGH>;

        simple-audio-card,cpu {
                sound-dai = <&sai2>;
        };

        simple-audio-card,codec {
                sound-dai = <&sgtl5000>;
                clocks = <&clk IMX8MQ_CLK_SAI2_ROOT>;
                frame-master;
                bitclock-master;
        };
};
```

ASoC DAPM

DAPM

- DAPM stands for Dynamic Audio Power Management.
- The goal is to save as much power as possible by shutting down audio routes that are not in use.
- This may affect the whole card or just part of it.
- To achieve this, the topology needs to be described. For this we have two objects: DAPM widgets and DAPM routes.
- The DAPM widgets represent various components of an audio system, such as audio inputs, outputs, mixers, and amplifiers.
- The routes are connecting widgets together.

snd_soc_dapm_widget

- An array of `struct snd_soc_dapm_widget` is registered by the component.
- Many helpers exist to avoid filling the struct manually:

include/sound/soc-dapm.h

```
#define SND_SOC_DAPM_INPUT(wname) \
{       .id = snd_soc_dapm_input, .name = wname, .kcontrol_news = NULL, \
        .num_kcontrols = 0, .reg = SND_SOC_NOPM }
#define SND_SOC_DAPM_OUTPUT(wname) \
{       .id = snd_soc_dapm_output, .name = wname, .kcontrol_news = NULL, \
        .num_kcontrols = 0, .reg = SND_SOC_NOPM }
#define SND_SOC_DAPM_MIC(wname, wevent) \
{       .id = snd_soc_dapm_mic, .name = wname, .kcontrol_news = NULL, \
        .num_kcontrols = 0, .reg = SND_SOC_NOPM, .event = wevent, \
        .event_flags = SND_SOC_DAPM_PRE_PMU | SND_SOC_DAPM_POST_PMD}
[...]
#define SND_SOC_DAPM_PGA(wname, wreg, wshift, winvert,\
         wcontrols, wncontrols) \
{       .id = snd_soc_dapm_pga, .name = wname, \
        SND_SOC_DAPM_INIT_REG_VAL(wreg, wshift, winvert), \
        .kcontrol_news = wcontrols, .num_kcontrols = wncontrols}
[...]
#define SND_SOC_DAPM_MUX(wname, wreg, wshift, winvert, wcontrols) \
{       .id = snd_soc_dapm_mux, .name = wname, \
        SND_SOC_DAPM_INIT_REG_VAL(wreg, wshift, winvert), \
        .kcontrol_news = wcontrols, .num_kcontrols = 1}
#define SND_SOC_DAPM_DEMUX(wname, wreg, wshift, winvert, wcontrols) \
{       .id = snd_soc_dapm_demux, .name = wname, \
        SND_SOC_DAPM_INIT_REG_VAL(wreg, wshift, winvert), \
```

snd_soc_dapm_widget

include/sound/soc-dapm.h

```
#define SND_SOC_DAPM_DAC(wname, stname, wreg, wshift, winvert) \
{       .id = snd_soc_dapm_dac, .name = wname, .sname = stname, \
        SND_SOC_DAPM_INIT_REG_VAL(wreg, wshift, winvert) }
#define SND_SOC_DAPM_DAC_E(wname, stname, wreg, wshift, winvert, \
                           wevent, wflags)                       \
{       .id = snd_soc_dapm_dac, .name = wname, .sname = stname, \
        SND_SOC_DAPM_INIT_REG_VAL(wreg, wshift, winvert), \
        .event = wevent, .event_flags = wflags}

#define SND_SOC_DAPM_ADC(wname, stname, wreg, wshift, winvert) \
{       .id = snd_soc_dapm_adc, .name = wname, .sname = stname, \
        SND_SOC_DAPM_INIT_REG_VAL(wreg, wshift, winvert), }
#define SND_SOC_DAPM_ADC_E(wname, stname, wreg, wshift, winvert, \
                           wevent, wflags)                       \
{       .id = snd_soc_dapm_adc, .name = wname, .sname = stname, \
        SND_SOC_DAPM_INIT_REG_VAL(wreg, wshift, winvert), \
        .event = wevent, .event_flags = wflags}


/* generic widgets */
#define SND_SOC_DAPM_REG(wid, wname, wreg, wshift, wmask, won_val, woff_val) \
{       .id = wid, .name = wname, .kcontrol_news = NULL, .num_kcontrols = 0, \
        .reg = wreg, .shift = wshift, .mask = wmask, \
        .on_val = won_val, .off_val = woff_val, }
#define SND_SOC_DAPM_SUPPLY(wname, wreg, wshift, winvert, wevent, wflags) \
{       .id = snd_soc_dapm_supply, .name = wname, \
        SND_SOC_DAPM_INIT_REG_VAL(wreg, wshift, winvert), \
        .event = wevent, .event_flags = wflags}
#define SND_SOC_DAPM_REGULATOR_SUPPLY(wname, udelay, wflags)              \
```

DAPM example

sound/soc/codecs/pcm3168a.c

```
static const struct snd_soc_dapm_widget pcm3168a_dapm_widgets[] = {
        SND_SOC_DAPM_DAC("DAC1", "Playback", PCM3168A_DAC_OP_FLT,
                         PCM3168A_DAC_OPEDA_SHIFT, 1),
        SND_SOC_DAPM_DAC("DAC2", "Playback", PCM3168A_DAC_OP_FLT,
                         PCM3168A_DAC_OPEDA_SHIFT + 1, 1),
        SND_SOC_DAPM_DAC("DAC3", "Playback", PCM3168A_DAC_OP_FLT,
                         PCM3168A_DAC_OPEDA_SHIFT + 2, 1),
        SND_SOC_DAPM_DAC("DAC4", "Playback", PCM3168A_DAC_OP_FLT,
                         PCM3168A_DAC_OPEDA_SHIFT + 3, 1),

        SND_SOC_DAPM_OUTPUT("AOUT1L"),
        SND_SOC_DAPM_OUTPUT("AOUT1R"),
        SND_SOC_DAPM_OUTPUT("AOUT2L"),
        SND_SOC_DAPM_OUTPUT("AOUT2R"),
        SND_SOC_DAPM_OUTPUT("AOUT3L"),
        SND_SOC_DAPM_OUTPUT("AOUT3R"),
        SND_SOC_DAPM_OUTPUT("AOUT4L"),
        SND_SOC_DAPM_OUTPUT("AOUT4R"),
```

Note: on the PCM3168A DACs and ADCs can only be powered down in pairs.

DAPM example

sound/soc/codecs/pcm3168a.c

```
        SND_SOC_DAPM_ADC("ADC1", "Capture", PCM3168A_ADC_PWR_HPFB,
                         PCM3168A_ADC_PSVAD_SHIFT, 1),
        SND_SOC_DAPM_ADC("ADC2", "Capture", PCM3168A_ADC_PWR_HPFB,
                         PCM3168A_ADC_PSVAD_SHIFT + 1, 1),
        SND_SOC_DAPM_ADC("ADC3", "Capture", PCM3168A_ADC_PWR_HPFB,
                         PCM3168A_ADC_PSVAD_SHIFT + 2, 1),

        SND_SOC_DAPM_INPUT("AIN1L"),
        SND_SOC_DAPM_INPUT("AIN1R"),
        SND_SOC_DAPM_INPUT("AIN2L"),
        SND_SOC_DAPM_INPUT("AIN2R"),
        SND_SOC_DAPM_INPUT("AIN3L"),
        SND_SOC_DAPM_INPUT("AIN3R")
};
```

snd_soc_dapm_route

- An array of `struct snd_soc_dapm_route` is registered by the component to define the routes.

include/sound/soc-dapm.h

```
struct snd_soc_dapm_route {
        const char *sink;
        const char *control;
        const char *source;

        /* Note: currently only supported for links where source is a supply */
        int (*connected)(struct snd_soc_dapm_widget *source,
                         struct snd_soc_dapm_widget *sink);

        struct snd_soc_dobj dobj;
};
```

DAPM routes example

sound/soc/codecs/pcm3168a.c

```
static const struct snd_soc_dapm_route pcm3168a_dapm_routes[] = {
        /* Playback */
        { "AOUT1L", NULL, "DAC1" },
        { "AOUT1R", NULL, "DAC1" },

        { "AOUT2L", NULL, "DAC2" },
        { "AOUT2R", NULL, "DAC2" },

        { "AOUT3L", NULL, "DAC3" },
        { "AOUT3R", NULL, "DAC3" },

        { "AOUT4L", NULL, "DAC4" },
        { "AOUT4R", NULL, "DAC4" },

        /* Capture */
        { "ADC1", NULL, "AIN1L" },
        { "ADC1", NULL, "AIN1R" },

        { "ADC2", NULL, "AIN2L" },
        { "ADC2", NULL, "AIN2R" },

        { "ADC3", NULL, "AIN3L" },
        { "ADC3", NULL, "AIN3R" }
};
```

CPU DAI driver

CPU DAI driver

- The CPU DAI driver is now a component driver, like the codec ones.
- However, it is usually more complex as it need to handle IRQs and take care of pinmuxing, clocks and DMA.
- Also, the list of supported format and rates is usually very large.

DMA handling

- When a DMA controller is available, handling DMA in ALSA is done almost completely in the core, through `dmaengine_pcm`.
- The DMA is simply registered using `devm_snd_dmaengine_pcm_register()`. This handles parsing the device tree if necessary.
- In the DAI driver probe callback, the DMA engine is simply configured using `snd_soc_dai_init_dma_data()` which takes the DMA configuration for playback and capture.

DMA handling example

sound/soc/atmel/atmel-i2s.c

```
struct atmel_i2s_dev {
        struct device                           *dev;
        struct regmap                           *regmap;
        struct clk                              *pclk;
        struct clk                              *gclk;
        struct snd_dmaengine_dai_dma_data       playback;
        struct snd_dmaengine_dai_dma_data       capture;
        unsigned int                            fmt;
        const struct atmel_i2s_gck_param        *gck_param;
        const struct atmel_i2s_caps             *caps;
        int                                     clk_use_no;
};
[...]
static int atmel_i2s_dai_probe(struct snd_soc_dai *dai)
{
        struct atmel_i2s_dev *dev = snd_soc_dai_get_drvdata(dai);

        snd_soc_dai_init_dma_data(dai, &dev->playback, &dev->capture);
        return 0;
}
```

DMA handling example

sound/soc/atmel/atmel-i2s.c

```
static int atmel_i2s_probe(struct platform_device *pdev)
{
[...]
        /* Prepare DMA config. */
        dev->playback.addr        = (dma_addr_t)mem->start + ATMEL_I2SC_THR;
        dev->playback.maxburst        = 1;
        dev->capture.addr        = (dma_addr_t)mem->start + ATMEL_I2SC_RHR;
        dev->capture.maxburst        = 1;

        if (of_property_match_string(np, "dma-names", "rx-tx") == 0)
                pcm_flags |= SND_DMAENGINE_PCM_FLAG_HALF_DUPLEX;
        err = devm_snd_dmaengine_pcm_register(&pdev->dev, NULL, pcm_flags);
        if (err) {
                dev_err(&pdev->dev, "failed to register PCM: %d\n", err);
                clk_disable_unprepare(dev->pclk);
                return err;
        }
[...]
}
```

DMA handling

- When a peripheral DMA controller is used, this is more complex.
- The driver will have to handle all the aspects of the PCM stream life cycle.
- Understandable example in `sound/soc/atmel/atmel-pcm-pdc.c`

Userspace ALSA

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

alsa-lib

alsa-lib

- The main way to interact with ALSA devices is to use alsa-lib.
- https://github.com/alsa-project/alsa-lib.git
- It provides mainly access to the devices but also goes further and allows handling audio in userspace.
- The library itself is actually named `libasound`
- The include file is `alsa/asoundlib.h`

alsa-lib API

- `int snd_pcm_open(snd_pcm_t ** pcmp, const char * name, snd_pcm_stream_t stream, int mode)`
  - `name` is the name of the PCM to be opened.
  - `stream` can be either `SND_PCM_STREAM_PLAYBACK` or `SND_PCM_STREAM_CAPTURE`
  - `mode` can be a combination of `SND_PCM_NONBLOCK` and `SND_PCM_ASYNC`
- `int snd_pcm_close(snd_pcm_t *pcm)`
  - Closes the PCM.

PCM name

- This can be specified as a hardware device. The three arguments (in order: CARD,DEV,SUBDEV) specify card number or identifier, device number and subdevice number (-1 means any). For example: `hw:0` or `hw:1,0`. Instead of the index, the card name can be used: `hw:STM32MP15DK,0`
- Or through the `plug` plugin: `plug:mypcmdef`, `plug:hw:0,0`.
- The list of available names can be generated using `snd_card_next` to iterate over all the physical cards. See `device_list` in `aplay`.

alsa-lib API - PCM

- The next step is to handle the PCM stream parameters

```
snd_pcm_hw_params_t *hw_params;
int snd_pcm_hw_params_malloc(snd_pcm_hw_params_t ** ptr)
int snd_pcm_hw_params_any(snd_pcm_t * pcm, snd_pcm_hw_params_t * params)
```

- This will allocate a `snd_pcm_hw_params_t` and fill it with the range of parameters supported by `pcm`.

```
int snd_pcm_hw_params_set_access(snd_pcm_t *pcm, snd_pcm_hw_params_t *params,
                                 snd_pcm_access_t _access)
```

- This set the proper access type: interleaved or non-interleaved, mmap or not.

```
int snd_pcm_hw_params_set_format(snd_pcm_t *pcm, snd_pcm_hw_params_t *params,
                                 snd_pcm_format_t val)
```

- This set the format, using a `SND_PCM_FORMAT_` macro.

alsa-lib API - PCM

```
int snd_pcm_hw_params_set_channels(snd_pcm_t *pcm, snd_pcm_hw_params_t *params,
                                   unsigned int val)
```

- Sets the number of channels.

```
int snd_pcm_hw_params_set_rate_near(snd_pcm_t *pcm, snd_pcm_hw_params_t *params,
                                    unsigned int *val, int *dir)
```

- Sets the sample rate, setting `dir` to 0 will require the exact rate.

```
int snd_pcm_hw_params_set_periods(snd_pcm_t *pcm, snd_pcm_hw_params_t *params,
                                  unsigned int val, int dir)
int snd_pcm_hw_params_set_period_size(snd_pcm_t *pcm, snd_pcm_hw_params_t *params,
                                      snd_pcm_uframes_t val, int dir)
int snd_pcm_hw_params_set_buffer_size(snd_pcm_t *pcm, snd_pcm_hw_params_t *params,
                                      snd_pcm_uframes_t val)
```

- Sets the number of periods and the period size in the buffer or the buffer size.

alsa-lib API - PCM

- `int snd_pcm_hw_params(snd_pcm_t * pcm, snd_pcm_hw_params_t * params)`
  - Installs the parameters and calls `snd_pcm_prepare` on the stream.
- `void snd_pcm_hw_params_free(snd_pcm_hw_params_t * obj)`
  - Frees the allocated `snd_pcm_hw_params_t`.
- `int snd_pcm_prepare(snd_pcm_t * pcm)`
  - Prepares the stream.
- `int snd_pcm_wait(snd_pcm_t * pcm, int timeout)`
  - Waits for the PCM to be ready.

alsa-lib API - PCM

```
snd_pcm_sframes_t snd_pcm_writei(snd_pcm_t *pcm, const void *buffer, snd_pcm_uframes_t size)
snd_pcm_sframes_t snd_pcm_readi(snd_pcm_t *pcm, void *buffer, snd_pcm_uframes_t size)
snd_pcm_sframes_t snd_pcm_writen(snd_pcm_t *pcm, void **bufs, snd_pcm_uframes_t size)
snd_pcm_sframes_t snd_pcm_readn(snd_pcm_t *pcm, void **bufs, snd_pcm_uframes_t size)
```

- Write or read from an interleaved or non-interleaved buffer.

```
int snd_pcm_mmap_begin(snd_pcm_t *pcm, const snd_pcm_channel_area_t **areas,
                       snd_pcm_uframes_t *offset, snd_pcm_uframes_t *frames)
snd_pcm_sframes_t snd_pcm_mmap_commit(snd_pcm_t *pcm, snd_pcm_uframes_t offset,
                                      snd_pcm_uframes_t frames)
snd_pcm_sframes_t snd_pcm_mmap_writei(snd_pcm_t *pcm, const void *buffer,
                                      snd_pcm_uframes_t size)
snd_pcm_sframes_t snd_pcm_mmap_readi(snd_pcm_t *pcm, void *buffer, snd_pcm_uframes_t size)
snd_pcm_sframes_t snd_pcm_mmap_writen(snd_pcm_t *pcm, void **bufs, snd_pcm_uframes_t size)
snd_pcm_sframes_t snd_pcm_mmap_readn(snd_pcm_t *pcm, void **bufs, snd_pcm_uframes_t size)
```

- Write or read from an interleaved or non-interleaved mmap buffer.

alsa-lib API - controls

- It is possible to set controls programatically.

```
snd_ctl_t *handle;
int snd_ctl_open (snd_ctl_t **ctl, const char *name, int mode)
```

- Opens the sound card to be controlled.

```
snd_ctl_elem_id_t *id;
#define snd_ctl_elem_id_alloca(ptr)
snd_ctl_elem_value_t *value;
#define snd_ctl_elem_value_alloca(ptr)
```

- Allocate a `snd_ctl_elem_id_t`, referring to a particular control and a `snd_ctl_elem_value_t` to be set for this control.

```
void snd_ctl_elem_id_set_interface(snd_ctl_elem_id_t *obj, snd_ctl_elem_iface_t val)
void snd_ctl_elem_id_set_name(snd_ctl_elem_id_t *obj, const char *val)
```

- Set the interface and name of the control to be set.

alsa-lib API - controls

- A lookup is needed to fill the `snd_ctl_elem_id_t` completely

```
int lookup_id(snd_ctl_elem_id_t *id, snd_ctl_t *handle)
{
    int err;
    snd_ctl_elem_info_t *info;
    snd_ctl_elem_info_alloca(&info);

    snd_ctl_elem_info_set_id(info, id);
    if ((err = snd_ctl_elem_info(handle, info)) < 0) {
        return err;
    }
    snd_ctl_elem_info_get_id(info, id);

    return 0;
}
```

alsa-lib API - controls

- ```
  void snd_ctl_elem_value_set_id(snd_ctl_elem_value_t *obj, const snd_ctl_elem_id_t *ptr)
  ```
  - Links the value with the control id.
- ```
  void snd_ctl_elem_value_set_boolean(snd_ctl_elem_value_t *obj, unsigned int idx, long val)
  void snd_ctl_elem_value_set_integer(snd_ctl_elem_value_t *obj, unsigned int idx, long val)
  void snd_ctl_elem_value_set_integer64(snd_ctl_elem_value_t *obj, unsigned int idx,
                                        long long val)
  void snd_ctl_elem_value_set_enumerated(snd_ctl_elem_value_t *obj, unsigned int idx,
                                         unsigned int val)
  void snd_ctl_elem_value_set_byte(snd_ctl_elem_value_t *obj, unsigned int idx,
                                   unsigned char val)
  void snd_ctl_elem_set_bytes(snd_ctl_elem_value_t *obj, void *data, size_t size)
  ```
  - Set the value in `snd_ctl_elem_value_t`.
- ```
  int snd_ctl_elem_write(snd_ctl_t *ctl, snd_ctl_elem_value_t *data)
  ```
  - Actually set the control.
  ```

Going further

- UCM: The ALSA Use Case Configuration: https://www.alsa-project.org/alsa-doc/alsa-lib/group__ucm__conf.html
- ALSA topology: https://www.alsa-project.org/wiki/ALSA_topology

Demo - Card configuration examples

Using `alsa-lib` tools to:

- Reorder channels
- Split channels
- Resample
- Mix samples
- Apply effects

alsa-utils

alsa-utils

- `alsa-utils` is a repository of tools to interact with ALSA devices
- https://github.com/alsa-project/alsa-utils

Controls

- `alsamixer` provides a `ncurse` based graphical interface to modify sound cards controls.
- `amixer` is a command line tool to set controls.
  - The `scontrols` and `controls` commands list the available controls.
  - The `contents` commands list the available controls and shows their content.
  - The `cset` and `sset` commands allows modifying the controls.
  - The `cget` and `sget` commands show the content of a specific control.
- `alsactl` is a tool that can save the control values to a file and restore them from a file.

Playback and capture

- `speaker-test` can generate tones or noises to play on specific channels with a specified rate.
- `aplay` plays an audio file. It is able to set many stream parameters.
- `arecord` can record an audio stream to a file.

Demo - Userspace tools

Using userspace tools to:

- configure sound card controls
- load and store default values for controls
- play sound
- record

Troubleshooting

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Troubleshooting: no sound

Audio seems to play for the correct duration but there is no sound:

- Unmute `Master` and the relevant controls
- Turn up the volume
- Check the codec analog muxing and mixing (use alsamixer)
- Check the amplifier configuration
- Check the routing

Troubleshooting: no sound

When trying to play sound but it seems stuck:

- Check pinmuxing
- Check the configured clock directions
- Check the producer/consumer configuration
- Check the clocks using an oscilloscope
- Check pinmuxing
- Some SoCs also have more muxing (NXP i.Mx AUDMUX, TI McASP)

Troubleshooting: write error

```
# aplay test.wav
Playing WAVE 'test.wav' : Signed 16 bit Little Endian, Rate 44100 Hz, Stereo
aplay: pcm_write:1737: write error: Input/output error
```

- Usually caused by an issue in the routing
- Check that the codec driver exposes a stream named "Playback"
- Use `vizdapm`: https://github.com/mihais/asoc-tools

Troubleshooting: over/underruns

```
# aplay test.wav
Playing WAVE 'test.wav' : Signed 16 bit Little Endian, Rate 44100 Hz, Stereo
underrun!!! (at least 1.899 ms long)
underrun!!! (at least 0.818 ms long)
underrun!!! (at least 2.912 ms long)
underrun!!! (at least 8.558 ms long)
```

- Usually caused by an imprecise BCLK
- Try to find a better PLL and dividers combination

Troubleshooting: going further

- Use `speaker-test` to generate audio and play tones.
- Be careful with the 440Hz tone, it may not expose all the errors. Rather play something that is not commonly divisible (e.g. 441Hz)
- Generate tone with fade in and fade out as this allows to catch DMA transfer issues more easily.

Troubleshooting: going further

- Have a look at the CPU DAI driver and its callback. In particular: `.set_clkdiv` and `.set_sysclk` to understand how the various clock dividers are setup. `.hw_params` or `.set_dai_fmt` may do some muxing
- Have a look at the codec driver callbacks, `.set_sysclk` as the `clk_id` parameter is codec specific.
- Remember using a codec as a clock consumer is an uncommon configuration and is probably untested.
- When in doubt, use `devmem` or `i2cget`

Demo - Troubleshooting

- Using debugfs to find issues
- Using vizdapm
- Using ftrace to trace register writes and DAPM states

PipeWire

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Introduction

Introduction

- A realtime multimedia data graph
- Works across processes
- Why?
  - Sharing of devices across processes
  - Dynamic routing at runtime
  - Implements format negociation & conversion
  - Modular audio processing, spread across Linux processes
  - Low overhead: shared memory for data and no roundtrip to daemon
- Same abstraction layer (alternatives)
  - PulseAudio
  - JACK Audio Connection Kit
- Technical stack: C (gnu11), Meson & Ninja

Concepts — objects

- The graph state representation is a list of objects.
- That object list is handled by the `Core` object, hosted by the PipeWire daemon.
- Each connected process is represented by a `Client` object.

Example with `pw-play audio.wav` and `pw-record --target pw-play rec.wav`:

```
$ pw-cli ls Core
    id 0, type PipeWire:Interface:Core/4
        object.serial = "0"
        core.name = "pipewire-0"
```

```
$ pw-cli ls Client
    id 35, type PipeWire:Interface:Client/3
        object.serial = "35"
        pipewire.sec.pid = "2718"
        application.name = "pipewire"
    id 129, type PipeWire:Interface:Client/3
        object.serial = "11608"
        pipewire.sec.pid = "466490"
        application.name = "pw-cli"
    id 145, type PipeWire:Interface:Client/3
        object.serial = "11572"
        pipewire.sec.pid = "465686"
        application.name = "pw-cat"
    id 168, type PipeWire:Interface:Client/3
        object.serial = "11593"
        pipewire.sec.pid = "466186"
        application.name = "pw-cat"
    ...
```

Concepts — nodes, ports & links (1)

- The graph itself is represented by the following object types:
  - A `Node` processes samples
  - A `Port` represents a node input or output
  - A `Link` connects an output port with an input port

Concepts — nodes, ports & links (2)

```
$ pw-cli ls Node
    id 137, type PipeWire:Interface:Node/3
        client.id = "145"
        node.name = "pw-play"
        media.class = "Stream/Output/Audio"
    id 111, type PipeWire:Interface:Node/3
        client.id = "168"
        node.name = "pw-record"
        media.class = "Stream/Input/Audio"
    ...

$ pw-cli ls Link
    id 119, type PipeWire:Interface:Link/3
        client.id = "33"
        link.output.port = "116"
        link.input.port = "139"
        link.output.node = "137"
        link.input.node = "111"
    id 97, type PipeWire:Interface:Link/3
        client.id = "33"
        link.output.port = "115"
        link.input.port = "117"
        link.output.node = "137"
        link.input.node = "111"
    ...
```

```
$ pw-cli ls Port
    id 116, type PipeWire:Interface:Port/3
        format.dsp = "32 bit float mono audio"
        node.id = "137"
        audio.channel = "FL"
        port.alias = "pw-play:output_FL"
    id 115, type PipeWire:Interface:Port/3
        format.dsp = "32 bit float mono audio"
        node.id = "137"
        audio.channel = "FR"
        port.alias = "pw-play:output_FR"
    id 139, type PipeWire:Interface:Port/3
        format.dsp = "32 bit float mono audio"
        node.id = "111"
        audio.channel = "FL"
        port.alias = "pw-record:input_FL"
    id 117, type PipeWire:Interface:Port/3
        format.dsp = "32 bit float mono audio"
        node.id = "111"
        audio.channel = "FR"
        port.alias = "pw-record:input_FR"
    ...
```

Concepts — object properties and params

- Objects are defined by their ID and type.
- Objects also contain **properties**: a list of string key-value pairs. Those can only be modified by the client hosting the node.
- Some object types also contain **params**. Those might be configurable by other clients.
  - They get used for format negociation & conversion, volume control, etc.

```
$ pw-cli info 94
  type: PipeWire:Interface:Node/3
* properties:
*   application.name = "pw-play"
*   node.name = "pw-play"
*   media.type = "Audio"
*   media.category = "Playback"
*   media.role = "Music"
*   node.rate = "1/44100"
*   node.latency = "4410/44100"
*   node.autoconnect = "true"
*   node.want-driver = "true"
*   media.class = "Stream/Output/Audio"
*   factory.id = "8"
*   clock.quantum-limit = "8192"
*   library.name = "audioconvert/libspa-audioconvert"
*   client.id = "151"
*   object.id = "94"
*   object.serial = "2005"
*   ...
* params: (8)
*   3 (Spa:Enum:ParamId:EnumFormat) r-
*   2 (Spa:Enum:ParamId:Props) rw
*   4 (Spa:Enum:ParamId:Format) rw
*   ...
```

Concepts — devices

- Another object type is `Device`. Those map to physical devices, to which are assigned one or more nodes. Device configuration is done via those objects.
- Providers can be alsa-lib, BlueZ, libcamera, V4L, etc.

Concepts — graph execution logic (1)

- PipeWire structures itself as multiple subgraphs. In each one of those, there is exactly one **driver** node, and zero or more **follower** nodes.
- The driver node is responsible for triggering the start of execution cycles, based on a timer or hardware interrupt for example.
- Each node keeps two counters:
  1. `required`: the number of dependencies on other nodes;
  2. `pending`: how many remaining nodes need to be executed before it can run in this cycle. A value of zero means the node can be executed. Its reset value is `required`.
- Nodes also keep a list of nodes that depend on them (called targets); a node is responsible for decrementing its targets' `pending` counters and signal them using IPC.
- See the documentation for more details. The graph evaluation is implemented by `pw_context_recalc_graph()`.

Concepts — graph execution logic (2)

- The driver node is picked based on the `priority.driver` property.
- A good default is to set higher priority to capture driver nodes.

Concepts — graph execution logic (3)

- PipeWire clients and modules can create independent nodes rather than a single one with input and output ports. That allows having multiple subgraphs, each driven by a different driver node.
- **Virtual loopbacks** are such an example: they allow sending samples from a subgraph to another while still decoupling driver clocks.

Concepts — graph execution logic (4)

- The number of samples to be generated during a cycle is called **the quantum**.
- There are global settings for minimum and maximum, and nodes can request specific values for the subgraph they take part in.
- Nodes can also request for a locked quantum: that it does not get changed across recalculations of the graph. This gets used for applications that require fixed quantum (such as the JACK compatibility layer).
- The **rate** is similar: it can be different for each subgraph. The PipeWire config has a list of allowed rates.

Concepts — modules

- Modules are libraries loaded by PipeWire clients to implement various features.
- Example modules:
  - `module-rt`: requests realtime scheduling priority using `setpriority(2)` and `pthread_setschedparam(3)`.
  - `module-loopback`: create two virtual loopback nodes.
  - `module-protocol-native`: implements the communication between the daemon and clients.
  - `module-profiler`: implements the profiling logic, attached to the daemon.
  - etc.

PipeWire communication protocols — IPC

- `socket(AF_UNIX, SOCK_STREAM, 0)` for communication with the daemon process. The socket is named `pipewire-0` by default or `$PIPEWIRE_REMOTE`. Directory look-up order:
  1. `$PIPEWIRE_RUNTIME_DIR`
  2. `$XDG_RUNTIME_DIR`
  3. `$USERPROFILE`
- `eventfd(2)` is the wakeup method.
- `memfd_create(2)` is used for sharing multimedia data across related clients (without data going through the daemon).
- PipeWire provides an event-loop implementation that relies upon `epoll(7)`. All clients use it. They also use `signalfd(2)` to handle signals.

PipeWire communication protocols — D-Bus optional dependency

- Happens on the session bus
- Flatpak permission support through XDG Desktop Portal, see `libpipewire-module-portal`
- Audio device reservation through the org.freedesktop.ReserveDevice1, see `libwireplumber-module-reserve-device`
- For Bluetooth support through BlueZ, see PipeWire's `libspa-bluez5`

Configuration

Configuration — location (1)

- Each client locates and reads its configuration at startup.
- Those configuration files follow a PipeWire-specific format.
- Look-up order:
  1. `$XDG_CONFIG_HOME/pipewire/`
     environment variable, often `~/.config/pipewire/` in distributions
  2. `$sysconfdir/pipewire/`
     compile-time variable, often `/etc/pipewire/`
  3. `$datadir/pipewire/`
     compile-time variable, often `/usr/share/pipewire/`

Configuration — location (2)

- A client that loads a config file named `client-rt.conf` will load the first file named as such in the above folders, but will also load all config sections from:
  1. `$datadir/pipewire/client-rt.conf.d/`
  2. `$sysconfdir/pipewire/client-rt.conf.d/`
  3. `$XDG_CONFIG_HOME/pipewire/client-rt.conf.d/`

Configuration — sections (1)

- `context.properties` configures the PipeWire instance.
- Most properties target the daemon (`default.clock.allowed-rates`, `default.clock.max-quantum`, etc.) but some also apply to other clients (`log.level`, `mem.mlock-all`, etc.).

```
context.properties = {
    link.max-buffers = 16
    log.level        = 2

    core.daemon = true        # listening for socket connections
    core.name   = pipewire-0  # core name and socket name

    # Properties for the DSP configuration.
    default.clock.rate          = 48000
    default.clock.allowed-rates = [ 48000 ]
    default.clock.quantum       = 1024
    default.clock.min-quantum   = 32
    default.clock.max-quantum   = 2048
    default.clock.quantum-limit = 8192
    # ...
}
```

Configuration — sections (2)

- `context.spa-libs` maps plugin features with globs to a SPA library.
- That defines the shared object to be used to implement the given factories. A way to look at this is that keys are interfaces used by PipeWire for various features, and values are the shared objects that implement those.

```
context.spa-libs = {
    # <factory-name regex> = <library-name>
    # Maps a SPA factory to its parent library.

    audio.convert.* = audioconvert/libspa-audioconvert
    avb.*           = avb/libspa-avb
    api.alsa.*      = alsa/libspa-alsa
    api.v4l2.*      = v4l2/libspa-v4l2
    api.libcamera.* = libcamera/libspa-libcamera
    api.bluez5.*    = bluez5/libspa-bluez5
    api.vulkan.*    = vulkan/libspa-vulkan
    api.jack.*      = jack/libspa-jack
    support.*       = support/libspa-support
    # ...
}
```

Configuration — sections (3)

- `context.modules` is an array of dictionaries. It lists modules to instantiate, with optional arguments (`args`), `flags` and a conditional expression (`condition`).
- A module can be loaded more than once: it will be instantiated multiple times.
- Two flags exist to turn panics into warnings:
  1. `ifexists` on unknown modules;
  2. `nofail` on module init failures.

```
context.modules = [
    # { name = <module-name>
    #     ( args  = { <key> = <value> ... } )
    #     ( flags = [ ( ifexists ) ( nofail ) ] )
    #     ( condition = [ { <key> = <value> ... } ... ] )
    # }

    # ...
]
```

Configuration — sections (4)

- `context.modules` example:

```
context.modules = [
    # The profiler module. Allows application to access profiler
    # and performance data. It provides an interface that is used
    # by pw-top and pw-profiler.
    { name = libpipewire-module-profiler }

    # Uses realtime scheduling to boost the audio thread
    # priorities. This uses RTKit if the user doesn't have
    # permission to use regular realtime scheduling.
    { name = libpipewire-module-rt
        args = {
            nice.level    = -11
            #rt.prio      = 88
            #rt.time.soft = -1
            #rt.time.hard = -1
        }
        flags = [ ifexists nofail ]
    }

    # ...
]
```

Configuration — sections (5)

- `context.objects` is an array of dictionaries. It lists objects that should be statically created by this client. This requires a `factory` to be used and arguments (`args`) to be passed to it.
- As previously, the `flags` property can configure the reaction to errors. For `context.objects`, only `nofail` exists.
- `condition` also exists for this section.

```
context.objects = [
    # { factory = <factory-name>
    #     ( args  = { <key> = <value> ... } )
    #     ( flags = [ ( nofail ) ] )
    #     ( condition = [ { <key> = <value> ... } ... ] )
    # }

    # ...
]
```

Configuration — sections (6)

- `context.objects` example:

```
context.objects = [
    # Create a fake source node. It will be stereo
    # because of its audio.position property.
    { factory = adapter
        args = {
            factory.name     = support.null-audio-sink
            node.name        = "my-mic"
            node.description = "Microphone"
            media.class      = "Audio/Source/Virtual"
            audio.position   = "FL,FR"
        }
    }

    # ...
]
```

Configuration — sections (7)

- `context.exec` is an array of dictionaries. Each entry is an executable that will be run on startup of the client as a child process.
- This used to be the recommended way to run the session & policy manager. This changed and the recommended way is to rely on your init system, be it systemd or any other.
- Using this section is therefore **deprecated**, except for simple development environments.

```
context.exec = [
    { path = "/usr/bin/pipewire-media-session"
        args = ""
        condition = [
            { exec.session-manager = null }
            { exec.session-manager = true }
        ] }
]
```

Tools rundown

Tools rundown — the PIPEWIRE_DEBUG variable

- Every client listens to the PIPEWIRE_DEBUG environment variable which allows overwriting the log.level from the configuration file.
- It takes as value the log level:
  - 0 or X: No logging is enabled.
  - 1 or E: Error logging is enabled.
  - 2 or W: Warnings are enabled.
  - 3 or I: Informational messages are enabled.
  - 4 or D: Debug messages are enabled.
  - 5 or T: Trace messages are enabled.
- This should be **the first debugging step** to increase verbosity and therefore better understand why a PipeWire client is facing issues. Careful with PIPEWIRE_DEBUG=5 which most likely will cause underruns issues. Level 3 is often good enough for debugging.

Tools rundown — pw-config

- pw-config is a small utility that allows dumping a given config file, taking into account its overrides. It is best used to ensure config changes are effective and overrides are applied as we expect.
- pw-config paths lists config paths, including overrides.
- pw-config list details all config sections.

```
$ pw-config --name custom.conf paths
{
  "config.path": "/usr/share/pipewire/custom.conf",
  "override.1.0.config.path": "/home/tleb/.config/pipewire/custom.conf.d/alsa-udev.conf",
  "override.1.1.config.path": "/home/tleb/.config/pipewire/custom.conf.d/source-rnnoise.conf"
}
```

Tools rundown — pw-dump

- `pw-dump` prints the graph as a JSON array of all exported objects known to `Core`.
- Its main goal is to allow sharing the graph's overall state when reporting a bug or describing a situation.
- Filtering: `pw-dump` takes a parameter which can be an object type (careful, it must be capitalised), ID or name (`object.path`, `object.serial` or `*.name`).
- Its output is rather verbose and for more interactive debugging sessions, `pw-cli` is more adapted.

Tools rundown — pw-cli (1)

- pw-cli is the main command-line interface tool to interact with PipeWire. It connects to PipeWire as a new client.
- It has two modes: (1) it can either answer to commands given as argument such as pw-cli help and stop afterwards or (2) run in interactive mode when given no argument. In that second mode, it also logs new objects that join the core object list.
- pw-cli help lists all existing commands. It includes arguments (inbetween square brackets when optional) and command aliases.

Tools rundown — pw-cli (2)

- It can expose many information about the graph:
  - pw-cli ls [<filter>] lists objects with their ID, type and a few of their core properties. <filter> is the same as pw-dump's argument.
  - pw-cli info <filter> gives all possible information about a given object. That includes all of its properties and params.
  - pw-cli enum-params <filter> <param-id> gives the content of a param associated with an object.
- But, it also allows modifying objects:
  - pw-cli set-param <filter> <param-id> <param-json> to edit a param value;
  - pw-cli permissions <client-id> <object> <permission> to modify permissions on a given object.
- As well as creating objects dynamically, that will be hosted by the pw-cli client: load-module, create-device, create-node, create-link.

Tools rundown — pw-top

- top for PipeWire.
- Appropriate tool to get a quick overview of the current graph nodes and structure.
- Status: S for stopped and R for running.

```
S   ID  QUANT   RATE    WAIT    BUSY   W/Q   B/Q  ERR FORMAT            NAME
S   28      0      0     ---     ---   ---   ---    0                  Dummy-Driver
S   29      0      0     ---     ---   ---   ---    0                  Freewheel-Driver
S   37      0      0     ---     ---   ---   ---    0                  Midi-Bridge
S   53      0      0     ---     ---   ---   ---    0                  v4l2_input.pci-0000_05_00.0-usb-0_2.3_1.2
S   55      0      0     ---     ---   ---   ---    0                  v4l2_input.pci-0000_00_14.0-usb-0_6_1.0
S   57      0      0     ---     ---   ---   ---    0                  v4l2_input.pci-0000_00_14.0-usb-0_6_1.2
S   59      0      0     ---     ---   ---   ---    0                  libcamera_input.__SB_.PC00.TRP0.PXSX-2.3_1.2-046d_0826
S   61      0      0     ---     ---   ---   ---    0                  libcamera_input.__SB_.PC00.XHCI.RHUB.HS06-6_1.0-0c45_6a15
S   63      0      0     ---     ---   ---   ---    0                  libcamera_input.__SB_.PC00.XHCI.RHUB.HS06-6_1.2-0c45_6a15
S   69      0      0     ---     ---   ---   ---    0                  alsa_output.usb-CalDigit__Inc._CalDigit_Thunderbolt_3_Audio-00.analog-stereo
S   70      0      0     ---     ---   ---   ---    0                  alsa_input.usb-CalDigit__Inc._CalDigit_Thunderbolt_3_Audio-00.analog-stereo
R   71    256  48000 174.2us   0.4us  0.03  0.00    0    S16LE 1 48000 alsa_input.usb-FuZhou_Kingwayinfo_CO._LTD_TONOR_TC30_Audio_Device_20200707-00.mono-fallback
R   95      0  48000 125.7us   3.5us  0.02  0.00    0    S16LE 2 48000  + pw-record
S   35      0      0     ---     ---   ---   ---    0                  alsa_input.usb-046d_HD_Webcam_C525_D1B361B0-00.mono-fallback
S   68      0      0     ---     ---   ---   ---    0                  alsa_output.pci-0000_00_1f.3.hdmi-stereo
R   88   2048  48000 295.3us 213.6us  0.01  0.01    0    S16LE 2 48000 bluez_output.60_AB_D2_44_9A_BF.1
R   84   8192  44100 147.1us  60.0us  0.00  0.00    0    F32LE 2 44100  + spotify
```

Tools rundown — pw-profiler

- Allows profiling of all running nodes: it records many time durations while running then generates graphs once the command is stopped.
- Here is an example with a single `pw-play` node, first started with `PIPEWIRE_CONFIG_NAME` equal to `client.conf` then with `client-rt.conf` on a loaded system.

Tools rundown — pw-dot

- `pw-dot` creates a file named `pw.dot` which is a Graphviz textual graph description file (DOT).
- By default, it connects to the PipeWire daemon and creates a graph representation of the global objects. It can also work from the output of `pw-dump` using the `--json` flag.
- That file can be turned into a graphical representation and viewed on a host using: `dot -Tsvg pw.dot > pw.svg && xdg-open pw.svg`

Tools rundown — pw-cat

- Aliased to `pw-play`, `pw-record` and others, it is a simple tool to play or record media files.
- It uses libsndfile for a large audio format support.
- It has many options available to control the exposed props and params:
  - `--target` allows asking to be routed to a given node;
  - `--latency` asks for a given latency (therefore buffer size);
  - `--quality` controls the adaptive resampling;
  - `--rate`, `--channels`, `--channel-map`, `--format`, `--volume` are self-describing.
  - etc.

Tools rundown — and a few others

- `pw-link`: it allows listing, creating and deleting links.
- `pw-mon`: it monitors and dumps various events: it prints when a global object is added or removed, displays information relative to the `Core`, etc.
- `pw-loopback`: it creates two nodes that act as a virtual loopback.
- `pw-metadata`: it allows editing metadata, which are runtime-writable settings stored by the daemon. The allowed rates and quantum can be controlled at runtime using that method.

Tools rundown — helvum (1)

- Helvum is a real-time 2D patchbay.
- It gives an overview of the graph with the existing nodes and their ports. It also can create and delete links, allowing manual editing of the graph.

Tools rundown — helvum (2)

- Helvum is a GUI software. We can however run it on our host and monitor our target if we have networking on the target.
- We use socat on the target to bridge the Unix socket from our target daemon over TCP/IP. We then use socat on the host to bridge the TCP/IP to a Unix socket that we will use as our PipeWire Unix socket for Helvum.

Tools rundown — helvum (3)

```
# We run socat on the target, creating a redirection from the Unix
# socket /run/pipewire-0 to a TCP/IP server on port 8000.
ssh $login@$ip "socat TCP4-LISTEN:8000 UNIX-CONNECT:/run/pipewire-0" &

# We run socat on the host, creating the redirection from the TCP/IP
# port 8000 on the target to the Unix socket /tmp/pipewire-0 on the
# host.
socat UNIX-LISTEN:/tmp/pipewire-0 TCP4:$ip:8000 &

# And we connect on the redirected Unix socket.
PIPEWIRE_RUNTIME_DIR=/tmp helvum
```

Demo 1 — running PipeWire

Demo 1 — introduction

- Demo time!
- We will play audio to an `alsa-lib` device from an audio file.
- We will let our session manager discover ALSA devices and connect an output node to the ALSA sink node.
- The steps will be:
  1. Start a PipeWire daemon;
  2. Start a WirePlumber daemon;
  3. Start a `pw-play` client;
  4. Study the graph status using various tools (`pw-dot`, `pw-top`, `pw-cli`, etc).

Demo 1 — pointers

1. Start a PipeWire daemon.
   - Running `pipewire` without arguments will start a client using `pipewire.conf`, which by default runs in daemon mode.
   - At this state, the graph is rather empty. Objects are mostly modules and factories attached to the core client, and the client objects.
2. Start a WirePlumber daemon.
   - It also picks its config automatically, no arguments required.
   - Once started, we can notice that ALSA devices and attached nodes are created in the graph.
   - Its log level is controlled using `WIREPLUMBER_DEBUG`.
3. Start a `pw-play` client;
   - `pw-play <file>`
4. Study the graph status using various tools (`pw-dot`, `pw-top`, `pw-cli`, etc).

Demo 2 — PipeWire filter-chains

Demo 2 — introduction

- We will keep our previous setup, but add a client that does equalization on the samples.
- The steps will be:
  1. To create a new configuration file, for the client hosting the effect;
  2. Start a client using this config;
  3. Update links manually to make `pw-play` be routed to the effect, then to the ALSA sink node.

Demo 2 — pointers

1. To create a new configuration file, for the client hosting the effect.
   - Recent PipeWire versions have a `filter-chain.conf` example with snippets for various needs (LADSPA with RNNoise, builtin effects, etc.).
   - When modules spawn objects, they often give their own properties to children, and take arguments to set specific properties for each node. See `capture.props` and `playback.props`.
2. Start a client using this config.
   - `pipewire -c filter-chain.conf`
3. Update links manually to make `pw-play` be routed to the effect, then to the ALSA sink node.
   - This can be done using Helvum with its GUI.
   - Otherwise, `pw-dot` or `pw-link --links` to get an overview then `pw-link <output-port> <input-port>` to create a new link.

WirePlumber

WirePlumber — session manager concept

- **PipeWire** handles the processing of the media graph.
- An additional layer is required to implement the desired configuration of devices and the connections between nodes. That is implemented by the **session & policy manager**.
- Two known open-source implementations exist:
  - pipewire-media-session: the initial implementation, deprecated;
  - WirePlumber: recommended implementation.
- **WirePlumber** implements a modular approach: it provides a high-level API and exposes it to Lua scripts. Those implement the management logic.
- Technical stack: C, GLib (GObject), Lua engine, Meson & Ninja.
- Documentation.

WirePlumber — default behavior

- **WirePlumber** has a default behavior that tries to replicate the PulseAudio behavior, i.e. a desktop setup.
- It enumerates and adds `Device` objects for ALSA, BlueZ and others. It also puts those devices into a best-guess profile.
- Those devices get their associated nodes created automatically.
- Audio routing is based on two default nodes:
  - An `Audio/Sink` node is for applications that want to emit audio. All `Output/Audio` nodes get routed to it.
  - An `Audio/Source` node is for applications that require a microphone input. All `Input/Audio` nodes get routed to it.
- Nodes can also request to be routed to:
  1. a target node using `target.object` (for example `pw-cat --target`);
  2. nothing automatically using `node.autoconnect`. WirePlumber will not create any automatic link, letting any PipeWire client create the desired links;
  3. a target node using `target.object` inside `default` metadata (at runtime).

WirePlumber — configuration (1)

- The config lookup logic is the same as PipeWire's.
- Configuration files define components to load when starting. Components include PipeWire modules, WirePlumber modules and **Lua scripts**.
- Lua scripts will **monitor** the PipeWire graph, and **react to events**. Configuration files also defines options passed to scripts, including arrays matching object descriptions to behavior for such objects.
- The default configuration is called `wireplumber.conf`, see `/usr/share/wireplumber/wireplumber.conf`. Default scripts are located alongside, in `scripts/`.

WirePlumber — configuration (2)

- Let's look at the configuration of the ALSA monitor in `scripts/monitors/alsa.lua`:

```
-- /usr/share/wireplumber/scripts/monitors/alsa.lua
config = {}
config.reserve_device = Core.test_feature ("monitor.alsa.reserve-device")
config.properties = Conf.get_section_as_properties ("monitor.alsa.properties")
config.rules = Conf.get_section_as_json ("monitor.alsa.rules", Json.Array {})
```

WirePlumber — configuration (3)

- To configure ALSA monitor, using a custom configuration fragments:

```
-- /etc/wireplumber/wireplumber.conf.d/20-alsa-config.conf
monitor.alsa.properties = {
  -- See documentation: man pipewire-devices(7)
  alsa.udev.expose-busy = true
}

monitor.alsa.rules = [
  {
    matches = [ { device.name = "~alsa_card.*" } ]
    actions = {
      update-props = {
        api.alsa.use-acp  = true
        api.acp.auto-port = false
        device.disabled   = false
      }
    }
  }
}
```

WirePlumber — configuration (4)

- Example of disabling D-Bus support:

```
-- /etc/wireplumber/wireplumber.conf.d/10-disable-dbus.conf
wireplumber.profiles = {
   main = {
      support.dbus = disabled

      # Avoid warnings
      support.reserve-device = disabled
      support.portal-permissionstore = disabled
      script.client.access-portal = disabled
      monitor.alsa.reserve-device = disabled
   }
}
```

WirePlumber — permission handling

- Another task of the session & policy manager is **permission management**.
- That is handled, in PipeWire >= 0.3.83, using two PipeWire daemon sockets:
  - Clients joining `pipewire-0-manager` have full permissions, seen using property `pipewire.access = "unrestricted"`.
  - Client joining `pipewire-0` must be given permissions by the session manager, i.e. WirePlumber. Propery `pipewire.access` is `"default"`.
- Permissions can be granted on a per-object-basis for each client. Else each client has a default permission assigned to it.

WirePlumber — DSP filtering on sinks and sources (1)

- WirePlumber now features a way to attach a filter-chain to a sink or source. This allows pre-processing before outputs for example.
- This is called "Automatic Software DSP", see documentation about the topic.
- Script implementing this behavior is `scripts/node/software-dsp.lua`.

WirePlumber — DSP filtering on sinks and sources (2)

- Example configuration:

```
node.software-dsp.rules = [
  {
    matches = [
      { "node.name" = "alsa_output.platform-sound.HiFi__Speaker__sink" }
      { "alsa.id" = "~WeirdHardware*" }
    ]

    actions = {
      create-filter = {
        # Arguments passed to libpipewire-module-filter-chain
        # For inspiration, look into /usr/share/pipewire/filter-chain/*.conf
        filter-path = "/etc/wireplumber/filter-config.json"
        hide-parent = true
      }
    }
  }
]
```

Demo 3 — interacting with WirePlumber

WirePlumber — demo time! (1)

- We'll use our previous setup, focusing on WirePlumber abilities.
- The steps will be:
  1. Start PipeWire and WirePlumber;
  2. Target a specific node;
  3. Modify the default playback node;
  4. Have a look at device profiles.

WirePlumber — demo time! (2)

1. Start PipeWire and WirePlumber.
   - See demo 1 for explainations.
2. Target a specific node.
   - This is done by nodes using `target.object` (previously `node.target`).
   - It can be a node ID, node name or object path (see WirePlumber scripts for the logic).
   - A node's properties are controlled when spawning it, so by its config or by its client (WirePlumber for example).
3. Modify the default playback node.
   - `wpctl set-default <id>` controls this.
   - Nodes must have `media.class` equal to `Audio/Sink` (or similar) to appear in this list. That does not include filter-chains, which are handled specifically.
4. Have a look at device profiles.
   - Those are params on the device objects. See `EnumProfile` and `Profile`.

C API

C API — introduction

- `libpipewire`: reference implementation, and currently the only one.
- Allows connecting to the daemon as a client.
- Rust bindings: pipewire-rs.
- See `pkg-config` for CFLAGS and LDFLAGS:

  ```
  $ pkg-config --cflags --libs libpipewire-0.3
  ```

- To initialise the library (logging, randomness, etc.), call:

  ```
  void pw_init(int *argc, char **argv[]);
  ```

C API — SPA

- A building block is worth mentioning, **Simple Plugin API** (SPA). It contains the following:
  - A plugin format encapsulating shared objects, allowing runtime introspection of the plugin content.
  - A Type-Length-Value data container called POD. It is header-only, with support for basic types (int, float, string, etc.) and nested types (array, struct, objects).
  - Utility functions as header-only: string handling utilities, relaxed JSON parsing (used for config files), a ringbuffer implementation, etc.
  - Support interfaces provided by the system, with multiple possible implementations: logging, file-descriptor polling, etc.
- Platform resources (ALSA, bluez5, vulkan, etc.) are exposed as SPA plugins and used internally by PipeWire or WirePlumber.

C API — event-loop

- At the core of each client: an `epoll(2)`-based event-loop is running.
- `pw_main_loop` is a wrapper around `pw_loop` providing a simple-to-use API.

```
/** Create a new main loop. */
struct pw_main_loop *
pw_main_loop_new(const struct spa_dict *props);

/** Get the loop implementation */
struct pw_loop * pw_main_loop_get_loop(struct pw_main_loop *loop);

/** Destroy a loop */
void pw_main_loop_destroy(struct pw_main_loop *loop);

/** Run a main loop. This blocks until \ref pw_main_loop_quit is called */
int pw_main_loop_run(struct pw_main_loop *loop);

/** Quit a main loop */
int pw_main_loop_quit(struct pw_main_loop *loop);
```

C API — context

- A `pw_context` instance is at the heart of the C API. It allows connection to the daemon and it manages locally available resources.
- It does the following:
  - Parsing of the appropriate configuration.
  - Start of the processing thread & associated data loop.
  - Handling of local resources: memory pool, work queue, **proxies**, local modules

```
/** Make a new context object for a given main_loop */
struct pw_context * pw_context_new(struct pw_loop *main_loop,
                struct pw_properties *props, size_t user_data_size);

/** Connect to a PipeWire instance */
struct pw_core * pw_context_connect(struct pw_context *context,
                struct pw_properties *properties, size_t user_data_size);
```

C API — proxies

- Think as **proxies** as file descriptors for PipeWire objects. They are local references to global PipeWire objects.
- The equivalent on the daemon side are called **resources**.
- A client starts with two proxies:
  1. One pointing to the `Core` object.
  2. Another one to the global `Client` object that represents itself.

C API — registry

- The PipeWire daemon handles a list of objects. Those are known as **global** objects and are represented by `pw_global` structures.
- `pw_registry` is a singleton structure that allows clients to track existing globals. It works by registering a callback to be called on new global object events.

```
struct pw_registry_events {
#define PW_VERSION_REGISTRY_EVENTS  0
        uint32_t version;
        void (*global) (void *data, uint32_t id, uint32_t permissions,
                const char *type, uint32_t version,
                const struct spa_dict *props);
        void (*global_remove) (void *data, uint32_t id);
};

struct pw_registry * pw_core_get_registry(struct pw_core *core,
        uint32_t version, size_t user_data_size);

void pw_registry_add_listener(struct pw_registry *registry,
        struct spa_hook *hook, struct pw_registry_events *events,
        void *data);
```

C API — example 1, monitoring global objects

```c
/* We will run indefinitely, getting events for each added and removed global
 * object.
 *
 * An influx of Registry::Global events will come in at the start to list all
 * already-existing globals. Use the Core::Sync method and Core::Done event to
 * know when that initial sync is done. See pw_core_sync(). */
#include <pipewire/pipewire.h>

static void registry_event_global(void *data, uint32_t id, uint32_t permissions,
                const char *type, uint32_t version, const struct spa_dict *props) {
        printf("object added: id:%u\ttype:%s/%d\n", id, type, version);
}

static void registry_event_global_remove(void *data, uint32_t id) {
        printf("object removed: id:%u\n", id);
}

static const struct pw_registry_events registry_events = {
        PW_VERSION_REGISTRY_EVENTS,
        .global = registry_event_global,
        .global_remove = registry_event_global_remove,
};
```

C API — example 1, monitoring global objects

```c
int main(int argc, char **argv) {
  pw_init(&argc, &argv);

  struct pw_main_loop *loop = pw_main_loop_new(NULL);
  struct pw_context *context = pw_context_new(pw_main_loop_get_loop(loop), NULL, 0);
  struct pw_core *core = pw_context_connect(context, NULL, 0);
  struct pw_registry *registry = pw_core_get_registry(core, PW_VERSION_REGISTRY, 0);

  struct spa_hook registry_listener;
  spa_zero(registry_listener);
  pw_registry_add_listener(registry, &registry_listener, &registry_events, NULL);

  pw_main_loop_run(loop);

  pw_proxy_destroy((struct pw_proxy*)registry);
  pw_core_disconnect(core);
  pw_context_destroy(context);
  pw_main_loop_destroy(loop);

  return 0;
}
```

C API — node implementations

- Implementing a raw node is not straight-forward, requiring to implement many book-keeping methods (see `struct spa_node_methods`).
- PipeWire provides two abstractions for implementing nodes:
  - `pw_filter`: DSP-type work, works on raw `f32` samples, without additional buffering.
  - `pw_stream`: more high level, it provides the following features:
    - **Buffering:** a stream can emit more samples than the current cycle quantum and those will be buffered.
    - **Format negociation:** the client can expose multiple supported formats and negociation will occur when changing from idle to running.
    - **Format conversion:** sample type, planar/interleaved, channel mapping, rate resampling.
- See example implementations of source nodes:
  - Filter: `src/examples/audio-dsp-src.c`
  - Stream: `src/examples/audio-src.c`

C API — pw_filter process event

```c
static void on_process(void *userdata, struct spa_io_position *position) {
  struct data *data = userdata;
  double *acc = data->out_port->accumulator;
  uint64_t n_samples = position->clock.duration;

  /* Fetch the sample buffer. The first argument is the port user data
   * (as returned by pw_filter_add_port), it is used to identify our
   * port (think container_of). */
  float *out = pw_filter_get_dsp_buffer(data->out_port, n_samples);
  if (out == NULL)
    return;

  for (uint64_t i = 0; i < n_samples; i++) {
    *acc += 2 * M_PI * 440 / 44100;   /* Grow our accumulator */
    *acc = remainder(*acc, 2 * M_PI); /* Avoid overflows */
    *out++ = sin(*acc) * 0.7;         /* Compute a sample */
  }
}

static const struct pw_filter_events filter_events = {
  PW_VERSION_FILTER_EVENTS,
  .process = on_process,
};
```

C API — pw_stream process event

```c
static void on_process(void *userdata) {
  struct data *data = userdata;

  struct pw_buffer *b = pw_stream_dequeue_buffer(
      data->stream);
  assert(b != NULL);

  struct spa_buffer *buf = b->buffer;
  uint8_t *p = buf->datas[0].data;
  assert(p != NULL);

  int stride = sizeof(float) * CHANNELS;
  int n_frames = SPA_MIN(b->requested,
      buf->datas[0].maxsize / stride);

  fill_f32(&data->accumulator, p, n_frames);

  buf->datas[0].chunk->offset = 0;
  buf->datas[0].chunk->stride = stride;
  buf->datas[0].chunk->size = n_frames * stride;

  pw_stream_queue_buffer(data->stream, b);
}
```

```c
#define CHANNELS 2
#define FREQ     440
#define RATE     44100

static void fill_f32(float *acc, float *dest,
    int n_frames) {
  for (int i = 0; i < n_frames; i++) {
    *acc += M_PI_M2 * FREQ / RATE;
    *acc = remainder(*acc, 2 * M_PI);

    float val = sin(*acc) * 0.7;
    for (int c = 0; c < CHANNELS; c++)
      *dst++ = val;
  }
}

static const
struct pw_stream_events stream_events = {
  PW_VERSION_STREAM_EVENTS,
  .process = on_process,
};
```

Going further

Going further

- For MIDI support, see `pw-cat --midi`, `pw-mididump` and the documentation.
- For the PulseAudio compatibility layer, see module-protocol-pulse and this documentation page.
- For the JACK compatibility layer, look at `pw-jack`.
- For video support, see many examples in `src/examples/`.
- For audio over IP, see modules `roc-*`, `pulse-tunnel`, `netjack2-*`, `rtp-*`, `protocol-simple`, `avb`.
- To understand why timer-based audio scheduling (`tsched`) is useful, see this blog post.

GStreamer

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Introduction

- `Gstreamer` is an open-source multimedia framework that provides a pipeline-based architecture for handling multimedia data such as audio and video.
- https://gstreamer.freedesktop.org/
- `GStreamer` provides a unified framework for handling various multimedia formats and tasks.
- It supports a wide range of codecs, formats, and protocols.
- Its modular architecture supports plugins and allows the addition of new elements, codecs, and functionality.

Architecture

- `GStreamer` is object oriented, it adheres to the `GObject` model of `GLib 2.0`.
- The main object is an `Element`. Each element has a specific function e.g. reading, writing, encoding or decoding data. By chaining elements, its is possible to create a `pipeline` to achieve a task.
- Elements communicate with each other through `pads`. A pad is a connection point that can be an input (`sink`) or output (`source`). Elements are linked by connecting pads. A pad can restrict the type of data that flows through it. Links are only allowed between two pads when the allowed data types (capabilities) of the two pads are compatible.
- A `bin` is a container for a collection of elements. It can be controlled just like an element
- A `pipeline` is a top level bin. Allowing to control and synchronize all its children.

Example

*E*xample of a GStreamer pipeline

Plugins

- Plugins are selfcontained libraries loaded at runtime.
- All relevant aspects of plugins can be queried at run-time.
- All the properties can be set using the GObject properties, there is no need for header files.
- Core plugins:
  - audiotestsrc, videotestsrc: Generates test audio or video patterns.
  - autoaudiosink, autovideosink: Automatically selects an output and plays audio or displays video.
  - filesrc, filesink: Read from and write to files.
  - decodebin: Automatically selects and configures decoders based on media content.
  - playbin: Automatically plays audio and video from a location

Useful Plugins

alsasink	Sink Audio	Output to a sound card via ALSA
alsasrc	Source Audio	Read from a sound card via ALSA
audioconvert	Filter Converter Audio	Convert audio to different formats
audiodynamic	Filter Effect Audio	Compressor and Expander
audiolatency	Audio Util	Measures the audio latency between the source and the sink
audioloudnorm	Filter Effect Audio	Normalizes perceived loudness of an audio stream
audiomixmatrix	Filter Audio	Mixes a number of input channels into a number of output channels according to a transformation matrix
audioresample	Filter Converter Audio	Resamples audio
clocksync	Generic	Synchronise buffers to the clock
dtmfdetect	Filter Analyzer Audio	This element detects DTMF tones
dtmfsrc	Source Audio	Generates DTMF tones
jackaudiosink	Sink Audio	Output audio to a JACK server
jackaudiosrc	Source Audio	Captures audio from a JACK server

Command line tools

- `gst-inspect-1.0` is a tool that prints out information on GStreamer plugins and elements.
- Without any arguments, it prints a list of all plugins and elements it knows about.
- `gst-launch-1.0` builds and runs a GStreamer pipeline on GStreamer plugins and elements.
- It takes a pipeline description as an argument, this is a list of elements separated by exclamation marks (!). Properties may be appended to elements in the form property=value.
- `gst-launch-1.0` is a tool useful for debugging but shouldn't be used as a standalone application.
- For example, to play an ogg file using ALSA:
  `gst-launch-1.0 filesrc location=music.ogg ! oggdemux ! vorbisdec ! audioconvert ! audioresample ! alsasink`

Debugging

- `gst-launch-1.0` has a `-v` option to make it verbose
- GStreamer also uses the `GST_DEBUG` environment variable. It takes a debug level from 0 (none) to 9 (memdump). This can also be filtered by element and categories. For example, `GST_DEBUG=2,audiotestsrc:6`, will use level 6 for the `audiotestsrc` element, and 2 for all the others.
- When `GST_DEBUG_DUMP_DOT_DIR` environment variable is set and point to a folder, `gst-launch-1.0` will create a `.dot` file at each state change. `graphviz` can then be used to generate a graph.
  - `gst-launch-1.0 filesrc location=Media/test_32_16.wav ! decodebin ! alsasink`
  - `dot -Kfdp -Tpng -o pipeline.png 0.00.00.021721659-gst-launch.PAUSED_PLAYING.dot`

Debugging - graph

Resources

- Documentation: https://gstreamer.freedesktop.org/documentation/. This includes documentation of the API to write application and plugins.
- Plugin list: https://gstreamer.freedesktop.org/documentation/plugins_doc.html

Demo - Gstreamer

- Inspect plugins and elements using `gst-inspect`
- Prepare multiple pipelines with `gst-launch`

Last slides

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Last slide

Thank you!

And may the Source be with you

Rights to copy

© Copyright 2004-2025, Bootlin
License: Creative Commons Attribution - Share Alike 3.0
https://creativecommons.org/licenses/by-sa/3.0/legalcode
You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

- **Attribution**. You must give the original author credit.
- **Share Alike**. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.
- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

Document sources: https://github.com/bootlin/training-materials/