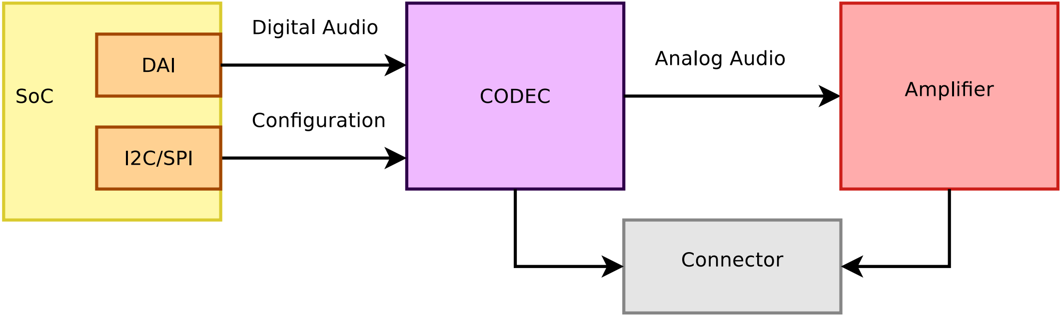
ASoC: Supporting Audio on an Embedded Board

Alexandre Belloni
Bootlin
alexandre.belloni@bootlin.com

Alexandre Belloni

- Embedded Linux engineer at Bootlin
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
- Open-source contributor
  - Maintainer for the Linux kernel **RTC subsystem**
  - Co-Maintainer of **kernel support for Atmel ARM processors**

Anatomy



SoC
DAI
I2C/SPI
Digital Audio
Configuration
CODEC
Analog Audio
Amplifier
Connector

Anatomy

- codec configuration usually happens on a simple serial bus, I2C or SPI.
- SoC DAI: the SoC Digital Audio Interface.
  - sometimes called synchronous serial interface
  - provides audio data to the codec
  - formats are usually AC97, I2S, PCM (TDM, network mode), DSP A/B
  - Examples: Atmel SSC, NXP SSI, TI McASP.
  - Some SoCs have a separate SPDIF controller
- Amplifier is optional

Some SoCs (Allwinner A33, Atmel SAMA5D2) have the codec and the amplifier on the SoC itself.

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.
- Machine drivers: board specific driver that serves as a glue between the SoC interface driver and the codec driver. It describes how both are connected. If you properly selected your hardware components, this is the only driver that needs to e written.

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

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` 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.
         */
        const char *cpu_name;
        struct device_node *cpu_of_node;
        /*
         * 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.
         */
        const char *cpu_dai_name;
```

```
/*
 * You MUST specify the link's codec, either by device name, or by
 * DT/OF node, but not both.
 */
const char *codec_name;
struct device_node *codec_of_node;
/* You MUST specify the DAI name within the codec */
const char *codec_dai_name;

struct snd_soc_dai_link_component *codecs;
unsigned int num_codecs;
```

```
        /*
         * You MAY specify the link's platform/PCM/DMA driver, either by
         * device name, or by DT/OF node, but not both. Some forms of link
         * do not need a platform.
         */
        const char *platform_name;
        struct device_node *platform_of_node;
        int id;        /* optional ID for machine driver link identification */

        const struct snd_soc_pcm_stream *params;
        unsigned int num_params;

        unsigned int dai_fmt;           /* format to set on init */
};
```

Example 1

sound/soc/atmel/atmel_wm8904.c

```
static struct snd_soc_dai_link atmel_asoc_wm8904_dailink = {
	.name = "WM8904",
	.stream_name = "WM8904 PCM",
	.codec_dai_name = "wm8904-hifi",
	.dai_fmt = SND_SOC_DAIFMT_I2S
		| SND_SOC_DAIFMT_NB_NF
		| SND_SOC_DAIFMT_CBM_CFM,
	.ops = &atmel_asoc_wm8904_ops,
};

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->cpu_of_node = cpu_np;
        dailink->platform_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->codec_of_node = 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->cpu_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);
[...]
}
```

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`.

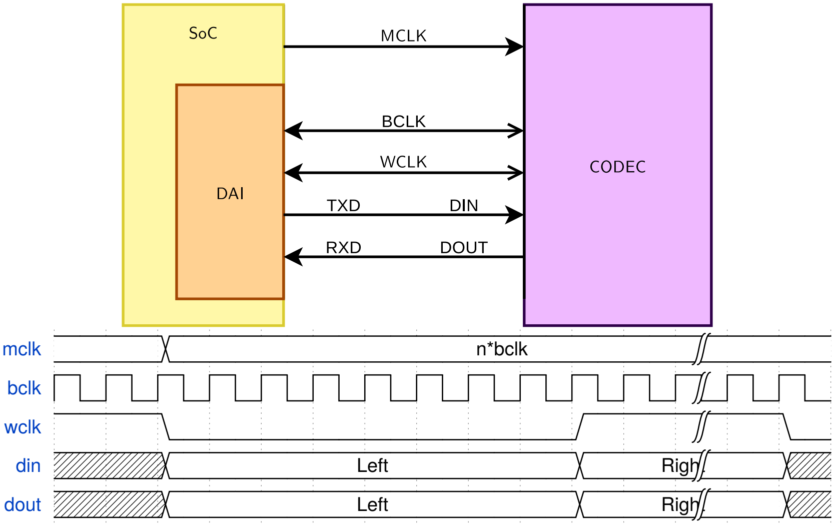
`sound/soc/atmel/atmel_wm8904.c`

```
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

- depending on characteristics (sample rate, bit depth) of the audio to be played or recorded, it may be necessary to reconfigure the clocks
- one of the most difficult part when writing a machine driver
- the SoC or the codec may be driving the clocks (master)

Clocking: I²S



Clocking

- MCLK is the codec clock. The IC needs it to be working. Some codecs are able to use BLCK as their clock. It is sometimes referred as the system clock.
- BCLK is the bit clock. It is provided by the *master* (either the SoC or the codec)
- WCLK is the word clock. It is often called LRCLK (Left Right clock) or FCLK/FSCLK (Frame clock). It is provided by the *master* (either the SoC or the codec). Its rate is the sample rate.
- DIN and DOUT are the data lines
- The relationship between BLCK and WCLK is: $Bclk = Wclk * Nchannels * BitDepth$
- usually the codecs will expect MCLK to be a multiple of BCLK. Usually specified as a multiple of Fs.
- however, some codec have a great set of PLLs and dividers, allowing to get a precise BCLK from many different MCLK rate
- quite often, not the case for the SoC, then use the codec as master!

Clocking: master/slave

The master/slave relationship is declared part of the `.dai_fmt` field of `struct snd_soc_dai_link`.

include/sound/soc.h

```
/*
 * DAI hardware clock masters.
 *
 * This is wrt the codec, the inverse is true for the interface
 * i.e. if the codec is clk and FRM master then the interface is
 * clk and frame slave.
 */
#define SND_SOC_DAIFMT_CBM_CFM          (1 << 12) /* codec clk & FRM master */
#define SND_SOC_DAIFMT_CBS_CFM          (2 << 12) /* codec clk slave & FRM master */
#define SND_SOC_DAIFMT_CBM_CFS          (3 << 12) /* codec clk master & frame slave */
#define SND_SOC_DAIFMT_CBS_CFS          (4 << 12) /* codec clk & FRM slave */
```

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;
        }
```

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,
};
```

simple-card

You may not need to know all that!

- A driver, `simple-card`, can be configured from device tree.
- It allows to specify the connection between the SoC and the codec.
- Documented in `Documentation/devicetree/bindings/sound/simple-card.txt`

```
sound {
        compatible = "simple-audio-card";
        simple-audio-card,name = "VF610-Tower-Sound-Card";
        simple-audio-card,format = "left_j";
        simple-audio-card,bitclock-master = <&dailink0_master>;
        simple-audio-card,frame-master = <&dailink0_master>;
        simple-audio-card,widgets =
                "Microphone", "Microphone Jack",
                "Headphone", "Headphone Jack",
                "Speaker", "External Speaker";
        simple-audio-card,routing =
                "MIC_IN", "Microphone Jack",
                "Headphone Jack", "HP_OUT",
                "External Speaker", "LINE_OUT";

        simple-audio-card,cpu {
                sound-dai = <&sh_fsi2 0>;
        };

        dailink0_master: simple-audio-card,codec {
                sound-dai = <&ak4648>;
                clocks = <&osc>;
```

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

Troubleshooting: no sound

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

- Unmute `Master` and the relevant widgets
- 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 master/slave 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`: `git://opensource.wolfsonmicro.com/asoc-tools.git`

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

- 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 slave is an uncommon configuration and is probably untested.
- When in doubt, use `devmem` or `i2cget`

References

- Documentation/sound/alsa/soc/
- Common Inter-IC Digital Interfaces for Audio Data Transfer by Jerad Lewis, Analog Devices, Inc. http://www.analog.com/media/en/technical-documentation/technical-articles/MS-2275.pdf?doc=an-1327.pdf
- I^{2}S specification https://web.archive.org/web/20060702004954/http://www.semiconductors.philips.com/acrobat_download/various/I2SBUS.pdf

Questions? Suggestions? Comments?

Alexandre Belloni
alexandre.belloni@bootlin.com

`http://bootlin.com/pub/conferences/2016/elce/belloni-alsa-asoc/`