ASoC Codec Driver

The codec driver is generic and hardware independent code that configures the codec to provide audio capture and playback. It should contain no code that is specific to the target platform or machine. All platform and machine specific code should be added to the platform and machine drivers respectively.

Each codec driver *must* provide the following features:-

1) Codec DAI and PCM configuration
2) Codec control IO - using I2C, 3 Wire(SPI) or both APIs
3) Mixers and audio controls
4) Codec audio operations

Optionally, codec drivers can also provide:-

5) DAPM description.
6) DAPM event handler.
7) DAC Digital mute control.

Its probably best to use this guide in conjunction with the existing codec driver code in sound/soc/codecs/

ASoC Codec driver breakdown

1 - Codec DAI and PCM configuration

Each codec driver must have a struct snd_soc_codec_dai to define its DAI and PCM capabilities and operations. This struct is exported so that it can be registered with the core by your machine driver.

e.g.

```
struct snd_soc_codec_dai wm8731_dai = {
        .name = "WM8731",
        /* playback capabilities */
        .playback = {
                .stream_name = "Playback",
                .channels_min = 1,
                .channels_max = 2,
                .rates = WM8731_RATES,
                .formats = WM8731_FORMATS,},
        /* capture capabilities */
        .capture = {
                .stream_name = "Capture",
                .channels_min = 1,
                .channels_max = 2,
                .rates = WM8731_RATES,
                .formats = WM8731_FORMATS,},
        /* pcm operations - see section 4 below */
        .ops = {
                .prepare = wm8731_pcm_prepare,
                .hw_params = wm8731_hw_params,
                .shutdown = wm8731_shutdown,
```

```
        },
        /* DAI operations - see DAI.txt */
        .dai_ops = {
                .digital_mute = wm8731_mute,
                .set_sysclk = wm8731_set_dai_sysclk,
                .set_fmt = wm8731_set_dai_fmt,
        }
};
EXPORT_SYMBOL_GPL(wm8731_dai);
```

2 - Codec control IO

The codec can usually be controlled via an I2C or SPI style interface (AC97 combines control with data in the DAI). The codec drivers provide functions to read and write the codec registers along with supplying a register cache:-

```
        /* IO control data and register cache */
        void *control_data; /* codec control (i2c/3wire) data */
        void *reg_cache;
```

Codec read/write should do any data formatting and call the hardware read write below to perform the IO. These functions are called by the core and ALSA when performing DAPM or changing the mixer:-

```
    unsigned int (*read)(struct snd_soc_codec *, unsigned int);
    int (*write)(struct snd_soc_codec *, unsigned int, unsigned int);
```

Codec hardware IO functions - usually points to either the I2C, SPI or AC97 read/write:-

```
        hw_write_t hw_write;
        hw_read_t hw_read;
```

3 - Mixers and audio controls

All the codec mixers and audio controls can be defined using the convenience macros defined in soc.h.

```
    #define SOC_SINGLE(xname, reg, shift, mask, invert)
```

Defines a single control as follows:-

```
  xname = Control name e.g. "Playback Volume"
  reg = codec register
  shift = control bit(s) offset in register
  mask = control bit size(s) e.g. mask of 7 = 3 bits
  invert = the control is inverted
```

Other macros include:-

```
    #define SOC_DOUBLE(xname, reg, shift_left, shift_right, mask, invert)
```

A stereo control

```
    #define SOC_DOUBLE_R(xname, reg_left, reg_right, shift, mask, invert)
```

A stereo control spanning 2 registers

```
    #define SOC_ENUM_SINGLE(xreg, xshift, xmask, xtexts)
```

Defines an single enumerated control as follows:-

```
   xreg = register
   xshift = control bit(s) offset in register
   xmask = control bit(s) size
   xtexts = pointer to array of strings that describe each setting

   #define SOC_ENUM_DOUBLE(xreg, xshift_l, xshift_r, xmask, xtexts)
```

Defines a stereo enumerated control

4 - Codec Audio Operations

The codec driver also supports the following ALSA operations:-

```
/* 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 *);
};
```

Please refer to the ALSA driver PCM documentation for details.
http://www.alsa-project.org/~iwai/writing-an-alsa-driver/c436.htm

5 - DAPM description.

The Dynamic Audio Power Management description describes the codec power components and their relationships and registers to the ASoC core.
Please read dapm.txt for details of building the description.

Please also see the examples in other codec drivers.

6 - DAPM event handler

This function is a callback that handles codec domain PM calls and system domain PM calls (e.g. suspend and resume). It is used to put the codec to sleep when not in use.

Power states:-

```
        SNDRV_CTL_POWER_D0: /* full On */
        /* vref/mid, clk and osc on, active */
```

```
        SNDRV_CTL_POWER_D1: /* partial On */
        SNDRV_CTL_POWER_D2: /* partial On */

        SNDRV_CTL_POWER_D3hot: /* Off, with power */
        /* everything off except vref/vmid, inactive */

        SNDRV_CTL_POWER_D3cold: /* Everything Off, without power */
```

7 - Codec DAC digital mute control

Most codecs have a digital mute before the DACs that can be used to minimise any system noise. The mute stops any digital data from entering the DAC.

A callback can be created that is called by the core for each codec DAI when the mute is applied or freed.

i.e.

```
static int wm8974_mute(struct snd_soc_codec *codec,
        struct snd_soc_codec_dai *dai, int mute)
{
        u16 mute_reg = wm8974_read_reg_cache(codec, WM8974_DAC) & 0xffbf;
        if(mute)
                wm8974_write(codec, WM8974_DAC, mute_reg | 0x40);
        else
                wm8974_write(codec, WM8974_DAC, mute_reg);
        return 0;
}
```