Jz4740

32 Bits Microprocessor

Application Notes 02

Embedded CODEC

Revision: 0.1
Date: Nov. 2007

北京君正集成电路有限公司
Ingenic Semiconductor Co. Ltd

Jz4740 32 Bits Microprocessor

Application Notes 02

Embedded CODEC

Release history

Date	Revision	Change
Nov. 2007	0.1	Before release

Disclaimer

This documentation is provided for use with Ingenic products. No license to Ingenic property rights is granted. Ingenic assumes no liability, provides no warranty either expressed or implied relating to the usage, or intellectual property right infringement except as provided for by Ingenic Terms and Conditions of Sale.

Ingenic products are not designed for and should not be used in any medical or life sustaining or supporting equipment.

All information in this document should be treated as preliminary. Ingenic may make changes to this document without notice. Anyone relying on this documentation should contact Ingenicfor the current documentation and errata.

Ingenic Semiconductor Co. Ltd

Room 801C, Power Creative E, No.1 B/D ShangDi East Road, Haidian District,
Beijing 100085, China
Tel: 86-10-58851002
Fax: 86-10-58851005
Http: //www.ingenic.cn

Jz4740 Embedded CODEC

本文介绍君正 32 位处理器 JZ4740 内部音频 CODEC 的使用方法和注意事项，包括 1）系统上电过程 CODEC 的初始化；2）消除 POP 音的几种不同方法；3）放音暂停和中间停止时的处理；4）音量调节；5）录音和增益设置；6）连续播放 mute（sample 0）超过 5ms 时的处理。

1. 系统上电初始化

在 Jz4740 内部 CODEC 工作前，应先初始化 GPIO 和设置 AIC 相关寄存器的值，参考代码如下：

```
void codec_init(void)
{
  /* 初始化 AIC 和 CODEC 使硬件工作*/
        __i2s_internal_codec();
        __i2s_as_slave();
        __i2s_select_i2s();
        __aic_select_i2s();
        REG_ICDC_CDCCR1 = 0x001b2303;
        mdelay(1);
        REG_ICDC_CDCCR1 = 0x001b2302;
}
```

2. 消除 POP 音

在CODEC打开和关闭时，HPVOL上的电压变化大时会产生POP音。原则上可以通过两种方法来消除POP音：1)软件处理;2)硬件处理。这两种方法各有优缺点，软件处理时在播放声音前增加一些延时(大概2.4ms)，在播放过程和结束时都需要做相应处理，比较复杂；而硬件处理方法则需要增加两个MOS管和一个GPIO控制管脚，好处是软件控制代码比较简单，而且去除POP音比较彻底。

采用方法一，在打开CODEC时先对CODEC HPOUT放电，再对VREF充电，之后再播放约400ms的正弦波，sample从0x1ffff变化到0x0，把HPOUT的电压从0V升到1.5V，对应下面函数。

```
    void open_codec_method1(void)
{
                REG_ICDC_CDCCR1 = 0x001b2303;
  mdelay(1);
  REG_ICDC_CDCCR1 = 0x001b2302;
  mdelay(2);
  REG_ICDC_CDCCR1 = 0x03000100;
  mdelay(500);
  mdelay(500);
```

```
mdelay(500);
mdelay(500);

__aic_play_lastsample();
reg_val = REG_ICDC_CDCCR2;
reg_val = reg_val & 0xfffff0f0;
reg_val = reg_val | 0x00000803;
REG_ICDC_CDCCR2 = reg_val;
__i2s_set_oss_sample_size(18);
__aic_enable_mono2stereo();

REG_AIC_DR = 0x1ffff;
REG_AIC_DR = 0x1ffff;
REG_AIC_DR = 0x1ffff;
REG_AIC_DR = 0x1ffff;
REG_AIC_DR = 0x1ffff;
REG_AIC_DR = 0x1ffff;
REG_AIC_DR = 0x1ffff;
REG_AIC_DR = 0x1ffff;

REG_AIC_DR = 0x1ffff;
REG_AIC_DR = 0x1ffff;
REG_AIC_DR = 0x1ffff;
REG_AIC_DR = 0x1ffff;

__aic_enable_replay();
__i2s_enable();

/*!!! must wait for 2 ms to reduce pop */
mdelay(2);
sample = 0x1ffff;
REG_ICDC_CDCCR1 = 0x03000000;
//sta1 = jiffies;
for (cnt = 0; cnt < sample_count; cnt++) {
    while(1) {
        tx_con = (REG_AIC_SR & 0x00003f00) >> 8;
        if (tx_con < 32) {
            REG_AIC_DR = (signed long)sample_arr[cnt];
            break;
        }
    }
}

tx_con = (REG_AIC_SR & 0x00003f00) >> 8;
```

```
        while(tx_con > 0) {
            tx_con = (REG_AIC_SR & 0x00003f00) >> 8;
        }

        REG_AIC_DR = 0x0;
        REG_AIC_DR = 0x0;
        REG_AIC_DR = 0x0;
        REG_AIC_DR = 0x0;
        REG_AIC_DR = 0x0;
        REG_AIC_DR = 0x0;

        tx_con = (REG_AIC_SR & 0x00003f00) >> 8;
        while(tx_con > 0) {
            tx_con = (REG_AIC_SR & 0x00003f00) >> 8;
        }

        __aic_disable_mono2stereo();
        __i2s_set_oss_sample_size(16);

        REG_AIC_DR = 0x1;
        REG_AIC_DR = 0x1;
        REG_AIC_DR = 0x1;
        REG_AIC_DR = 0x1;
        REG_AIC_DR = 0x1;
        REG_AIC_DR = 0x1;
        tx_con = (REG_AIC_SR & 0x00003f00) >> 8;
        while (tx_con > 0) {
            tx_con = (REG_AIC_SR & 0x00003f00) >> 8;
        }
        REG_ICDC_CDCCR1 = 0x03002000;
        REG_AIC_DR = 0x1;
}
```

采用方法一，在关闭 CODEC 时播放约 400ms 的正弦波，sample 从 0x0 变化到 0x1ffff，把 HPOUT 的电压从 1.5V 降到 0V，对应下面函数。

```
    void close_codec_method1(void)
    {
            REG_AIC_DR = 0x0;
        REG_AIC_DR = 0x0;
        REG_AIC_DR = 0x0;
        REG_AIC_DR = 0x0;

        tx_con = (REG_AIC_SR & 0x00003f00) >> 8;
```

```
    while(tx_con > 0) {
        tx_con = (REG_AIC_SR & 0x00003f00) >> 8;
    }
    __aic_play_lastsample();
    reg_val = REG_ICDC_CDCCR2;
    reg_val = reg_val & 0xfffff0f0;
    reg_val = reg_val | 0x00000803;
    REG_ICDC_CDCCR2 = reg_val;

    __i2s_set_oss_sample_size(18);
    __aic_enable_mono2stereo();
    REG_AIC_DR = 0x0;
    REG_AIC_DR = 0x0;

    sample = 0x0;
    REG_ICDC_CDCCR1 = 0x03000000;

    For (cnt = (sample_count-1) ; cnt >= 0; cnt--) {
        while(1) {
            tx_con = (REG_AIC_SR & 0x00003f00) >> 8;
            if (tx_con < 32) {
                REG_AIC_DR = (signed long)sample_arr[cnt];
                break;
            }
        }
    }

    tx_con = (REG_AIC_SR & 0x00003f00) >> 8;
    while(tx_con > 0) {
        tx_con = (REG_AIC_SR & 0x00003f00) >> 8;
    }

    REG_AIC_DR = 0x1ffff;
    REG_AIC_DR = 0x1ffff;
    __i2s_disable_replay();
    __i2s_disable();
    REG_ICDC_CDCCR1 = 0x001b2303;
    mdelay(1);
    REG_ICDC_CDCCR1 = 0x001b2302;
}
```

采用方法二时，需在 HPOUT 左右声道上各接一个 MOS 管，在 CODEC 打开时用 GPIO 控制 MOS 管断开，等 POP 音过去后，再控制 MOS 管接上，通过这种方法也能消除 POP 音。

采用方法二时，先初试化 CODEC 和 GPIO 设置：

```
void codec_gpio_init(void)
    {
            __gpio_as_output(GPIO_AMP_CE);      /* GPIO_AMP_CE控制MOS管的连通和断开
    */
            REG_ICDC_CDCCR1 = 0x00033300;
      REG_ICDC_CDCCR1 = 0x14023300
    }
```

采用方法二时,在打开 CODEC 之前先断开 MOS 管:

```
void open_codec_method2(void)
    {
            __gpio_set_pin(GPIO_AMP_CE);          //断开
      REG_ICDC_CDCCR1 = 0x00037302;
      mdelay(2);
      REG_ICDC_CDCCR1 = 0x03006000;
      mdelay(2);
      REG_ICDC_CDCCR1 = 0x03002000;
    }
```

在放音之前再用 GPIO 连上 MOS 管:

```
    void write_sample(char *buf, int size)
    {
      .
      .
            __gpio_clear_pin(GPIO_AMP_CE);
            .
            .
    }
```

采用方法二时,在放完音之后关闭 CODEC，用 GPIO 断开 MOS 管，并关闭 CODEC。

```
void close_codec_method(void)
    {
            __gpio_set_pin(GPIO_AMP_CE);
      REG_ICDC_CDCCR1 = 0x00033300;
    }
```

3. 放音暂停和中间停止时的处理

在打开CODEC后，如果超过5ms没有播放声音，CODEC会自动关闭，此时会产生POP音。为了防止这种情况出现，要求软件在超过5ms没有声音播放时，必须播放额外的数据0x1，以防止CODEC自动关闭。

例如暂停播放或者停止播放时，当暂停时间超过 5ms 时，CODEC 自动关闭，这时就会产生 POP 声。如果播放的最后一个 sample 不是 0x0，还需要播放一段 sample 线性升/降到 0x0。这种情况下，我们需要进行两个步骤的处理：首先，播放一段 sample，从当前播放的 sample 线性升/降到 0x0 的音（当前 sample 是正时，线性降到 0x0；当前 sample 是负，线性升到 0x0）；然后，设置 AIC 工作模式以持续播放 sample 0x1，达到消除 POP 音的目的。下面是一段参考代码：

```
/* 播放 sample 为 16-bit 时的参考处理代码
 * l_sample：最后播放的左声道 sample
* r_sample：最后播放的右声道 sample
 * jz_audio_speed 是当前播放频率 (例如 48000, 44100, 8000 等)
 * pop_turn_onoff_buf 是存放左、右声道最后一个 sample 线性收到 0 时的所有 sample 的地址区
 */
void write_mute_to_dma_buffer(signed long l_sample, signed long r_sample)
{
  int i,step_len;
  unsigned long *pop_buf = (unsigned long*)pop_turn_onoff_buf;
  unsigned int sample_oss = (REG_AIC_CR & 0x00380000) >> 19;
  unsigned long l_sample_count,r_sample_count,sample_count;
  struct jz_i2s_controller_info *controller = i2s_controller;

  signed int left_sam, right_sam, l_val, r_val;

  left_sam = (signed int)l_sample;
  right_sam = (signed int)r_sample;

  if(left_sam == 0 && right_sam == 0)   /* 左、右声道 sample 是 0 时，不用处理 */
      return;

  /*******************************************************
      * 以下构造线性升/降到 0x0 的 samples，并播放这些 samples
      *******************************************************/
  switch(sample_oss)
  {
  case 0x0:      /* 8-bit */
      /* 请参考 16-bit 流程做相应处理 */
      break;
  case 0x1:      /* 16-bit */
      /** ************************************************
```

```
     * 开始构造 samples
**************************************************/
step_len = jz_audio_speed / 10 * 3; /* 300ms 内的 sample 个数*/
step_len = step_len / 2;              /* 共有两个声道,每个声道分别计算 */
step_len = 0x7fff / step_len + 1;         /* 每次减少的步长 */

l_sample_count = 0;
l_val = left_sam;

while(1) {       /* 先计算左声道收到 0 时的 sample 个数 */
     if (l_val > 0) {    /* 暂停或关闭时，左声道最后一个 sample 是正数 */

          if (l_val >= step_len) {          /* sample 大于等于步长时 */
               l_val -= step_len;   /* 每次减少步长 */
               l_sample_count ++;      /* 同时计数增 1 */
          } else
               break;

     }

     if (l_val < 0) {  /* 暂停或关闭时，左声道最后一个 sample 是负数 */

          if (l_val <= -step_len) {    /* sample 小于等于步长时 */
               l_val += step_len; /* 每次增加步长 */
               l_sample_count ++; /* 同时计数增 1 */

          } else
               break;

     }

     if (l_val == 0) /* sample 等于 0 时，结束*/
          break;
}

r_sample_count = 0;
r_val = right_sam;
while(1) { /* 再计算左声道收到 0 时的 sample 个数 */
     if (r_val > 0) { /* 暂停或关闭时，右声道最后一个 sample 是正数 */

          if (r_val >= step_len) { /* sample 大于等于步长时 */
               r_val -= step_len; /* 每次减少步长 */
               r_sample_count ++; /* 同时计数增 1 */
          } else
```

```
                    break /* sample 小于步长时，结束 */
            }

            if (r_val < 0) { /* 暂停或关闭时，右声道最后一个 sample 是负数 */

                if (r_val <= -step_len) { /* sample 小于等于步长时 */
                    r_val += step_len; /* 每次增加步长 */
                    r_sample_count ++; /* 同时计数增 1 */
                } else
                    break; /* sample 大于步长时，结束 */

            }

            if(r_val == 0) /* sample 等于 0 时，结束 */
                break;

        }

        //fill up
        if(l_sample_count > r_sample_count) /* sample 数是左、右声道收到 0 时个数的最大数
*/
            sample_count = l_sample_count;
        else
            sample_count = r_sample_count;

        l_val = left_sam;
        r_val = right_sam;
        for (i=0;i <= sample_count;i++) {   /* 循环填充 pop_buf */

            *pop_buf = (unsigned long)l_val; /* 首先填充左声道 */

            pop_buf ++;                          /*   buffer 地址增加 */

            if (l_val > step_len) {
                l_val -= step_len;      /* 左声道是正时，减步长 */
            } else if (l_val < -step_len) {
                l_val += step_len;     /* 左声道是负时，加步长 */
            } else if (l_val >= -step_len && l_val <= step_len) {
                l_val = 0;   /* 左声道是在正和负步长之间时，左声道置 0 */
            }

            *pop_buf = (unsigned long)r_val; /* 首先填充右声道 */

            pop_buf ++;                           /*   buffer 地址增加 */
```

```
                if (r_val > step_len) {
                    r_val -= step_len;      /* 右声道是正时，减步长 */
                } else if (r_val < -step_len) {
                    r_val += step_len;      /* 右声道是负时，加步长 */
                } else if (r_val >= -step_len && r_val <= step_len) {
                    r_val = 0; /* 右声道是在正和负步长之间时，右声道置 0 */
                }

            }

            *pop_buf = 0;   /* 左声道填如 0 */

            pop_buf ++;
            *pop_buf = 0; /* 右声道填如 0 */

            pop_buf ++;
            sample_count += 2; /* sample 总数再加上 2 个 */

            /* 回写，总数是左、右声道的字节数 */
            dma_cache_wback_inv(pop_turn_onoff_buf, sample_count*8);


            /** ****************************************************
             * samples 构造完毕，启动 DMA 传输播放 samples
                        ****************************************************/
            pop_dma_flag = 1;

    audio_start_dma(controller->dma1,controller,pop_turn_onoff_pbuf,sample_count*8,DMA_MO
DE_WRITE);
            sleep_on(&pop_wait_queue);     /* 休眠等待 DMA 结束 */
            pop_dma_flag = 0;                   /* 此处已唤醒 */

            break;
        case 0x2:       /* 18-bit */
            /* 请参考 16-bit 流程做相应处理 */
            break;
        case 0x3:       /* 20-bit */
            /* 请参考 16-bit 流程做相应处理 */
            break;
        case 0x4:       /* 24-bit */
            /* 请参考 16-bit 流程做相应处理 */
            break;
        }
```

```
        /* 到此第一阶段 sample 播放完毕，进入第二阶段 */
    __aic_play_lastsample();        /* 进入持续播放最后一个 sample 模式 */
    REG_AIC_DR = 0x1;          /* 左声道 sample */
    REG_AIC_DR = 0x1;          /* 右声道 sample */
}
```

4. 耳机音量调节

CODEC 中硬件调节音量有四级，和软件调节相结和，可以支持多个级别的音量调节。下面给出建议的调节公式和参数列表，可以实现 32 级的音量调节，其中级别越大表示音量越大，0 表示没有声音输出。

$$Sample_{NEW} = (Sasmple_{Original} \times FixP8\_Factor) >> 8$$ (32-bit signed integer operation)

Volume Level	CDCCR1.HPMUTE	CDCCR2.HPVOL	Sample Factor	FixP8_Factor
31	0	11	1	256
30			0.966	247
29			0.931	238
28			0.897	230
27			0.862	221
26			0.828	212
25		10	1	256
24			0.959	245
23			0.917	235
22			0.876	224
21			0.835	214
20		01	1	256
19			0.948	243
18			0.897	230
17			0.845	216
16		00	1	256
15			0.938	240
14			0.875	224
13			0.813	208
12			0.750	192
11			0.688	176
10			0.625	160
9			0.563	144
8			0.500	128
7			0.438	112
6			0.375	96
5			0.313	80
4			0.250	64
3			0.188	48
2			0.125	32
1			0.063	16
0	1		0.000	0

下面这段参考代码是用纯软件调节音量的事例，假设音量级别是由用户指定的 0 – 10，0 音量最低，10 音量最高。

```
/* val 值为用户指定的音量级别 0-10 */
    void audio_set_volume(int val)
    {
            codec_volume = 3;              /* CODEC DAC 音量固定为 3 （6dB） */
            codec_volue_shift = val;              /* 在下面修改音量函数中用到 */
            REG_ICDC_CDCCR2 = ((REG_ICDC_CDCCR2 & ~(0x3)) | codec_volume);
    }
```

修改的音量会在下面的函数中改 sample。

```
/* src_start 为音频数据的源地址
 * count 为音频数据的字节数
 * dst_start 为处理后的数据存放的地址
  */
    void audio_sample_modify(signed long src_start, int count, signed long dst_start)
    {
      int cnt = 0;
      signed short d1;
      signed long l1;
      volatile signed short *s = (signed short *)src_start;
      volatile signed long *dp = (signed long *)dst_start;

      while (count > 0)
      {
                count -= 2;
                cnt += 2;
                d1 = *(s++);

                l1 = (signed long)d1;            /* 原始的 sample */
           l1 = l1 * codec_volue_shift / 10;          /* 修改后的 sample */

                *(dp ++) = l1;
      }
    }
```

5. 录音及增益设置

这里我们给出一组录音常用的 CODEC 设置参数，用户可以根据自己实际情况进行调整。

```
void codec_record(void)
{
  REG_ICDC_CDCCR1 = 0x14024300; /* 录音使能设置 */
        REG_ICDC_CDCCR2 = 0x00100833; /* 录音频率和增益设置*/
  mdelay(50);
}
```

6. 连续播放 mute（sample 0）超过 5ms 时的处理

在 codec 打开时（即 CDCCR1 设置是 0x03002000），如果播放 mute 音（0x0）累积长达 5ms，codec 会关闭 DAC，当下一个非 mute 音频数据到来时，codec 会打开 DAC，这时会产生 POP 声。

为了消除这个 POP 声，需要在播放音频 sample 的过程中计算连续播放 mute(0x0)的个数，当连续播放 mute 的个数超过 5ms 时，需要修改最后一个 sample 为 0x1，这样 codec 就不会自动关闭 DAC，也就不会有 POP 音了。请参考下面的代码：

```
/*
*       src_start 是源地址
*       count 是 sample 的字节数
*       dst_start 是 DMA 缓冲区地址
*       jz_audio_rate 是音频的频率（e.g. 48000，44100，8000 等）
*/
void audio_sample_prehandle(signed long src_start, int count, signed long dst_start)
{
  int cnt = 0;
  signed short d1;
  signed long l1;
  int mute_cnt = 0;   /* 统计连续 mute 的个数 */

  volatile signed short *s = (signed short *)src_start;
  volatile signed long *dp = (signed long*)(*(dst_start));

  int sam_rate = jz_audio_rate / 200;     /* 5ms 中的 sample 个数 */

  while (count > 0)
  {
            count = count - 2;
            cnt = cnt + 2;
            d1 = *(s++);

            l1 = (signed long)d1;

        /* 当 sample 是 mute 时，开始计算连续 mute 的个数 */
```

```
        if (l1 == 0) {
                        mute_cnt ++;   /* 每有一个 mute 时，mute 计数加 1*/

            /*   当连续 mute 的个数大于等于 5ms  中的 sample 个数时  */
                        if (mute_cnt >= sam_rate) {
                            *(dp-1) = (signed long)1; /* 把一个声道的 mute 修改成 0x1 */
                *(dp) = (signed long)1;     /* 把另一个声道的 mute 修改成 0x1 */
                mute_cnt = 0;     /* mute 计数清零,本次连续的 mute 统计结束  */
            }
        } else {
            /* 在 5ms 中有非 mute，不用修改 mute，本次连续的 mute 统计结束*/
            mute_cnt = 0;
            (dp ++) = l1; / *  原数据保持不变  */
                }
     }
    }
```

注：

1） 寄存器定义请参考 JZ4740 音频 CODEC 的 spec

2） 参考代码中使用到的宏定义请参考君正发布的 linux 内核代码