aic8800 dsp开发指南

本文档为aic8800系列芯片内置DSP的软件开发指南

1 DSP基本参数

Speed	480M Hz
Memory	384K，0x100000~0x15FFFF。

DSP可以使用的memory共384K（以下统称为dsp memory），dsp程序的text、rodata、data、bss和stack段必须按照这个范围来划分。

主CPU也可以直接访问dsp memory，但是尽量不要与DSP同时访问，以免影响DSP的执行效率。

2 开发流程

DSP有一套单独的SDK，和主CPU的SDK使用相同编译环境。

1. 编写DSP端的程序，所用的SDK为aci8800-dsp-sdk。在编译脚本中加入DSP=on。默认编译脚本build_dsp.sh，不使用rtos。链接脚本为map_ram_cm4.txt。
2. 编写主CPU端的程序，所用SDK为aic8800-sdk。在编译脚本中加入DSP=on。将第1步编译出的bin文件转换后存放在主CPU端的数组dsp_image_table中。使用编译脚本build_test_case_dsp.sh可以编译dsp的test case，对应的链接脚本为map_ram_cm4.txt。使用其他链接脚本时，注意添加此脚本中的.shared_dsp段。
3. 将第2步编译出的bin烧录在8800的flash中并运行。运行时，主CPU会先启动，在主CPU的dsp task中，通过调用dsp_image_load将第2步中所说的dsp_image_table加载到dsp memory中，然后通过调用dsp_launch启动DSP。

3 主CPU和DSP之间的交互

主CPU和DSP之间通过ipc进行交互，同时二者共同维护一个dsp_ipc_env结构体，包含msg和data的链表。

在命名上，"a2e"是主CPU至DSP方向，"e2a"是DSP至主CPU方向。

目前主CPU至DSP方向定义了4个ipc信号，在DSP至CPU方向也定义了4个信号。

信号	方向	说明
IPC_DSP_A2E_MSG_BIT	主CPU至DSP	主CPU发送MSG给DSP
IPC_DSP_A2E_DATA_BIT	主CPU至DSP	主CPU发送DATA给DSP
IPC_DSP_A2E_MSG_ACK_BIT	主CPU至DSP	主CPU收到DSP的MSG后，发送MSG ACK给DSP（暂未使用）
IPC_DSP_A2E_DATA_ACK_BIT	主CPU至DSP	主CPU收到DSP的DATA后，发送DATA ACK给DSP（暂未使用）
IPC_DSP_E2A_MSG_BIT	DSP至主CPU	DSP发送MSG给主CPU
IPC_DSP_E2A_DATA_BIT	DSP至主CPU	DSP发送DATA给主CPU
IPC_DSP_E2A_MSG_ACK_BIT	DSP至主CPU	DSP收到主CPU的MSG后，发送MSG ACK给主CPU（暂未使用）
IPC_DSP_E2A_DATA_ACK_BIT	DSP至主CPU	DSP收到主CPU的DATA后，发送DATA ACK给主CPU（暂未使用）

这些ipc信号的含义，客户可以自行更改。

dsp_ipc_env结构体如下：

需要注意的是，在主*CPU*和*DSP*的链接脚本中，要保证*dsp_ipc_env*处在同一地址。目前在主*CPU*和*DSP*端的链接脚本中定义了相同的.*shared_dsp*段，用来存放*dsp_ipc_env*。

同时，主*CPU*和*DSP*需要互斥访问*dsp_ipc_env*，在访问前调用*dsp_ipc_env_lock*，在访问后调用*dsp_ipc_env_unlock*来实现。

```
struct dsp_ipc_env_tag {
    struct co_list a2e_msg_sent;

    struct co_list a2e_msg_free;

    struct co_list a2e_data_sent;

    struct co_list a2e_data_free;

    struct co_list e2a_msg_sent;

    struct co_list e2a_msg_free;

    struct co_list e2a_data_sent;

    struct co_list e2a_data_free;

    uint16_t a2e_msg_cnt;

    uint16_t a2e_data_cnt;
```

```
21
22      uint16_t e2a_msg_cnt;
23
24      uint16_t e2a_data_cnt;
25  };
26
27  __SHARED_DSP static struct dsp_ipc_env_tag dsp_ipc_env;
```

同时，在DSP端准备了4个内存池，分别为a2e_msg_pool、e2a_msg_pool、a2e_data_pool和e2a_data_pool。

```
1   struct dsp_msg_elt {
2       /// List header
3       struct co_list_hdr hdr;
4       uint16_t id;
5       uint16_t seq;
6       uint16_t len;
7       uint8_t param[DSP_MSG_MAX_LEN];
8   };
9
10  struct dsp_data_elt {
11      /// List header
12      struct co_list_hdr hdr;
13      uint16_t id;
14      uint16_t seq;
15      uint16_t len;
16      uint16_t max_len;
17      uint8_t *data_ptr;
18  };
19
20  static struct dsp_msg_elt a2e_msg_pool[DSP_MSG_MAX_NB];
21  static struct dsp_msg_elt e2a_msg_pool[DSP_MSG_MAX_NB];
22  static struct dsp_data_elt a2e_data_pool[DSP_DATA_MAX_NB];
23  static struct dsp_data_elt e2a_data_pool[DSP_DATA_MAX_NB];
```

msg_pool和data_pool都位于DSP memory当中。对于msg_pool，在struct dsp_msg_elt中，成员param[DSP_MSG_MAX_LEN]负责存放msg的内容，即msg的携带的内容有最大长度限制，msg_pool完全是静态分配好的。

对于data_pool，在struct dsp_data_elt中，成员data_ptr指向data buffer。不管是a2e_data_pool还是e2a_data_pool，都由DSP端来分配data buffer，data_ptr所指向的地址均位于dsp memory中。目前的实现是，由主CPU端来指定data buffer的长度，通过msg来告知DSP，DSP在收到msg后，根据msg中携带的长度再为data_pool中的各个元素分配data buffer。

以链表a2e_msg_sent、a2e_msg_free和内存池a2e_msg_pool为例：

1. 在DSP端初始化时，将a2e_msg_pool中的元素，依次插入链表a2e_msg_free中，参考dsp_ipc_env_init。
2. DSP端初始化完成后，进入主循环，若无事件需要处理，则执行WFI进入休眠状态。
3. 当主CPU要给DSP发送msg时，先从链表a2e_msg_free中取出一个元素，填充内容后，将其插入链表a2e_msg_sent，并设置ipc的IPC_DSP_A2E_MSG_BIT。

4. DSP端收到对应的ipc中断后，在中断服务程序中调用dsp_event_set(DSP_EVENT_MSG)，将dsp_event_env.event_field中的DSP_EVENT_MSG位置1，然后退出中断服务程序。ipc中断会导致DSP退出休眠状态，继续执行主循环。
5. DSP主循环进入dsp_schedule，在dsp_schedule中调用DSP_EVENT_MSG对应的回调函数dsp_msg_handler。
6. 在dsp_msg_handler中，从链表a2e_msg_sent取出一个元素并处理，处理完之后将此元素再插入链表a2e_msg_free以完成回收。
7. 检查一下a2e_msg_sent链表是否为非空。如果是非空，则再次调用dsp_event_set(DSP_EVENT_MSG)，将dsp_event_env.event_field中的DSP_EVENT_MSG位置1，然后DSP回到第5步继续执行。

主CPU端和DSP端对msg和data的处理方式基本一致。目前的区别是，主CPU端单独创建了一个task，在task中进行处理。DSP端没有使用task，二是在主循环中进行处理。

4 主CPU端API介绍

```
/*
 * 获取访问dsp_ipc_env的互斥锁，获取成功后返回true。
 */
bool dsp_ipc_env_lock(void);

/*
 * 释放访问dsp_ipc_env的互斥锁。
 */
void dsp_ipc_env_unlock(void);

/*
 * 从dsp_ipc_env.a2e_msg_free中取出一个msg element。
 */
struct dsp_msg_elt *dsp_a2e_msg_malloc(void);

/*
 * 将msg element插入dsp_ipc_env.a2e_msg_free。
 */
void dsp_a2e_msg_free(struct dsp_msg_elt *msg);

/*
 * 将msg element插入dsp_ipc_env.e2a_msg_free。
 */
void dsp_e2a_msg_free(struct dsp_msg_elt *msg);

/*
 * 从dsp_ipc_env.a2e_data_free中取出一个data element。
 */
struct dsp_data_elt *dsp_a2e_data_malloc(void);

/*
 * 将data element插入dsp_ipc_env.a2e_data_free。
 */
void dsp_a2e_data_free(struct dsp_data_elt *data);

/*
 * 将data element插入dsp_ipc_env.e2a_data_free。
```

```
 */
void dsp_e2a_data_free(struct dsp_data_elt *data);

/*
 * 构建主CPU至DSP方向的msg element。
 * @param[in] msg, pointer points to the msg element
 * @param[in] id, msg id
 * @param[in] param, stores params that will be carried by the msg element
 * @param[in] len, length of params
 * @return 0: success
 */
int dsp_a2e_msg_build(struct dsp_msg_elt *msg, uint16_t id, void *param,
uint16_t len);

/*
 * 构建主CPU至DSP方向的data element，将数据从主CPU的memory搬运至dsp memory。
 * @param[in] data, pointer points to the data element
 * @param[in] buf, stores data that will be carried by the data element
 * @param[in] offset
 * @param[in] len, length of the buf
 * @param[in] use_dma, use dma or not, better use dma when length is above
32 bytes
 * @return 0: success
 */
int dsp_a2e_data_build(struct dsp_data_elt *data, uint8_t *buf, uint16_t
offset, uint16_t len, bool use_dma);

/*
 * 将e2a data element元素中的数据从dsp memory搬运至主CPU的memory。
 * @param[in] dst, destination
 * @param[in] src, source
 * @param[in] len, length
 * @param[in] use_dma, use dma or not, better use dma when length is above
32 bytes
 * @return 0: success
 */
int dsp_e2a_data_copy(uint8_t *dst, uint8_t *src, uint16_t len, bool
use_dma);

/*
 * 将dsp的image加载至对应的dsp memory中。
 * @param[in] addr, start addr of the image
 * @param[in] use_dma, use dma or not
 * @param[in] dma_ch, dma channel
 * @param[in] image_check, check whether the image
 * @return 0: success
 */
void dsp_image_load(uint32_t addr, bool use_dma, uint8_t dma_ch, bool
image_check);

/*
 * 启动DSP，addr必须与image加载的地址一致。
 */
void dsp_lauch(addr);
```

5 DSP端API介绍

DSP端API与主APP端API相似，不再介绍。