AIC8800 DSP Development Guide

This document is a software development guide for the built-in DSP of the aic8800 series chips

1 DSP basic parameters

Speed	**480Mhz**
Memory	384Kÿ0x100000~0x15FFFFÿ

The total memory available to DSP is 384K (hereinafter referred to as DSP memory). The text, rodata, data, bss and stack segments of the DSP program must be divided according to this range.

The main CPU can also directly access the DSP memory, but try not to access it at the same time as the DSP to avoid affecting the execution efficiency of the DSP.

2 Development Process

DSP has a separate SDK, which uses the same compilation environment as the SDK of the main CPU.

1. Write the DSP program, the SDK used is aci8800-dsp-sdk. Add DSP=on in the compilation script. The default compilation script is build_dsp.sh, which does not use rtos. The link script is map_ram_cm4.txt.
2. Write the program for the main CPU, using the aic8800-sdk SDK. Add DSP=on to the compilation script. Convert the bin file compiled in step 1 and store it in the array dsp_image_table on the main CPU. Use the compilation script build_test_case_dsp.sh to compile the dsp test case, and the corresponding link script is map_ram_cm4.txt. When using other link scripts, be sure to add the .shared_dsp section in this script. 3. Burn the bin compiled in step 2 into the flash of 8800 and run it. When running, the main CPU will start first, and the dsp

of the main CPU will be

In the task, load the dsp_image_table mentioned in step 2 into the dsp memory by calling dsp_image_load, and then start the DSP by calling dsp_launch.

3 Interaction between the main CPU and DSP

The main CPU and DSP interact through ipc, and both of them jointly maintain a dsp_ipc_env structure, which contains linked lists of msg and data.

In terms of naming, "a2e" is the direction from the main CPU to the DSP, and "e2a" is the direction from the DSP to the main CPU.

Currently, 4 IPC signals are defined in the direction from the main CPU to the DSP, and 4 signals are also defined in the direction from the DSP to the CPU.

Signal	direction	illustrate
IPC_DSP_A2E_MSG_BIT	Main CPU to DSP	The main CPU sends MSG to the DSP
IPC_DSP_A2E_DATA_BIT	Main CPU to DSP	The main CPU sends DATA to the DSP
IPC_DSP_A2E_MSG_ACK_BIT	Main CPU to DSP	After receiving the MSG from the DSP, the main CPU sends a MSG ACK to the DSP (not used yet)
IPC_DSP_A2E_DATA_ACK_BIT	Main CPU to DSP	After the main CPU receives the DATA from the DSP, it sends a DATA ACK to the DSP (not used yet)
IPC_DSP_E2A_MSG_BIT	DSP to Main CPU	DSP sends MSG to main CPU
IPC_DSP_E2A_DATA_BIT	DSP to Main CPU	DSP sends DATA to the main CPU
IPC_DSP_E2A_MSG_ACK_BIT	DSP to Main CPU	After receiving the MSG from the main CPU, the DSP sends a MSG ACK to the main CPU. CPU (not used yet)
IPC_DSP_E2A_DATA_ACK_BIT	DSP to Main CPU	After receiving the DATA from the main CPU, the DSP sends a DATA ACK to the main CPU. CPU (not used yet)

The meaning of these ipc signals can be changed by the customer.

The dsp_ipc_env structure is as follows:

It should be noted that in the link scripts of the main CPU and DSP, dsp_ipc_env must be at the same address. Currently, the same .shared_dsp section is defined in the link scripts of the main CPU and DSP to store dsp_ipc_env.

At the same time, the main CPU and DSP need to mutually exclusive access dsp_ipc_env, ***which is achieved by*** **calling dsp_ipc_env_lock before access and dsp_ipc_env_unlock after access .**

```
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;
```

At the same time, four memory pools are prepared on the DSP side, namely a2e_msg_pool, e2a_msg_pool, a2e_data_pool and e2a_data_pool.

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

Both msg_pool and data_pool are located in DSP memory. For msg_pool, in struct dsp_msg_elt, member param[DSP_MSG_MAX_LEN] is responsible for storing the content of msg, that is, the content carried by msg has a maximum length limit, and msg_pool is completely statically allocated.

For data_pool, in struct dsp_data_elt, the member data_ptr points to the data buffer. Whether it is a2e_data_pool or e2a_data_pool, the data buffer is allocated by the DSP side, and the address pointed to by data_ptr is located in the dsp memory. The current implementation is that the main CPU side specifies the length of the data buffer and informs the DSP through msg. After receiving the msg, the DSP allocates the data buffer for each element in the data_pool according to the length carried in the msg.

Take the linked lists a2e_msg_sent, a2e_msg_free and the memory pool a2e_msg_pool as examples:

1. When the DSP is initialized, insert the elements in a2e_msg_pool into the linked list a2e_msg_free in sequence.

   dsp_ipc_env_initÿ

2. After the DSP is initialized, it enters the main loop. If there is no event to be processed, it executes WFI to enter the sleep state. 3.

   When the main CPU wants to send a msg to the DSP, it first takes an element from the linked list a2e_msg_free, fills it with content, and inserts it into the linked list.

   Table a2e_msg_sent and set IPC_DSP_A2E_MSG_BIT of ipc.

4. After the DSP receives the corresponding ipc interrupt, it calls dsp_event_set(DSP_EVENT_MSG) in the interrupt service routine.
   The DSP_EVENT_MSG bit in dsp_event_env.event_field is set to 1, and then the interrupt service routine is exited. The ipc interrupt causes the DSP to exit the sleep state and continue to execute the main loop.
5. The DSP main loop enters dsp_schedule and calls the callback function dsp_msg_handler corresponding to DSP_EVENT_MSG in dsp_schedule.
6. In dsp_msg_handler, take an element from the linked list a2e_msg_sent and process it. After processing, insert the element into the linked list a2e_msg_free to complete the recycling. 7.
Check whether the linked list a2e_msg_sent is not empty. If it is not empty, call
   dsp_event_set(DSP_EVENT_MSG), set the DSP_EVENT_MSG position in dsp_event_env.event_field to 1, and then the DSP returns to step 5 to continue execution.

The main CPU and DSP sides process msg and data in basically the same way. The current difference is that the main CPU side creates a task and processes it in the task. The DSP side does not use tasks and processes it in the main loop.

4 Main CPU API Introduction

```
/* *
  Get the mutex lock for accessing dsp_ipc_env, and return true if successful.
  */
bool dsp_ipc_env_lock(void);

/* *
  Release the mutex for accessing dsp_ipc_env. */

void dsp_ipc_env_unlock(void);

/* *
  Take a msg element from dsp_ipc_env.a2e_msg_free . */ struct dsp_msg_elt

*dsp_a2e_msg_malloc(void);

/* *
  Insert msg element into dsp_ipc_env.a2e_msg_free. */

void dsp_a2e_msg_free(struct dsp_msg_elt *msg);

/* *
  Insert msg element into dsp_ipc_env.e2a_msg_free. */

void dsp_e2a_msg_free(struct dsp_msg_elt *msg);

/* *
  Take a data element from dsp_ipc_env.a2e_data_free . */

struct dsp_data_elt *dsp_a2e_data_malloc(void);

/* *
  Insert data element into dsp_ipc_env.a2e_data_free. */

void dsp_a2e_data_free(struct dsp_data_elt *data);

/* *
  Insert data element into dsp_ipc_env.e2a_data_free.
```

```
38    */
39  void dsp_e2a_data_free(struct dsp_data_elt *data);
40
41  /* *
42    ÿÿÿCPUÿDSPÿÿÿmsg elementÿ * @param[in] msg, pointer
43    points to the msg element * @param[in] id, msg id * @param[in] param, stores params that will
44    be carried by the msg element * @param[in]
45    len, length of params
46
47    * @return 0: success
48    */
49  int dsp_a2e_msg_build(struct dsp_msg_elt *msg, uint16_t id, void *param, uint16_t len);
50
51  /*
52    * ÿÿÿCPUÿDSPÿÿÿdata elementÿÿÿÿÿÿCPUÿmemoryÿÿÿdsp memoryÿ * @param[in] data, pointer points to the data
53    element * @param[in] buf, stores data that will be carried by the data element * @param[in]
54    offset * @param[in] len, length of the buf * @param[in] use_dma, use dma or not, better use dma when length is above
55    32 bytes * @return 0: success
56
57
58
59    */
60  int dsp_a2e_data_build(struct dsp_data_elt *data, uint8_t *buf, uint16_t offset, uint16_t len, bool use_dma);
61
62  /*
63    * Move the data in e2a data element from dsp memory to main CPU memory. * @param[in] dst,
64    destination * @param[in] src, source * @param[in]
65    len, length * @param[in] use_dma, use
66    dma or not, better use dma when length
67    is above 32 bytes * @return 0: success
68
69    */
70  int dsp_e2a_data_copy(uint8_t *dst, uint8_t *src, uint16_t len, bool use_dma);
71
72  /*
73    * ÿdspÿimageÿÿÿÿÿÿdsp memoryÿÿ * @param[in] addr, start
74    addr of the image * @param[in] use_dma, use dma or not * @param[in]
75    dma_ch, dma channel * @param[in] image_check, check
76    whether the image * @return 0: success
77
78
79    */
80  void dsp_image_load(uint32_t addr, bool use_dma, uint8_t dma_ch, bool image_check);
81
82  /* *
83    Start DSP, addr must be consistent with the address loaded by
84    image. */
85  void dsp_lauch(addr);
```

The DSP-side API is similar to the main APP-side API and will not be introduced here.