版本	V1.0
日期	202012

APT32F102 系列

CSI 代码结构和使用说明

相关文档：

各IP对应的使用说明

剑池CDKV2.6版本特性

版权所有©深圳市爱普特微电子有限公司

本资料内容为深圳市爱普特微电子有限公司在现有数据资料基础上慎重且力求准确无误编制而成，本资料中所记载的实例以正确的使用方法和标准操作为前提，使用方在应用该等实例时请充分考虑外部诸条件，深圳市爱普特微电子有限公司不担保或确认该等实例在使用方的适用性、适当性或完整性，深圳市爱普特微电子有限公司亦不对使用方因使用本资料所有内容而可能或已经带来的风险或后果承担任何法律责任。基于使本资料的内容更加完善等原因，公司保留未经预告的修改权。

Revision History

版本	日期	描述	作者
V1.0	2020-12	新建	魏柠柠

目录

1. 概述 1
2. 代码结构 1
3. 芯片资源分配 2
4. 时钟设置 3
5. 初始化 3
5.1 上电初始化 4
5.2 管脚初始化 4
5.3 设备初始化 4
5.3.1 说明 4
5.3.2 实现 5
6. 中断的实现 6
6.1.1 说明 6
6.1.2 实现 7
7. Q & A 7
7.1 Q1：如果 CSI 的代码架构不能满足系统对实时性的要求怎么办？ 7
7.2 Q2：如果 CSI 现有代码（API）不能满足特定的应用场合，怎么办？ 8
7.3 Q3：CSI 代码可以在老版本 CDK 上运行吗？ 8
8. 附录 1 剑池 CDKV2.6 版本使用注意事项 8
8.1 删除操作 8
8.2 新增功能 8

1. 概述

本文主要说明于 APT32F102x 系列 CSI（Chip Standard Interface）代码结构和使用说明。

2. 代码结构

CSI 充分考虑应用，将 csp 层驱动代码合理组织打包。csp 层只对 CSI 层和 sys 层开放。原则上，用户层只能调用 CSI 层和 sys 层的 API。

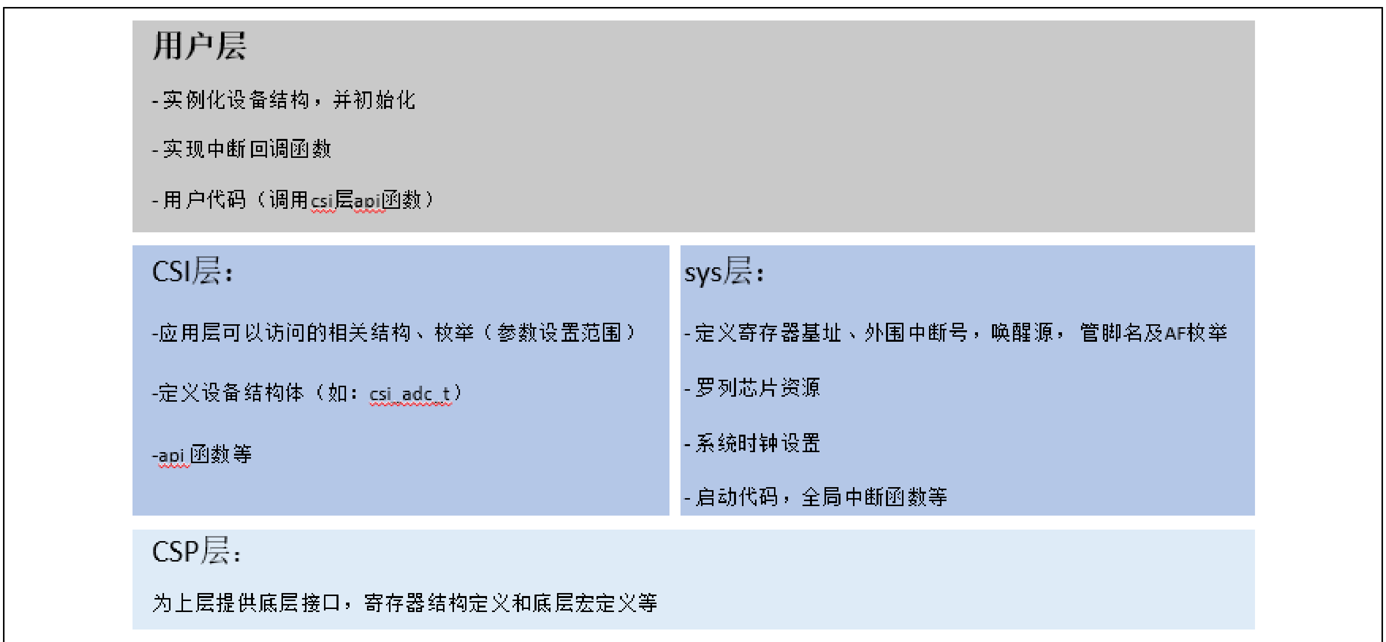

Figure 1 代码层次结构

在CDK中，开发例程呈现如下：

Figure 2 工程目录结构

注 1：只有绿色框包含的文件是芯片使用者可能需要改动的文件。

注 2：黄色文本框里的内容是对 CDK2.6（含）以上版本中组件概念的说明。新版本特性详见***剑池 CDK V2.6 版本特性.pdf***

3. 芯片资源分配

资源分配包括两部分内容：IP资源和管脚资源。

工程视图中，需要修改的文件位于：sdk_102 -> board -> include -> board_config.h

实际文件路径为：工程目录\board\include

```
/* example pin manager */

#define CONSOLE_IDX              0
#define CONSOLE_TXD              PA00
#define CONSOLE_RXD              PA01
#define CONSOLE_TXD_FUNC         PA01_UART0_TX
#define CONSOLE_RXD_FUNC         PA00_UART0_RX

#define EXI_PIN                  PA09
#define EXI_PIN_FUNC             PIN_FUNC_INPUT

#define SPI_IDX                  0
#define SPI_MOSI_PIN             PA014
#define SPI_MISO_PIN             PA015
#define SPI_NSS_PIN              PB05
#define SPI_SCK_PIN              PB04
#define SPI_MOSI_PIN_FUNC        PA014_SPI_MOSI
#define SPI_MISO_PIN_FUNC        PA015_SPI_MISO
#define SPI_NSS_PIN_FUNC         PB05_SPI_NSS
#define SPI_SCK_PIN_FUNC         PB04_SPI_SCK
```

Figure 3 管脚资源配置示例

xxx_IDX 即表示 IP 资源的使用情况：0 表示第 0 个，如 UART0。芯片的片上资源罗列在 g_soc_info[ ] 这个表里，位于 sdk_102 -> chip -> sys –> devices.c 中。

```
const csi_perip_info_t g_soc_info[] = {

    {CK801_ADDR_BASE,          CORET_IRQn,          0,    DEV_CORET_TAG},
    {APB_SYS_BASE,             SYSCON_IRQn,         0,    DEV_SYSCON_TAG},
    {APB_IFC_BASE,             IFC_IRQn,            0,    DEV_IFC_TAG},
    {APB_ADC0_BASE,            ADC_IRQn,            0,    DEV_ADC_TAG},
    {APB_EPT0_BASE,            EPT0_IRQn,           0,    DEV_EPT_TAG},
```

Figure 4 芯片资源表

4. 时钟设置

工程视图中，需要修改的文件位于：sdk_102 -> board -> src -> board_config.c

实际文件路径为：工程目录\board\src

```
/// system clock configuration parameters to define source, source freq(if selectable), sdiv and pdiv
const system_clk_config_t g_tSystemClkConfig[1] = {

    {SRC_HFOSC, HFOSC_48M_VALUE, SCLK_DIV2, PCLK_DIV1}
    //{SRC_EMOSC, 20000000, SCLK_DIV1, PCLK_DIV2}
    //{SRC_IMOSC, IMOSC_5M_VALUE, SCLK_DIV1, PCLK_DIV1}
    //{SRC_HFOSC, HFOSC_48M_VALUE, SCLK_DIV2, PCLK_DIV1}
    //{SRC_IMOSC, IMOSC_5M_VALUE, SCLK_DIV1, PCLK_DIV1}
};
```

Figure 5 芯片时钟配置

数组g_tSystemClkConfig包含4个成员，分别是时钟源、时钟源频率，SCLK分频值和PCLK分频值。可在sys_clk.h（sdk_102 -> chip -> sys）中查询枚举值。

5. 初始化

5.1 上电初始化

芯片上电后，启动文件（汇编）会调用__main()完成SRAM数据初始化后，跳转main（）。

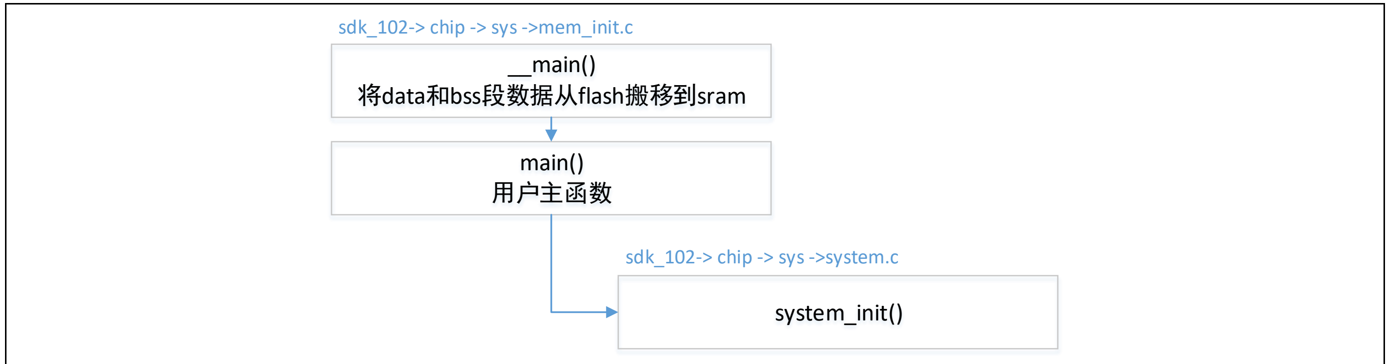

Figure 6 上电启动流程

main（）开始的时候调用system_init()实现程序正式运行前的准备工作，需要包含但不限于下述内容。

```
void system_init(void)
{
    CK_CPU_DISALLNORMALIRQ;

    csi_reliability_init();
    csi_wdt_init(&tIwdt, 0);
    csi_wdt_stop(&tIwdt);
    soc_sysclk_config();
    soc_pclk_config();
    soc_get_cpu_freq();
    soc_get_apb_freq(0);
    csi_tick_init();

    CK_CPU_ENALLNORMALIRQ;
}
```

调试或要更改IWDT默认配置时

必须有

用delay函数时

Figure 7 system_init()函数

5.2 管脚初始化

会用到资源分配章节里提到的board_config.h中的宏定义。

```
csi_pin_set_mux(ADC_PIN, ADC_PIN_FUNC);
```

Figure 8 管脚初始化示例

5.3 设备初始化

5.3.1 说明

在CSI的代码结构里，芯片所有的片上资源都是设备的概念。默认情况下，所有设备不可用。如果应用程序需要使用某个设备，比如ADC，必须先做初始化的工作。这包括两个部分：

- 实例化一个设备结构体。每个设备的结构体有共通的部分，也可能有设备独特的部分。比如在ADC中，

```
typedef struct csi_adc csi_adc_t;
struct csi_adc {
    csi_dev_t               dev;        ///< Hw-device info
    void (*callback)(csi_adc_t *adc, csi_adc_event_t event, void *arg);   ///< User callback ,signaled by driver event
    void                   *arg;        ///< 
    uint32_t               *data;       ///< Data buf
    uint32_t                num;        ///< Data size
    csi_error_t            (*start)(csi_adc_t *adc);   ///< Start function
    csi_error_t            (*stop)(csi_adc_t *adc);    ///< Stop function
    csi_state_t            state;       ///< current state
    void                   *priv;
};
```


Figure 9 ADC 的设备结构体

- 调用对应的初始化函数。每个设备都有初始化函数CSI_xxx_init（）。其中会打开设备时钟、初始化设备结构体及一些设备特有的初始化操作。

```
csi_error_t csi_adc_init(csi_adc_t *adc, uint32_t idx)
{
    CSI_PARAM_CHK(adc, CSI_ERROR);
    csi_error_t ret = CSI_OK;
    csp_adc_t *adc_base;

    adc->priv = 0U;          //clr adc seq num

    if (target_get(DEV_ADC_TAG, idx, &adc->dev) != CSI_OK)
        ret = CSI_ERROR;
    else
    {
        adc_base = (csp_adc_t *)HANDLE_REG_BASE(adc);

        adc->state.writeable = 1U;
        adc->state.readable  = 1U;
        adc->state.error     = 0U;
        adc->callback        = NULL;
        adc->arg             = NULL;
        adc->data            = NULL;
        adc->start           = NULL;
        adc->stop            = NULL;

        csi_clk_enable(&adc->dev);                          //adc peripheral clk en
        csp_adc_def_Init(adc_base);                         //reset all registers
        csp_adc_set_clk(adc_base,ENABLE);                   //ADC CLK ENABLE
        csp_adc_set_bit_num(adc_base,ADC12_12BIT);          //12BIT AD
        csp_adc_set_vref(adc_base,VERF_VDD_VSS);            //ADC VREF
        csp_adc_en(adc_base);                               //enable  adc
    }
    return ret;

}
```


Figure 10 ADC设备初始化函数

5.3.2 实现

综上所述，要使用片上任何一个设备，至少需要做以下两步：

```
csi_adc_t    g_tAdc;

csi_adc_init(&g_tAdc, ADC_IDX);
```

Figure 11 ADC 设备初始化

初始化函数的第二个参数，是第几个 ADC 的意思。这在一个芯片内有多个一样的 IP 时是有意义的。如果只有一个，比如 APT32F102 里只有一个 ADC 的情况，第二参数就设为 0。芯片的片上资源罗列在 g_soc_info[ ] 表里，位于 sdk_102 -> chip -> sys –> devices.c 中。

6. 中断的实现

6.1.1 说明

```
// External interrupts
.long do_irq        //CORETHandler
.long do_irq        //SYSCONIntHandler
.long csp_ifc_irq_handler       //IFCIntHandler
.long do_irq        //ADCIntHandler
.long do_irq        //EPT0IntHandler
.long DummyHandler//EPT0EMIntHandler
.long do_irq        //WWDTHandler
.long do_irq        //EXI0IntHandler
.long do_irq        //EXI1IntHandler
.long do_irq        //GPT0IntHandler
.long DummyHandler//GPT1IntHandler
.long DummyHandler
.long do_irq        //RTCIntHandler
.long do_irq        //UART0IntHandler
.long do_irq        //UART1IntHandler
.long do_irq        //UART2IntHandler//USARTIntHandler
.long DummyHandler
.long do_irq        //I2CIntHandler
.long DummyHandler
.long do_irq        //SPI0IntHandler
.long do_irq        //SIO0IntHandler
.long do_irq        //EXI2to3IntHandler
.long do_irq        //EXI4to9IntHandler
.long do_irq        //EXI10to13IntHandler
.long do_irq        //CNTAIntHandler
.long do_irq        //TKEYIntHandler
.long do_irq        //LPTIntHandler
.long DummyHandler//LEDIntHandler
.long do_irq        //BT0IntHandler
.long do_irq        //BT1IntHandler
.long DummyHandler//BT2IntHandler
.long DummyHandler//BT3IntHandler
```

Figure 12 中断向量表

上图为启动代码里的中断向量表，可以看出几乎所有的中断发生后都会跳转到do_irq()。do_irq()位于sdk_102 -> chip -> sys -> irq.c，会通过查表g_irq_table[CONFIG_IRQ_NUM] 的方式选择对应的设备中断处理函数。

- 表g_irq_table[CONFIG_IRQ_NUM]在上电初期是空的，会在CSI_xxx_attach_callback（）注册后在线生成（xxx是设备名，如adc）。

- 设备的中断处理函数。CSI代码结构中，中断是使用回调的方式来实现的。每个支持中断的设备在对应的xxx.c文件中都会有apt_xxx_irqhander()函数。这就是do_irq()查表后会调用的函数。这个函数会处理清中断标志位的工作，最终会调用用户回调函数。如需使用，要通过CSI_xxx_attach_callback（）传入函数指针。

```
static void apt_adc_irqhandler(void *args)
{
    csi_adc_t *adc      = (csi_adc_t *)args;
    csp_adc_t *adc_base = (csp_adc_t *)HANDLE_REG_BASE(adc);

    uint8_t i;
    uint32_t wChnlNUm = (uint32_t)adc->priv;

    if(adc->data != NULL)
    {
        if(adc->num > 0)
        {
            for(i = 0; i < wChnlNUm; i++)
            {
                if(csp_adc_get_status(adc_base,ADC12_SEQ(i)))
                {
                    *(adc->data + i*s_byBufLen + adc->num- 1) = csp_adc_get_data(adc_base, i);
                    csp_adc_clr_status(adc_base,ADC12_SEQ(i));
                }
            }
            adc->num -- ;
        }

        if(adc->num == 0)
        {
            if (adc->callback)
            {
                adc->callback(adc, ADC_EVENT_CONVERT_COMPLETE, adc->arg);
                adc->state.readable = 10;
            }
        }
    }
}
```

Figure 13 adc_irqhandler 函数

6.1.2 实现

综上所述，如果程序需要用到某个设备的中断，需要做两个事情：

1. 调用相应的 attach_callback 函数。

```
CSI_adc_attach_callback(&g_tAdc, user_adc_event, (void *)arg);
```

2. 写用户中断回调函数。上述注册函数的第二个参量就是用户代码中的中断处理函数名。

```
void user_adc_event(CSI_adc_t *adc, CSI_adc_event_t event, void *arg)
```

arg 的使用需要参看具体的设备.c 文件。

7. Q & A

7.1 Q1：如果CSI的代码架构不能满足系统对实时性的要求怎么办？

A1：有好几种方法可以提高系统实时性。

1) 使用ETCB设备

尽可能使用 ETCB 设备，设置好输入和输出，实现不同设备间的硬件互动。

2) 使用独立的中断处理函数

如果某个设备对实时性敏感，可以将启动代码中对应位置（参看使用手册中断章节）的 do_irq（）改掉。Figure 11 中的 csp_ifc_irq_handler 就是根据需要另外定义的独立的中断入口函数。

sdk_102 -> chip -> sys -> interrupt.c 文件中已经预先架构了所有设备的独立中断处理函数。考虑到实时性，interrupt.c 文件包含了 csp.h，这意味着在这个文件里可以直接进行最底层的操作，比如 GPIOA0->CONLR = 0x22222222 这样的操作。

3) 把 csp.h 加到对执行速度敏感的源文件中，就可以在这个文件内调动 csp 接口函数或者直接进行寄存器操作。

7.2 Q2：如果CSI现有代码（API）不能满足特定的应用场合，怎么办？

A2：原则上，用户只调用 CSI 的 API 接口。但因为标准代码的覆盖率有限，所以可能会出现现有 API 不满足特定应用场合的情况。解决方法：把 csp.h 加到源文件中，就可以直接调用 csp 的接口，或者进行寄存器操作。

7.3 Q3：CSI代码可以在老版本CDK上运行吗？

A3：CSI代码结构和CDK版本并没有直接的关系。只是CDK2.6（含）以上的版本支持组件的概念，所以工程形式上存在差异。爱普特准备了两套CSI代码，一套支持CDK新版本，一套延续CDK老版本的工程结构。

8. 附录1 剑池CDKV2.6版本使用注意事项

本文只简述和常规使用密切相关的内容。具体如组件的概念请参看CDK 帮助文件和***剑池CDK V2.6版本特性.pdf。***

8.1 删除操作

CDK2.6 不再有 virtual folder 的概念，工程视图将和文件系统保持一致。这意味着，在工程视图里的删除操作将直接删除文件。

8.2 新增功能

CDK2.6新增一个进入调试的接口，点击后不下载代码，直接进入调试状态，方便查看上次运行后的flash内容。

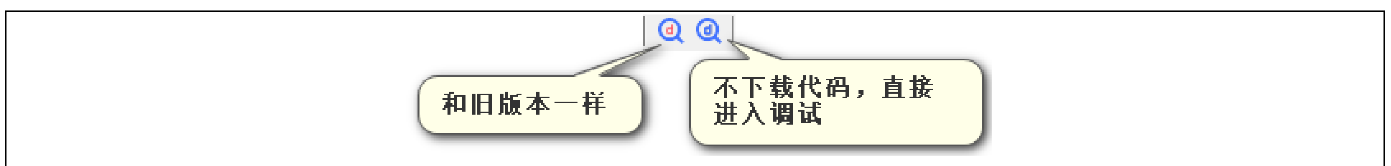

Figure 14 进入调试的按钮