K210
Standalone SDK 编程指南

KENDRYTE
勘智

关于本手册

本文档为用户提供基于 Kendryte K210 Standalone SDK 开发时的编程指南.

对应 SDK 版本

Kendryte Standalone SDK v0.5.3 (*7ba017918e595714eeeb92ce5695db90e59b8950*)

发布说明

日期	版本	发布说明
2018-10-10	V0.1.0	初始版本
2018-10-20	V0.2.0	发布对应 Standalone SDK v0.4.0 的文档
2018-11-02	V0.3.0	发布对应 Standalone SDK v0.5.1 的文档
2019-01-11	V0.4.0	发布对应 Standalone SDK v0.5.3 的文档

免责声明

本文中的信息，包括参考的 URL 地址，如有变更，恕不另行通知。文档“按现状”提供，不负任何担保责任，包括对适销性、适用于特定用途或非侵权性的任何担保，和任何提案、规格或样品在他处提到的任何担保。本文档不负任何责任，包括使用本文档内信息产生的侵犯任何专利权行为的责任。本文档在此未以禁止反言或其他方式授予任何知识产权使用许可，不管是明示许可还是暗示许可。文中提到的所有商标名称、商标和注册商标均属其各自所有者的财产，特此声明。

版权公告

目录

第 1 章

神经网络处理器（KPU）

1.1 概述

KPU 是通用的神经网络处理器，它可以在低功耗的情况下实现卷积神经网络计算，实时获取被检测目标的大小、坐标和种类，对人脸或者物体进行检测和分类。使用 kpu 时，必须结合 model compiler。

1.2 功能描述

KPU 具备以下几个特点：

- 支持主流训练框架按照特定限制规则训练出来的定点化模型
- 对网络层数无直接限制，支持每层卷积神经网络参数单独配置，包括输入输出通道数目、输入输出行宽列高
- 支持两种卷积内核 1x1 和 3x3
- 支持任意形式的激活函数
- 实时工作时最大支持神经网络参数大小为 5.5MiB 到 5.9MiB
- 非实时工作时最大支持网络参数大小为（Flash 容量-软件体积）

1.3 API 参考

对应的头文件 kpu.h
为用户提供以下接口

- kpu_task_init（0.6.0 以后不再支持，请使用 kpu_single_task_init）
- kpu_run（0.6.0 以后不再支持，请使用 kpu_start）
- kpu_get_output_buf（0.6.0 以后不再支持）

- kpu_release_output_buf（0.6.0 以后不再支持）
- kpu_start
- kpu_single_task_init
- kpu_single_task_deinit
- kpu_model_load_from_buffer

1.3.1　kpu_task_init

1.3.1.1　描述

初始化 kpu 任务句柄，该函数具体实现在 model compiler 生成的 gencode_output.c 中。

1.3.1.2　函数定义

```
kpu_task_t* kpu_task_init(kpu_task_t* task)
```

1.3.1.3　参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入

1.3.1.4　返回值

KPU 任务句柄。

1.3.2　kpu_run

1.3.2.1　描述

启动 KPU，进行 AI 运算。

1.3.2.2　函数原型

```
int kpu_run(kpu_task_t* v_task, dmac_channel_number_t dma_ch, const void *src, void*
    dest, plic_irq_callback_t callback)
```

1.3.2.3　参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入
dma_ch	DMA 通道	输入
src	输入图像数据	输入
dest	运算输出结果	输出
callback	运算完成回调函数	输入

1.3.2.4　返回值

返回值	描述
0	成功
非 0	KPU 忙，失败

1.3.3　kpu_get_output_buf

1.3.3.1　描述

获取 KPU 输出结果的缓存。

1.3.3.2　函数原型

```
uint8_t *kpu_get_output_buf(kpu_task_t* task)
```

1.3.3.3　参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入

1.3.3.4　返回值

KPU 输出结果的缓存的指针。

1.3.4　kpu_release_output_buf

1.3.4.1　描述

释放 KPU 输出结果缓存。

1.3.4.2　函数原型

```
void kpu_release_output_buf(uint8_t *output_buf)
```

1.3.4.3　参数

参数名称	描述	输入输出
output_buf	KPU 输出结果缓存	输入

1.3.4.4　返回值

无

1.3.5　kpu_start

1.3.5.1　描述

启动 KPU，进行 AI 运算。

1.3.5.2　函数原型

```
int kpu_start(kpu_task_t *task)
```

1.3.5.3　参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入

1.3.5.4　返回值

返回值	描述
0	成功
非 0	KPU 忙，失败

1.3.6　kpu_single_task_init

1.3.6.1　描述

初始化 kpu 任务句柄。

1.3.6.2　函数原型

```
int kpu_single_task_init(kpu_task_t *task)
```

1.3.6.3　参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入

1.3.6.4　返回值

返回值	描述
0	成功
非 0	失败

1.3.7　kpu_single_task_deinit

1.3.7.1　描述

注销 kpu 任务。

1.3.7.2　函数原型

```
int kpu_single_task_deinit(kpu_task_t *task)
```

1.3.7.3　参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入

1.3.7.4　返回值

返回值	描述
0	成功
非 0	失败

1.3.8　kpu_model_load_from_buffer

1.3.8.1　描述

解析 kmodel 并初始化 kpu 句柄。

1.3.8.2　函数原型

```
int kpu_model_load_from_buffer(kpu_task_t *task, uint8_t *buffer,
    kpu_model_layer_metadata_t **meta);
```

1.3.8.3　参数

参数名称	描述	输入输出
task	KPU 任务句柄	输入
buffer	kmodel 数据	输入
meta	内部测试数据，用户设置为 NULL	输出

1.3.8.4　返回值

返回值	描述
0	成功
非 0	失败

1.3.9　举例

```
/* 通过MC生成kpu_task_gencode_output_init,设置源数据为g_ai_buf,使用DMA5,kpu完成后调
    用ai_done函数 */
kpu_task_t task;
volatile uint8_t g_ai_done_flag;
static int ai_done(void *ctx)
{
    g_ai_done_flag = 1;
    return 0;
```

```
}

/* 初始化kpu */
kpu_task_gencode_output_init(&task); /* MC生成的函数 */
task.src = g_ai_buf;
task.dma_ch = 5;
task.callback = ai_done;
kpu_single_task_init(&task);

/* 启动kpu */
kpu_start(&task);
```

1.4　数据类型

相关数据类型、数据结构定义如下：

- kpu_task_t：kpu 任务结构体。

1.4.1　kpu_task_t

1.4.1.1　描述

kpu 任务结构体。

1.4.1.2　定义

```
typedef struct
{
    kpu_layer_argument_t *layers;
    kpu_layer_argument_t *remain_layers;
    plic_irq_callback_t callback;
    void *ctx;
    uint64_t *src;
    uint64_t *dst;
    uint32_t src_length;
    uint32_t dst_length;
    uint32_t layers_length;
    uint32_t remain_layers_length;
    dmac_channel_number_t dma_ch;
    uint32_t eight_bit_mode;
    float output_scale;
    float output_bias;
    float input_scale;
    float input_bias;
} kpu_task_t;
```

1.4.1.3　成员

成员名称	描述
layers	KPU 参数指针 (MC 初始化，用户不必关心)
remain_layers	KPU 参数指针（运算过程中使用，用户不必关心）
callback	运算完成回调函数（需要用户设置）
ctx	回调函数的参数（非空需要用户设置）
src	运算源数据（需要用户设置）
dst	运算结果输出指针（KPU 初始化赋值，用户不必关心）
src_length	源数据长度 (MC 初始化，用户不必关心)
dst_length	运算结果长度 (MC 初始化，用户不必关心)
layers_length	层数 (MC 初始化，用户不必关心)
remain_layers_length	剩余层数（运算过程中使用，用户不必关心）
dma_ch	使用的 DMA 通道号（需要用户设置）
eight_bit_mode	是否是 8 比特模式 (MC 初始化，用户不必关心)
output_scale	输出 scale 值 (MC 初始化，用户不必关心)
output_bias	输出 bias 值 (MC 初始化，用户不必关心)
input_scale	输入 scale 值 (MC 初始化，用户不必关心)
input_bias	输入 bias 值 (MC 初始化，用户不必关心)

第 2 章

高级加密加速器 (AES)

2.1 功能描述

K210 内置 AES(高级加密加速器)，相对于软件可以极大的提高 AES 运算速度。AES 加速器支持多种加密/解密模式 (ECB,CBC,GCM)，多种长度的 KEY(128,192,256) 的运算。

2.2 API 参考

对应的头文件 aes.h
为用户提供以下接口

- aes_ecb128_hard_encrypt
- aes_ecb128_hard_decrypt
- aes_ecb192_hard_encrypt
- aes_ecb192_hard_decrypt
- aes_ecb256_hard_encrypt
- aes_ecb256_hard_decrypt
- aes_cbc128_hard_encrypt
- aes_cbc128_hard_decrypt
- aes_cbc192_hard_encrypt
- aes_cbc192_hard_decrypt
- aes_cbc256_hard_encrypt
- aes_cbc256_hard_decrypt
- aes_gcm128_hard_encrypt
- aes_gcm128_hard_decrypt
- aes_gcm192_hard_encrypt

- aes_gcm192_hard_decrypt
- aes_gcm256_hard_encrypt
- aes_gcm256_hard_decrypt
- aes_ecb128_hard_encrypt_dma
- aes_ecb128_hard_decrypt_dma
- aes_ecb192_hard_encrypt_dma
- aes_ecb192_hard_decrypt_dma
- aes_ecb256_hard_encrypt_dma
- aes_ecb256_hard_decrypt_dma
- aes_cbc128_hard_encrypt_dma
- aes_cbc128_hard_decrypt_dma
- aes_cbc192_hard_encrypt_dma
- aes_cbc192_hard_decrypt_dma
- aes_cbc256_hard_encrypt_dma
- aes_cbc256_hard_decrypt_dma
- aes_gcm128_hard_encrypt_dma
- aes_gcm128_hard_decrypt_dma
- aes_gcm192_hard_encrypt_dma
- aes_gcm192_hard_decrypt_dma
- aes_gcm256_hard_encrypt_dma
- aes_gcm256_hard_decrypt_dma
- aes_init
- aes_process
- gcm_get_tag

2.2.1　aes_ecb128_hard_encrypt

2.2.1.1　描述

AES-ECB-128 加密运算。输入输出数据都使用 cpu 传输。ECB 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。ECB 模式没有用到向量。

2.2.1.2　函数原型

```
void aes_ecb128_hard_encrypt(uint8_t *input_key, uint8_t *input_data, size_t input_len,
     uint8_t *output_data)
```

2.2.1.3　参数

参数名称	描述	输入输出
input_key	AES-ECB-128 加密的密钥	输入
input_data	AES-ECB-128 待加密的明文数据	输入
input_len	AES-ECB-128 待加密明文数据的长度	输入
output_data	AES-ECB-128 加密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.1.4　返回值

无。

2.2.2　aes_ecb128_hard_decrypt

2.2.2.1　描述

AES-ECB-128 解密运算。输入输出数据都使用 cpu 传输。ECB 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。ECB 模式没有用到向量。

2.2.2.2　函数原型

```
void aes_ecb128_hard_decrypt(uint8_t *input_key, uint8_t *input_data, size_t input_len,
     uint8_t *output_data)
```

2.2.2.3　参数

参数名称	描述	输入输出
input_key	AES-ECB-128 解密的密钥	输入
input_data	AES-ECB-128 待解密的密文数据	输入
input_len	AES-ECB-128 待解密密文数据的长度	输入

参数名称	描述	输入输出
output_data	AES-ECB-128 解密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.2.4　返回值

无。

2.2.3　aes_ecb192_hard_encrypt

2.2.3.1　描述

AES-ECB-192 加密运算。输入输出数据都使用 cpu 传输。ECB 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。ECB 模式没有用到向量。

2.2.3.2　函数原型

```
void aes_ecb192_hard_encrypt(uint8_t *input_key, uint8_t *input_data, size_t input_len,
     uint8_t *output_data)
```

2.2.3.3　参数

参数名称	描述	输入输出
input_key	AES-ECB-192 加密的密钥	输入
input_data	AES-ECB-192 待加密的明文数据	输入
input_len	AES-ECB-192 待加密明文数据的长度	输入
output_data	AES-ECB-192 加密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.3.4　返回值

无。

2.2.4　aes_ecb192_hard_decrypt

2.2.4.1　描述

AES-ECB-192 解密运算。输入输出数据都使用 cpu 传输。ECB 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。ECB 模式没有用到向量。

2.2.4.2　函数原型

```
void aes_ecb192_hard_decrypt(uint8_t *input_key, uint8_t *input_data, size_t input_len,
     uint8_t *output_data)
```

2.2.4.3　参数

参数名称	描述	输入输出
input_key	AES-ECB-192 解密的密钥	输入
input_data	AES-ECB-192 待解密的密文数据	输入
input_len	AES-ECB-192 待解密密文数据的长度	输入
output_data	AES-ECB-192 解密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.4.4　返回值

无。

2.2.5　aes_ecb256_hard_encrypt

2.2.5.1　描述

AES-ECB-256 加密运算。输入输出数据都使用 cpu 传输。ECB 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。ECB 模式没有用到向量。

2.2.5.2　函数原型

```
void aes_ecb256_hard_encrypt(uint8_t *input_key, uint8_t *input_data, size_t input_len,
     uint8_t *output_data)
```

2.2.5.3　参数

参数名称	描述	输入输出
input_key	AES-ECB-256 加密的密钥	输入
input_data	AES-ECB-256 待加密的明文数据	输入
input_len	AES-ECB-256 待加密明文数据的长度	输入
output_data	AES-ECB-256 加密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.5.4　返回值

无。

2.2.6　aes_ecb256_hard_decrypt

2.2.6.1　描述

AES-ECB-256 解密运算。输入输出数据都使用 cpu 传输。ECB 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。ECB 模式没有用到向量。

2.2.6.2　函数原型

```
void aes_ecb256_hard_decrypt(uint8_t *input_key, uint8_t *input_data, size_t input_len,
     uint8_t *output_data)
```

2.2.6.3　参数

参数名称	描述	输入输出
input_key	AES-ECB-256 解密的密钥	输入
input_data	AES-ECB-256 待解密的密文数据	输入

参数名称	描述	输入输出
input_len	AES-ECB-256 待解密密文数据的长度	输入
output_data	AES-ECB-256 解密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.6.4　返回值

无。

2.2.7　aes_cbc128_hard_encrypt

2.2.7.1　描述

AES-CBC-128 加密运算。输入输出数据都使用 cpu 传输。CBC 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。

2.2.7.2　函数原型

```
void aes_cbc128_hard_encrypt(cbc_context_t *context, uint8_t *input_data, size_t
    input_len, uint8_t *output_data)
```

2.2.7.3　参数

参数名称	描述	输入输出
context	AES-CBC-128 加密计算的结构体，包含加密密钥与偏移向量	输入
input_data	AES-CBC-128 待加密的明文数据	输入
input_len	AES-CBC-128 待加密明文数据的长度	输入
output_data	AES-CBC-128 加密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.7.4　返回值

无。

2.2.8　aes_cbc128_hard_decrypt

2.2.8.1　描述

AES-CBC-128 解密运算。输入输出数据都使用 cpu 传输。CBC 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。

2.2.8.2　函数原型

```
void aes_cbc128_hard_decrypt(cbc_context_t *context, uint8_t *input_data, size_t
    input_len, uint8_t *output_data)
```

2.2.8.3　参数

参数名称	描述	输入输出
context	AES-CBC-128 解密计算的结构体，包含解密密钥与偏移向量	输入
input_data	AES-CBC-128 待解密的密文数据	输入
input_len	AES-CBC-128 待解密密文数据的长度	输入
output_data	AES-CBC-128 解密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.8.4　返回值

无。

2.2.9　aes_cbc192_hard_encrypt

2.2.9.1　描述

AES-CBC-192 加密运算。输入输出数据都使用 cpu 传输。CBC 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。

2.2.9.2　函数原型

```
void aes_cbc192_hard_encrypt(cbc_context_t *context, uint8_t *input_data, size_t
    input_len, uint8_t *output_data)
```

2.2.9.3　参数

参数名称	描述	输入输出
context	AES-CBC-192 加密计算的结构体，包含加密密钥与偏移向量	输入
input_data	AES-CBC-192 待加密的明文数据	输入
input_len	AES-CBC-192 待加密明文数据的长度	输入
output_data	AES-CBC-192 加密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.9.4　返回值

无。

2.2.10　aes_cbc192_hard_decrypt

2.2.10.1　描述

AES-CBC-192 解密运算。输入输出数据都使用 cpu 传输。CBC 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。

2.2.10.2　函数原型

```
void aes_cbc192_hard_decrypt(cbc_context_t *context, uint8_t *input_data, size_t
    input_len, uint8_t *output_data)
```

2.2.10.3　参数

参数名称	描述	输入输出
context	AES-CBC-192 解密计算的结构体，包含解密密钥与偏移向量	输入
input_data	AES-CBC-192 待解密的密文数据	输入
input_len	AES-CBC-192 待解密密文数据的长度	输入
output_data	AES-CBC-192 解密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.10.4　返回值

无。

2.2.11　aes_cbc256_hard_encrypt

2.2.11.1　描述

AES-CBC-256 加密运算。输入输出数据都使用 cpu 传输。CBC 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。

2.2.11.2　函数原型

```
void aes_cbc256_hard_encrypt(cbc_context_t *context, uint8_t *input_data, size_t
    input_len, uint8_t *output_data)
```

2.2.11.3　参数

参数名称	描述	输入输出
context	AES-CBC-256 加密计算的结构体，包含加密密钥与偏移向量	输入
input_data	AES-CBC-256 待加密的明文数据	输入
input_len	AES-CBC-256 待加密明文数据的长度	输入

参数名称	描述	输入输出
output_data	AES-CBC-256 加密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.11.4　返回值

无。

2.2.12　aes_cbc256_hard_decrypt

2.2.12.1　描述

AES-CBC-256 解密运算。输入输出数据都使用 cpu 传输。CBC 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。

2.2.12.2　函数原型

```
void aes_cbc256_hard_decrypt(uint8_t *input_key, uint8_t *input_data, size_t input_len,
     uint8_t *output_data)
```

2.2.12.3　参数

参数名称	描述	输入输出
context	AES-CBC-256 解密计算的结构体，包含解密密钥与偏移向量	输入
input_data	AES-CBC-256 待解密的密文数据	输入
input_len	AES-CBC-256 待解密密文数据的长度	输入
output_data	AES-CBC-256 解密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.12.4　返回值

无。

2.2.13　aes_gcm128_hard_encrypt

2.2.13.1　描述

AES-GCM-128 加密运算。输入输出数据都使用 cpu 传输。

2.2.13.2　函数原型

```
void aes_gcm128_hard_encrypt(gcm_context_t *context, uint8_t *input_data, size_t
    input_len, uint8_t *output_data, uint8_t *gcm_tag)
```

2.2.13.3　参数

参数名称	描述	输入输出
context	AES-GCM-128 加密计算的结构体，包含加密密钥/偏移向量/aad/aad 长度	输入
input_data	AES-GCM-128 待加密的明文数据	输入
input_len	AES-GCM-128 待加密明文数据的长度	输入
output_data	AES-GCM-128 加密运算后的结果存放在这个 buffer	输出
gcm_tag	AES-GCM-128 加密运算后的 tag 存放在这个 buffer。这个 buffer 大小需要保证为 16bytes	输出

2.2.13.4　返回值

无。

2.2.14　aes_gcm128_hard_decrypt

2.2.14.1　描述

AES-GCM-128 解密运算。输入输出数据都使用 cpu 传输。

2.2.14.2　函数原型

```
void aes_gcm128_hard_decrypt(gcm_context_t *context, uint8_t *input_data, size_t
    input_len, uint8_t *output_data, uint8_t *gcm_tag)
```

2.2.14.3　参数

参数名称	描述	输入输出
context	AES-GCM-128 解密计算的结构体，包含解密密钥/偏移向量/aad/aad 长度	输入
input_data	AES-GCM-128 待解密的密文数据	输入
input_len	AES-GCM-128 待解密密文数据的长度	输入
output_data	AES-GCM-128 解密运算后的结果存放在这个 buffer	输出
gcm_tag	AES-GCM-128 解密运算后的 tag 存放在这个 buffer。这个 buffer 大小需要保证为 16bytes	输出

2.2.14.4　返回值

无。

2.2.15　aes_gcm192_hard_encrypt

2.2.15.1　描述

AES-GCM-192 加密运算。输入输出数据都使用 cpu 传输。

2.2.15.2　函数原型

```
void aes_gcm192_hard_encrypt(gcm_context_t *context, uint8_t *input_data, size_t
    input_len, uint8_t *output_data, uint8_t *gcm_tag)
```

2.2.15.3　参数

参数名称	描述	输入输出
context	AES-GCM-192 加密计算的结构体，包含加密密钥/偏移向量/aad/aad 长度	输入
input_data	AES-GCM-192 待加密的明文数据	输入
input_len	AES-GCM-192 待加密明文数据的长度	输入
output_data	AES-GCM-192 加密运算后的结果存放在这个 buffer	输出
gcm_tag	AES-GCM-192 加密运算后的 tag 存放在这个 buffer。这个 buffer 大小需要保证为 16bytes	输出

2.2.15.4　返回值

无。

2.2.16　aes_gcm192_hard_decrypt

2.2.16.1　描述

AES-GCM-192 解密运算。输入输出数据都使用 cpu 传输。

2.2.16.2　函数原型

```
void aes_gcm192_hard_decrypt(gcm_context_t *context, uint8_t *input_data, size_t
    input_len, uint8_t *output_data, uint8_t *gcm_tag)
```

2.2.16.3　参数

参数名称	描述	输入输出
context	AES-GCM-192 解密计算的结构体，包含解密密钥/偏移向量/aad/aad 长度	输入
input_data	AES-GCM-192 待解密的密文数据	输入
input_len	AES-GCM-192 待解密密文数据的长度	输入
output_data	AES-GCM-192 解密运算后的结果存放在这个 buffer	输出
gcm_tag	AES-GCM-192 解密运算后的 tag 存放在这个 buffer。这个 buffer 大小需要保证为 16bytes	输出

2.2.16.4　返回值

无。

2.2.17　aes_gcm256_hard_encrypt

2.2.17.1　描述

AES-GCM-256 加密运算。输入输出数据都使用 cpu 传输。

2.2.17.2　函数原型

```
void aes_gcm256_hard_encrypt(gcm_context_t *context, uint8_t *input_data, size_t
    input_len, uint8_t *output_data, uint8_t *gcm_tag)
```

2.2.17.3　参数

参数名称	描述	输入输出
context	AES-GCM-256 加密计算的结构体，包含加密密钥/偏移向量/aad/aad 长度	输入
input_data	AES-GCM-256 待加密的明文数据	输入
input_len	AES-GCM-256 待加密明文数据的长度	输入
output_data	AES-GCM-256 加密运算后的结果存放在这个 buffer	输出
gcm_tag	AES-GCM-256 加密运算后的 tag 存放在这个 buffer。这个 buffer 大小需要保证为 16bytes	输出

2.2.17.4　返回值

无。

2.2.18　aes_gcm256_hard_decrypt

2.2.18.1　描述

AES-GCM-256 解密运算。输入输出数据都使用 cpu 传输。

2.2.18.2　函数原型

```
void aes_gcm256_hard_decrypt(gcm_context_t *context, uint8_t *input_data, size_t
    input_len, uint8_t *output_data, uint8_t *gcm_tag)
```

2.2.18.3　参数

参数名称	描述	输入输出
context	AES-GCM-256 解密计算的结构体，包含解密密钥/偏移向量/aad/aad 长度	输入
input_data	AES-GCM-256 待解密的密文数据	输入
input_len	AES-GCM-256 待解密密文数据的长度	输入
output_data	AES-GCM-256 解密运算后的结果存放在这个 buffer	输出
gcm_tag	AES-GCM-256 解密运算后的 tag 存放在这个 buffer。这个 buffer 大小需要保证为 16bytes	输出

2.2.18.4　返回值

无。

2.2.19　aes_ecb128_hard_encrypt_dma

2.2.19.1　描述

AES-ECB-128 加密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。ECB 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。ECB 模式没有用到向量。

2.2.19.2　函数原型

```
void aes_ecb128_hard_encrypt_dma(dmac_channel_number_t dma_receive_channel_num, uint8_t
     *input_key, uint8_t *input_data, size_t input_len, uint8_t *output_data)
```

2.2.19.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
input_key	AES-ECB-128 加密的密钥	输入
input_data	AES-ECB-128 待加密的明文数据	输入
input_len	AES-ECB-128 待加密明文数据的长度	输入
output_data	AES-ECB-128 加密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.19.4　返回值

无。

2.2.20　aes_ecb128_hard_decrypt_dma

2.2.20.1　描述

AES-ECB-128 解密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。ECB 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。ECB 模式没有用到向量。

2.2.20.2　函数原型

```
void aes_ecb128_hard_decrypt_dma(dmac_channel_number_t dma_receive_channel_num, uint8_t
    *input_key, uint8_t *input_data, size_t input_len, uint8_t *output_data)
```

2.2.20.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
input_key	AES-ECB-128 解密的密钥	输入
input_data	AES-ECB-128 待解密的密文数据	输入
input_len	AES-ECB-128 待解密密文数据的长度	输入

参数名称	描述	输入输出
output_data	AES-ECB-128 解密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.20.4　返回值

无。

2.2.21　aes_ecb192_hard_encrypt_dma

2.2.21.1　描述

AES-ECB-192 加密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。ECB 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。ECB 模式没有用到向量。

2.2.21.2　函数原型

```
void aes_ecb192_hard_encrypt_dma(dmac_channel_number_t dma_receive_channel_num, uint8_t
    *input_key, uint8_t *input_data, size_t input_len, uint8_t *output_data)
```

2.2.21.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
input_key	AES-ECB-192 加密的密钥	输入
input_data	AES-ECB-192 待加密的明文数据	输入
input_len	AES-ECB-192 待加密明文数据的长度	输入
output_data	AES-ECB-192 加密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.21.4　返回值

无。

2.2.22　aes_ecb192_hard_decrypt_dma

2.2.22.1　描述

AES-ECB-192 解密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。ECB 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。ECB 模式没有用到向量。

2.2.22.2　函数原型

```
void aes_ecb192_hard_decrypt_dma(dmac_channel_number_t dma_receive_channel_num, uint8_t
    *input_key, uint8_t *input_data, size_t input_len, uint8_t *output_data)
```

2.2.22.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
input_key	AES-ECB-192 解密的密钥	输入
input_data	AES-ECB-192 待解密的密文数据	输入
input_len	AES-ECB-192 待解密密文数据的长度	输入
output_data	AES-ECB-192 解密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.22.4　返回值

无。

2.2.23　aes_ecb256_hard_encrypt_dma

2.2.23.1　描述

AES-ECB-256 加密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。ECB 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。ECB 模式没有用到向量。

2.2.23.2　函数原型

```
void aes_ecb256_hard_encrypt_dma(dmac_channel_number_t dma_receive_channel_num, uint8_t
    *input_key, uint8_t *input_data, size_t input_len, uint8_t *output_data)
```

2.2.23.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
input_key	AES-ECB-256 加密的密钥	输入
input_data	AES-ECB-256 待加密的明文数据	输入
input_len	AES-ECB-256 待加密明文数据的长度	输入
output_data	AES-ECB-256 加密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.23.4　返回值

无。

2.2.24　aes_ecb256_hard_decrypt_dma

2.2.24.1　描述

AES-ECB-256 解密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。ECB 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。ECB 模式没有用到向量。

2.2.24.2　函数原型

```
void aes_ecb256_hard_decrypt_dma(dmac_channel_number_t dma_receive_channel_num, uint8_t
    *input_key, uint8_t *input_data, size_t input_len, uint8_t *output_data)
```

2.2.24.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
input_key	AES-ECB-256 解密的密钥	输入

参数名称	描述	输入输出
input_data	AES-ECB-256 待解密的密文数据	输入
input_len	AES-ECB-256 待解密密文数据的长度	输入
output_data	AES-ECB-256 解密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.24.4　返回值

无。

2.2.25　aes_cbc128_hard_encrypt_dma

2.2.25.1　描述

AES-CBC-128 加密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。CBC 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。

2.2.25.2　函数原型

```
void aes_cbc128_hard_encrypt_dma(dmac_channel_number_t dma_receive_channel_num,
    cbc_context_t *context, uint8_t *input_data, size_t input_len, uint8_t *
    output_data)
```

2.2.25.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
context	AES-CBC-128 加密计算的结构体，包含加密密钥与偏移向量	输入
input_data	AES-CBC-128 待加密的明文数据	输入
input_len	AES-CBC-128 待加密明文数据的长度	输入

参数名称	描述	输入输出
output_data	AES-CBC-128 加密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.25.4　返回值

无。

2.2.26　aes_cbc128_hard_decrypt_dma

2.2.26.1　描述

AES-CBC-128 解密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。CBC 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。

2.2.26.2　函数原型

```
void aes_cbc128_hard_decrypt_dma(dmac_channel_number_t dma_receive_channel_num,
    cbc_context_t *context, uint8_t *input_data, size_t input_len, uint8_t *
    output_data)
```

2.2.26.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
context	AES-CBC-128 解密计算的结构体，包含解密密钥与偏移向量	输入
input_data	AES-CBC-128 待解密的密文数据	输入
input_len	AES-CBC-128 待解密密文数据的长度	输入
output_data	AES-CBC-128 解密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.26.4　返回值

无。

2.2.27　aes_cbc192_hard_encrypt_dma

2.2.27.1　描述

AES-CBC-192 加密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。CBC 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。

2.2.27.2　函数原型

```
void aes_cbc192_hard_encrypt_dma(dmac_channel_number_t dma_receive_channel_num,
    cbc_context_t *context, uint8_t *input_data, size_t input_len, uint8_t *
    output_data)
```

2.2.27.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
context	AES-CBC-192 加密计算的结构体，包含加密密钥与偏移向量	输入
input_data	AES-CBC-192 待加密的明文数据	输入
input_len	AES-CBC-192 待加密明文数据的长度	输入
output_data	AES-CBC-192 加密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.27.4　返回值

无。

2.2.28　aes_cbc192_hard_decrypt_dma

2.2.28.1　描述

AES-CBC-192 解密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。CBC 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。

2.2.28.2　函数原型

```
void aes_cbc192_hard_decrypt_dma(dmac_channel_number_t dma_receive_channel_num,
    cbc_context_t *context, uint8_t *input_data, size_t input_len, uint8_t *
    output_data)
```

2.2.28.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
context	AES-CBC-192 解密计算的结构体，包含解密密钥与偏移向量	输入
input_data	AES-CBC-192 待解密的密文数据	输入
input_len	AES-CBC-192 待解密密文数据的长度	输入
output_data	AES-CBC-192 解密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.28.4　返回值

无。

2.2.29　aes_cbc256_hard_encrypt_dma

2.2.29.1　描述

AES-CBC-256 加密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。CBC 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。

2.2.29.2　函数原型

```
void aes_cbc256_hard_encrypt_dma(dmac_channel_number_t dma_receive_channel_num,
    cbc_context_t *context, uint8_t *input_data, size_t input_len, uint8_t *
    output_data)
```

2.2.29.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
context	AES-CBC-256 加密计算的结构体，包含加密密钥与偏移向量	输入
input_data	AES-CBC-256 待加密的明文数据	输入
input_len	AES-CBC-256 待加密明文数据的长度	输入
output_data	AES-CBC-256 加密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.29.4　返回值

无。

2.2.30　aes_cbc256_hard_decrypt_dma

2.2.30.1　描述

AES-CBC-256 解密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。CBC 加密将明文按照固定大小 16bytes 的块进行加密的，块大小不足则进行填充。

2.2.30.2　函数原型

```
void aes_cbc256_hard_decrypt_dma(dmac_channel_number_t dma_receive_channel_num, uint8_t
     *input_key, uint8_t *input_data, size_t input_len, uint8_t *output_data)
```

2.2.30.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
context	AES-CBC-256 解密计算的结构体，包含解密密钥与偏移向量	输入
input_data	AES-CBC-256 待解密的密文数据	输入
input_len	AES-CBC-256 待解密密文数据的长度	输入
output_data	AES-CBC-256 解密运算后的结果存放在这个 buffer。这个 buffer 的大小需要保证 16bytes 对齐。	输出

2.2.30.4　返回值

无。

2.2.31　aes_gcm128_hard_encrypt_dma

2.2.31.1　描述

AES-GCM-128 加密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。

2.2.31.2　函数原型

```
void aes_gcm128_hard_encrypt_dma(dmac_channel_number_t dma_receive_channel_num,
    gcm_context_t *context, uint8_t *input_data, size_t input_len, uint8_t *
    output_data, uint8_t *gcm_tag)
```

2.2.31.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入

参数名称	描述	输入输出
context	AES-GCM-128 加密计算的结构体，包含加密密钥/偏移向量/aad/aad 长度	输入
input_data	AES-GCM-128 待加密的明文数据	输入
input_len	AES-GCM-128 待加密明文数据的长度。	输入
output_data	AES-GCM-128 加密运算后的结果存放在这个 buffer。。由于 DMA 搬运数据的最小粒度为 4bytes，所以需要保证这个 buffer 大小至少为 4bytes 的整数倍。	输出
gcm_tag	AES-GCM-128 加密运算后的 tag 存放在这个 buffer。这个 buffer 大小需要保证为 16bytes	输出

2.2.31.4　返回值

无。

2.2.32　aes_gcm128_hard_decrypt_dma

2.2.32.1　描述

AES-GCM-128 解密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。

2.2.32.2　函数原型

```
void aes_gcm128_hard_decrypt_dma(dmac_channel_number_t dma_receive_channel_num,
    gcm_context_t *context, uint8_t *input_data, size_t input_len, uint8_t *
    output_data, uint8_t *gcm_tag)
```

2.2.32.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
context	AES-GCM-128 解密计算的结构体，包含解密密钥/偏移向量/aad/aad 长度	输入
input_data	AES-GCM-128 待解密的密文数据	输入
input_len	AES-GCM-128 待解密密文数据的长度。	输入
output_data	AES-GCM-128 解密运算后的结果存放在这个 buffer。由于 DMA 搬运数据的最小粒度为 4bytes，所以需要保证这个 buffer 大小至少为 4bytes 的整数倍。	输出
gcm_tag	AES-GCM-128 解密运算后的 tag 存放在这个 buffer。这个 buffer 大小需要保证为 16bytes	输出

2.2.32.4　返回值

无。

2.2.33　aes_gcm192_hard_encrypt_dma

2.2.33.1　描述

AES-GCM-192 加密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。

2.2.33.2　函数原型

```
void aes_gcm192_hard_encrypt_dma(dmac_channel_number_t dma_receive_channel_num,
    gcm_context_t *context, uint8_t *input_data, size_t input_len, uint8_t *
    output_data, uint8_t *gcm_tag)
```

2.2.33.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
context	AES-GCM-192 加密计算的结构体，包含加密密钥/偏移向量/aad/aad 长度	输入
input_data	AES-GCM-192 待加密的明文数据	输入
input_len	AES-GCM-192 待加密明文数据的长度。	输入
output_data	AES-GCM-192 加密运算后的结果存放在这个 buffer。由于 DMA 搬运数据的最小粒度为 4bytes，所以需要保证这个 buffer 大小至少为 4bytes 的整数倍。	输出
gcm_tag	AES-GCM-192 加密运算后的 tag 存放在这个 buffer。这个 buffer 大小需要保证为 16bytes	输出

2.2.33.4　返回值

无。

2.2.34　aes_gcm192_hard_decrypt_dma

2.2.34.1　描述

AES-GCM-192 解密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。

2.2.34.2　函数原型

```
void aes_gcm192_hard_decrypt_dma(dmac_channel_number_t dma_receive_channel_num,
    gcm_context_t *context, uint8_t *input_data, size_t input_len, uint8_t *
    output_data, uint8_t *gcm_tag)
```

2.2.34.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
context	AES-GCM-192 解密计算的结构体，包含解密密钥/偏移向量/aad/aad 长度	输入
input_data	AES-GCM-192 待解密的密文数据	输入
input_len	AES-GCM-192 待解密密文数据的长度。	输入
output_data	AES-GCM-192 解密运算后的结果存放在这个 buffer。由于 DMA 搬运数据的最小粒度为 4bytes，所以需要保证这个 buffer 大小至少为 4bytes 的整数倍。	输出
gcm_tag	AES-GCM-192 解密运算后的 tag 存放在这个 buffer。这个 buffer 大小需要保证为 16bytes	输出

2.2.34.4　返回值

无。

2.2.35　aes_gcm256_hard_encrypt_dma

2.2.35.1　描述

AES-GCM-256 加密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。

2.2.35.2　函数原型

```
void aes_gcm256_hard_encrypt_dma(dmac_channel_number_t dma_receive_channel_num,
    gcm_context_t *context, uint8_t *input_data, size_t input_len, uint8_t *
    output_data, uint8_t *gcm_tag)
```

2.2.35.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
context	AES-GCM-256 加密计算的结构体，包含加密密钥/偏移向量/aad/aad 长度	输入
input_data	AES-GCM-256 待加密的明文数据	输入
input_len	AES-GCM-256 待加密明文数据的长度。	输入
output_data	AES-GCM-256 加密运算后的结果存放在这个 buffer。由于 DMA 搬运数据的最小粒度为 4bytes，所以需要保证这个 buffer 大小至少为 4bytes 的整数倍。	输出
gcm_tag	AES-GCM-256 加密运算后的 tag 存放在这个 buffer。这个 buffer 大小需要保证为 16bytes	输出

2.2.35.4　返回值

无。

2.2.36　aes_gcm256_hard_decrypt_dma

2.2.36.1　描述

AES-GCM-256 解密运算。输入数据使用 cpu 传输，输出数据都使用 dma 传输。

2.2.36.2　函数原型

```
void aes_gcm256_hard_decrypt_dma(dmac_channel_number_t dma_receive_channel_num,
    gcm_context_t *context, uint8_t *input_data, size_t input_len, uint8_t *
    output_data, uint8_t *gcm_tag)
```

2.2.36.3　参数

参数名称	描述	输入输出
dma_receive_channel_num	AES 输出数据的 DMA 通道号	输入
context	AES-GCM-256 解密计算的结构体，包含解密密钥/偏移向量/aad/aad 长度	输入
input_data	AES-GCM-256 待解密的密文数据	输入
input_len	AES-GCM-256 待解密密文数据的长度。	输入
output_data	AES-GCM-256 解密运算后的结果存放在这个 buffer。由于 DMA 搬运数据的最小粒度为 4bytes，所以需要保证这个 buffer 大小至少为 4bytes 的整数倍。	输出
gcm_tag	AES-GCM-256 解密运算后的 tag 存放在这个 buffer。这个 buffer 大小需要保证为 16bytes	输出

2.2.36.4　返回值

无。

2.2.37　aes_init

2.2.37.1　描述

AES 硬件模块的初始化

2.2.37.2　函数原型

```
void aes_init(uint8_t *input_key, size_t input_key_len, uint8_t *iv,size_t iv_len,
    uint8_t *gcm_aad,aes_cipher_mode_t cipher_mode, aes_encrypt_sel_t encrypt_sel,
    size_t gcm_aad_len, size_t input_data_len)
```

2.2.37.3　参数

参数名称	描述	输入输出
input_key	待加密/解密的密钥	输入
input_key_len	待加密/解密密钥的长度	输入
iv	AES 加密解密用到的 iv 数据	输入
iv_len	AES 加密解密用到的 iv 数据的长度，CBC 固定为 16bytes，GCM 固定为 12bytes	输出
gcm_aad	AES-GCM 加密解密用到的 aad 数据	输出
cipher_mode	AES 硬件模块执行的加密解密类型，支持 AES_CBC/AES_ECB/AES_GCM	输入
encrypt_sel	AES 硬件模块执行的模式：加密或解密	输入
gcm_aad_len	AES-GCM 加密解密用到的 aad 数据的长度	输入
input_data_len	待加密/解密的数据长度	输入

2.2.37.4　返回值

无。

2.2.38　aes_process

2.2.38.1　描述

AES 硬件模块执行加密解密操作

2.2.38.2　函数原型

```
void aes_process(uint8_t *input_data, uint8_t *output_data, size_t input_data_len,
    aes_cipher_mode_t cipher_mode)
```

2.2.38.3　参数

参数名称	描述	输入输出
input_data	这个 buffer 存放待加密/解密的数据	输入
output_data	这个 buffer 存放加密/解密的输出结果	输出
input_data_len	待加密/解密的数据的长度	输入
cipher_mode	AES 硬件模块执行的加密解密类型，支持 AES_CBC/AES_ECB/AES_GCM	输入

2.2.38.4　返回值

无。

2.2.39　gcm_get_tag

2.2.39.1　描述

获取 AES-GCM 计算结束后的 tag

2.2.39.2　函数原型

```
void gcm_get_tag(uint8_t *gcm_tag)
```

2.2.39.3　参数

参数名称	描述	输入输出
gcm_tag	这个 buffer 存放 AES-GCM 加密/解密后的 tag，固定为 16bytes 的大小	输入出

2.2.39.4　返回值

无。

2.2.40　举例

```
cbc_context_t cbc_context;
cbc_context.input_key = cbc_key;
cbc_context.iv = cbc_iv;
aes_cbc128_hard_encrypt(&cbc_context, aes_input_data, 16L, aes_output_data);
memcpy(aes_input_data, aes_output_data, 16L);
aes_cbc128_hard_decrypt(&cbc_context, aes_input_data, 16L, aes_output_data);
```

2.3　数据类型

相关数据类型、数据结构定义如下：

- aes_cipher_mode_t：AES 加密/解密的方式。

2.3.1　aes_cipher_mode_t

2.3.1.1　描述

AES 加密/解密的方式。

2.3.1.2　定义

```
typedef enum _aes_cipher_mode
{
    AES_ECB = 0,
    AES_CBC = 1,
    AES_GCM = 2,
    AES_CIPHER_MAX
} aes_cipher_mode_t;
```

- gcm_context_t：AES-GCM 加密/解密时参数用到的结构体

2.3.2　gcm_context_t

2.3.2.1　描述

AES-GCM 参数用到的结构体，包括密钥、偏移向量、aad 数据、aad 数据长度。

2.3.2.2　定义

```
typedef struct _gcm_context
{
    uint8_t *input_key;
    uint8_t *iv;
    uint8_t *gcm_aad;
    size_t gcm_aad_len;
} gcm_context_t;
```

- cbc_context_t：AES-CBC 加密/解密时参数用到的结构体

2.3.3　cbc_context_t

2.3.3.1　描述

AES-CBC 参数用到的结构体，包括密钥、偏移向量。

2.3.3.2　定义

```
typedef struct _cbc_context
{
    uint8_t *input_key;
    uint8_t *iv;
```

```
} cbc_context_t;
```

2.3.3.3　成员

成员名称	描述
AES_ECB	ECB 加密/解密
AES_CBC	CBC 加密/解密
AES_GCM	GCM 加密/解密

第 3 章

中断 PLIC

3.1 概述

可以将任一外部中断源单独分配到每个 CPU 的外部中断上。这提供了强大的灵活性，能适应不同的应用需求。

3.2 功能描述

PLIC 模块具有以下功能：

- 启用或禁用中断
- 设置中断处理程序
- 配置中断优先级

3.3 API 参考

对应头文件 plic.h
为用户提供以下接口

- plic_init
- plic_irq_enable
- plic_irq_disable
- plic_set_priority
- plic_get_priority
- plic_irq_register
- plic_irq_deregister

3.3.1　plic_init

3.3.1.1　描述

PLIC 初始化外部中断。

3.3.1.2　函数原型

```
void plic_init(void)
```

3.3.1.3　参数

无。

3.3.1.4　返回值

无。

3.3.2　plic_irq_enable

3.3.2.1　描述

使能外部中断。

3.3.2.2　函数原型

```
int plic_irq_enable(plic_irq_t irq_number)
```

3.3.2.3　参数

参数名称	描述	输入输出
irq_number	中断号	输入

3.3.2.4　返回值

返回值	描述
0	成功
非 0	失败

3.3.3　plic_irq_disable

3.3.3.1　描述

禁用外部中断。

3.3.3.2　函数原型

```
int plic_irq_disable(plic_irq_t irq_number)
```

3.3.3.3　参数

参数名称	描述	输入输出
irq_number	中断号	输入

3.3.3.4　返回值

返回值	描述
0	成功
非 0	失败

3.3.4　plic_set_priority

3.3.4.1　描述

设置中断优先级。

3.3.4.2　函数原型

```
int plic_set_priority(plic_irq_t irq_number, uint32_t priority)
```

3.3.4.3　参数

参数名称	描述	输入输出
irq_number	中断号	输入
priority	中断优先级	输入

3.3.4.4　返回值

返回值	描述
0	成功
非 0	失败

3.3.5　plic_get_priority

3.3.5.1　描述

获取中断优先级。

3.3.5.2　函数原型

```
uint32_t plic_get_priority(plic_irq_t irq_number)
```

3.3.5.3　参数

参数名称	描述	输入输出
irq_number	中断号	输入

3.3.5.4　返回值

irq_number 中断的优先级。

3.3.6　plic_irq_register

3.3.6.1　描述

注册外部中断函数。

3.3.6.2　函数原型

```
int plic_irq_register(plic_irq_t irq, plic_irq_callback_t callback, void* ctx)
```

3.3.6.3　参数

参数名称	描述	输入输出
irq	中断号	输入

参数名称	描述	输入输出
callback	中断回调函数	输入
ctx	回调函数的参数	输入

3.3.6.4　返回值

返回值	描述
0	成功
非 0	失败

3.3.7　plic_irq_deregister

3.3.7.1　描述

注销外部中断函数。

3.3.7.2　函数原型

```
int plic_irq_deregister(plic_irq_t irq)
```

3.3.7.3　参数

参数名称	描述	输入输出
irq	中断号	输入

3.3.7.4　返回值

返回值	描述
0	成功
非 0	失败

3.3.8　举例

```
/* 设置GPIOHS0的触发中断 */
int count = 0;
int gpiohs_pin_onchange_isr(void *ctx)
```

```
{
    int *userdata = (int *)ctx;
    *userdata++;
}
plic_init();
plic_set_priority(IRQN_GPIOHS0_INTERRUPT, 1);
plic_irq_register(IRQN_GPIOHS0_INTERRUPT, gpiohs_pin_onchange_isr, &count);
plic_irq_enable(IRQN_GPIOHS0_INTERRUPT);
sysctl_enable_irq();
```

3.4　数据类型

相关数据类型、数据结构定义如下：

- plic_irq_t：外部中断号。
- plic_irq_callback_t：外部中断回调函数。

3.4.1　plic_irq_t

3.4.1.1　描述

外部中断号。

3.4.1.2　定义

```
typedef enum _plic_irq
{
    IRQN_NO_INTERRUPT        = 0, /*!< The non-existent interrupt */
    IRQN_SPI0_INTERRUPT      = 1, /*!< SPI0 interrupt */
    IRQN_SPI1_INTERRUPT      = 2, /*!< SPI1 interrupt */
    IRQN_SPI_SLAVE_INTERRUPT = 3, /*!< SPI_SLAVE interrupt */
    IRQN_SPI3_INTERRUPT      = 4, /*!< SPI3 interrupt */
    IRQN_I2S0_INTERRUPT      = 5, /*!< I2S0 interrupt */
    IRQN_I2S1_INTERRUPT      = 6, /*!< I2S1 interrupt */
    IRQN_I2S2_INTERRUPT      = 7, /*!< I2S2 interrupt */
    IRQN_I2C0_INTERRUPT      = 8, /*!< I2C0 interrupt */
    IRQN_I2C1_INTERRUPT      = 9, /*!< I2C1 interrupt */
    IRQN_I2C2_INTERRUPT      = 10, /*!< I2C2 interrupt */
    IRQN_UART1_INTERRUPT     = 11, /*!< UART1 interrupt */
    IRQN_UART2_INTERRUPT     = 12, /*!< UART2 interrupt */
    IRQN_UART3_INTERRUPT     = 13, /*!< UART3 interrupt */
    IRQN_TIMER0A_INTERRUPT   = 14, /*!< TIMER0 channel 0 or 1 interrupt */
    IRQN_TIMER0B_INTERRUPT   = 15, /*!< TIMER0 channel 2 or 3 interrupt */
    IRQN_TIMER1A_INTERRUPT   = 16, /*!< TIMER1 channel 0 or 1 interrupt */
    IRQN_TIMER1B_INTERRUPT   = 17, /*!< TIMER1 channel 2 or 3 interrupt */
    IRQN_TIMER2A_INTERRUPT   = 18, /*!< TIMER2 channel 0 or 1 interrupt */
    IRQN_TIMER2B_INTERRUPT   = 19, /*!< TIMER2 channel 2 or 3 interrupt */
```

```
    IRQN_RTC_INTERRUPT       = 20, /*!< RTC tick and alarm interrupt */
    IRQN_WDT0_INTERRUPT      = 21, /*!< Watching dog timer0 interrupt */
    IRQN_WDT1_INTERRUPT      = 22, /*!< Watching dog timer1 interrupt */
    IRQN_APB_GPIO_INTERRUPT  = 23, /*!< APB GPIO interrupt */
    IRQN_DVP_INTERRUPT       = 24, /*!< Digital video port interrupt */
    IRQN_AI_INTERRUPT        = 25, /*!< AI accelerator interrupt */
    IRQN_FFT_INTERRUPT       = 26, /*!< FFT accelerator interrupt */
    IRQN_DMA0_INTERRUPT      = 27, /*!< DMA channel0 interrupt */
    IRQN_DMA1_INTERRUPT      = 28, /*!< DMA channel1 interrupt */
    IRQN_DMA2_INTERRUPT      = 29, /*!< DMA channel2 interrupt */
    IRQN_DMA3_INTERRUPT      = 30, /*!< DMA channel3 interrupt */
    IRQN_DMA4_INTERRUPT      = 31, /*!< DMA channel4 interrupt */
    IRQN_DMA5_INTERRUPT      = 32, /*!< DMA channel5 interrupt */
    IRQN_UARTHS_INTERRUPT    = 33, /*!< Hi-speed UART0 interrupt */
    IRQN_GPIOHS0_INTERRUPT   = 34, /*!< Hi-speed GPIO0 interrupt */
    IRQN_GPIOHS1_INTERRUPT   = 35, /*!< Hi-speed GPIO1 interrupt */
    IRQN_GPIOHS2_INTERRUPT   = 36, /*!< Hi-speed GPIO2 interrupt */
    IRQN_GPIOHS3_INTERRUPT   = 37, /*!< Hi-speed GPIO3 interrupt */
    IRQN_GPIOHS4_INTERRUPT   = 38, /*!< Hi-speed GPIO4 interrupt */
    IRQN_GPIOHS5_INTERRUPT   = 39, /*!< Hi-speed GPIO5 interrupt */
    IRQN_GPIOHS6_INTERRUPT   = 40, /*!< Hi-speed GPIO6 interrupt */
    IRQN_GPIOHS7_INTERRUPT   = 41, /*!< Hi-speed GPIO7 interrupt */
    IRQN_GPIOHS8_INTERRUPT   = 42, /*!< Hi-speed GPIO8 interrupt */
    IRQN_GPIOHS9_INTERRUPT   = 43, /*!< Hi-speed GPIO9 interrupt */
    IRQN_GPIOHS10_INTERRUPT  = 44, /*!< Hi-speed GPIO10 interrupt */
    IRQN_GPIOHS11_INTERRUPT  = 45, /*!< Hi-speed GPIO11 interrupt */
    IRQN_GPIOHS12_INTERRUPT  = 46, /*!< Hi-speed GPIO12 interrupt */
    IRQN_GPIOHS13_INTERRUPT  = 47, /*!< Hi-speed GPIO13 interrupt */
    IRQN_GPIOHS14_INTERRUPT  = 48, /*!< Hi-speed GPIO14 interrupt */
    IRQN_GPIOHS15_INTERRUPT  = 49, /*!< Hi-speed GPIO15 interrupt */
    IRQN_GPIOHS16_INTERRUPT  = 50, /*!< Hi-speed GPIO16 interrupt */
    IRQN_GPIOHS17_INTERRUPT  = 51, /*!< Hi-speed GPIO17 interrupt */
    IRQN_GPIOHS18_INTERRUPT  = 52, /*!< Hi-speed GPIO18 interrupt */
    IRQN_GPIOHS19_INTERRUPT  = 53, /*!< Hi-speed GPIO19 interrupt */
    IRQN_GPIOHS20_INTERRUPT  = 54, /*!< Hi-speed GPIO20 interrupt */
    IRQN_GPIOHS21_INTERRUPT  = 55, /*!< Hi-speed GPIO21 interrupt */
    IRQN_GPIOHS22_INTERRUPT  = 56, /*!< Hi-speed GPIO22 interrupt */
    IRQN_GPIOHS23_INTERRUPT  = 57, /*!< Hi-speed GPIO23 interrupt */
    IRQN_GPIOHS24_INTERRUPT  = 58, /*!< Hi-speed GPIO24 interrupt */
    IRQN_GPIOHS25_INTERRUPT  = 59, /*!< Hi-speed GPIO25 interrupt */
    IRQN_GPIOHS26_INTERRUPT  = 60, /*!< Hi-speed GPIO26 interrupt */
    IRQN_GPIOHS27_INTERRUPT  = 61, /*!< Hi-speed GPIO27 interrupt */
    IRQN_GPIOHS28_INTERRUPT  = 62, /*!< Hi-speed GPIO28 interrupt */
    IRQN_GPIOHS29_INTERRUPT  = 63, /*!< Hi-speed GPIO29 interrupt */
    IRQN_GPIOHS30_INTERRUPT  = 64, /*!< Hi-speed GPIO30 interrupt */
    IRQN_GPIOHS31_INTERRUPT  = 65, /*!< Hi-speed GPIO31 interrupt */
    IRQN_MAX
} plic_irq_t;
```

3.4.1.3　成员

成员名称	描述
IRQN_NO_INTERRUPT	不存在
IRQN_SPI0_INTERRUPT	SPI0 中断
IRQN_SPI1_INTERRUPT	SPI1 中断
IRQN_SPI_SLAVE_INTERRUPT	从 SPI 中断
IRQN_SPI3_INTERRUPT	SPI3 中断
IRQN_I2S0_INTERRUPT	I2S0 中断
IRQN_I2S1_INTERRUPT	I2S1 中断
IRQN_I2S2_INTERRUPT	I2S2 中断
IRQN_I2C0_INTERRUPT	I2C0 中断
IRQN_I2C1_INTERRUPT	I2C1 中断
IRQN_I2C2_INTERRUPT	I2C2 中断
IRQN_UART1_INTERRUPT	UART1 中断
IRQN_UART2_INTERRUPT	UART2 中断
IRQN_UART3_INTERRUPT	UART3 中断
IRQN_TIMER0A_INTERRUPT	TIMER0 通道 0 和 1 中断
IRQN_TIMER0B_INTERRUPT	TIMER0 通道 2 和 3 中断
IRQN_TIMER1A_INTERRUPT	TIMER1 通道 0 和 1 中断
IRQN_TIMER1B_INTERRUPT	TIMER1 通道 2 和 3 中断
IRQN_TIMER2A_INTERRUPT	TIMER2 通道 0 和 1 中断
IRQN_TIMER2B_INTERRUPT	TIMER2 通道 2 和 3 中断
IRQN_RTC_INTERRUPT	RTC 滴答中断和报警中断
IRQN_WDT0_INTERRUPT	看门狗 0 中断
IRQN_WDT1_INTERRUPT	看门狗 1 中断
IRQN_APB_GPIO_INTERRUPT	普通 GPIO 中断
IRQN_DVP_INTERRUPT	数字摄像头（DVP）中断
IRQN_AI_INTERRUPT	AI 加速器中断
IRQN_FFT_INTERRUPTFFT	傅里叶加速器中断
IRQN_DMA0_INTERRUPT	DMA 通道 0 中断
IRQN_DMA1_INTERRUPT	DMA 通道 1 中断
IRQN_DMA2_INTERRUPT	DMA 通道 2 中断
IRQN_DMA3_INTERRUPT	DMA 通道 3 中断
IRQN_DMA4_INTERRUPT	DMA 通道 4 中断
IRQN_DMA5_INTERRUPT	DMA 通道 5 中断
IRQN_UARTHS_INTERRUPT	高速 UART 中断
IRQN_GPIOHS0_INTERRUPT	高速 GPIO0 中断

成员名称	描述
IRQN_GPIOHS1_INTERRUPT	高速 GPIO1 中断
IRQN_GPIOHS2_INTERRUPT	高速 GPIO2 中断
IRQN_GPIOHS3_INTERRUPT	高速 GPIO3 中断
IRQN_GPIOHS4_INTERRUPT	高速 GPIO4 中断
IRQN_GPIOHS5_INTERRUPT	高速 GPIO5 中断
IRQN_GPIOHS6_INTERRUPT	高速 GPIO6 中断
IRQN_GPIOHS7_INTERRUPT	高速 GPIO7 中断
IRQN_GPIOHS8_INTERRUPT	高速 GPIO8 中断
IRQN_GPIOHS9_INTERRUPT	高速 GPIO9 中断
IRQN_GPIOHS10_INTERRUPT	高速 GPIO10 中断
IRQN_GPIOHS11_INTERRUPT	高速 GPIO11 中断
IRQN_GPIOHS12_INTERRUPT	高速 GPIO12 中断
IRQN_GPIOHS13_INTERRUPT	高速 GPIO13 中断
IRQN_GPIOHS14_INTERRUPT	高速 GPIO14 中断
IRQN_GPIOHS15_INTERRUPT	高速 GPIO15 中断
IRQN_GPIOHS16_INTERRUPT	高速 GPIO16 中断
IRQN_GPIOHS17_INTERRUPT	高速 GPIO17 中断
IRQN_GPIOHS18_INTERRUPT	高速 GPIO18 中断
IRQN_GPIOHS19_INTERRUPT	高速 GPIO19 中断
IRQN_GPIOHS20_INTERRUPT	高速 GPIO20 中断
IRQN_GPIOHS21_INTERRUPT	高速 GPIO21 中断
IRQN_GPIOHS22_INTERRUPT	高速 GPIO22 中断
IRQN_GPIOHS23_INTERRUPT	高速 GPIO23 中断
IRQN_GPIOHS24_INTERRUPT	高速 GPIO24 中断
IRQN_GPIOHS25_INTERRUPT	高速 GPIO25 中断
IRQN_GPIOHS26_INTERRUPT	高速 GPIO26 中断
IRQN_GPIOHS27_INTERRUPT	高速 GPIO27 中断
IRQN_GPIOHS28_INTERRUPT	高速 GPIO28 中断
IRQN_GPIOHS29_INTERRUPT	高速 GPIO29 中断
IRQN_GPIOHS30_INTERRUPT	高速 GPIO30 中断
IRQN_GPIOHS31_INTERRUPT	高速 GPIO31 中断

3.4.2 plic_irq_callback_t

3.4.2.1 描述

外部中断回调函数。

3.4.2.2　定义

```
typedef int (*plic_irq_callback_t)(void *ctx);
```

第 4 章

通用输入/输出（GPIO）

4.1 概述

芯片有 8 个通用 GPIO。

4.2 功能描述

GPIO 模块具有以下功能：

- 可配置上下拉驱动模式

4.3 API 参考

对应的头文件 `gpio.h`
为用户提供以下接口

- `gpio_init`
- `gpio_set_drive_mode`
- `gpio_set_pin`
- `gpio_get_pin`

4.3.1 gpio_init

4.3.1.1 描述

初始化 GPIO。

4.3.1.2　函数原型

```
int gpio_init(void)
```

4.3.1.3　返回值

返回值	描述
0	成功
非 0	失败

4.3.2　gpio_set_drive_mode

4.3.2.1　描述

设置 GPIO 驱动模式。

4.3.2.2　函数原型

```
void gpio_set_drive_mode(uint8_t pin, gpio_drive_mode_t mode)
```

4.3.2.3　参数

参数名称	描述	输入输出
pin	GPIO 管脚	输入
mode	GPIO 驱动模式	输入

4.3.2.4　返回值

无。

4.3.3　gpio_set_pin

4.3.3.1　描述

设置 GPIO 管脚值。

4.3.3.2　函数原型

```
void gpio_set_pin(uint8_t pin, gpio_pin_value_t value)
```

4.3.3.3　参数

参数名称	描述	输入输出
pin	GPIO 管脚	输入
value	GPIO 值	输入

4.3.3.4　返回值

无。

4.3.4　gpio_get_pin

4.3.4.1　描述

获取 GPIO 管脚值。

4.3.4.2　函数原型

```
gpio_pin_value_t gpio_get_pin(uint8_t pin)
```

4.3.4.3　参数

参数名称	描述	输入输出
pin	GPIO 管脚	输入

4.3.4.4　返回值

获取的 GPIO 管脚值。

4.3.5　举例

```
/* 设置IO13为输出并置为高 */
gpio_init();
fpioa_set_function(13, FUNC_GPIO3);
gpio_set_drive_mode(3, GPIO_DM_OUTPUT);
gpio_set_pin(3, GPIO_PV_High);
```

4.4　数据类型

相关数据类型、数据结构定义如下：

- gpio_drive_mode_t：GPIO 驱动模式。
- gpio_pin_value_t：GPIO 值。

4.4.1　gpio_drive_mode_t

4.4.1.1　描述

GPIO 驱动模式。

4.4.1.2　定义

```
typedef enum _gpio_drive_mode
{
    GPIO_DM_INPUT,
    GPIO_DM_INPUT_PULL_DOWN,
    GPIO_DM_INPUT_PULL_UP,
    GPIO_DM_OUTPUT,
} gpio_drive_mode_t;
```

4.4.1.3　成员

成员名称	描述
GPIO_DM_INPUT	输入
GPIO_DM_INPUT_PULL_DOWN	输入下拉
GPIO_DM_INPUT_PULL_UP	输入上拉
GPIO_DM_OUTPUT	输出

4.4.2　gpio_pin_value_t

4.4.2.1　描述

GPIO 值。

4.4.2.2　定义

```
typedef enum _gpio_pin_value
```

```
{
    GPIO_PV_LOW,
    GPIO_PV_HIGH
} gpio_pin_value_t;
```

4.4.2.3　成员

成员名称	描述
GPIO_PV_LOW	低
GPIO_PV_HIGH	高

第 5 章

通用高速输入/输出（GPIOHS）

5.1 概述

芯片有 32 个高速 GPIO。与普通 GPIO 相似，管脚反转能力更强。

5.2 功能描述

GPIOHS 模块具有以下功能：

- 可配置上下拉驱动模式
- 支持上升沿、下降沿和双沿触发

5.3 API 参考

对应的头文件 `gpiohs.h`
为用户提供以下接口

- `gpiohs_set_drive_mode`
- `gpiohs_set_pin`
- `gpiohs_get_pin`
- `gpiohs_set_pin_edge`
- `gpiohs_set_irq`（0.6.0 后不再支持，请使用 `gpiohs_irq_register`）
- `gpiohs_irq_register`
- `gpiohs_irq_unregister`

5.3.1　gpiohs_set_drive_mode

5.3.1.1　描述

设置 GPIO 驱动模式。

5.3.1.2　函数原型

```
void gpiohs_set_drive_mode(uint8_t pin, gpio_drive_mode_t mode)
```

5.3.1.3　参数

参数名称	描述	输入输出
pin	GPIO 管脚	输入
mode	GPIO 驱动模式	输入

5.3.1.4　返回值

无。

5.3.2　gpio_set_pin

5.3.2.1　描述

设置 GPIO 管脚值。

5.3.2.2　函数原型

```
void gpiohs_set_pin(uint8_t pin, gpio_pin_value_t value)
```

5.3.2.3　参数

参数名称	描述	输入输出
pin	GPIO 管脚	输入
value	GPIO 值	输入

5.3.2.4　返回值

无。

5.3.3　gpio_get_pin

5.3.3.1　描述

获取 GPIO 管脚值。

5.3.3.2　函数原型

```
gpio_pin_value_t gpiohs_get_pin(uint8_t pin)
```

5.3.3.3　参数

参数名称	描述	输入输出
pin	GPIO 管脚	输入

5.3.3.4　返回值

获取的 GPIO 管脚值。

5.3.4　gpiohs_set_pin_edge

5.3.4.1　描述

设置高速 GPIO 中断触发模式。

5.3.4.2　函数原型

```
void gpiohs_set_pin_edge(uint8_t pin, gpio_pin_edge_t edge)
```

5.3.4.3　参数

参数名称	描述	输入输出
pin	GPIO 管脚	输入
edge	中断触发方式	输入

5.3.4.4　返回值

无。

5.3.5　gpiohs_set_irq

5.3.5.1　描述

设置高速 GPIO 的中断回调函数。

5.3.5.2　函数原型

```
void gpiohs_set_irq(uint8_t pin, uint32_t priority, void(*func)());
```

5.3.5.3　参数

参数名称	描述	输入输出
pin	GPIO 管脚	输入
priority	中断优先级	输入
func	中断回调函数	输入

5.3.5.4　返回值

无。

5.3.6　gpiohs_irq_register

5.3.6.1　描述

设置高速 GPIO 的中断回调函数。

5.3.6.2　函数原型

```
void gpiohs_irq_register(uint8_t pin, uint32_t priority, plic_irq_callback_t callback,
    void *ctx)
```

5.3.6.3　参数

参数名称	描述	输入输出
pin	GPIO 管脚	输入
priority	中断优先级	输入
plic_irq_callback_t	中断回调函数	输入

参数名称	描述	输入输出
ctx	回调函数参数	输入

5.3.6.4　返回值

无。

5.3.7　gpiohs_irq_unregister

5.3.7.1　描述

注销 GPIOHS 中断。

5.3.8　函数原型

```
void gpiohs_irq_unregister(uint8_t pin)
```

5.3.8.1　参数

参数名称	描述	输入输出
pin	GPIO 管脚	输入

5.3.8.2　返回值

无。

5.3.9　举例

```
void irq_gpiohs2(void *ctx)
{
    printf("Hello world\n");
}
/* 设置IO13为高速GPIO，输出模式并置为高 */
fpioa_set_function(13, FUNC_GPIOHS3);
gpiohs_set_drive_mode(3, GPIO_DM_OUTPUT);
gpiohs_set_pin(3, GPIO_PV_High);
/* 设置IO14为高速GPIO 输入模式 双沿触发中断时打印Hello world */
plic_init();
fpioa_set_function(14, FUNC_GPIOHS2);
gpiohs_set_drive_mode(2, GPIO_DM_INPUT);
gpiohs_set_pin_edge(2, GPIO_PE_BOTH);
```

```
gpiohs_irq_register(2, 1, irq_gpiohs2, NULL);
sysctl_enable_irq();
```

5.4　数据类型

相关数据类型、数据结构定义如下：

- gpio_drive_mode_t：GPIO 驱动模式。
- gpio_pin_value_t：GPIO 值。
- gpio_pin_edge_t：GPIO 边沿触发模式。

5.4.1　gpio_drive_mode_t

5.4.1.1　描述

GPIO 驱动模式。

5.4.1.2　定义

```
typedef enum _gpio_drive_mode
{
    GPIO_DM_INPUT,
    GPIO_DM_INPUT_PULL_DOWN,
    GPIO_DM_INPUT_PULL_UP,
    GPIO_DM_OUTPUT,
} gpio_drive_mode_t;
```

5.4.1.3　成员

成员名称	描述
GPIO_DM_INPUT	输入
GPIO_DM_INPUT_PULL_DOWN	输入下拉
GPIO_DM_INPUT_PULL_UP	输入上拉
GPIO_DM_OUTPUT	输出

5.4.2　gpio_pin_value_t

5.4.2.1　描述

GPIO 值。

5.4.2.2　定义

```
typedef enum _gpio_pin_value
{
    GPIO_PV_LOW,
    GPIO_PV_HIGH
} gpio_pin_value_t;
```

5.4.2.3　成员

成员名称	描述
GPIO_PV_LOW	低
GPIO_PV_HIGH	高

5.4.3　gpio_pin_edge_t

5.4.3.1　描述

高速 GPIO 边沿触发模式。

5.4.3.2　定义

```
typedef enum _gpio_pin_edge
{
    GPIO_PE_NONE,
    GPIO_PE_FALLING,
    GPIO_PE_RISING,
    GPIO_PE_BOTH,
    GPIO_PE_LOW,
    GPIO_PE_HIGH = 8,
} gpio_pin_edge_t;
```

5.4.3.3　成员

成员名称	描述
GPIO_PE_NONE	不触发
GPIO_PE_FALLING	下降沿触发
GPIO_PE_RISING	上升沿触发
GPIO_PE_BOTH	双沿触发

成员名称	描述
GPIO_PE_LOW	低电平触发
GPIO_PE_HIGH	高电平触发

第 6 章

现场可编程 IO 阵列 (FPIOA)

6.1 概述

FPIOA（现场可编程 IO 阵列）允许用户将 255 个内部功能映射到芯片外围的 48 个自由 IO 上。

6.2 功能描述

- 支持 IO 的可编程功能选择
- 支持 IO 输出的 8 种驱动能力选择
- 支持 IO 的内部上拉电阻选择
- 支持 IO 的内部下拉电阻选择
- 支持 IO 输入的内部施密特触发器设置
- 支持 IO 输出的斜率控制
- 支持内部输入逻辑的电平设置

6.3 API 参考

对应的头文件 `fpioa.h`
为用户提供以下接口

- `fpioa_set_function`
- `fpioa_get_io_by_function`
- `fpioa_set_io`
- `fpioa_get_io`
- `fpioa_set_tie_enable`
- `fpioa_set_tie_value`

- fpioa_set_io_pull
- fpioa_get_io_pull
- fpioa_set_io_driving
- fpioa_get_io_driving

6.3.1 fpioa_set_function

6.3.1.1 描述

设置 IO0-IO47 管脚复用功能。

6.3.1.2 函数原型

```
int fpioa_set_function(int number, fpioa_function_t function)
```

6.3.1.3 参数

参数名称	描述	输入输出
number	IO 管脚号	输入
function	管脚功能号	输入

6.3.1.4 返回值

返回值	描述
0	成功
非 0	失败

6.3.2 fpioa_get_io_by_function

6.3.2.1 描述

根据功能号获取 IO 管脚号。

6.3.2.2 函数原型

```
int fpioa_get_io_by_function(fpioa_function_t function)
```

6.3.2.3　参数

参数名称	描述	输入输出
function	功能管脚编号	输入

6.3.2.4　返回值

返回值	描述
大于等于 0	IO 管脚号
小于 0	失败

6.3.3　fpioa_get_io

6.3.3.1　描述

获得 IO 管脚的配置.

6.3.3.2　函数原型

```
int fpioa_get_io(int number, fpioa_io_config_t *cfg)
```

6.3.3.3　参数

参数名称	描述	输入输出
number	IO 管脚号	输入
cfg	管脚功能结构体	输出

6.3.3.4　返回值

返回值	描述
0	成功
非 0	失败

6.3.4　fpioa_set_io

6.3.4.1　描述

设置 IO 管脚的配置.

6.3.4.2　函数原型

```
int fpioa_set_io(int number, fpioa_io_config_t *cfg)
```

6.3.4.3　参数

参数名称	描述	输入输出
number	IO 管脚号	输入
cfg	管脚功能结构体	输入

6.3.4.4　返回值

返回值	描述
0	成功
非 0	失败

6.3.5　fpioa_set_tie_enable

6.3.5.1　描述

使能禁用功能管脚 TIE.

6.3.5.2　函数原型

```
int fpioa_set_tie_enable(fpioa_function_t function, int enable)
```

6.3.5.3　参数

参数名称	描述	输入输出
function	管脚功能号	输入
enable	TIE 使能位 0: 禁用 1: 使能	输入

6.3.5.4　返回值

返回值	描述
0	成功
非 0	失败

6.3.6　fpioa_set_tie_value

6.3.6.1　描述

设置功能管脚 TIE 上拉下拉。

6.3.6.2　函数原型

```
int fpioa_set_tie_value(fpioa_function_t function, int value)
```

6.3.6.3　参数

参数名称	描述	输入输出
function	管脚功能号	输入
value	TIE 值 0: 下拉 1: 上拉	输入

6.3.6.4　返回值

返回值	描述
0	成功
非 0	失败

6.3.7　fpioa_set_pull

6.3.7.1　描述

设置 IO 的上拉下拉。

6.3.7.2　函数原型

```
int fpioa_set_io_pull(int number, fpioa_pull_t pull)
```

6.3.7.3　参数

参数名称	描述	输入输出
number	IO 编号	输入
pull	上下拉值	输入

6.3.7.4　返回值

返回值	描述
0	成功
非 0	失败

6.3.8　fpioa_get_pull

6.3.8.1　描述

获取 IO 管脚上下拉值。

6.3.8.2　函数原型

```
int fpioa_get_io_pull(int number)
```

6.3.8.3　参数

参数名称	描述	输入输出
number	IO 编号	输入

6.3.8.4　返回值

返回值	描述
0	无上下拉
1	下拉
2	上拉

6.3.9　fpioa_set_io_driving

6.3.10　描述

设置 IO 管脚的驱动能力。

6.3.10.1　函数原型

```
int fpioa_set_io_driving(int number, fpioa_driving_t driving)
```

6.3.10.2　参数

参数名称	描述	输入输出
number	IO 编号	输入
driving	驱动能力	输入

6.3.10.3　返回值

返回值	描述
0	成功
非 0	失败

6.3.11　fpioa_get_io_driving

6.3.11.1　描述

获取驱动能力。

6.3.11.2　函数原型

```
int fpioa_get_io_driving(int number)
```

6.3.11.3　参数

参数名称	描述	输入输出
number	IO 编号	输入

6.3.11.4　返回值

返回值	描述
大于等于 0	驱动能力
小于 0	失败

6.4　数据类型

相关数据类型、数据结构定义如下：

- fpioa_function_t：管脚的功能编号。
- fpioa_io_config_t：FPIOA 的配置。
- fpioa_pull_t：FPIOA 功能管脚上拉下拉值。
- fpioa_driving_t：FPIOA 驱动能力编号。

6.4.1　fpioa_io_config_t

6.4.1.1　描述

FPIOA 的配置。

6.4.1.2　定义

```
typedef struct _fpioa_io_config
{
    uint32_t ch_sel : 8;
    uint32_t ds : 4;
    uint32_t oe_en : 1;
    uint32_t oe_inv : 1;
    uint32_t do_sel : 1;
    uint32_t do_inv : 1;
    uint32_t pu : 1;
    uint32_t pd : 1;
    uint32_t resv0 : 1;
    uint32_t sl : 1;
    uint32_t ie_en : 1;
    uint32_t ie_inv : 1;
    uint32_t di_inv : 1;
    uint32_t st : 1;
    uint32_t resv1 : 7;
    uint32_t pad_di : 1;
} __attribute__((packed, aligned(4))) fpioa_io_config_t;
```

6.4.1.3　成员

成员名称	描述
ch_sel	配置引脚功能序号
ds	选择驱动能力，参考驱动能力选择表
oe_en	1:输出使能 0:输出关闭
oe_inv	1:使能反转输出 0:禁止反转输出
do_sel	1:输出高电平 0:输出低电平
do_inv	反转输出电平
pu	1:上拉使能 0:上拉关闭
pd	1:下拉使能 0:下拉关闭
resv0	保留位
sl	电压转换速率控制使能
ie_en	1:输入使能 0:输入关闭
ie_inv	1:使能反转输入 0:禁止反转输入
di_inv	反转输入数据
st	施密特触发
resv1	保留位
pad_di	读取当前引脚的输入电平

6.4.2　驱动能力选择表

6.4.2.1　低电平输入电流

ds	Min(mA)	Typ(mA)	Max(mA)
0	3.2	5.4	8.3
1	4.7	8.0	12.3
2	6.3	10.7	16.4
3	7.8	13.2	20.2
4	9.4	15.9	24.2
5	10.9	18.4	28.1
6	12.4	20.9	31.8
7	13.9	23.4	35.5

6.4.2.2　高电平输出电流

ds	Min(mA)	Typ(mA)	Max(mA)
0	5.0	7.6	11.2
1	7.5	11.4	16.8
2	10.0	15.2	22.3
3	12.4	18.9	27.8
4	14.9	22.6	33.3
5	17.4	26.3	38.7
6	19.8	30.0	44.1
7	22.3	33.7	49.5

6.4.3 fpioa_pull_t

6.4.3.1 描述

FPIOA 功能管脚上拉下拉值。

6.4.3.2 定义

```
typedef enum _fpioa_pull
{
    FPIOA_PULL_NONE,
    FPIOA_PULL_DOWN,
    FPIOA_PULL_UP,
    FPIOA_PULL_MAX
} fpioa_pull_t;
```

6.4.3.3 成员

成员名称	描述
FPIOA_PULL_NONE	无上下拉
FPIOA_PULL_DOWN	下拉
FPIOA_PULL_UP	上拉

6.4.4 fpioa_driving_t

6.4.4.1 描述

FPIOA 驱动能力编号，参见驱动能力选择表。

6.4.4.2　定义

```
typedef enum _fpioa_driving
{
    FPIOA_DRIVING_0,
    FPIOA_DRIVING_1,
    FPIOA_DRIVING_2,
    FPIOA_DRIVING_3,
    FPIOA_DRIVING_4,
    FPIOA_DRIVING_5,
    FPIOA_DRIVING_6,
    FPIOA_DRIVING_7,
} fpioa_driving_t;
```

6.4.4.3　成员

成员名称	描述
FPIOA_DRIVING_0	驱动能力 0
FPIOA_DRIVING_1	驱动能力 1
FPIOA_DRIVING_2	驱动能力 2
FPIOA_DRIVING_3	驱动能力 3
FPIOA_DRIVING_4	驱动能力 4
FPIOA_DRIVING_5	驱动能力 5
FPIOA_DRIVING_6	驱动能力 6
FPIOA_DRIVING_7	驱动能力 7

6.4.5　fpioa_function_t

6.4.5.1　描述

管脚的功能编号。

6.4.5.2　定义

```
typedef enum _fpioa_function
{
    FUNC_JTAG_TCLK        = 0,  /*!< JTAG Test Clock */
    FUNC_JTAG_TDI         = 1,  /*!< JTAG Test Data In */
    FUNC_JTAG_TMS         = 2,  /*!< JTAG Test Mode Select */
    FUNC_JTAG_TDO         = 3,  /*!< JTAG Test Data Out */
    FUNC_SPI0_D0          = 4,  /*!< SPI0 Data 0 */
    FUNC_SPI0_D1          = 5,  /*!< SPI0 Data 1 */
```

```
    FUNC_SPI0_D2          = 6,  /*!< SPI0 Data 2 */
    FUNC_SPI0_D3          = 7,  /*!< SPI0 Data 3 */
    FUNC_SPI0_D4          = 8,  /*!< SPI0 Data 4 */
    FUNC_SPI0_D5          = 9,  /*!< SPI0 Data 5 */
    FUNC_SPI0_D6          = 10, /*!< SPI0 Data 6 */
    FUNC_SPI0_D7          = 11, /*!< SPI0 Data 7 */
    FUNC_SPI0_SS0         = 12, /*!< SPI0 Chip Select 0 */
    FUNC_SPI0_SS1         = 13, /*!< SPI0 Chip Select 1 */
    FUNC_SPI0_SS2         = 14, /*!< SPI0 Chip Select 2 */
    FUNC_SPI0_SS3         = 15, /*!< SPI0 Chip Select 3 */
    FUNC_SPI0_ARB         = 16, /*!< SPI0 Arbitration */
    FUNC_SPI0_SCLK        = 17, /*!< SPI0 Serial Clock */
    FUNC_UARTHS_RX        = 18, /*!< UART High speed I2S_RECEIVER */
    FUNC_UARTHS_TX        = 19, /*!< UART High speed I2S_TRANSMITTER */
    FUNC_RESV6            = 20, /*!< Reserved function */
    FUNC_RESV7            = 21, /*!< Reserved function */
    FUNC_CLK_SPI1         = 22, /*!< Clock SPI1 */
    FUNC_CLK_I2C1         = 23, /*!< Clock I2C1 */
    FUNC_GPIOHS0          = 24, /*!< GPIO High speed 0 */
    FUNC_GPIOHS1          = 25, /*!< GPIO High speed 1 */
    FUNC_GPIOHS2          = 26, /*!< GPIO High speed 2 */
    FUNC_GPIOHS3          = 27, /*!< GPIO High speed 3 */
    FUNC_GPIOHS4          = 28, /*!< GPIO High speed 4 */
    FUNC_GPIOHS5          = 29, /*!< GPIO High speed 5 */
    FUNC_GPIOHS6          = 30, /*!< GPIO High speed 6 */
    FUNC_GPIOHS7          = 31, /*!< GPIO High speed 7 */
    FUNC_GPIOHS8          = 32, /*!< GPIO High speed 8 */
    FUNC_GPIOHS9          = 33, /*!< GPIO High speed 9 */
    FUNC_GPIOHS10         = 34, /*!< GPIO High speed 10 */
    FUNC_GPIOHS11         = 35, /*!< GPIO High speed 11 */
    FUNC_GPIOHS12         = 36, /*!< GPIO High speed 12 */
    FUNC_GPIOHS13         = 37, /*!< GPIO High speed 13 */
    FUNC_GPIOHS14         = 38, /*!< GPIO High speed 14 */
    FUNC_GPIOHS15         = 39, /*!< GPIO High speed 15 */
    FUNC_GPIOHS16         = 40, /*!< GPIO High speed 16 */
    FUNC_GPIOHS17         = 41, /*!< GPIO High speed 17 */
    FUNC_GPIOHS18         = 42, /*!< GPIO High speed 18 */
    FUNC_GPIOHS19         = 43, /*!< GPIO High speed 19 */
    FUNC_GPIOHS20         = 44, /*!< GPIO High speed 20 */
    FUNC_GPIOHS21         = 45, /*!< GPIO High speed 21 */
    FUNC_GPIOHS22         = 46, /*!< GPIO High speed 22 */
    FUNC_GPIOHS23         = 47, /*!< GPIO High speed 23 */
    FUNC_GPIOHS24         = 48, /*!< GPIO High speed 24 */
    FUNC_GPIOHS25         = 49, /*!< GPIO High speed 25 */
    FUNC_GPIOHS26         = 50, /*!< GPIO High speed 26 */
    FUNC_GPIOHS27         = 51, /*!< GPIO High speed 27 */
    FUNC_GPIOHS28         = 52, /*!< GPIO High speed 28 */
    FUNC_GPIOHS29         = 53, /*!< GPIO High speed 29 */
    FUNC_GPIOHS30         = 54, /*!< GPIO High speed 30 */
    FUNC_GPIOHS31         = 55, /*!< GPIO High speed 31 */
    FUNC_GPIO0            = 56, /*!< GPIO pin 0 */
    FUNC_GPIO1            = 57, /*!< GPIO pin 1 */
    FUNC_GPIO2            = 58, /*!< GPIO pin 2 */
```

```
    FUNC_GPIO3              = 59, /*!< GPIO pin 3 */
    FUNC_GPIO4              = 60, /*!< GPIO pin 4 */
    FUNC_GPIO5              = 61, /*!< GPIO pin 5 */
    FUNC_GPIO6              = 62, /*!< GPIO pin 6 */
    FUNC_GPIO7              = 63, /*!< GPIO pin 7 */
    FUNC_UART1_RX           = 64, /*!< UART1 I2S_RECEIVER */
    FUNC_UART1_TX           = 65, /*!< UART1 I2S_TRANSMITTER */
    FUNC_UART2_RX           = 66, /*!< UART2 I2S_RECEIVER */
    FUNC_UART2_TX           = 67, /*!< UART2 I2S_TRANSMITTER */
    FUNC_UART3_RX           = 68, /*!< UART3 I2S_RECEIVER */
    FUNC_UART3_TX           = 69, /*!< UART3 I2S_TRANSMITTER */
    FUNC_SPI1_D0            = 70, /*!< SPI1 Data 0 */
    FUNC_SPI1_D1            = 71, /*!< SPI1 Data 1 */
    FUNC_SPI1_D2            = 72, /*!< SPI1 Data 2 */
    FUNC_SPI1_D3            = 73, /*!< SPI1 Data 3 */
    FUNC_SPI1_D4            = 74, /*!< SPI1 Data 4 */
    FUNC_SPI1_D5            = 75, /*!< SPI1 Data 5 */
    FUNC_SPI1_D6            = 76, /*!< SPI1 Data 6 */
    FUNC_SPI1_D7            = 77, /*!< SPI1 Data 7 */
    FUNC_SPI1_SS0           = 78, /*!< SPI1 Chip Select 0 */
    FUNC_SPI1_SS1           = 79, /*!< SPI1 Chip Select 1 */
    FUNC_SPI1_SS2           = 80, /*!< SPI1 Chip Select 2 */
    FUNC_SPI1_SS3           = 81, /*!< SPI1 Chip Select 3 */
    FUNC_SPI1_ARB           = 82, /*!< SPI1 Arbitration */
    FUNC_SPI1_SCLK          = 83, /*!< SPI1 Serial Clock */
    FUNC_SPI_SLAVE_D0       = 84, /*!< SPI Slave Data 0 */
    FUNC_SPI_SLAVE_SS       = 85, /*!< SPI Slave Select */
    FUNC_SPI_SLAVE_SCLK     = 86, /*!< SPI Slave Serial Clock */
    FUNC_I2S0_MCLK          = 87, /*!< I2S0 Master Clock */
    FUNC_I2S0_SCLK          = 88, /*!< I2S0 Serial Clock(BCLK) */
    FUNC_I2S0_WS            = 89, /*!< I2S0 Word Select(LRCLK) */
    FUNC_I2S0_IN_D0         = 90, /*!< I2S0 Serial Data Input 0 */
    FUNC_I2S0_IN_D1         = 91, /*!< I2S0 Serial Data Input 1 */
    FUNC_I2S0_IN_D2         = 92, /*!< I2S0 Serial Data Input 2 */
    FUNC_I2S0_IN_D3         = 93, /*!< I2S0 Serial Data Input 3 */
    FUNC_I2S0_OUT_D0        = 94, /*!< I2S0 Serial Data Output 0 */
    FUNC_I2S0_OUT_D1        = 95, /*!< I2S0 Serial Data Output 1 */
    FUNC_I2S0_OUT_D2        = 96, /*!< I2S0 Serial Data Output 2 */
    FUNC_I2S0_OUT_D3        = 97, /*!< I2S0 Serial Data Output 3 */
    FUNC_I2S1_MCLK          = 98, /*!< I2S1 Master Clock */
    FUNC_I2S1_SCLK          = 99, /*!< I2S1 Serial Clock(BCLK) */
    FUNC_I2S1_WS            = 100,    /*!< I2S1 Word Select(LRCLK) */
    FUNC_I2S1_IN_D0         = 101,    /*!< I2S1 Serial Data Input 0 */
    FUNC_I2S1_IN_D1         = 102,    /*!< I2S1 Serial Data Input 1 */
    FUNC_I2S1_IN_D2         = 103,    /*!< I2S1 Serial Data Input 2 */
    FUNC_I2S1_IN_D3         = 104,    /*!< I2S1 Serial Data Input 3 */
    FUNC_I2S1_OUT_D0        = 105,    /*!< I2S1 Serial Data Output 0 */
    FUNC_I2S1_OUT_D1        = 106,    /*!< I2S1 Serial Data Output 1 */
    FUNC_I2S1_OUT_D2        = 107,    /*!< I2S1 Serial Data Output 2 */
    FUNC_I2S1_OUT_D3        = 108,    /*!< I2S1 Serial Data Output 3 */
    FUNC_I2S2_MCLK          = 109,    /*!< I2S2 Master Clock */
    FUNC_I2S2_SCLK          = 110,    /*!< I2S2 Serial Clock(BCLK) */
    FUNC_I2S2_WS            = 111,    /*!< I2S2 Word Select(LRCLK) */
```

```
    FUNC_I2S2_IN_D0       = 112,    /*!< I2S2 Serial Data Input 0 */
    FUNC_I2S2_IN_D1       = 113,    /*!< I2S2 Serial Data Input 1 */
    FUNC_I2S2_IN_D2       = 114,    /*!< I2S2 Serial Data Input 2 */
    FUNC_I2S2_IN_D3       = 115,    /*!< I2S2 Serial Data Input 3 */
    FUNC_I2S2_OUT_D0      = 116,    /*!< I2S2 Serial Data Output 0 */
    FUNC_I2S2_OUT_D1      = 117,    /*!< I2S2 Serial Data Output 1 */
    FUNC_I2S2_OUT_D2      = 118,    /*!< I2S2 Serial Data Output 2 */
    FUNC_I2S2_OUT_D3      = 119,    /*!< I2S2 Serial Data Output 3 */
    FUNC_RESV0            = 120,    /*!< Reserved function */
    FUNC_RESV1            = 121,    /*!< Reserved function */
    FUNC_RESV2            = 122,    /*!< Reserved function */
    FUNC_RESV3            = 123,    /*!< Reserved function */
    FUNC_RESV4            = 124,    /*!< Reserved function */
    FUNC_RESV5            = 125,    /*!< Reserved function */
    FUNC_I2C0_SCLK        = 126,    /*!< I2C0 Serial Clock */
    FUNC_I2C0_SDA         = 127,    /*!< I2C0 Serial Data */
    FUNC_I2C1_SCLK        = 128,    /*!< I2C1 Serial Clock */
    FUNC_I2C1_SDA         = 129,    /*!< I2C1 Serial Data */
    FUNC_I2C2_SCLK        = 130,    /*!< I2C2 Serial Clock */
    FUNC_I2C2_SDA         = 131,    /*!< I2C2 Serial Data */
    FUNC_CMOS_XCLK        = 132,    /*!< DVP System Clock */
    FUNC_CMOS_RST         = 133,    /*!< DVP System Reset */
    FUNC_CMOS_PWDN        = 134,    /*!< DVP Power Down Mode */
    FUNC_CMOS_VSYNC       = 135,    /*!< DVP Vertical Sync */
    FUNC_CMOS_HREF        = 136,    /*!< DVP Horizontal Reference output */
    FUNC_CMOS_PCLK        = 137,    /*!< Pixel Clock */
    FUNC_CMOS_D0          = 138,    /*!< Data Bit 0 */
    FUNC_CMOS_D1          = 139,    /*!< Data Bit 1 */
    FUNC_CMOS_D2          = 140,    /*!< Data Bit 2 */
    FUNC_CMOS_D3          = 141,    /*!< Data Bit 3 */
    FUNC_CMOS_D4          = 142,    /*!< Data Bit 4 */
    FUNC_CMOS_D5          = 143,    /*!< Data Bit 5 */
    FUNC_CMOS_D6          = 144,    /*!< Data Bit 6 */
    FUNC_CMOS_D7          = 145,    /*!< Data Bit 7 */
    FUNC_SCCB_SCLK        = 146,    /*!< SCCB Serial Clock */
    FUNC_SCCB_SDA         = 147,    /*!< SCCB Serial Data */
    FUNC_UART1_CTS        = 148,    /*!< UART1 Clear To Send */
    FUNC_UART1_DSR        = 149,    /*!< UART1 Data Set Ready */
    FUNC_UART1_DCD        = 150,    /*!< UART1 Data Carrier Detect */
    FUNC_UART1_RI         = 151,    /*!< UART1 Ring Indicator */
    FUNC_UART1_SIR_IN     = 152,    /*!< UART1 Serial Infrared Input */
    FUNC_UART1_DTR        = 153,    /*!< UART1 Data Terminal Ready */
    FUNC_UART1_RTS        = 154,    /*!< UART1 Request To Send */
    FUNC_UART1_OUT2       = 155,    /*!< UART1 User-designated Output 2 */
    FUNC_UART1_OUT1       = 156,    /*!< UART1 User-designated Output 1 */
    FUNC_UART1_SIR_OUT    = 157,    /*!< UART1 Serial Infrared Output */
    FUNC_UART1_BAUD       = 158,    /*!< UART1 Transmit Clock Output */
    FUNC_UART1_RE         = 159,    /*!< UART1 I2S_RECEIVER Output Enable */
    FUNC_UART1_DE         = 160,    /*!< UART1 Driver Output Enable */
    FUNC_UART1_RS485_EN   = 161,    /*!< UART1 RS485 Enable */
    FUNC_UART2_CTS        = 162,    /*!< UART2 Clear To Send */
    FUNC_UART2_DSR        = 163,    /*!< UART2 Data Set Ready */
    FUNC_UART2_DCD        = 164,    /*!< UART2 Data Carrier Detect */
```

```
    FUNC_UART2_RI         = 165,    /*!< UART2 Ring Indicator */
    FUNC_UART2_SIR_IN     = 166,    /*!< UART2 Serial Infrared Input */
    FUNC_UART2_DTR        = 167,    /*!< UART2 Data Terminal Ready */
    FUNC_UART2_RTS        = 168,    /*!< UART2 Request To Send */
    FUNC_UART2_OUT2       = 169,    /*!< UART2 User-designated Output 2 */
    FUNC_UART2_OUT1       = 170,    /*!< UART2 User-designated Output 1 */
    FUNC_UART2_SIR_OUT    = 171,    /*!< UART2 Serial Infrared Output */
    FUNC_UART2_BAUD       = 172,    /*!< UART2 Transmit Clock Output */
    FUNC_UART2_RE         = 173,    /*!< UART2 I2S_RECEIVER Output Enable */
    FUNC_UART2_DE         = 174,    /*!< UART2 Driver Output Enable */
    FUNC_UART2_RS485_EN   = 175,    /*!< UART2 RS485 Enable */
    FUNC_UART3_CTS        = 176,    /*!< UART3 Clear To Send */
    FUNC_UART3_DSR        = 177,    /*!< UART3 Data Set Ready */
    FUNC_UART3_DCD        = 178,    /*!< UART3 Data Carrier Detect */
    FUNC_UART3_RI         = 179,    /*!< UART3 Ring Indicator */
    FUNC_UART3_SIR_IN     = 180,    /*!< UART3 Serial Infrared Input */
    FUNC_UART3_DTR        = 181,    /*!< UART3 Data Terminal Ready */
    FUNC_UART3_RTS        = 182,    /*!< UART3 Request To Send */
    FUNC_UART3_OUT2       = 183,    /*!< UART3 User-designated Output 2 */
    FUNC_UART3_OUT1       = 184,    /*!< UART3 User-designated Output 1 */
    FUNC_UART3_SIR_OUT    = 185,    /*!< UART3 Serial Infrared Output */
    FUNC_UART3_BAUD       = 186,    /*!< UART3 Transmit Clock Output */
    FUNC_UART3_RE         = 187,    /*!< UART3 I2S_RECEIVER Output Enable */
    FUNC_UART3_DE         = 188,    /*!< UART3 Driver Output Enable */
    FUNC_UART3_RS485_EN   = 189,    /*!< UART3 RS485 Enable */
    FUNC_TIMER0_TOGGLE1   = 190,    /*!< TIMER0 Toggle Output 1 */
    FUNC_TIMER0_TOGGLE2   = 191,    /*!< TIMER0 Toggle Output 2 */
    FUNC_TIMER0_TOGGLE3   = 192,    /*!< TIMER0 Toggle Output 3 */
    FUNC_TIMER0_TOGGLE4   = 193,    /*!< TIMER0 Toggle Output 4 */
    FUNC_TIMER1_TOGGLE1   = 194,    /*!< TIMER1 Toggle Output 1 */
    FUNC_TIMER1_TOGGLE2   = 195,    /*!< TIMER1 Toggle Output 2 */
    FUNC_TIMER1_TOGGLE3   = 196,    /*!< TIMER1 Toggle Output 3 */
    FUNC_TIMER1_TOGGLE4   = 197,    /*!< TIMER1 Toggle Output 4 */
    FUNC_TIMER2_TOGGLE1   = 198,    /*!< TIMER2 Toggle Output 1 */
    FUNC_TIMER2_TOGGLE2   = 199,    /*!< TIMER2 Toggle Output 2 */
    FUNC_TIMER2_TOGGLE3   = 200,    /*!< TIMER2 Toggle Output 3 */
    FUNC_TIMER2_TOGGLE4   = 201,    /*!< TIMER2 Toggle Output 4 */
    FUNC_CLK_SPI2         = 202,    /*!< Clock SPI2 */
    FUNC_CLK_I2C2         = 203,    /*!< Clock I2C2 */
    FUNC_INTERNAL0        = 204,    /*!< Internal function signal 0 */
    FUNC_INTERNAL1        = 205,    /*!< Internal function signal 1 */
    FUNC_INTERNAL2        = 206,    /*!< Internal function signal 2 */
    FUNC_INTERNAL3        = 207,    /*!< Internal function signal 3 */
    FUNC_INTERNAL4        = 208,    /*!< Internal function signal 4 */
    FUNC_INTERNAL5        = 209,    /*!< Internal function signal 5 */
    FUNC_INTERNAL6        = 210,    /*!< Internal function signal 6 */
    FUNC_INTERNAL7        = 211,    /*!< Internal function signal 7 */
    FUNC_INTERNAL8        = 212,    /*!< Internal function signal 8 */
    FUNC_INTERNAL9        = 213,    /*!< Internal function signal 9 */
    FUNC_INTERNAL10       = 214,    /*!< Internal function signal 10 */
    FUNC_INTERNAL11       = 215,    /*!< Internal function signal 11 */
    FUNC_INTERNAL12       = 216,    /*!< Internal function signal 12 */
    FUNC_INTERNAL13       = 217,    /*!< Internal function signal 13 */
```

```
    FUNC_INTERNAL14       = 218,    /*!< Internal function signal 14 */
    FUNC_INTERNAL15       = 219,    /*!< Internal function signal 15 */
    FUNC_INTERNAL16       = 220,    /*!< Internal function signal 16 */
    FUNC_INTERNAL17       = 221,    /*!< Internal function signal 17 */
    FUNC_CONSTANT         = 222,    /*!< Constant function */
    FUNC_INTERNAL18       = 223,    /*!< Internal function signal 18 */
    FUNC_DEBUG0           = 224,    /*!< Debug function 0 */
    FUNC_DEBUG1           = 225,    /*!< Debug function 1 */
    FUNC_DEBUG2           = 226,    /*!< Debug function 2 */
    FUNC_DEBUG3           = 227,    /*!< Debug function 3 */
    FUNC_DEBUG4           = 228,    /*!< Debug function 4 */
    FUNC_DEBUG5           = 229,    /*!< Debug function 5 */
    FUNC_DEBUG6           = 230,    /*!< Debug function 6 */
    FUNC_DEBUG7           = 231,    /*!< Debug function 7 */
    FUNC_DEBUG8           = 232,    /*!< Debug function 8 */
    FUNC_DEBUG9           = 233,    /*!< Debug function 9 */
    FUNC_DEBUG10          = 234,    /*!< Debug function 10 */
    FUNC_DEBUG11          = 235,    /*!< Debug function 11 */
    FUNC_DEBUG12          = 236,    /*!< Debug function 12 */
    FUNC_DEBUG13          = 237,    /*!< Debug function 13 */
    FUNC_DEBUG14          = 238,    /*!< Debug function 14 */
    FUNC_DEBUG15          = 239,    /*!< Debug function 15 */
    FUNC_DEBUG16          = 240,    /*!< Debug function 16 */
    FUNC_DEBUG17          = 241,    /*!< Debug function 17 */
    FUNC_DEBUG18          = 242,    /*!< Debug function 18 */
    FUNC_DEBUG19          = 243,    /*!< Debug function 19 */
    FUNC_DEBUG20          = 244,    /*!< Debug function 20 */
    FUNC_DEBUG21          = 245,    /*!< Debug function 21 */
    FUNC_DEBUG22          = 246,    /*!< Debug function 22 */
    FUNC_DEBUG23          = 247,    /*!< Debug function 23 */
    FUNC_DEBUG24          = 248,    /*!< Debug function 24 */
    FUNC_DEBUG25          = 249,    /*!< Debug function 25 */
    FUNC_DEBUG26          = 250,    /*!< Debug function 26 */
    FUNC_DEBUG27          = 251,    /*!< Debug function 27 */
    FUNC_DEBUG28          = 252,    /*!< Debug function 28 */
    FUNC_DEBUG29          = 253,    /*!< Debug function 29 */
    FUNC_DEBUG30          = 254,    /*!< Debug function 30 */
    FUNC_DEBUG31          = 255,    /*!< Debug function 31 */
    FUNC_MAX              = 256,    /*!< Function numbers */
} fpioa_function_t;
```

6.4.5.3　成员

成员名称	描述
FUNC_JTAG_TCLK	JTAG 时钟接口
FUNC_JTAG_TDI	JTAG 数据输入接口
FUNC_JTAG_TMS	JTAG 控制 TAP 状态机的转换
FUNC_JTAG_TDO	JTAG 数据输出接口
FUNC_SPI0_D0	SPI0 数据线 0

成员名称	描述
FUNC_SPI0_D1	SPI0 数据线 1
FUNC_SPI0_D2	SPI0 数据线 2
FUNC_SPI0_D3	SPI0 数据线 3
FUNC_SPI0_D4	SPI0 数据线 4
FUNC_SPI0_D5	SPI0 数据线 5
FUNC_SPI0_D6	SPI0 数据线 6
FUNC_SPI0_D7	SPI0 数据线 7
FUNC_SPI0_SS0	SPI0 片选信号 0
FUNC_SPI0_SS1	SPI0 片选信号 1
FUNC_SPI0_SS2	SPI0 片选信号 2
FUNC_SPI0_SS3	SPI0 片选信号 3
FUNC_SPI0_ARB	SPI0 仲裁信号
FUNC_SPI0_SCLK	SPI0 时钟
FUNC_UARTHS_RX	UART 高速接收数据接口
FUNC_UARTHS_TX	UART 高速发送数据接口
FUNC_RESV6	保留功能
FUNC_RESV7	保留功能
FUNC_CLK_SPI1	SPI1 时钟
FUNC_CLK_I2C1	I2C1 时钟
FUNC_GPIOHS0	高速 GPIO0
FUNC_GPIOHS1	高速 GPIO1
FUNC_GPIOHS2	高速 GPIO2
FUNC_GPIOHS3	高速 GPIO3
FUNC_GPIOHS4	高速 GPIO4
FUNC_GPIOHS5	高速 GPIO5
FUNC_GPIOHS6	高速 GPIO6
FUNC_GPIOHS7	高速 GPIO7
FUNC_GPIOHS8	高速 GPIO8
FUNC_GPIOHS9	高速 GPIO9
FUNC_GPIOHS10	高速 GPIO10
FUNC_GPIOHS11	高速 GPIO11
FUNC_GPIOHS12	高速 GPIO12
FUNC_GPIOHS13	高速 GPIO13
FUNC_GPIOHS14	高速 GPIO14
FUNC_GPIOHS15	高速 GPIO15

成员名称	描述
FUNC_GPIOHS16	高速 GPIO16
FUNC_GPIOHS17	高速 GPIO17
FUNC_GPIOHS18	高速 GPIO18
FUNC_GPIOHS19	高速 GPIO19
FUNC_GPIOHS20	高速 GPIO20
FUNC_GPIOHS21	高速 GPIO21
FUNC_GPIOHS22	高速 GPIO22
FUNC_GPIOHS23	高速 GPIO23
FUNC_GPIOHS24	高速 GPIO24
FUNC_GPIOHS25	高速 GPIO25
FUNC_GPIOHS26	高速 GPIO26
FUNC_GPIOHS27	高速 GPIO27
FUNC_GPIOHS28	高速 GPIO28
FUNC_GPIOHS29	高速 GPIO29
FUNC_GPIOHS30	高速 GPIO30
FUNC_GPIOHS31	高速 GPIO31
FUNC_GPIO0	GPIO0
FUNC_GPIO1	GPIO1
FUNC_GPIO2	GPIO2
FUNC_GPIO3	GPIO3
FUNC_GPIO4	GPIO4
FUNC_GPIO5	GPIO5
FUNC_GPIO6	GPIO6
FUNC_GPIO7	GPIO7
FUNC_UART1_RX	UART1 接收数据接口
FUNC_UART1_TX	UART1 发送数据接口
FUNC_UART2_RX	UART2 接收数据接口
FUNC_UART2_TX	UART2 发送数据接口
FUNC_UART3_RX	UART3 接收数据接口
FUNC_UART3_TX	UART3 发送数据接口
FUNC_SPI1_D0	SPI1 数据线 0
FUNC_SPI1_D1	SPI1 数据线 1
FUNC_SPI1_D2	SPI1 数据线 2
FUNC_SPI1_D3	SPI1 数据线 3
FUNC_SPI1_D4	SPI1 数据线 4

成员名称	描述
FUNC_SPI1_D5	SPI1 数据线 5
FUNC_SPI1_D6	SPI1 数据线 6
FUNC_SPI1_D7	SPI1 数据线 7
FUNC_SPI1_SS0	SPI1 片选信号 0
FUNC_SPI1_SS1	SPI1 片选信号 1
FUNC_SPI1_SS2	SPI1 片选信号 2
FUNC_SPI1_SS3	SPI1 片选信号 3
FUNC_SPI1_ARB	SPI1 仲裁信号
FUNC_SPI1_SCLK	SPI1 时钟
FUNC_SPI_SLAVE_D0	SPI 从模式数据线 0
FUNC_SPI_SLAVE_SS	SPI 从模式片选信号
FUNC_SPI_SLAVE_SCLK	SPI 从模式时钟
FUNC_I2S0_MCLK	I2S0 主时钟（系统时钟）
FUNC_I2S0_SCLK	I2S0 串行时钟（位时钟）
FUNC_I2S0_WS	I2S0 帧时钟
FUNC_I2S0_IN_D0	I2S0 串行输入数据接口 0
FUNC_I2S0_IN_D1	I2S0 串行输入数据接口 1
FUNC_I2S0_IN_D2	I2S0 串行输入数据接口 2
FUNC_I2S0_IN_D3	I2S0 串行输入数据接口 3
FUNC_I2S0_OUT_D0	I2S0 串行输出数据接口 0
FUNC_I2S0_OUT_D1	I2S0 串行输出数据接口 1
FUNC_I2S0_OUT_D2	I2S0 串行输出数据接口 2
FUNC_I2S0_OUT_D3	I2S0 串行输出数据接口 3
FUNC_I2S1_MCLK	I2S1 主时钟（系统时钟）
FUNC_I2S1_SCLK	I2S1 串行时钟（位时钟）
FUNC_I2S1_WS	I2S1 帧时钟
FUNC_I2S1_IN_D0	I2S1 串行输入数据接口 0
FUNC_I2S1_IN_D1	I2S1 串行输入数据接口 1
FUNC_I2S1_IN_D2	I2S1 串行输入数据接口 2
FUNC_I2S1_IN_D3	I2S1 串行输入数据接口 3
FUNC_I2S1_OUT_D0	I2S1 串行输出数据接口 0
FUNC_I2S1_OUT_D1	I2S1 串行输出数据接口 1
FUNC_I2S1_OUT_D2	I2S1 串行输出数据接口 2
FUNC_I2S1_OUT_D3	I2S1 串行输出数据接口 3
FUNC_I2S2_MCLK	I2S2 主时钟（系统时钟）

成员名称	描述
FUNC_I2S2_SCLK	I2S2 串行时钟（位时钟）
FUNC_I2S2_WS	I2S2 帧时钟
FUNC_I2S2_IN_D0	I2S2 串行输入数据接口 0
FUNC_I2S2_IN_D1	I2S2 串行输入数据接口 1
FUNC_I2S2_IN_D2	I2S2 串行输入数据接口 2
FUNC_I2S2_IN_D3	I2S2 串行输入数据接口 3
FUNC_I2S2_OUT_D0	I2S2 串行输出数据接口 0
FUNC_I2S2_OUT_D1	I2S2 串行输出数据接口 1
FUNC_I2S2_OUT_D2	I2S2 串行输出数据接口 2
FUNC_I2S2_OUT_D3	I2S2 串行输出数据接口 3
FUNC_RESV0	保留功能
FUNC_RESV1	保留功能
FUNC_RESV2	保留功能
FUNC_RESV3	保留功能
FUNC_RESV4	保留功能
FUNC_RESV5	保留功能
FUNC_I2C0_SCLK	I2C0 串行时钟
FUNC_I2C0_SDA	I2C0 串行数据接口
FUNC_I2C1_SCLK	I2C1 串行时钟
FUNC_I2C1_SDA	I2C1 串行数据接口
FUNC_I2C2_SCLK	I2C2 串行时钟
FUNC_I2C2_SDA	I2C2 串行数据接口
FUNC_CMOS_XCLK	DVP 系统时钟
FUNC_CMOS_RST	DVP 系统复位信号
FUNC_CMOS_PWDN	DVP 使能信号
FUNC_CMOS_VSYNC	DVP 场同步
FUNC_CMOS_HREF	DVP 行参考信号
FUNC_CMOS_PCLK	像素时钟
FUNC_CMOS_D0	像素数据 0
FUNC_CMOS_D1	像素数据 1
FUNC_CMOS_D2	像素数据 2
FUNC_CMOS_D3	像素数据 3
FUNC_CMOS_D4	像素数据 4
FUNC_CMOS_D5	像素数据 5
FUNC_CMOS_D6	像素数据 6

成员名称	描述
FUNC_CMOS_D7	像素数据 7
FUNC_SCCB_SCLK	SCCB 时钟
FUNC_SCCB_SDA	SCCB 串行数据信号
FUNC_UART1_CTS	UART1 清除发送信号
FUNC_UART1_DSR	UART1 数据设备准备信号
FUNC_UART1_DCD	UART1 数据载波检测
FUNC_UART1_RI	UART1 振铃指示
FUNC_UART1_SIR_IN	UART1 串行红外输入信号
FUNC_UART1_DTR	UART1 数据终端准备信号
FUNC_UART1_RTS	UART1 发送请求信号
FUNC_UART1_OUT2	UART1 用户指定输出信号 2
FUNC_UART1_OUT1	UART1 用户指定输出信号 1
FUNC_UART1_SIR_OUT	UART1 串行红外输出信号
FUNC_UART1_BAUD	UART1 时钟
FUNC_UART1_RE	UART1 接收使能
FUNC_UART1_DE	UART1 发送使能
FUNC_UART1_RS485_EN	UART1 使能 RS485
FUNC_UART2_CTS	UART2 清除发送信号
FUNC_UART2_DSR	UART2 数据设备准备信号
FUNC_UART2_DCD	UART2 数据载波检测
FUNC_UART2_RI	UART2 振铃指示
FUNC_UART2_SIR_IN	UART2 串行红外输入信号
FUNC_UART2_DTR	UART2 数据终端准备信号
FUNC_UART2_RTS	UART2 发送请求信号
FUNC_UART2_OUT2	UART2 用户指定输出信号 2
FUNC_UART2_OUT1	UART2 用户指定输出信号 1
FUNC_UART2_SIR_OUT	UART2 串行红外输出信号
FUNC_UART2_BAUD	UART2 时钟
FUNC_UART2_RE	UART2 接收使能
FUNC_UART2_DE	UART2 发送使能
FUNC_UART2_RS485_EN	UART2 使能 RS485
FUNC_UART3_CTS	清除发送信号
FUNC_UART3_DSR	数据设备准备信号
FUNC_UART3_DCD	UART3 数据载波检测
FUNC_UART3_RI	UART3 振铃指示

成员名称	描述
FUNC_UART3_SIR_IN	UART3 串行红外输入信号
FUNC_UART3_DTR	UART3 数据终端准备信号
FUNC_UART3_RTS	UART3 发送请求信号
FUNC_UART3_OUT2	UART3 用户指定输出信号 2
FUNC_UART3_OUT1	UART3 用户指定输出信号 1
FUNC_UART3_SIR_OUT	UART3 串行红外输出信号
FUNC_UART3_BAUD	UART3 时钟
FUNC_UART3_RE	UART3 接收使能
FUNC_UART3_DE	UART3 发送使能
FUNC_UART3_RS485_EN	UART3 使能 RS485
FUNC_TIMER0_TOGGLE1	TIMER0 输出信号 1
FUNC_TIMER0_TOGGLE2	TIMER0 输出信号 2
FUNC_TIMER0_TOGGLE3	TIMER0 输出信号 3
FUNC_TIMER0_TOGGLE4	TIMER0 输出信号 4
FUNC_TIMER1_TOGGLE1	TIMER1 输出信号 1
FUNC_TIMER1_TOGGLE2	TIMER1 输出信号 2
FUNC_TIMER1_TOGGLE3	TIMER1 输出信号 3
FUNC_TIMER1_TOGGLE4	TIMER1 输出信号 4
FUNC_TIMER2_TOGGLE1	TIMER2 输出信号 1
FUNC_TIMER2_TOGGLE2	TIMER2 输出信号 2
FUNC_TIMER2_TOGGLE3	TIMER2 输出信号 3
FUNC_TIMER2_TOGGLE4	TIMER2 输出信号 4
FUNC_CLK_SPI2	SPI2 时钟
FUNC_CLK_I2C2	I2C2 时钟
FUNC_INTERNAL0	内部功能 0
FUNC_INTERNAL1	内部功能 1
FUNC_INTERNAL2	内部功能 2
FUNC_INTERNAL3	内部功能 3
FUNC_INTERNAL4	内部功能 4
FUNC_INTERNAL5	内部功能 5
FUNC_INTERNAL6	内部功能 6
FUNC_INTERNAL7	内部功能 7
FUNC_INTERNAL8	内部功能 8
FUNC_INTERNAL9	内部功能 9
FUNC_INTERNAL10	内部功能 10

成员名称	描述
FUNC_INTERNAL11	内部功能 11
FUNC_INTERNAL12	内部功能 12
FUNC_INTERNAL13	内部功能 13
FUNC_INTERNAL14	内部功能 14
FUNC_INTERNAL15	内部功能 15
FUNC_INTERNAL16	内部功能 16
FUNC_INTERNAL17	内部功能 17
FUNC_CONSTANT	常量
FUNC_INTERNAL18	内部功能 18
FUNC_DEBUG0	调试功能 0
FUNC_DEBUG1	调试功能 1
FUNC_DEBUG2	调试功能 2
FUNC_DEBUG3	调试功能 3
FUNC_DEBUG4	调试功能 4
FUNC_DEBUG5	调试功能 5
FUNC_DEBUG6	调试功能 6
FUNC_DEBUG7	调试功能 7
FUNC_DEBUG8	调试功能 8
FUNC_DEBUG9	调试功能 9
FUNC_DEBUG10	调试功能 10
FUNC_DEBUG11	调试功能 11
FUNC_DEBUG12	调试功能 12
FUNC_DEBUG13	调试功能 13
FUNC_DEBUG14	调试功能 14
FUNC_DEBUG15	调试功能 15
FUNC_DEBUG16	调试功能 16
FUNC_DEBUG17	调试功能 17
FUNC_DEBUG18	调试功能 18
FUNC_DEBUG19	调试功能 19
FUNC_DEBUG20	调试功能 20
FUNC_DEBUG21	调试功能 21
FUNC_DEBUG22	调试功能 22
FUNC_DEBUG23	调试功能 23
FUNC_DEBUG24	调试功能 24
FUNC_DEBUG25	调试功能 25

成员名称	描述
FUNC_DEBUG26	调试功能 26
FUNC_DEBUG27	调试功能 27
FUNC_DEBUG28	调试功能 28
FUNC_DEBUG29	调试功能 29
FUNC_DEBUG30	调试功能 30
FUNC_DEBUG31	调试功能 31

第 7 章

数字摄像头接口（DVP）

7.1 概述

DVP 是摄像头接口模块，支持把摄像头输入图像数据转发给 AI 模块或者内存。

7.2 功能描述

DVP 模块具有以下功能：

- RGB565 和 RGB24Planar 共 2 个视频数据输出端口
- 支持丢弃不需要处理的帧

7.3 API 参考

对应的头文件 dvp.h
为用户提供以下接口

- dvp_init
- dvp_set_output_enable
- dvp_set_image_format
- dvp_set_image_size
- dvp_set_ai_addr
- dvp_set_display_addr
- dvp_config_interrupt
- dvp_get_interrupt
- dvp_clear_interrupt

- dvp_start_convert
- dvp_enable_burst
- dvp_disable_burst
- dvp_enable_auto
- dvp_disable_auto
- dvp_sccb_send_data
- dvp_sccb_receive_data
- dvp_sccb_set_clk_rate
- dvp_set_xclk_rate

7.3.1 dvp_init

7.3.1.1 描述

初始化 DVP。

7.3.1.2 函数原型

```
void dvp_init(uint8_t reg_len)
```

7.3.1.3 参数

参数名称	描述	输入输出
reg_len	sccb 寄存器长度	输入

7.3.1.4 返回值

无

7.3.2 dvp_set_output_enable

7.3.2.1 描述

设置输出模式使能或禁用。

7.3.2.2 函数原型

```
void dvp_set_output_enable(dvp_output_mode_t index, int enable)
```

7.3.2.3　参数

参数名称	描述	输入输出
index	图像输出至内存或 AI	输入
enable	0：禁用 1：使能	输入

7.3.2.4　返回值

无。

7.3.3　dvp_set_image_format

7.3.3.1　描述

设置图像接收模式，RGB 或 YUV。

7.3.3.2　函数原型

```
void dvp_set_image_format(uint32_t format)
```

7.3.3.3　参数

参数名称	描述	输入输出
format	图像模式 DVP_CFG_RGB_FORMAT RGB 模式 DVP_CFG_YUV_FORMAT YUV 模式	输入

7.3.3.4　返回值

无

7.3.4　dvp_set_image_size

7.3.4.1　描述

设置 DVP 图像采集尺寸。

7.3.4.2　函数原型

```
void dvp_set_image_size(uint32_t width, uint32_t height)
```

7.3.4.3　参数

参数名称	描述	输入输出
width	图像宽度	输入
height	图像高度	输入

7.3.4.4　返回值

无

7.3.5　dvp_set_ai_addr

7.3.5.1　描述

设置 AI 存放图像的地址，供 AI 模块进行算法处理。

7.3.5.2　函数原型

```
void dvp_set_ai_addr(uint32_t r_addr, uint32_t g_addr, uint32_t b_addr)
```

7.3.5.3　参数

参数名称	描述	输入输出
r_addr	红色分量地址	输入
g_addr	绿色分量地址	输入
b_addr	蓝色分量地址	输入

7.3.5.4　返回值

无

7.3.6　dvp_set_display_addr

7.3.6.1　描述

设置采集图像在内存中的存放地址，可以用来显示。

7.3.6.2　函数原型

```
void dvp_set_display_addr(uint32_t addr)
```

7.3.6.3　参数

参数名称	描述	输入输出
addr	存放图像的内存地址	输入

7.3.6.4　返回值

无

7.3.7　dvp_config_interrupt

配置 DVP 中断类型。

7.3.7.1　函数原型

```
void dvp_config_interrupt(uint32_t interrupt, uint8_t enable)
```

7.3.7.2　描述

设置图像开始和结束中断状态，使能或禁用。

7.3.7.3　参数

参数名称	描述	输入输出
interrupt	中断类型 DVP_CFG_START_INT_ENABLE 图像开始采集中断 DVP_CFG_FINISH_INT_ENABLE 图像结束采集中断	输入
enable	0：禁止 1：使能	输入

7.3.7.4　返回值

无。

7.3.8　dvp_get_interrupt

7.3.8.1　描述

判断是否是输入的中断类型。

7.3.8.2　函数原型

```
int dvp_get_interrupt(uint32_t interrupt)
```

7.3.8.3　参数

参数名称	描述	输入输出
interrupt	中断类型 DVP_CFG_START_INT_ENABLE 图像开始采集中断 DVP_CFG_FINISH_INT_ENABLE 图像结束采集中断	输入

7.3.8.4　返回值

返回值	描述
0	否
非 0	是

7.3.9　dvp_clear_interrupt

7.3.9.1　描述

清除中断。

7.3.9.2　函数原型

```
void dvp_clear_interrupt(uint32_t interrupt)
```

7.3.9.3　参数

参数名称	描述	输入输出
interrupt	中断类型 DVP_CFG_START_INT_ENABLE 图像开始采集中断 DVP_CFG_FINISH_INT_ENABLE 图像结束采集中断	输入

7.3.9.4　返回值

无。

7.3.10　dvp_start_convert

7.3.10.1　描述

开始采集图像，在确定图像采集开始中断后调用。

7.3.10.2　函数原型

```
void dvp_start_convert(void)
```

7.3.10.3　参数

无。

7.3.10.4　返回值

无。

7.3.11　dvp_enable_burst

7.3.11.1　描述

使能突发传输模式。

7.3.11.2　函数原型

```
void dvp_enable_burst(void)
```

7.3.11.3　参数

无。

7.3.11.4　返回值

无。

7.3.12　dvp_disable_burst

7.3.12.1　描述

禁用突发传输模式。

7.3.12.2　函数原型

```
void dvp_disable_burst(void)
```

7.3.12.3　参数

无。

7.3.12.4　返回值

无。

7.3.13　dvp_enable_auto

7.3.13.1　描述

使能自动接收图像模式。

7.3.13.2　函数原型

```
void dvp_enable_auto(void)
```

7.3.13.3　参数

无。

7.3.13.4　返回值

无。

7.3.14　dvp_disable_auto

7.3.14.1　描述

禁用自动接收图像模式。

7.3.14.2　函数原型

```
void dvp_disable_auto(void)
```

7.3.14.3　参数

无。

7.3.14.4　返回值

无。

7.3.15　dvp_sccb_send_data

7.3.15.1　描述

通过 sccb 发送数据。

7.3.15.2　函数原型

```
void dvp_sccb_send_data(uint8_t dev_addr, uint16_t reg_addr, uint8_t reg_data)
```

7.3.15.3　参数

参数名称	描述	输入输出
dev_addr	外设图像传感器 SCCB 地址	输入
reg_addr	外设图像传感器寄存器	输入
reg_data	发送的数据	输入

7.3.15.4　返回值

无

7.3.16　dvp_sccb_receive_data

7.3.16.1　描述

通过 SCCB 接收数据。

7.3.16.2　函数原型

```
uint8_t dvp_sccb_receive_data(uint8_t dev_addr, uint16_t reg_addr)
```

7.3.16.3　参数

参数名称	描述	输入输出
dev_addr	外设图像传感器 SCCB 地址	输入
reg_addr	外设图像传感器寄存器	输入

7.3.16.4　返回值

读取寄存器的数据。

7.3.17　dvp_set_xclk_rate

7.3.17.1　描述

设置 xclk 的速率。

7.3.17.2　函数原型

```
uint32_t dvp_set_xclk_rate(uint32_t xclk_rate)
```

7.3.17.3　参数

参数名称	描述	输入输出
xclk_rate	xclk 的速率	输入

7.3.17.4　返回值

xclk 的实际速率。

7.3.18　dvp_sccb_set_clk_rate

7.3.18.1　描述

设置 sccb 的速率。

7.3.18.2　函数原型

```
uint32_t dvp_sccb_set_clk_rate(uint32_t clk_rate)
```

7.3.18.3　参数

参数名称	描述	输入输出
clk_rate	sccb 的速率	输入

7.3.18.4　返回值

返回值	描述
0	失败，设置的速率太低无法满足，请使用 I2C
非 0	实际的 sccb 速率

7.3.19　举例

```
/* 采集320 * 240的RGB图像数据传输至lcd_gram0，及0x40600000 0x40612C00 0x40625800 地址处
    */
uint32_t lcd_gram0[38400] __attribute__((aligned(64)));

int on_irq_dvp(void* ctx)
{
    if (dvp_get_interrupt(DVP_STS_FRAME_FINISH))
    {
        dvp_clear_interrupt(DVP_STS_FRAME_FINISH);
    }
    else
    {
        dvp_start_convert();
        dvp_clear_interrupt(DVP_STS_FRAME_START);
    }
    return 0;
}
plic_init();
dvp_init(8);
dvp_set_xclk_rate(12000000);
dvp_enable_burst();
dvp_set_output_enable(DVP_OUTPUT_AI, 1);
dvp_set_output_enable(DVP_OUTPUT_DISPLAY, 1);
dvp_set_image_format(DVP_CFG_RGB_FORMAT);
dvp_set_image_size(320, 240);
dvp_set_ai_addr((uint32_t)0x40600000, (uint32_t)0x40612C00, (uint32_t)0x40625800);
dvp_set_display_addr(lcd_gram0);
dvp_config_interrupt(DVP_CFG_START_INT_ENABLE | DVP_CFG_FINISH_INT_ENABLE, 0);
dvp_disable_auto();
plic_set_priority(IRQN_DVP_INTERRUPT, 1);
plic_irq_register(IRQN_DVP_INTERRUPT, on_irq_dvp, NULL);
plic_irq_enable(IRQN_DVP_INTERRUPT);
dvp_clear_interrupt(DVP_STS_FRAME_START | DVP_STS_FRAME_FINISH);
dvp_config_interrupt(DVP_CFG_START_INT_ENABLE | DVP_CFG_FINISH_INT_ENABLE, 1);
sysctl_enable_irq();
```

```
/* 通过SCCB向地址0x60的外设0xFF寄存器发送0x01,从寄存器0x1D读取数据 */
dvp_sccb_send_data(0x60, 0xFF, 0x01);
dvp_sccb_receive_data(0x60, 0x1D)
```

7.4　数据类型

相关数据类型、数据结构定义如下：

- dvp_output_mode_t：DVP 输出图像的模式。

7.4.1　dvp_output_mode_t

7.4.1.1　描述

DVP 输入图像的模式。

7.4.1.2　定义

```
typedef enum _dvp_output_mode
{
    DVP_OUTPUT_AI,
    DVP_OUTPUT_DISPLAY,
} dvp_output_mode_t;
```

7.4.1.3　成员

成员名称	描述
DVP_OUTPUT_AI	AI 输出
DVP_OUTPUT_DISPLAY	向内存输出用于显示

第 8 章

快速傅里叶变换加速器 (FFT)

8.1 概述

FFT 模块是用硬件的方式来实现 FFT 的基 2 时分运算加速。

8.2 功能描述

目前该模块可以支持 64 点、128 点、256 点以及 512 点的 FFT 以及 IFFT。在 FFT 内部有两块大小为 512*32bit 的 SRAM，在配置完成后 FFT 会向 DMA 发送 TX 请求，将 DMA 送来的送据放到其中的一块 SRAM 中去，直到满足当前 FFT 运算所需要的数据量并开始 FFT 运算，蝶形运算单元从包含有有效数据的 SRAM 中读出数据，运算结束后将数据写到另外一块 SRAM 中去，下次蝶形运算再从刚写入的 SRAM 中读出数据，运算结束后并写入另外一块 SRAM，如此反复交替进行直到完成整个 FFT 运算。

8.3 API 参考

对应的头文件 `fft.h`
为用户提供以下接口

- `fft_complex_uint16_dma`

8.3.1 fft_complex_uint16_dma

8.3.1.1 描述

FFT 运算。

8.3.1.2　函数原型

```
void fft_complex_uint16_dma(dmac_channel_number_t dma_send_channel_num,
    dmac_channel_number_t dma_receive_channel_num, uint16_t shift, fft_direction_t
    direction, const uint64_t *input, size_t point_num, uint64_t *output);
```

8.3.1.3　参数

参数名称	描述	输入输出
dma_send_channel_num	发送数据使用的 DMA 通道号	输入
dma_receive_channel_num	接收数据使用的 DMA 通道号	输入
shift	FFT 模块 16 位寄存器导致数据溢出 (-32768~32767)，FFT 变换有 9 层，shift 决定哪一层需要移位操作 (如 0x1ff 表示 9 层都做移位操作；0x03 表示第第一层与第二层做移位操作)，防止溢出。如果移位了，则变换后的幅值不是正常 FFT 变换的幅值，对应关系可以参考 fft_test 测试 demo 程序。包含了求解频率点、相位、幅值的示例	输入
direction	FFT 正变换或是逆变换	输入
input	输入的数据序列，格式为 RIRI..，实部与虚部的精度都为 16bit	输入
point_num	待运算的数据点数，只能为 512/256/128/64	输入
output	运算后结果。格式为 RIRI..，实部与虚部的精度都为 16bit	输出

8.3.1.4　返回值

无。

8.3.2　举例

```
#define FFT_N                512U
#define FFT_FORWARD_SHIFT    0x0U
#define FFT_BACKWARD_SHIFT   0x1ffU
#define PI                   3.14159265358979323846
complex_hard_t data_hard[FFT_N] = {0};
for (i = 0; i < FFT_N; i++)
{
    tempf1[0] = 0.3 * cosf(2 * PI * i / FFT_N + PI / 3) * 256;
    tempf1[1] = 0.1 * cosf(16 * 2 * PI * i / FFT_N - PI / 9) * 256;
    tempf1[2] = 0.5 * cosf((19 * 2 * PI * i / FFT_N) + PI / 6) * 256;
    data_hard[i].real = (int16_t)(tempf1[0] + tempf1[1] + tempf1[2] + 10);
    data_hard[i].imag = (int16_t)0;
}
for (int i = 0; i < FFT_N / 2; ++i)
{
    input_data = (fft_data_t *)&buffer_input[i];
    input_data->R1 = data_hard[2 * i].real;
    input_data->I1 = data_hard[2 * i].imag;
    input_data->R2 = data_hard[2 * i + 1].real;
    input_data->I2 = data_hard[2 * i + 1].imag;
}
fft_complex_uint16_dma(DMAC_CHANNEL0, DMAC_CHANNEL1, FFT_FORWARD_SHIFT, FFT_DIR_FORWARD
    , buffer_input, FFT_N, buffer_output);
for (i = 0; i < FFT_N / 2; i++)
{
    output_data = (fft_data_t*)&buffer_output[i];
    data_hard[2 * i].imag = output_data->I1 ;
    data_hard[2 * i].real = output_data->R1 ;
    data_hard[2 * i + 1].imag = output_data->I2 ;
    data_hard[2 * i + 1].real = output_data->R2 ;
}
for (int i = 0; i < FFT_N / 2; ++i)
{
    input_data = (fft_data_t *)&buffer_input[i];
    input_data->R1 = data_hard[2 * i].real;
    input_data->I1 = data_hard[2 * i].imag;
    input_data->R2 = data_hard[2 * i + 1].real;
    input_data->I2 = data_hard[2 * i + 1].imag;
}
fft_complex_uint16_dma(DMAC_CHANNEL0, DMAC_CHANNEL1, FFT_BACKWARD_SHIFT,
    FFT_DIR_BACKWARD, buffer_input, FFT_N, buffer_output);
for (i = 0; i < FFT_N / 2; i++)
{
    output_data = (fft_data_t*)&buffer_output[i];
    data_hard[2 * i].imag = output_data->I1 ;
    data_hard[2 * i].real = output_data->R1 ;
    data_hard[2 * i + 1].imag = output_data->I2 ;
    data_hard[2 * i + 1].real = output_data->R2 ;
```

```
}
```

8.4　数据类型

相关数据类型、数据结构定义如下：

- fft_data_t：FFT 运算传入的数据格式。
- fft_direction_t：FFT 变换模式。

8.4.1　fft_data_t

8.4.1.1　描述

FFT 运算传入的数据格式。

8.4.1.2　定义

```
typedef struct tag_fft_data
{
    int16_t I1;
    int16_t R1;
    int16_t I2;
    int16_t R2;
} fft_data_t;
```

8.4.1.3　成员

成员名称	描述
I1	第一个数据的虚部
R1	第一个数据的实部
I2	第二个数据的虚部
R2	第二个数据的实部

8.4.2　fft_direction_t

8.4.2.1　描述

FFT 变换模式

8.4.2.2　定义

```
typedef enum _fft_direction
{
    FFT_DIR_BACKWARD,
    FFT_DIR_FORWARD,
    FFT_DIR_MAX,
} fft_direction_t;
```

8.4.2.3　成员

成员名称	描述
FFT_DIR_BACKWARD	FFT 逆变换
FFT_DIR_FORWARD	FFT 正变换

第 9 章

安全散列算法加速器 (SHA256)

9.1 功能描述

- 支持 SHA-256 的计算

9.2 API 参考

对应的头文件 sha256.h

为用户提供以下接口

- sha256_init
- sha256_update
- sha256_final
- sha256_hard_calculate

9.2.1 sha256_init

9.2.1.1 描述

初始化 SHA256 加速器外设.

9.2.1.2 函数原型

```
void sha256_init(sha256_context_t *context, size_t input_len)
```

9.2.1.3 参数

参数名称	描述	输入输出
context	SHA256 的上下文对象	输入
input_len	待计算 SHA256 hash 的消息的长度	输入

9.2.1.4　返回值

无。

9.2.2　举例

```
sha256_context_t context;
sha256_init(&context, 128U);
```

9.2.3　sha256_update

9.2.3.1　描述

传入一个数据块参与 SHA256 Hash 计算

9.2.3.2　函数原型

```
void sha256_update(sha256_context_t *context, const void *input, size_t input_len)
```

9.2.3.3　参数

参数名称	描述	输入输出
context	SHA256 的上下文对象	输入
input	待加入计算的 SHA256 计算的数据块	输入
buf_len	待加入计算的 SHA256 计算数据块的长度	输入

9.2.3.4　返回值

无。

9.2.4　sha256_final

9.2.4.1　描述

结束对数据的 SHA256 Hash 计算

9.2.4.2　函数原型

```
void sha256_final(sha256_context_t *context, uint8_t *output)
```

9.2.4.3　参数

参数名称	描述	输入输出
context	SHA256 的上下文对象	输入
output	存放 SHA256 计算的结果，需保证传入这个 buffer 的大小为 32Bytes 以上	输出

9.2.4.4　返回值

无。

9.2.5　sha256_hard_calculate

9.2.5.1　描述

一次性对连续的数据计算它的 SHA256 Hash

9.2.5.2　函数原型

```
void sha256_hard_calculate(const uint8_t *input, size_t input_len, uint8_t *output)
```

9.2.5.3　参数

参数名称	描述	输入输出
input	待 SHA256 计算的数据	输入
input_len	待 SHA256 计算数据的长度	输入
output	存放 SHA256 计算的结果，需保证传入这个 buffer 的大小为 32Bytes 以上	输出

9.2.5.4　返回值

无。

9.2.6　举例

```
uint8_t hash[32];
sha256_hard_calculate((uint8_t *)"abc", 3, hash);
```

9.3　例程

9.3.1　进行一次计算

```
sha256_context_t context;
sha256_init(&context, input_len);
sha256_update(&context, input, input_len);
sha256_final(&context, output);
```

或者可以直接调用 sha256_hard_calculate 函数

```
sha256_hard_calculate(input, input_len, output);
```

9.3.2　进行分块计算

```
sha256_context_t context;
sha256_init(&context, input_piece1_len + input_piece2_len);
sha256_update(&context, input_piece1, input_piece1_len);
sha256_update(&context, input_piece2, input_piece2_len);
sha256_final(&context, output);
```

第 10 章

通用异步收发传输器（UART）

10.1 概述

嵌入式应用通常要求一个简单的并且占用系统资源少的方法来传输数据。通用异步收发传输器（UART）即可以满足这些要求，它能够灵活地与外部设备进行全双工数据交换。

10.2 功能描述

UART 模块具有以下功能：

- 配置 UART 参数
- 自动收取数据到缓冲区

10.3 API 参考

对应的头文件 uart.h
为用户提供以下接口

- uart_init
- uart_config（0.6.0 后不再支持，请使用 uart_configure）
- uart_configure
- uart_send_data
- uart_send_data_dma
- uart_send_data_dma_irq
- uart_receive_data
- uart_receive_data_dma

- uart_receive_data_dma_irq
- uart_irq_register
- uart_irq_deregister

10.3.1 uart_init

10.3.1.1 描述

初始化 uart。

10.3.1.2 函数原型

```
void uart_init(uart_device_number_t channel)
```

10.3.1.3 参数

参数名称	描述	输入输出
channel	UART 号	输入

10.3.1.4 返回值

无。

10.3.2 uart_configure

10.3.2.1 描述

设置 UART 相关参数。该函数已废弃，替代函数为 uart_configure。

10.3.3 uart_configure

10.3.3.1 描述

设置 UART 相关参数。

10.3.3.2 函数原型

```
void uart_configure(uart_device_number_t channel, uint32_t baud_rate, uart_bitwidth_t
    data_width, uart_stopbit_t stopbit, uart_parity_t parity)
```

10.3.3.3　参数

参数名称	描述	输入输出
channel	UART 编号	输入
baud_rate	波特率	输入
data_width	数据位（5-8）	输入
stopbit	停止位	输入
parity	校验位	输入

10.3.3.4　返回值

无。

10.3.4　uart_send_data

10.3.4.1　描述

通过 UART 发送数据。

10.3.4.2　函数原型

```
int uart_send_data(uart_device_number_t channel, const char *buffer, size_t buf_len)
```

10.3.4.3　参数

参数名称	描述	输入输出
channel	UART 编号	输入
buffer	待发送数据	输入
buf_len	待发送数据的长度	输入

10.3.4.4　返回值

已发送数据的长度。

10.3.5　uart_send_data_dma

10.3.5.1　描述

UART 通过 DMA 发送数据。数据全部发送完毕后返回。

10.3.5.2　函数原型

```
void uart_send_data_dma(uart_device_number_t uart_channel, dmac_channel_number_t
    dmac_channel, const uint8_t *buffer, size_t buf_len)
```

10.3.5.3　参数

参数名称	描述	输入输出
uart_channel	UART 编号	输入
dmac_channel	DMA 通道	输入
buffer	待发送数据	输入
buf_len	待发送数据的长度	输入

10.3.5.4　返回值

无。

10.3.6　uart_send_data_dma_irq

10.3.6.1　描述

UART 通过 DMA 发送数据，并设置 DMA 发送完成中断函数，仅单次中断。

10.3.6.2　函数原型

```
void uart_send_data_dma_irq(uart_device_number_t uart_channel, dmac_channel_number_t
    dmac_channel,
                            const uint8_t *buffer, size_t buf_len, plic_irq_callback_t
                                uart_callback,
                            void *ctx, uint32_t priority)
```

10.3.6.3　参数

参数名称	描述	输入输出
uart_channel	UART 编号	输入
dmac_channel	DMA 通道	输入
buffer	待发送数据	输入
buf_len	待发送数据的长度	输入

参数名称	描述	输入输出
uart_callback	DMA 中断回调	输入
ctx	中断函数参数	输入
priority	中断优先级	输入

10.3.6.4　返回值

无。

10.3.7　uart_receive_data

10.3.7.1　描述

通过 UART 读取数据。

10.3.7.2　函数原型

```
int uart_receive_data(uart_device_number_t channel, char *buffer, size_t buf_len);
```

10.3.7.3　参数

参数名称	描述	输入输出
channel	UART 编号	输入
buffer	接收数据	输出
buf_len	接收数据的长度	输入

10.3.7.4　返回值

已接收到的数据长度。

10.3.8　uart_receive_data_dma

10.3.8.1　描述

UART 通过 DMA 接收数据。

10.3.8.2　函数原型

```
void uart_receive_data_dma(uart_device_number_t uart_channel, dmac_channel_number_t
    dmac_channel, uint8_t *buffer, size_t buf_len)
```

10.3.8.3　参数

参数名称	描述	输入输出
uart_channel	UART 编号	输入
dmac_channel	DMA 通道	输入
buffer	接收数据	输出
buf_len	接收数据的长度	输入

10.3.8.4　返回值

无。

10.3.9　uart_receive_data_dma_irq

10.3.9.1　描述

UART 通过 DMA 接收数据，并注册 DMA 接收完成中断函数，仅单次中断。

10.3.9.2　函数原型

```
void uart_receive_data_dma_irq(uart_device_number_t uart_channel, dmac_channel_number_t
     dmac_channel,
                                    uint8_t *buffer, size_t buf_len,
                                        plic_irq_callback_t uart_callback,
                                    void *ctx, uint32_t priority)
```

10.3.9.3　参数

参数名称	描述	输入输出
uart_channel	UART 编号	输入
dmac_channel	DMA 通道	输入
buffer	接收数据	输出
buf_len	接收数据的长度	输入
uart_callback	DMA 中断回调	输入
ctx	中断函数参数	输入
priority	中断优先级	输入

10.3.9.4　返回值

无。

10.3.10　uart_irq_register

10.3.10.1　描述

注册 UART 中断函数。

10.3.10.2　函数原型

```
void uart_irq_register(uart_device_number_t channel, uart_interrupt_mode_t
    interrupt_mode, plic_irq_callback_t uart_callback, void *ctx, uint32_t priority)
```

10.3.10.3　参数

参数名称	描述	输入输出
channel	UART 编号	输入
interrupt_mode	中断类型	输入
uart_callback	中断回调	输入
ctx	中断函数参数	输入
priority	中断优先级	输入

10.3.10.4　返回值

无。

10.3.11　uart_irq_deregister

10.3.11.1　描述

注销 UART 中断函数。

10.3.11.2　函数原型

```
void uart_irq_deregister(uart_device_number_t channel, uart_interrupt_mode_t
    interrupt_mode)
```

10.3.11.3　参数

参数名称	描述	输入输出
channel	UART 编号	输入
interrupt_mode	中断类型	输入

10.3.11.4　返回值

无。

10.3.12　举例

```
/* UART1 波特率115200，8位数据，1位停止位，无校验位 */
uart_init(UART_DEVICE_1);
uart_config(UART_DEVICE_1, 115200, UART_BITWIDTH_8BIT, UART_STOP_1, UART_PARITY_NONE);
char *v_hel = {"hello world!\n"};
/* 发送 hello world! */
uart_send_data(UART_DEVICE_1, hel, strlen(v_hel));
/* 接收数据 */
while(uart_receive_data(UART_DEVICE_1, &recv, 1))
{
    printf("%c ", recv);
}
```

10.4　数据类型

相关数据类型、数据结构定义如下：

- uart_device_number_t：UART 编号。
- uart_bitwidth_t：UART 数据位宽。
- uart_stopbits_t：UART 停止位。
- uart_parity_t：UART 校验位。
- uart_interrupt_mode_t：UART 中断类型，接收或发送。
- uart_send_trigger_t：发送中断或 DMA 触发 FIFO 深度。
- uart_receive_trigger_t：接收中断或 DMA 触发 FIFO 深度。

10.4.1　uart_device_number_t

10.4.1.1　描述

UART 编号。

10.4.1.2　定义

```
typedef enum _uart_device_number
{
    UART_DEVICE_1,
    UART_DEVICE_2,
    UART_DEVICE_3,
    UART_DEVICE_MAX,
} uart_device_number_t;
```

10.4.1.3　成员

成员名称	描述
UART_DEVICE_1	UART 1
UART_DEVICE_2	UART 2
UART_DEVICE_3	UART 3

10.4.2　uart_bitwidth_t

10.4.2.1　描述

UART 数据位宽。

10.4.2.2　定义

```
typedef enum _uart_bitwidth
{
    UART_BITWIDTH_5BIT = 0,
    UART_BITWIDTH_6BIT,
    UART_BITWIDTH_7BIT,
    UART_BITWIDTH_8BIT,
} uart_bitwidth_t;
```

10.4.2.3　成员

成员名称	描述
UART*BITWIDTH*5BIT	5 比特
UART*BITWIDTH*6BIT	6 比特
UART*BITWIDTH*7BIT	7 比特

成员名称	描述
UART*BITWIDTH*8BIT	8 比特

10.4.3　uart_stopbits_t

10.4.3.1　描述

UART 停止位。

10.4.3.2　定义

```
typedef enum _uart_stopbits
{
    UART_STOP_1,
    UART_STOP_1_5,
    UART_STOP_2
} uart_stopbits_t;
```

10.4.3.3　成员

成员名称	描述
UART_STOP_1	1 个停止位
UART_STOP_1_5	1.5 个停止位
UART_STOP_2	2 个停止位

10.4.4　uart_parity_t

10.4.4.1　描述

UART 校验位。

10.4.4.2　定义

```
typedef enum _uart_parity
{
    UART_PARITY_NONE,
    UART_PARITY_ODD,
    UART_PARITY_EVEN
} uart_parity_t;
```

10.4.4.3　成员

成员名称	描述
UART_PARITY_NONE	无校验位
UART_PARITY_ODD	奇校验
UART_PARITY_EVEN	偶校验

10.4.5 uart_interrupt_mode_t

10.4.5.1 描述

UART 中断类型，接收或发送。

10.4.5.2 定义

```
typedef enum _uart_interrupt_mode
{
    UART_SEND = 1,
    UART_RECEIVE = 2,
} uart_interrupt_mode_t;
```

10.4.5.3 成员

成员名称	描述
UART_SEND	UART 发送
UART_RECEIVE	UART 接收

10.4.6 uart_send_trigger_t

10.4.6.1 描述

发送中断或 DMA 触发 FIFO 深度。当 FIFO 中的数据小于等于该值时触发中断或 DMA 传输。FIFO 的深度为 16 字节。

10.4.6.2 定义

```
typedef enum _uart_send_trigger
{
    UART_SEND_FIFO_0,
    UART_SEND_FIFO_2,
    UART_SEND_FIFO_4,
    UART_SEND_FIFO_8,
} uart_send_trigger_t;
```

10.4.6.3　成员

成员名称	描述
UART_SEND_FIFO_0	FIFO 为空
UART_SEND_FIFO_2	FIFO 剩余 2 字节
UART_SEND_FIFO_4	FIFO 剩余 4 字节
UART_SEND_FIFO_8	FIFO 剩余 8 字节

10.4.7　uart_receive_trigger_t

10.4.7.1　描述

接收中断或 DMA 触发 FIFO 深度。当 FIFO 中的数据大于等于该值时触发中断或 DMA 传输。FIFO 的深度为 16 字节。

10.4.7.2　定义

```
typedef enum _uart_receive_trigger
{
    UART_RECEIVE_FIFO_1,
    UART_RECEIVE_FIFO_4,
    UART_RECEIVE_FIFO_8,
    UART_RECEIVE_FIFO_14,
} uart_receive_trigger_t;
```

10.4.7.3　成员

成员名称	描述
UART_RECEIVE_FIFO_1	FIFO 剩余 1 字节
UART_RECEIVE_FIFO_4	FIFO 剩余 2 字节
UART_RECEIVE_FIFO_8	FIFO 剩余 4 字节
UART_RECEIVE_FIFO_14	FIFO 剩余 8 字节

第 11 章

高速通用异步收发传输器（UARTHS）

11.1 概述

嵌入式应用通常要求一个简单的并且占用系统资源少的方法来传输数据。高速通用异步收发传输器（UARTHS）即可以满足这些要求，它能够灵活地与外部设备进行全双工数据交换。目前系统使用该串口做为调试串口，printf 时会调用该串口输出。

11.2 功能描述

UARTHS 模块具有以下功能：

- 配置 UARTHS 参数
- 自动收取数据到缓冲区

11.3 API 参考

对应的头文件 uarths.h
为用户提供以下接口

- uarths_init
- uarths_config
- uarths_receive_data
- uarths_send_data
- uarths_set_irq
- uarths_get_interrupt_mode
- uarths_set_interrupt_cnt

11.3.1　uarths_init

11.3.1.1　描述

初始化 UARTHS，系统默认波特率为 115200 8bit 1 位停止位无检验位。因为 uarths 时钟源为 PLL0，在设置 PLL0 后需要重新调用该函数设置波特率，否则会打印乱码。

11.3.1.2　函数原型

```
void uarths_init(void)
```

11.3.1.3　参数

无。

11.3.1.4　返回值

无。

11.3.2　uarths_config

11.3.2.1　描述

设置 UARTHS 的参数。默认 8bit 数据，无校验位。

11.3.2.2　函数原型

```
void uarths_config(uint32_t baud_rate, uarths_stopbit_t stopbit)
```

11.3.3　参数

参数名称	描述	输入输出
baud_rate	波特率	输入
stopbit	停止位	输入

11.3.3.1　返回值

无。

11.3.4　uarths_receive_data

11.3.4.1　描述

通过 UARTHS 读取数据。

11.3.4.2　函数原型

```
size_t uarths_receive_data(uint8_t *buf, size_t buf_len)
```

11.3.4.3　参数

参数名称	描述	输入输出
buf	接收数据	输出
buf_len	接收数据的长度	输入

11.3.4.4　返回值

已接收到的数据长度。

11.3.5　uarths_send_data

11.3.5.1　描述

通过 UART 发送数据。

11.3.5.2　函数原型

```
size_t uarths_send_data(const uint8_t *buf, size_t buf_len)
```

11.3.5.3　参数

参数名称	描述	输入输出
buf	待发送数据	输入
buf_len	待发送数据的长度	输入

11.3.5.4　返回值

已发送数据的长度。

11.3.6　uarths_set_irq

11.3.6.1　描述

设置 UARTHS 中断回调函数。

11.3.6.2　函数原型

```
void uarths_set_irq(uarths_interrupt_mode_t interrupt_mode, plic_irq_callback_t
    uarths_callback, void *ctx, uint32_t priority)
```

11.3.6.3　参数

参数名称	描述	输入输出
interrupt_mode	中断类型	输入
uarths_callback	中断回调函数	输入
ctx	回调函数的参数	输入
priority	中断优先级	输入

11.3.6.4　返回值

无。

11.3.7　uarths_get_interrupt_mode

11.3.7.1　描述

获取 UARTHS 的中断类型。接收、发送或接收发送同时中断。

11.3.7.2　函数原型

```
uarths_interrupt_mode_t uarths_get_interrupt_mode(void)
```

11.3.7.3　参数

无

11.3.7.4　返回值

当前中断的类型。

11.3.8　uarths_set_interrupt_cnt

11.3.8.1　描述

设置 UARTHS 中断时的 FIFO 深度。当中断类型为 UARTHS_SEND_RECEIVE，发送接收 FIFO 中断深度均为 cnt;

11.3.8.2　函数原型

```
void uarths_set_interrupt_cnt(uarths_interrupt_mode_t interrupt_mode, uint8_t cnt)
```

11.3.8.3　参数

参数名称	描述	输入输出
interrupt_mode	中断类型	输入
cnt	FIFO 深度	输入

11.3.8.4　返回值

无。

11.3.9　举例

```
/* 设置接收中断 中断FIFO深度为0，即接收到数据立即中断并读取接收到的数据。*/
    int uarths_irq(void *ctx)
    {
        if(!uarths_receive_data((uint8_t *)&receive_char, 1))
            printf("Uarths receive ERR!\n");
        return 0;
    }

    plic_init();
    uarths_set_interrupt_cnt(UARTHS_RECEIVE , 0);
    uarths_set_irq(UARTHS_RECEIVE ,uarths_irq, NULL, 4);
    sysctl_enable_irq();
```

11.4　数据类型

相关数据类型、数据结构定义如下：

- uarths_interrupt_mode_t：中断类型。
- uarths_stopbit_t：停止位。

11.4.1　uarths_interrupt_mode_t

11.4.2　描述

UARTHS 中断类型。

11.4.2.1　定义

```
typedef enum _uarths_interrupt_mode
{
    UARTHS_SEND = 1,
    UARTHS_RECEIVE = 2,
    UARTHS_SEND_RECEIVE = 3,
} uarths_interrupt_mode_t;
```

11.4.2.2　成员

成员名称	描述
UARTHS_SEND	发送中断
UARTHS_RECEIVE	接收中断
UARTHS_SEND_RECEIVE	发送接收中断

11.4.3　uarths_stopbit_t

11.4.3.1　描述：

UARTHS 停止位。

11.4.3.2　定义

```
typedef enum _uarths_stopbit
{
    UART_STOP_1,
    UART_STOP_2
} uarths_stopbit_t;
```

11.4.3.3　成员

成员名称	描述
UART_STOP_1	1 位停止位
UART_STOP_2	2 位停止位

第 12 章

看门狗定时器（WDT）

12.1 概述

WDT 提供系统出错或无响应时的恢复功能。

12.2 功能描述

WDT 模块具有以下功能：

- 配置超时时间
- 手动重启计时

12.3 API 参考

对应的头文件 `wdt.h`
为用户提供以下接口

- `wdt_init`
- `wdt_start`(0.6.0 后不再支持，请使用 `wdt_init`)
- `wdt_stop`
- `wdt_feed`
- `wdt_clear_interrupt`

12.3.1　wdt_init

12.3.1.1　描述

配置参数，启动看门狗。不使用中断的话，将 on_irq 设置为 NULL。

12.3.1.2　函数原型

```
uint32_t wdt_init(wdt_device_number_t id, uint64_t time_out_ms, plic_irq_callback_t
    on_irq, void *ctx)
```

12.3.1.3　参数

参数名称	描述	输入输出
id	看门狗编号	输入
time_out_ms	超时时间（毫秒）	输入
on_irq	中断回调函数	输入
ctx	回调函数参数	输入

12.3.1.4　返回值

看门狗超时重启的实际时间（毫秒）。与 time_out_ms 有差异，一般情况会大于这个时间。在外部晶振 26M 的情况下，最大超时时间为 330 毫秒。

12.3.2　wdt_start

12.3.2.1　描述

启动看门狗。

12.3.2.2　函数原型

```
void wdt_start(wdt_device_number_t id, uint64_t time_out_ms, plic_irq_callback_t on_irq
    )
```

12.3.2.3　参数

参数名称	描述	输入输出
id	看门狗编号	输入

参数名称	描述	输入输出
time_out_ms	超时时间（毫秒）	输入
on_irq	中断回调函数	输入

12.3.2.4　返回值

无

12.3.3　wdt_stop

12.3.3.1　描述

关闭看门狗。

12.3.3.2　函数原型

```
void wdt_stop(wdt_device_number_t id)
```

12.3.3.3　参数

参数名称	描述	输入输出
id	看门狗编号	输入

12.3.3.4　返回值

无。

12.3.4　wdt_feed

12.3.4.1　描述

喂狗。

12.3.4.2　函数原型

```
void wdt_feed(wdt_device_number_t id)
```

12.3.4.3　参数

参数名称	描述	输入输出
id	看门狗编号	输入

12.3.4.4　返回值

无。

12.3.5　wdt_clear_interrupt

12.3.5.1　描述

清除中断。如果在中断函数中清除中断，则看门狗不会重启。

12.3.5.2　函数原型

```
void wdt_clear_interrupt(wdt_device_number_t id)
```

12.3.5.3　参数

参数名称	描述	输入输出
id	看门狗编号	输入

12.3.5.4　返回值

无。

12.3.6　举例

```
/* 2秒后进入看门狗中断函数打印Hello_world,再过2s复位 */
int wdt0_irq(void *ctx)
{
    printf("Hello_world\n");
    return 0;
}
plic_init();
sysctl_enable_irq();
wdt_init(WDT_DEVICE_0, 2000, wdt0_irq, NULL);
```

12.4　数据类型

相关数据类型、数据结构定义如下：

- wdt_device_number_t

12.4.1　wdt_device_number_t

12.4.1.1　描述

看门狗编号。

12.4.1.2　定义

```
typedef enum _wdt_device_number
{
    WDT_DEVICE_0,
    WDT_DEVICE_1,
    WDT_DEVICE_MAX,
} wdt_device_number_t;
```

12.4.1.3　成员

成员名称	描述
WDT_DEVICE_0	看门狗 0
WDT_DEVICE_1	看门狗 1

第 13 章

直接内存存取控制器 (DMAC)

13.1 概述

直接存储访问 (Direct Memory Access, DMA) 用于在外设与存储器之间以及存储器与存储器之间提供高速数据传输。可以在无需任何 CPU 操作的情况下通过 DMA 快速移动数据，从而提高了 CPU 的效率。

13.2 功能描述

DMA 模块具有以下功能:

- 自动选择一路空闲的 DMA 通道用于传输
- 根据源地址和目标地址自动选择软件或硬件握手协议
- 支持 1、2、4、8 字节的元素大小，源和目标大小不必一致
- 异步或同步传输功能
- 循环传输功能，常用于刷新屏幕或音频录放等场景

13.3 API 参考

对应的头文件 `dmac.h`
为用户提供以下接口

- `dmac_init`
- `dmac_set_single_mode`
- `dmac_is_done`
- `dmac_wait_done`

- dmac_set_irq
- dmac_set_src_dest_length
- dmac_is_idle
- dmac_wait_idle

13.3.1　dmac_init

13.3.1.1　描述

初始化 DMA。

13.3.1.2　函数原型

```
void dmac_init(void)
```

13.3.1.3　参数

无。

13.3.1.4　返回值

无。

13.3.2　dmac_set_single_mode

13.3.2.1　描述

设置单路 DMA 参数。

13.3.2.2　函数原型

```
void dmac_set_single_mode(dmac_channel_number_t channel_num, const void *src, void *
    dest, dmac_address_increment_t src_inc, dmac_address_increment_t dest_inc,
    dmac_burst_trans_length_t dmac_burst_size, dmac_transfer_width_t dmac_trans_width,
     size_t block_size)
```

13.3.2.3　参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入
src	源地址	输入

参数名称	描述	输入输出
dest	目标地址	输出
src_inc	源地址是否自增	输入
dest_inc	目标地址是否自增	输入
dmac_burst_size	突发传输数量	输入
dmac_trans_width	单次传输数据位宽	输入
block_size	传输数据的个数	输入

13.3.2.4　返回值

无。

13.3.3　dmac_is_done

13.3.3.1　描述

用于 DMAC 启动后判断是否完成传输。用于 DMAC 启动传输后，如果在启动前判断会不准确。

13.3.3.2　函数原型

```
int dmac_is_done(dmac_channel_number_t channel_num)
```

13.3.3.3　参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入

13.3.3.4　返回值

返回值	描述
0	未完成
1	已完成

13.3.4　dmac_wait_done

13.3.4.1　描述

等待 DMA 完成工作。

13.3.4.2　函数原型

```
void dmac_wait_done(dmac_channel_number_t channel_num)
```

13.3.4.3　参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入

13.3.4.4　返回值

无。

13.3.5　dmac_set_irq

13.3.5.1　描述

设置 DMAC 中断的回调函数

13.3.5.2　函数原型

```
void dmac_set_irq(dmac_channel_number_t channel_num , plic_irq_callback_t dmac_callback
    , void *ctx, uint32_t priority)
```

13.3.5.3　参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入
dmac_callback	中断回调函数	输入
ctx	回调函数的参数	输入
priority	中断优先级	输入

13.3.5.4　返回值

无。

13.3.6　dmac_set_src_dest_length

13.3.6.1　描述

设置 DMAC 的源地址、目的地址和长度，然后启动 DMAC 传输。如果 src 为 NULL 则不设置源地址，dest 为 NULL 则不设置目的地址，len<=0 则不设置长度。

该函数一般用于 DMAC 中断中，使 DMA 继续传输数据，而不必再次设置 DMAC 的所有参数以节省时间。

13.3.6.2　函数原型

```
void dmac_set_src_dest_length(dmac_channel_number_t channel_num, const void *src, void
    *dest, size_t len)
```

13.3.6.3　参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入
src	中断回调函数	输入
dest	回调函数的参数	输入
len	中断优先级	输入

13.3.6.4　返回值

无。

13.3.7　dmac_is_idle

13.3.7.1　描述

判断 DMAC 当前通道是否空闲，该函数在传输前和传输后都可以用来判断 DMAC 状态。

13.3.7.2　函数原型

```
int dmac_is_idle(dmac_channel_number_t channel_num)
```

13.3.7.3　参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入

13.3.7.4　返回值

返回值	描述
0	忙
1	空闲

13.3.8　dmac_wait_idle

13.3.8.1　描述

等待 DMAC 进入空闲状态。

13.3.8.2　参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入

13.3.8.3　返回值

无。

13.3.9　举例

```
/* I2C通过DMA发送128个int数据 */
uint32_t buf[128];
dmac_wait_idle(SYSCTL_DMA_CHANNEL_0);
sysctl_dma_select(SYSCTL_DMA_CHANNEL_0, SYSCTL_DMA_SELECT_I2C0_TX_REQ);
dmac_set_single_mode(SYSCTL_DMA_CHANNEL_0, buf, (void*)(&i2c_adapter->data_cmd),
    DMAC_ADDR_INCREMENT, DMAC_ADDR_NOCHANGE, DMAC_MSIZE_4, DMAC_TRANS_WIDTH_32, 128);
dmac_wait_done(SYSCTL_DMA_CHANNEL_0);
```

13.4　数据类型

相关数据类型、数据结构定义如下：

- dmac_channel_number_t: DMA 通道编号。

- dmac_address_increment_t：地址增长方式。
- dmac_burst_trans_length_t：突发传输数量。
- dmac_transfer_width_t：单次传输数据位数。

13.4.1　dmac_channel_number_t

13.4.1.1　描述

DMA 通道编号。

13.4.1.2　定义

```
typedef enum _dmac_channel_number
{
    DMAC_CHANNEL0 = 0,
    DMAC_CHANNEL1 = 1,
    DMAC_CHANNEL2 = 2,
    DMAC_CHANNEL3 = 3,
    DMAC_CHANNEL4 = 4,
    DMAC_CHANNEL5 = 5,
    DMAC_CHANNEL_MAX
} dmac_channel_number_t;
```

13.4.1.3　成员

成员名称	描述
DMAC_CHANNEL0	DMA 通道 0
DMAC_CHANNEL1	DMA 通道 1
DMAC_CHANNEL2	DMA 通道 2
DMAC_CHANNEL3	DMA 通道 3
DMAC_CHANNEL4	DMA 通道 4
DMAC_CHANNEL5	DMA 通道 5

13.4.2　dmac_address_increment_t

13.4.2.1　描述

地址增长方式。

13.4.2.2　定义

```
typedef enum _dmac_address_increment
{
    DMAC_ADDR_INCREMENT = 0x0,
    DMAC_ADDR_NOCHANGE  = 0x1
} dmac_address_increment_t;
```

13.4.2.3　成员

成员名称	描述
DMAC_ADDR_INCREMENT	地址自动增长
DMAC_ADDR_NOCHANGE	地址不变

13.4.3　dmac_burst_trans_length_t

13.4.3.1　描述

突发传输数量。

13.4.3.2　定义

```
typedef enum _dmac_burst_trans_length
{
    DMAC_MSIZE_1   = 0x0,
    DMAC_MSIZE_4   = 0x1,
    DMAC_MSIZE_8   = 0x2,
    DMAC_MSIZE_16  = 0x3,
    DMAC_MSIZE_32  = 0x4,
    DMAC_MSIZE_64  = 0x5,
    DMAC_MSIZE_128 = 0x6,
    DMAC_MSIZE_256 = 0x7
} dmac_burst_trans_length_t;
```

13.4.3.3　成员

成员名称	描述
DMAC_MSIZE_1	单次传输数量乘 1
DMAC_MSIZE_4	单次传输数量乘 4
DMAC_MSIZE_8	单次传输数量乘 8
DMAC_MSIZE_16	单次传输数量乘 16
DMAC_MSIZE_32	单次传输数量乘 32
DMAC_MSIZE_64	单次传输数量乘 64

成员名称	描述
DMAC_MSIZE_128	单次传输数量乘 128
DMAC_MSIZE_256	单次传输数量乘 256

13.4.4　dmac_transfer_width_t

13.4.4.1　描述

单次传输数据位数。

13.4.4.2　定义

```
typedef enum _dmac_transfer_width
{
    DMAC_TRANS_WIDTH_8   = 0x0,
    DMAC_TRANS_WIDTH_16  = 0x1,
    DMAC_TRANS_WIDTH_32  = 0x2,
    DMAC_TRANS_WIDTH_64  = 0x3,
    DMAC_TRANS_WIDTH_128 = 0x4,
    DMAC_TRANS_WIDTH_256 = 0x5
} dmac_transfer_width_t;
```

13.4.4.3　成员

成员名称	描述
DMAC_TRANS_WIDTH_8	单次传输 8 位
DMAC_TRANS_WIDTH_16	单次传输 16 位
DMAC_TRANS_WIDTH_32	单次传输 32 位
DMAC_TRANS_WIDTH_64	单次传输 64 位
DMAC_TRANS_WIDTH_128	单次传输 128 位
DMAC_TRANS_WIDTH_256	单次传输 256 位

第 14 章

集成电路内置总线 (I²C)

14.1 概述

I2C 总线用于和多个外部设备进行通信。多个外部设备可以共用一个 I2C 总线。

14.2 功能描述

I2C 模块具有以下功能:

- 独立的 I2C 设备封装外设相关参数
- 自动处理多设备总线争用

14.3 API 参考

对应的头文件 i2c.h
为用户提供以下接口

- i2c_init
- i2c_init_as_slave
- i2c_send_data
- i2c_send_data_dma
- i2c_recv_data
- i2c_recv_data_dma

14.3.1　i2c_init

14.3.1.1　描述

配置 I²C 器件从地址、寄存器位宽度和 I²C 速率。

14.3.1.2　函数原型

```
void i2c_init(i2c_device_number_t i2c_num, uint32_t slave_address, uint32_t
    address_width, uint32_t i2c_clk)
```

14.3.1.3　参数

参数名称	描述	输入输出
i2c_num	I²C 号	输入
slave_address	I²C 器件从地址	输入
address_width	I²C 器件寄存器宽度 (7 或 10)	输入
i2c_clk	I²C 速率 (Hz)	输入

14.3.1.4　返回值

无。

14.3.2　i2c_init_as_slave

14.3.2.1　描述

配置 I²C 为从模式。

14.3.2.2　函数原型

```
void i2c_init_as_slave(i2c_device_number_t i2c_num, uint32_t slave_address, uint32_t
    address_width, const i2c_slave_handler_t *handler)
```

14.3.2.3　参数

参数名称	描述	输入输出
i2c_num	I²C 号	输入
slave_address	I²C 从模式的地址	输入

参数名称	描述	输入输出
address_width	I²C 器件寄存器宽度 (7 或 10)	输入
handler	I²C 从模式的中断处理函数	输入

14.3.2.4　返回值

无。

14.3.3　i2c_send_data

14.3.3.1　描述

写数据。

14.3.3.2　函数原型

```
int i2c_send_data(i2c_device_number_t i2c_num, const uint8_t *send_buf, size_t
    send_buf_len)
```

14.3.3.3　参数

参数名称	描述	输入输出
i2c_num	I²C 号	输入
send_buf	待传输数据	输入
send_buf_len	待传输数据长度	输入

14.3.3.4　返回值

返回值	描述
0	成功
非 0	失败

14.3.4　i2c_send_data_dma

14.3.4.1　描述

通过 DMA 写数据。

14.3.4.2　函数原型

```
void i2c_send_data_dma(dmac_channel_number_t dma_channel_num, i2c_device_number_t
    i2c_num, const uint8_t *send_buf, size_t send_buf_len)
```

14.3.4.3　参数

参数名称	描述	输入输出
dma_channel_num	使用的 dma 通道号	输入
i2c_num	I²C 号	输入
send_buf	待传输数据	输入
send_buf_len	待传输数据长度	输入

14.3.4.4　返回值

无

14.3.5　i2c_recv_data

14.3.5.1　描述

通过 CPU 读数据。

14.3.5.2　函数原型

```
int i2c_recv_data(i2c_device_number_t i2c_num, const uint8_t *send_buf, size_t
    send_buf_len, uint8_t *receive_buf, size_t receive_buf_len)
```

14.3.5.3　参数

参数名称	描述	输入输出
i2c_num	I²C 总线号	输入
send_buf	待传输数据，一般情况是 i2c 外设的寄存器，如果没有设置为 NULL	输入
send_buf_len	待传输数据长度，如果没有则写 0	输入
receive_buf	接收数据内存	输出
receive_buf_len	接收数据的长度	输入

14.3.5.4　返回值

返回值	描述
0	成功
非 0	失败

14.3.6　i2c_recv_data_dma

14.3.6.1　描述

通过 dma 读数据。

14.3.6.2　函数原型

```
void i2c_recv_data_dma(dmac_channel_number_t dma_send_channel_num,
    dmac_channel_number_t dma_receive_channel_num,
    i2c_device_number_t i2c_num, const uint8_t *send_buf, size_t send_buf_len, uint8_t
        *receive_buf, size_t receive_buf_len)
```

14.3.6.3　参数

参数名称	描述	输入输出
dma_send_channel_num	发送数据使用的 dma 通道	输入
dma_receive_channel_num	接收数据使用的 dma 通道	输入
i2c_num	I²C 总线号	输入
send_buf	待传输数据，一般情况是 i2c 外设的寄存器，如果没有设置为 NULL	输入
send_buf_len	待传输数据长度，如果没有则写 0	输入
receive_buf	接收数据内存	输出
receive_buf_len	接收数据的长度	输入

14.3.6.4　返回值

无

14.3.7　举例

```
/* i2c外设地址是0x32, 7位地址，速率200K */
i2c_init(I2C_DEVICE_0, 0x32, 7, 200000);
uint8_t reg = 0;
```

```
uint8_t data_buf[2] = {0x00,0x01}
data_buf[0] = reg;
/* 向0寄存器写0x01 */
i2c_send_data(I2C_DEVICE_0, data_buf, 2);
i2c_send_data_dma(DMAC_CHANNEL0, I2C_DEVICE_0, data_buf, 4);
/* 从0寄存器读取1字节数据 */
i2c_receive_data(I2C_DEVICE_0, &reg, 1, data_buf, 1);
i2c_receive_data_dma(DMAC_CHANNEL0, DMAC_CHANNEL1, I2C_DEVICE_0,&reg, 1, data_buf, 1);
```

14.4　数据类型

相关数据类型、数据结构定义如下：

- i2c_device_number_t: i2c 号。
- i2c_slave_handler_t: i2c 从模式的中断处理函数句柄

14.4.1　i2c_device_number_t

14.4.1.1　描述

i2c 编号。

14.4.1.2　定义

```
typedef enum _i2c_device_number
{
    I2C_DEVICE_0,
    I2C_DEVICE_1,
    I2C_DEVICE_2,
    I2C_DEVICE_MAX,
} i2c_device_number_t;
```

14.4.2　i2c_slave_handler_t

14.4.2.1　描述

i2c 从模式的中断处理函数句柄。根据不同的中断状态执行相应的函数操作。

14.4.2.2　定义

```
typedef struct _i2c_slave_handler
{
    void(*on_receive)(uint32_t data);
    uint32_t(*on_transmit)();
```

```
    void(*on_event)(i2c_event_t event);
} i2c_slave_handler_t;
```

14.4.2.3　成员

成员名称	描述
I2C_DEVICE_0	I2C 0
I2C_DEVICE_1	I2C 1
I2C_DEVICE_2	I2C 2

第 15 章

串行外设接口 (SPI)

15.1 概述

SPI 是一种高速的，全双工，同步的通信总线。

15.2 功能描述

SPI 模块具有以下功能:

- 独立的 SPI 设备封装外设相关参数
- 自动处理多设备总线争用
- 支持标准、双线、四线、八线模式
- 支持先写后读和全双工读写
- 支持发送一串相同的数据帧，常用于清屏、填充存储扇区等场景

15.3 API 参考

对应的头文件 `spi.h`
为用户提供以下接口

- `spi_init`
- `spi_init_non_standard`
- `spi_send_data_standard`
- `spi_send_data_standard_dma`
- `spi_receive_data_standard`
- `spi_receive_data_standard_dma`

- spi_send_data_multiple
- spi_send_data_multiple_dma
- spi_receive_data_multiple
- spi_receive_data_multiple_dma
- spi_fill_data_dma
- spi_send_data_normal_dma
- spi_set_clk_rate

15.3.1 spi_init

15.3.1.1 描述

设置 SPI 工作模式、多线模式和位宽。

15.3.1.2 函数原型

```
void spi_init(spi_device_num_t spi_num, spi_work_mode_t work_mode, spi_frame_format_t
    frame_format, size_t data_bit_length, uint32_t endian)
```

15.3.1.3 参数

参数名称	描述	输入输出
spi_num	SPI 号	输入
work_mode	极性相位的四种模式	输入
frame_format	多线模式	输入
data_bit_length	单次传输的数据的位宽	输入
endian	大小端 0：小端 1：大端	输入

15.3.1.4 返回值

无。

15.3.2 spi_config_non_standard

15.3.2.1 描述

多线模式下设置指令长度、地址长度、等待时钟数、指令地址传输模式。

15.3.2.2 函数原型

```
void spi_init_non_standard(spi_device_num_t spi_num, uint32_t instruction_length,
    uint32_t address_length, uint32_t wait_cycles,
    spi_instruction_address_trans_mode_t instruction_address_trans_mode)
```

15.3.2.3　参数

参数名称	描述	输入输出
spi_num	SPI 号	输入
instruction_length	发送指令的位数	输入
address_length	发送地址的位数	输入
wait_cycles	等待时钟个数	输入
instruction_address_trans_mode	指令地址传输的方式	输入

15.3.2.4　返回值

无

15.3.3　spi_send_data_standard

15.3.3.1　描述

SPI 标准模式传输数据。

15.3.3.2　函数原型

```
void spi_send_data_standard(spi_device_num_t spi_num, spi_chip_select_t chip_select,
    const uint8_t *cmd_buff, size_t cmd_len, const uint8_t *tx_buff, size_t tx_len)
```

15.3.3.3　参数

参数名称	描述	输入输出
spi_num	SPI 号	输入
chip_select	片选信号	输入
cmd_buff	外设指令地址数据，没有则设为 NULL	输入
cmd_len	外设指令地址数据长度，没有则设为 0	输入
tx_buff	发送的数据	输入
tx_len	发送数据的长度	输入

15.3.3.4　返回值

无

15.3.4　spi_send_data_standard_dma

15.3.4.1　描述

SPI 标准模式下使用 DMA 传输数据。

15.3.4.2　函数原型

```
void spi_send_data_standard_dma(dmac_channel_number_t channel_num, spi_device_num_t
    spi_num, spi_chip_select_t chip_select, const uint8_t *cmd_buff, size_t cmd_len,
    const uint8_t *tx_buff, size_t tx_len)
```

15.3.4.3　参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入
spi_num	SPI 号	输入
chip_select	片选信号	输入
cmd_buff	外设指令地址数据，没有则设为 NULL	输入
cmd_len	外设指令地址数据长度，没有则设为 0	输入
tx_buff	发送的数据	输入
tx_len	发送数据的长度	输入

15.3.4.4　返回值

无

15.3.5　spi_receive_data_standard

15.3.5.1　描述

标准模式下接收数据。

15.3.5.2　函数原型

```
void spi_receive_data_standard(spi_device_num_t spi_num, spi_chip_select_t chip_select,
     const uint8_t *cmd_buff, size_t cmd_len, uint8_t *rx_buff, size_t rx_len)
```

15.3.5.3　参数

参数名称	描述	输入输出
spi_num	SPI 号	输入
chip_select	片选信号	输入
cmd_buff	外设指令地址数据，没有则设为 NULL	输入
cmd_len	外设指令地址数据长度，没有则设为 0	输入
rx_buff	接收的数据	输出
rx_len	接收数据的长度	输入

15.3.5.4　返回值

无

15.3.6　spi_receive_data_standard_dma

15.3.6.1　描述

标准模式下通过 DMA 接收数据。

15.3.6.2　函数原型

```
void spi_receive_data_standard_dma(dmac_channel_number_t dma_send_channel_num,
    dmac_channel_number_t dma_receive_channel_num, spi_device_num_t spi_num,
    spi_chip_select_t chip_select, const uint8_t *cmd_buff, size_t cmd_len, uint8_t *
    rx_buff, size_t rx_len)
```

15.3.6.3　参数

参数名称	描述	输入输出
dma_send_channel_num	发送指令地址使用的 DMA 通道号	输入
dma_receive_channel_num	接收数据使用的 DMA 通道号	输入
spi_num	SPI 号	输入
chip_select	片选信号	输入
cmd_buff	外设指令地址数据，没有则设为 NULL	输入
cmd_len	外设指令地址数据长度，没有则设为 0	输入
rx_buff	接收的数据	输出
rx_len	接收数据的长度	输入

15.3.6.4　返回值

无

15.3.7　spi_send_data_multiple

15.3.7.1　描述

多线模式发送数据。

15.3.7.2　函数原型

```
void spi_send_data_multiple(spi_device_num_t spi_num, spi_chip_select_t chip_select,
    const uint32_t *cmd_buff, size_t cmd_len, uint8_t *tx_buff, size_t tx_len)
```

15.3.7.3　参数

参数名称	描述	输入输出
spi_num	SPI 号	输入
chip_select	片选信号	输入
cmd_buff	外设指令地址数据，没有则设为 NULL	输入
cmd_len	外设指令地址数据长度，没有则设为 0	输入
tx_buff	发送的数据	输入
tx_len	发送数据的长度	输入

15.3.7.4　返回值

无

15.3.8　spi_send_data_multiple_dma

15.3.8.1　描述

多线模式使用 DMA 发送数据。

15.3.8.2　函数原型

```
void spi_send_data_multiple_dma(dmac_channel_number_t channel_num,spi_device_num_t
    spi_num, spi_chip_select_t chip_select, const uint32_t *cmd_buff, size_t cmd_len,
    const uint8_t *tx_buff, size_t tx_len)
```

15.3.8.3　参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入
spi_num	SPI 号	输入
chip_select	片选信号	输入
cmd_buff	外设指令地址数据，没有则设为 NULL	输入
cmd_len	外设指令地址数据长度，没有则设为 0	输入
tx_buff	发送的数据	输入
tx_len	发送数据的长度	输入

15.3.8.4　返回值

无

15.3.9　spi_receive_data_multiple

15.3.9.1　描述

多线模式接收数据。

15.3.9.2　函数原型

```
void spi_receive_data_multiple(spi_device_num_t spi_num, spi_chip_select_t chip_select,
     const uint32_t *cmd_buff, size_t cmd_len, uint8_t *rx_buff, size_t rx_len)
```

15.3.9.3　参数

参数名称	描述	输入输出
spi_num	SPI 号	输入
chip_select	片选信号	输入
cmd_buff	外设指令地址数据，没有则设为 NULL	输入
cmd_len	外设指令地址数据长度，没有则设为 0	输入
rx_buff	接收的数据	输出
rx_len	接收数据的长度	输入

15.3.9.4　返回值

无

15.3.10　spi_receive_data_multiple_dma

15.3.10.1　描述

多线模式通过 DMA 接收。

15.3.10.2　函数原型

```
void spi_receive_data_multiple_dma(dmac_channel_number_t dma_send_channel_num,
    dmac_channel_number_t dma_receive_channel_num, spi_device_num_t spi_num,
    spi_chip_select_t chip_select, uint32_t const *cmd_buff, size_t cmd_len, uint8_t *
    rx_buff, size_t rx_len);
```

15.3.10.3　参数

参数名称	描述	输入输出
dma_send_channel_num	发送指令地址使用的 DMA 通道号	输入
dma_receive_channel_num	接收数据使用的 DMA 通道号	输入
spi_num	SPI 号	输入
chip_select	片选信号	输入
cmd_buff	外设指令地址数据，没有则设为 NULL	输入
cmd_len	外设指令地址数据长度，没有则设为 0	输入
rx_buff	接收的数据	输出
rx_len	接收数据的长度	输入

15.3.10.4　返回值

无

15.3.11　spi_fill_data_dma

15.3.11.1　描述

通过 DMA 始终发送同一个数据，可以用于刷新数据。

15.3.11.2　函数原型

```
void spi_fill_data_dma(dmac_channel_number_t channel_num,spi_device_num_t spi_num,
    spi_chip_select_t chip_select, const uint32_t *tx_buff, size_t tx_len);
```

15.3.11.3　参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入
spi_num	SPI 号	输入
chip_select	片选信号	输入
tx_buff	发送的数据，仅发送 tx_buff 这一个数据，不会自动增加	输入
tx_len	发送数据的长度	输入

15.3.11.4　返回值

无

15.3.12　spi_send_data_normal_dma

15.3.12.1　描述

通过 DMA 发送数据。不用设置指令地址。

15.3.12.2　函数原型

```
void spi_send_data_normal_dma(dmac_channel_number_t channel_num, spi_device_num_t
    spi_num, spi_chip_select_t chip_select, const void *tx_buff, size_t tx_len,
    spi_transfer_width_t spi_transfer_width)
```

15.3.12.3　参数

参数名称	描述	输入输出
channel_num	DMA 通道号	输入
spi_num	SPI 号	输入
chip_select	片选信号	输入
tx_buff	发送的数据，仅发送 tx_buff 这一个数据，不会自动增加	输入
tx_len	发送数据的长度	输入
spi_transfer_width	发送数据的位宽	输入

15.3.12.4　返回值

无

15.3.13　举例

```
/* SPI0 工作在MODE0模式 标准SPI模式 单次发送8位数据 */
spi_init(SPI_DEVICE_0, SPI_WORK_MODE_0, SPI_FF_STANDARD, 8, 0);
uint8_t cmd[4];
cmd[0] = 0x06;
cmd[1] = 0x01;
cmd[2] = 0x02;
cmd[3] = 0x04;
uint8_t data_buf[4] = {0,1,2,3};
/* SPI0 使用片选0 发送指令0x06 向地址0x010204 发送0，1，2，3 四个字节数据 */
spi_send_data_standard(SPI_DEVICE_0, SPI_CHIP_SELECT_0, cmd, 4, data_buf, 4);
/* SPI0 使用片选0 发送指令0x06 地址0x010204 接收4个字节的数据 */
spi_receive_data_standard(SPI_DEVICE_0, SPI_CHIP_SELECT_0, cmd, 4, data_buf, 4);

/* SPI0 工作在MODE0模式 四线SPI模式 单次发送8位数据 */
spi_init(SPI_DEVICE_0, SPI_WORK_MODE_0, SPI_FF_QUAD, 8, 0);
/* 8位指令长度 32位地址长度 发送指令地址后等待4个clk，指令通过标准SPI方式发送，地址通过
    四线方式发送 */
spi_init_non_standard(SPI_DEVICE_0, 8, 32, 4, SPI_AITM_ADDR_STANDARD);
uint32 cmd[2];
cmd[0] = 0x06;
cmd[1] = 0x010204;
uint8_t data_buf[4] = {0,1,2,3};
/* SPI0 使用片选0 发送指令0x06 向地址0x010204 发送0，1，2，3 四个字节数据 */
spi_send_data_multiple(SPI_DEVICE_0, SPI_CHIP_SELECT_0, cmd, 2, data_buf, 4);
/* SPI0 使用片选0 发送指令0x06 地址0x010204 接收4个字节的数据 */
spi_receive_data_multiple(SPI_DEVICE_0, SPI_CHIP_SELECT_0, cmd, 2, data_buf, 4);

/* SPI0 工作在MODE2模式 八线SPI模式 单次发送32位数据 */
spi_init(SPI_DEVICE_0, SPI_WORK_MODE_2, SPI_FF_OCTAL, 32, 0);
/* 无指令 32位地址长度 发送指令地址后等待0个clk，指令地址通过8线发送 */
spi_init_non_standard(SPI_DEVICE_0, 0, 32, 0, SPI_AITM_AS_FRAME_FORMAT);
uint32_t data_buf[256] = {0};
/* 使用DMA通道0 片选0 发送256个int数据*/
spi_send_data_normal_dma(DMAC_CHANNEL0, SPI_DEVICE_0, SPI_CHIP_SELECT_0, data_buf, 256,
     SPI_TRANS_INT);
uint32_t data = 0x55AA55AA;
/* 使用DMA通道0 片选0 连续发送256个 0x55AA55AA*/
spi_fill_data_dma(DMAC_CHANNEL0, SPI_DEVICE_0, SPI_CHIP_SELECT_0,&data, 256);
```

15.3.14　spi_set_clk_rate

15.3.14.1　描述

设置 SPI 的时钟频率

15.3.14.2　函数原型

```
uint32_t spi_set_clk_rate(spi_device_num_t spi_num, uint32_t spi_clk)
```

15.3.14.3　参数

参数名称	描述	输入输出
spi_num	SPI 号	输入
spi_clk	目标 SPI 设备的时钟频率	输入

15.3.14.4　返回值

设置完后的 SPI 设备的时钟频率

15.4　数据类型

相关数据类型、数据结构定义如下：

- spi_device_num_t: SPI 编号。
- spi_mode_t: SPI 模式。
- spi_frame_format_t: SPI 帧格式。
- spi_instruction_address_trans_mode_t: SPI 指令和地址的传输模式。

15.4.1　spi_device_num_t

15.4.1.1　描述

SPI 编号。

15.4.1.2　定义

```
typedef enum _spi_device_num
{
    SPI_DEVICE_0,
    SPI_DEVICE_1,
    SPI_DEVICE_2,
    SPI_DEVICE_3,
    SPI_DEVICE_MAX,
} spi_device_num_t;
```

15.4.1.3　成员

成员名称	描述
SPI*DEVICE*0	SPI 0 做为主设备
SPI*DEVICE*1	SPI 1 做为主设备
SPI*DEVICE*2	SPI 2 做为从设备
SPI*DEVICE*3	SPI 3 做为主设备

15.4.2　spi_mode_t

15.4.2.1　描述

SPI 模式。

15.4.2.2　定义

```
typedef enum _spi_mode
{
    SPI_WORK_MODE_0,
    SPI_WORK_MODE_1,
    SPI_WORK_MODE_2,
    SPI_WORK_MODE_3,
} spi_mode_t;
```

15.4.2.3　成员

成员名称	描述
SPI_WORK_MODE_0	SPI 模式 0
SPI_WORK_MODE_1	SPI 模式 1
SPI_WORK_MODE_2	SPI 模式 2
SPI_WORK_MODE_3	SPI 模式 3

15.4.3　spi_frame_format_t

15.4.3.1　描述

SPI 帧格式。

15.4.3.2　定义

```
typedef enum _spi_frame_format
{
    SPI_FF_STANDARD,
    SPI_FF_DUAL,
    SPI_FF_QUAD,
    SPI_FF_OCTAL
} spi_frame_format_t;
```

15.4.3.3　成员

成员名称	描述
SPI_FF_STANDARD	标准
SPI_FF_DUAL	双线
SPI_FF_QUAD	四线
SPI_FF_OCTAL	八线 (SPI3 不支持)

15.4.4　spi_instruction_address_trans_mode_t

15.4.4.1　描述

SPI 指令和地址的传输模式。

15.4.4.2　定义

```
typedef enum _spi_instruction_address_trans_mode
{
    SPI_AITM_STANDARD,
    SPI_AITM_ADDR_STANDARD,
    SPI_AITM_AS_FRAME_FORMAT
} spi_instruction_address_trans_mode_t;
```

15.4.4.3　成员

成员名称	描述
SPI_AITM_STANDARD	均使用标准帧格式
SPI_AITM_ADDR_STANDARD	指令使用配置的值，地址使用标准帧格式
SPI_AITM_AS_FRAME_FORMAT	均使用配置的值

第 16 章

集成电路内置音频总线（I2S）

16.1 概述

I2S 标准总线定义了三种信号：时钟信号 BCK、声道选择信号 WS 和串行数据信号 SD。一个基本的 I2S 数据总线有一个主机和一个从机。主机和从机的角色在通信过程中保持不变。I2S 模块包含独立的发送和接收声道，能够保证优良的通信性能。

16.2 功能描述

I2S 模块具有以下功能：

- 根据音频格式自动配置设备（支持 16、24、32 位深，44100 采样率，1 - 4 声道）
- 可配置为播放或录音模式
- 自动管理音频缓冲区

16.3 API 参考

对应的头文件 `i2s.h`
为用户提供以下接口

- `i2s_init`
- `i2s_send_data_dma`
- `i2s_recv_data_dma`
- `i2s_rx_channel_config`
- `i2s_tx_channel_config`
- `i2s_play`

- i2s_set_sample_rate：I2S 设置采样率。
- i2s_set_dma_divide_16：设置 dma*divide*16，16 位数据时设置 dma*divide*16，DMA 传输时自动将 32 比特 INT32 数据分成两个 16 比特的左右声道数据。
- i2s_get_dma_divide_16：获取 dma*divide*16 值。用于判断是否需要设置 dma*divide*16。

16.3.1　i2s_init

16.3.1.1　描述

初始化 I2S。

16.3.1.2　函数原型

```
void i2s_init(i2s_device_number_t device_num, i2s_transmit_t rxtx_mode, uint32_t
    channel_mask)
```

16.3.1.3　参数

成员名称	描述	输入输出
device_num	I2S 号	输入
rxtx_mode	接收或发送模式	输入
channel_mask	通道掩码	输入

16.3.1.4　返回值

无。

16.3.2　i2s_send_data_dma

16.3.2.1　描述

I2S 发送数据。

16.3.2.2　函数原型

```
void i2s_send_data_dma(i2s_device_number_t device_num, const void *buf, size_t buf_len,
     dmac_channel_number_t channel_num)
```

16.3.2.3　参数

成员名称	描述	输入输出
device_num	I2S 号	输入
buf	发送数据地址	输入
buf_len	数据长度	输入
channel_num	DMA 通道号	输入

16.3.2.4　返回值

无。

16.3.3　i2s_recv_data_dma

16.3.3.1　描述

I2S 接收数据。

16.3.3.2　函数原型

```
void i2s_recv_data_dma(i2s_device_number_t device_num, uint32_t *buf, size_t buf_len,
    dmac_channel_number_t channel_num)
```

16.3.3.3　参数

成员名称	描述	输入输出
device_num	I2S 号	输入
buf	接收数据地址	输出
buf_len	数据长度	输入
channel_num	DMA 通道号	输入

16.3.3.4　返回值

无

16.3.4　i2s_rx_channel_config

16.3.4.1　描述

设置接收通道参数。

16.3.4.2　函数原型

```
void i2s_rx_channel_config(i2s_device_number_t device_num, i2s_channel_num_t
    channel_num, i2s_word_length_t word_length, i2s_word_select_cycles_t
    word_select_size, i2s_fifo_threshold_t trigger_level, i2s_work_mode_t word_mode)
```

16.3.4.3　参数

成员名称	描述	输入输出
device_num	I2S 号	输入
channel_num	通道号	输入
word_length	接收数据位数	输出
word_select_size	单个数据时钟数	输入
trigger_level	DMA 触发时 FIFO 深度	输入
word_mode	工作模式	输入

16.3.4.4　返回值

无。

16.3.5　i2s_tx_channel_config

16.3.5.1　描述

设置发送通道参数。

16.3.5.2　函数原型

```
void i2s_tx_channel_config(i2s_device_number_t device_num, i2s_channel_num_t
    channel_num, i2s_word_length_t word_length, i2s_word_select_cycles_t
    word_select_size, i2s_fifo_threshold_t trigger_level, i2s_work_mode_t word_mode)
```

16.3.5.3　参数

成员名称	描述	输入输出
device_num	I2S 号	输入
channel_num	通道号	输入
word_length	接收数据位数	输出

成员名称	描述	输入输出
word_select_size	单个数据时钟数	输入
trigger_level	DMA 触发时 FIFO 深度	输入
word_mode	工作模式	输入

16.3.5.4　返回值

无。

16.3.6　i2s_play

16.3.6.1　描述

发送 PCM 数据，比如播放音乐

16.3.6.2　函数原型

```
void i2s_play(i2s_device_number_t device_num, dmac_channel_number_t channel_num,
              const uint8_t *buf, size_t buf_len, size_t frame, size_t bits_per_sample,
                   uint8_t track_num)
```

16.3.6.3　参数

成员名称	描述	输入输出
device_num	I2S 号	输入
channel_num	通道号	输入
buf	PCM 数据	输入
buf_len	PCM 数据长度	输入
frame	单次发送数量	输入
bits_per_sample	单次采样位宽	输入
track_num	声道数	输入

16.3.6.4　返回值

无。

16.3.7　i2s_set_sample_rate

16.3.7.1　描述

设置采样率。

16.3.7.2　函数原型

```
uint32_t i2s_set_sample_rate(i2s_device_number_t device_num, uint32_t sample_rate)
```

16.3.7.3　参数

成员名称	描述	输入输出
device_num	I2S 号	输入
sample_rate	采样率	输入

16.3.7.4　返回值

实际的采样率。

16.3.8　i2s_set_dma_divide_16

16.3.8.1　描述

设置 dma_divide_16，16 位数据时设置 dma_divide_16，DMA 传输时自动将 32 比特 INT32 数据分成两个 16 比特的左右声道数据。

16.3.8.2　函数原型

```
int i2s_set_dma_divide_16(i2s_device_number_t device_num, uint32_t enable)
```

16.3.8.3　参数

成员名称	描述	输入输出
device_num	I2S 号	输入
enable	0:禁用 1: 使能	输入

16.3.8.4　返回值

返回值	描述
0	成功
非 0	失败

16.3.9　i2s_get_dma_divide_16

16.3.9.1　描述

获取 dma_divide_16 值。用于判断是否需要设置 dma_divide_16。

16.3.9.2　函数原型

```
int i2s_get_dma_divide_16(i2s_device_number_t device_num)
```

16.3.9.3　参数

成员名称	描述	输入输出
device_num	I2S 号	输入

16.3.9.4　返回值

返回值	描述
1	使能
0	禁用
<0	失败

16.3.10　举例

```
/* I2S0 通道0 设置为接收通道，接收16位数据，单次传输32个时钟，FIFO深度为4，标准模式。接收8组数据*/
/* I2S2 通道1 设置为发送通道，发送16位数据，单次传输32个时钟，FIFO深度为4，右对齐模式。发送8组数据*/
uint32_t buf[8];
i2s_init(I2S_DEVICE_0, I2S_RECEIVER, 0x3);
i2s_init(I2S_DEVICE_2, I2S_TRANSMITTER, 0xC);
i2s_rx_channel_config(I2S_DEVICE_0, I2S_CHANNEL_0, RESOLUTION_16_BIT, SCLK_CYCLES_32,
    TRIGGER_LEVEL_4, STANDARD_MODE);
i2s_tx_channel_config(I2S_DEVICE_2, I2S_CHANNEL_1, RESOLUTION_16_BIT, SCLK_CYCLES_32,
    TRIGGER_LEVEL_4, RIGHT_JUSTIFYING_MODE);
i2s_recv_data_dma(I2S_DEVICE_0, rx_buf, 8, DMAC_CHANNEL1);
i2s_send_data_dma(I2S_DEVICE_2, buf, 8, DMAC_CHANNEL0);
```

16.4　数据类型

相关数据类型、数据结构定义如下：

- i2s_device_number_t：I2S 编号。
- i2s_channel_num_t：I2S 通道号。
- i2s_transmit_t：I2S 传输模式。
- i2s_work_mode_t：I2S 工作模式。
- i2s_word_select_cycles_t：I2S 单次传输时钟数。
- i2s_word_length_t：I2S 传输数据位数。
- i2s_fifo_threshold_t：I2S FIFO 深度。

16.4.1　i2s_device_number_t

16.4.1.1　描述

I2S 编号。

16.4.1.2　定义

```
typedef enum _i2s_device_number
{
    I2S_DEVICE_0 = 0,
    I2S_DEVICE_1 = 1,
    I2S_DEVICE_2 = 2,
    I2S_DEVICE_MAX
} i2s_device_number_t;
```

16.4.1.3　成员

成员名称	描述
I2S_DEVICE_0	I2S 0
I2S_DEVICE_1	I2S 1
I2S_DEVICE_2	I2S 2

16.4.2　i2s_channel_num_t

16.4.2.1　描述

I2S 通道号。

16.4.2.2　定义

```
typedef enum _i2s_channel_num
{
    I2S_CHANNEL_0 = 0,
    I2S_CHANNEL_1 = 1,
    I2S_CHANNEL_2 = 2,
    I2S_CHANNEL_3 = 3
} i2s_channel_num_t;
```

16.4.2.3　成员

成员名称	描述
I2S_CHANNEL_0	I2S 通道 0
I2S_CHANNEL_1	I2S 通道 1
I2S_CHANNEL_2	I2S 通道 2
I2S_CHANNEL_3	I2S 通道 3

16.4.3　i2s_transmit_t

16.4.3.1　描述

I2S 传输模式。

16.4.3.2　定义

```
typedef enum _i2s_transmit
{
    I2S_TRANSMITTER = 0,
    I2S_RECEIVER = 1
} i2s_transmit_t;
```

16.4.3.3　成员

成员名称	描述
I2S_TRANSMITTER	发送模式
I2S_RECEIVER	接收模式

16.4.4　i2s_work_mode_t

16.4.4.1　描述

I2S 工作模式。

16.4.4.2　定义

```
typedef enum _i2s_work_mode
{
    STANDARD_MODE = 1,
    RIGHT_JUSTIFYING_MODE = 2,
    LEFT_JUSTIFYING_MODE = 4
} i2s_work_mode_t;
```

16.4.4.3　成员

成员名称	描述
STANDARD_MODE	标准模式
RIGHT_JUSTIFYING_MODE	右对齐模式
LEFT_JUSTIFYING_MODE	左对齐模式

16.4.5　i2s_word_select_cycles_t

16.4.5.1　描述

I2S 单次传输时钟数。

16.4.5.2　定义

```
typedef enum _word_select_cycles
{
    SCLK_CYCLES_16 = 0x0,
    SCLK_CYCLES_24 = 0x1,
    SCLK_CYCLES_32 = 0x2
} i2s_word_select_cycles_t;
```

16.4.5.3　成员

成员名称	描述
SCLK_CYCLES_16	16 个时钟

成员名称	描述
SCLK_CYCLES_24	24 个时钟
SCLK_CYCLES_32	32 个时钟

16.4.6　i2s_word_length_t

16.4.6.1　描述

I2S 传输数据位数。

16.4.6.2　定义

```
typedef enum _word_length
{
    IGNORE_WORD_LENGTH = 0x0,
    RESOLUTION_12_BIT = 0x1,
    RESOLUTION_16_BIT = 0x2,
    RESOLUTION_20_BIT = 0x3,
    RESOLUTION_24_BIT = 0x4,
    RESOLUTION_32_BIT = 0x5
} i2s_word_length_t;
```

16.4.6.3　成员

成员名称	描述
IGNORE_WORD_LENGTH	忽略长度
RESOLUTION_12_BIT	12 位数据长度
RESOLUTION_16_BIT	16 位数据长度
RESOLUTION_20_BIT	20 位数据长度
RESOLUTION_24_BIT	24 位数据长度
RESOLUTION_32_BIT	32 位数据长度

16.4.7　i2s_fifo_threshold_t

16.4.7.1　描述

I2S FIFO 深度。

16.4.7.2　定义

```
typedef enum _fifo_threshold
{
    /* Interrupt trigger when FIFO level is 1 */
    TRIGGER_LEVEL_1 = 0x0,
    /* Interrupt trigger when FIFO level is 2 */
    TRIGGER_LEVEL_2 = 0x1,
    /* Interrupt trigger when FIFO level is 3 */
    TRIGGER_LEVEL_3 = 0x2,
    /* Interrupt trigger when FIFO level is 4 */
    TRIGGER_LEVEL_4 = 0x3,
    /* Interrupt trigger when FIFO level is 5 */
    TRIGGER_LEVEL_5 = 0x4,
    /* Interrupt trigger when FIFO level is 6 */
    TRIGGER_LEVEL_6 = 0x5,
    /* Interrupt trigger when FIFO level is 7 */
    TRIGGER_LEVEL_7 = 0x6,
    /* Interrupt trigger when FIFO level is 8 */
    TRIGGER_LEVEL_8 = 0x7,
    /* Interrupt trigger when FIFO level is 9 */
    TRIGGER_LEVEL_9 = 0x8,
    /* Interrupt trigger when FIFO level is 10 */
    TRIGGER_LEVEL_10 = 0x9,
    /* Interrupt trigger when FIFO level is 11 */
    TRIGGER_LEVEL_11 = 0xa,
    /* Interrupt trigger when FIFO level is 12 */
    TRIGGER_LEVEL_12 = 0xb,
    /* Interrupt trigger when FIFO level is 13 */
    TRIGGER_LEVEL_13 = 0xc,
    /* Interrupt trigger when FIFO level is 14 */
    TRIGGER_LEVEL_14 = 0xd,
    /* Interrupt trigger when FIFO level is 15 */
    TRIGGER_LEVEL_15 = 0xe,
    /* Interrupt trigger when FIFO level is 16 */
    TRIGGER_LEVEL_16 = 0xf
} i2s_fifo_threshold_t;
```

16.4.7.3　成员

成员名称	描述
TRIGGER_LEVEL_1	1 字节 FIFO 深度
TRIGGER_LEVEL_2	2 字节 FIFO 深度
TRIGGER_LEVEL_3	3 字节 FIFO 深度
TRIGGER_LEVEL_4	4 字节 FIFO 深度
TRIGGER_LEVEL_5	5 字节 FIFO 深度
TRIGGER_LEVEL_6	6 字节 FIFO 深度
TRIGGER_LEVEL_7	7 字节 FIFO 深度
TRIGGER_LEVEL_8	8 字节 FIFO 深度

成员名称	描述
TRIGGER_LEVEL_9	9 字节 FIFO 深度
TRIGGER_LEVEL_10	10 字节 FIFO 深度
TRIGGER_LEVEL_11	11 字节 FIFO 深度
TRIGGER_LEVEL_12	12 字节 FIFO 深度
TRIGGER_LEVEL_13	13 字节 FIFO 深度
TRIGGER_LEVEL_14	14 字节 FIFO 深度
TRIGGER_LEVEL_15	15 字节 FIFO 深度
TRIGGER_LEVEL_16	16 字节 FIFO 深度

第 17 章

定时器（TIMER）

17.1 概述

芯片有 3 个定时器，每个定时器有 4 路通道。可以配置为 PWM，详见 PWM 说明。

17.2 功能描述

TIMER 模块具有以下功能：

- 启用或禁用定时器
- 配置定时器触发间隔
- 配置定时器触发处理程序

17.3 API 参考

对应的头文件 `timer.h`
为用户提供以下接口

- `timer_init`
- `timer_set_interval`
- `timer_set_irq`（0.6.0 后不再支持，请使用 `timer_irq_register`）
- `timer_set_enable`
- `timer_irq_register`
- `timer_irq_deregister`

17.3.1　timer_init

17.3.1.1　描述

初始化定时器。

17.3.1.2　函数原型

```
void timer_init(timer_device_number_t timer_number)
```

17.3.1.3　参数

参数名称	描述	输入输出
timer_number	定时器号	输入

17.3.1.4　返回值

无。

17.3.2　timer_set_interval

17.3.2.1　描述

设置定时间隔。

17.3.2.2　函数原型

```
size_t timer_set_interval(timer_device_number_t timer_number, timer_channel_number_t
    channel, size_t nanoseconds)
```

17.3.2.3　参数

参数名称	描述	输入输出
timer_number	定时器号	输入
channel	定时器通道号	输入
nanoseconds	时间间隔（纳秒）	输入

17.3.2.4　返回值

实际的触发间隔（纳秒）。

17.3.3　timer_set_irq

17.3.3.1　描述

设置定时器触发中断回调函数，该函数已废弃，替代函数为 timer_irq_register。

17.3.3.2　函数原型

```
void timer_set_irq(timer_device_number_t timer_number, timer_channel_number_t channel,
    void(*func)(),  uint32_t priority)
```

17.3.3.3　参数

参数名称	描述	输入输出
timer_number	定时器号	输入
channel	定时器通道号	输入
func	回调函数	输入
priority	中断优先级	输入

17.3.3.4　返回值

无。

17.3.4　timer_set_enable

17.3.4.1　描述

使能禁用定时器。

17.3.4.2　函数原型

```
void timer_set_enable(timer_device_number_t timer_number, timer_channel_number_t
    channel, uint32_t enable)
```

17.3.4.3　参数

参数名称	描述	输入输出
timer_number	定时器号	输入
channel	定时器通道号	输入
enable	使能禁用定时器 0：禁用 1：使能	输入

17.3.4.4　返回值

无。

17.3.5　timer_irq_register

17.3.5.1　描述

注册定时器触发中断回调函数。

17.3.5.2　函数原型

```
int timer_irq_register(timer_device_number_t device, timer_channel_number_t channel,
    int is_single_shot, uint32_t priority, timer_callback_t callback, void *ctx);
```

17.3.5.3　参数

参数名称	描述	输入输出
device	定时器号	输入
channel	定时器通道号	输入
is_single_shot	是否单次中断	输入
priority	中断优先级	输入
callback	中断回调函数	输入
ctx	回调函数参数	输入

17.3.5.4　返回值

返回值	描述
0	成功
非 0	失败

17.3.6　timer_irq_deregister

17.3.6.1　描述

注销定时器中断函数。

17.3.6.2　函数原型

```
int timer_irq_deregister(timer_device_number_t device, timer_channel_number_t channel)
```

17.3.6.3　参数

参数名称	描述	输入输出
device	定时器号	输入
channel	定时器通道号	输入

17.3.6.4　返回值

返回值	描述
0	成功
非 0	失败

17.3.7　举例

```
/* 定时器0 通道0 定时1秒打印Time OK! */
void irq_time(void)
{
    printf("Time OK!\n");
}
plic_init();
timer_init(TIMER_DEVICE_0);
timer_set_interval(TIMER_DEVICE_0, TIMER_CHANNEL_0, 1e9);
timer_set_irq(TIMER_CHANNEL_0, TIMER_CHANNEL_0, irq_time, 1);
timer_set_enable(TIMER_CHANNEL_0, TIMER_CHANNEL_0, 1);
sysctl_enable_irq();
```

17.4　数据类型

相关数据类型、数据结构定义如下：

- timer_device_number_t：定时器编号。
- timer_channel_number_t：定时器通道号。
- timer_callback_t：定时器回调函数。

17.4.1　timer_device_number_t

17.4.1.1　描述

定时器编号

17.4.1.2　定义

```
typedef enum _timer_deivce_number
{
    TIMER_DEVICE_0,
    TIMER_DEVICE_1,
    TIMER_DEVICE_2,
    TIMER_DEVICE_MAX,
} timer_device_number_t;
```

17.4.1.3　成员

成员名称	描述
TIMER_DEVICE_0	定时器 0
TIMER_DEVICE_1	定时器 1
TIMER_DEVICE_2	定时器 2

17.4.2　timer_channel_number_t

17.4.2.1　描述

定时器通道号。

17.4.2.2　定义

```
typedef enum _timer_channel_number
{
    TIMER_CHANNEL_0,
    TIMER_CHANNEL_1,
    TIMER_CHANNEL_2,
    TIMER_CHANNEL_3,
    TIMER_CHANNEL_MAX,
```

```
} timer_channel_number_t;
```

17.4.2.3　成员

成员名称	描述
TIMER_CHANNEL_0	定时器通道 0
TIMER_CHANNEL_1	定时器通道 1
TIMER_CHANNEL_2	定时器通道 2
TIMER_CHANNEL_3	定时器通道 3

17.4.3　timer_callback_t

17.4.3.1　描述

定时器回调函数。

17.4.3.2　定义

```
typedef int (*timer_callback_t)(void *ctx);
```

第 18 章

实时时钟 (RTC)

18.1 概述

RTC 是用来计时的单元，在设置时间后具备计时功能。

注意 RTC 模块仅当 PLL0 使能，并且 CPU 频率大于 30MHz 时使用

18.2 功能描述

RTC 模块具有以下功能：

- 获取当前日期时刻
- 设置当前日期时刻

18.3 API 参考

对应的头文件 rtc.h

为用户提供以下接口

- rtc_init
- rtc_timer_set
- rtc_timer_get

18.3.1 rtc_init

18.3.1.1 描述

初始化 RTC。

18.3.1.2　函数原型

```
int rtc_init(void)
```

18.3.1.3　参数

无。

18.3.1.4　返回值

返回值	描述
0	成功
非 0	失败

18.3.2　rtc_timer_set

18.3.2.1　描述

设置日期时间。

18.3.2.2　函数原型

```
int rtc_timer_set(int year, int month, int day, int hour, int minute, int second)
```

18.3.2.3　参数

参数名称	描述	输入输出
year	年	输入
month	月	输入
day	日	输入
hour	时	输入
minute	分	输入
second	秒	输入

18.3.2.4　返回值

无

18.3.3　rtc_timer_get

18.3.3.1　描述

获取日期时间。

18.3.3.2　函数原型

```
int rtc_timer_get(int *year, int *month, int *day, int *hour, int *minute, int *second)
```

18.3.3.3　参数

参数名称	描述	输入输出
year	年	输出
month	月	输出
day	日	输出
hour	时	输出
minute	分	输出
second	秒	输出

18.3.3.4　返回值

返回值	描述
0	成功
非 0	失败

18.3.4　举例

```
rtc_init();
rtc_timer_set(2018, 9, 12, 23, 30, 29);
int year;
int month;
int day;
int hour;
int minute;
int second;
rtc_timer_get(&year, &month, &day, &hour, &minute, &second);
printf("%4d-%d-%d_%d:%d:%d\n", year, month, day, hour, minute, second);
```

第 19 章

脉冲宽度调制器 (PWM)

19.1 概述

PWM 用于控制脉冲输出的占空比。其本质是一个定时器，所以注意设置 PWM 号与通道时不要与 TIMER 定时器冲突。

19.2 功能描述

PWM 模块具有以下功能:

- 配置 PWM 输出频率
- 配置 PWM 每个管脚的输出占空比

19.3 API 参考

对应头文件 pwm.h
为用户提供以下接口

- pwm_init
- pwm_set_frequency
- pwm_set_enable

19.3.1 pwm_init

19.3.1.1 描述

初始化 PWM。

19.3.1.2　函数原型

```
void pwm_init(pwm_device_number_t pwm_number)
```

19.3.1.3　参数

参数名称	描述	输入输出
pwm_number	pwm 号	输入

19.3.1.4　返回值

无。

19.3.2　pwm_set_frequency

19.3.2.1　描述

设置频率及占空比。

19.3.2.2　函数原型

```
double pwm_set_frequency(pwm_device_number_t pwm_number, pwm_channel_number_t channel,
    double frequency, double duty)
```

19.3.2.3　参数

参数名称	描述	输入输出
pwm_number	PWM 号	输入
channel	PWM 通道号	输入
frequency	PWM 输出频率	输入
duty	占空比	输入

19.3.2.4　返回值

实际输出频率。

19.3.3　pwmsetenable

19.3.3.1　描述

使能禁用 PWM。

19.3.3.2　函数原型

```
void pwm_set_enable(pwm_device_number_t pwm_number, uint32_t channel, int enable)
```

19.3.3.3　参数

参数名称	描述	输入输出
pwm_number	PWM 号	输入
channel	PWM 通道号	输入
enable	使能禁用 PWM0：禁用 1：使能	输入

19.3.3.4　返回值

无。

19.3.4　举例

```
/* pwm0 channel 1 输出 200KHZ占空比为0.5的方波 */
/* 设置IO13作为PWM的输出管脚 */
fpioa_set_function(13, FUNC_TIMER0_TOGGLE1);
pwm_init(PWM_DEVICE_0);
pwm_set_frequency(PWM_DEVICE_0, PWM_CHANNEL_1, 200000, 0.5);
pwm_set_enable(PWM_DEVICE_0, PWM_CHANNEL_1, 1);
```

19.4　数据类型

- pwm_device_number_t：pwm 号。
- pwm_channel_number_t：pwm 通道号。

19.4.1　pwm_device_number_t

19.4.1.1　描述

pwm 号。

19.4.1.2　定义

```
typedef enum _pwm_device_number
{
    PWM_DEVICE_0,
    PWM_DEVICE_1,
    PWM_DEVICE_2,
    PWM_DEVICE_MAX,
} pwm_device_number_t;
```

19.4.1.3　成员

成员名称	描述
PWM_DEVICE_0	PWM0
PWM_DEVICE_1	PWM1
PWM_DEVICE_2	PWM2

19.4.2　pwm_channel_number_t

19.4.2.1　描述

pwm 通道号。

19.4.2.2　定义

```
typedef enum _pwm_channel_number
{
    PWM_CHANNEL_0,
    PWM_CHANNEL_1,
    PWM_CHANNEL_2,
    PWM_CHANNEL_3,
    PWM_CHANNEL_MAX,
} pwm_channel_number_t;
```

19.4.2.3　成员

成员名称	描述
PWM_CHANNEL_0	PWM 通道 0
PWM_CHANNEL_1	PWM 通道 1
PWM_CHANNEL_2	PWM 通道 2
PWM_CHANNEL_3	PWM 通道 3

第 20 章

系统控制

20.1 概述

系统控制模块提供对操作系统的配置功能。

20.2 功能描述

系统控制模块具有以下功能：

- 设置 PLL CPU 时钟频率。
- 设置各个模块时钟的分频值。
- 获取各个模块的时钟频率。
- 使能、禁用、复位各个模块。
- 设置 DMA 请求源。
- 使能禁用系统中断。

20.3 API 参考

对应的头文件 sysctl.h
为用户提供以下接口

- sysctl_cpu_set_freq
- sysctl_pll_set_freq
- sysctl_pll_get_freq
- sysctl_pll_enable
- sysctl_pll_disable

- sysctl_clock_set_threshold
- sysctl_clock_get_threshold
- sysctl_clock_set_clock_select
- sysctl_clock_get_clock_select
- sysctl_clock_get_freq
- sysctl_clock_enable
- sysctl_clock_disable
- sysctl_reset
- sysctl_dma_select
- sysctl_set_power_mode
- sysctl_enable_irq
- sysctl_disable_irq
- sysctl_get_time_us
- sysctl_get_reset_status

20.3.1 sysctl_cpu_set_freq

20.3.1.1 描述

设置 CPU 工作频率。是通过修改 PLL0 的频率实现的。

20.3.1.2 函数原型

```
uint32_t sysctl_cpu_set_freq(uint32_t freq)
```

20.3.1.3 参数

参数名称	描述	输入输出
freq	要设置的频率（Hz）	输入

20.3.1.4 返回值

设置后的实际频率（Hz）。

20.3.2 sysctl_set_pll_frequency

20.3.2.1 描述

设置 PLL 频率。

20.3.2.2　函数原型

```
uint32_t sysctl_pll_set_freq(sysctl_pll_t pll, uint32_t pll_freq)
```

20.3.2.3　参数

参数名称	描述	输入输出
pll	PLL 编号	输入
pll_freq	要设置的频率（Hz）	输入

20.3.2.4　返回值

设置后的实际频率（Hz）。

20.3.2.5　函数原型

```
uint32_t sysctl_pll_get_freq(sysctl_pll_t pll)
```

20.3.2.6　参数

参数名称	描述	输入输出
pll	PLL 编号	输入

20.3.2.7　返回值

对应 PLL 的频率（Hz）。

20.3.3　sysctl_pll_enable

20.3.3.1　描述

使能对应的 PLL。

20.3.3.2　函数原型

```
int sysctl_pll_enable(sysctl_pll_t pll)
```

20.3.3.3　参数

参数名称	描述	输入输出
pll	PLL 编号	输入

20.3.3.4　返回值

返回值	描述
0	成功
非 0	失败

20.3.4　sysctl_pll_disable

20.3.4.1　描述

禁用对应 PLL。

20.3.4.2　函数原型

```
int sysctl_pll_disable(sysctl_pll_t pll)
```

20.3.4.3　参数

参数名称	描述	输入输出
pll	PLL 编号	输入

20.3.4.4　返回值

返回值	描述
0	成功
非 0	失败

20.3.5　sysctl_clock_set_threshold

20.3.5.1　描述

设置对应时钟的分频值。

20.3.5.2　函数原型

```
void sysctl_clock_set_threshold(sysctl_threshold_t which, int threshold)
```

20.3.5.3　参数

参数名称	描述	输入输出
which	设置的时钟	输入
threshold	分频值	输入

20.3.5.4　返回值

返回值	描述
0	成功
非 0	失败

20.3.6　sysctl_clock_get_threshold

20.3.6.1　描述

获取对应时钟的分频值。

20.3.6.2　函数原型

```
int sysctl_clock_get_threshold(sysctl_threshold_t which)
```

参数名称	描述	输入输出
which	时钟	输入

20.3.6.3　返回值

对应时钟的分频值。

20.3.7　sysctl_clock_set_clock_select

20.3.7.1　描述

设置时钟源。

20.3.7.2　函数原型

```
int sysctl_clock_set_clock_select(sysctl_clock_select_t which, int select)
```

20.3.7.3　参数

参数名称	描述	输入输出
which	时钟	输入
select	时钟源	输入

20.3.7.4　返回值

返回值	描述
0	成功
非 0	失败

20.3.8　sysctl_clock_get_clock_select

20.3.8.1　描述

获取时钟对应的时钟源。

20.3.8.2　函数原型

```
int sysctl_clock_get_clock_select(sysctl_clock_select_t which)
```

20.3.8.3　参数

参数名称	描述	输入输出
which	时钟	输入

20.3.8.4　返回值

时钟对应的时钟源。

20.3.9　sysctl_clock_get_freq

20.3.9.1　描述

获取时钟的频率。

20.3.9.2　函数原型

```
uint32_t sysctl_clock_get_freq(sysctl_clock_t clock)
```

20.3.9.3　参数

参数名称	描述	输入输出
clock	时钟	输入

20.3.9.4　返回值

时钟的频率（Hz）

20.3.10　sysctl_clock_enable

20.3.10.1　描述

使能时钟。PLL 要使用 sysctl_pll_enable。

20.3.10.2　函数原型

```
int sysctl_clock_enable(sysctl_clock_t clock)
```

20.3.10.3　参数

参数名称	描述	输入输出
clock	时钟	输入

20.3.10.4　返回值

返回值	描述
0	成功
非 0	失败

20.3.11　sysctl_clock_disable

20.3.11.1　描述

禁用时钟，PLL 使用 sysctl_pll_disable。

20.3.11.2　函数原型

```
int sysctl_clock_disable(sysctl_clock_t clock)
```

20.3.11.3　参数

参数名称	描述	输入输出
clock	时钟	输入

20.3.11.4　返回值

返回值	描述
0	成功
非 0	失败

20.3.12　sysctl_reset

20.3.12.1　描述

复位各个模块。

20.3.12.2　函数原型

```
void sysctl_reset(sysctl_reset_t reset)
```

20.3.12.3　参数

参数名称	描述	输入输出
reset	预复位模块	输入

20.3.12.4　返回值

无。

20.3.13　sysctl_dma_select

20.3.13.1　描述

设置 DMA 请求源。与 DMAC 的 API 配合使用。

20.3.13.2　函数原型

```
int sysctl_dma_select(sysctl_dma_channel_t channel, sysctl_dma_select_t select)
```

20.3.13.3　参数

参数名称	描述	输入输出
channel	DMA 通道号	输入
select	DMA 请求源	输入

20.3.13.4　返回值

返回值	描述
0	成功
非 0	失败

20.3.14　sysctl_set_power_mode

20.3.14.1　描述

设置 FPIOA 的对应电源域的电压。

20.3.14.2　原型

```
void sysctl_set_power_mode(sysctl_power_bank_t power_bank, sysctl_io_power_mode_t
    io_power_mode)
```

20.3.14.3　参数

参数名称	描述	输入输出
power_bank	IO 电源域编号	输入
io_power_mode	设置的电压值 1.8V 或 3.3V	输入

20.3.14.4　返回值

无。

20.3.15　sysctl_enable_irq

20.3.15.1　描述

使能系统中断，如果使用中断一定要开启系统中断。

20.3.15.2　函数原型

```
void sysctl_enable_irq(void)
```

20.3.15.3　参数

无。

20.3.15.4　返回值

无。

20.3.16　sysctl_disable_irq

20.3.16.1　描述

禁用系统中断。

20.3.16.2　函数原型

```
void sysctl_disable_irq(void)
```

20.3.16.3　参数

无。

20.3.16.4　返回值

无。

20.3.17　sysctl_get_time_us

20.3.17.1　描述

开机至今的时间（微秒）。

20.3.17.2　函数原型

```
uint64_t sysctl_get_time_us(void)
```

20.3.17.3　参数

无。

20.3.17.4　返回值

开机至今的时间（微秒）。

20.3.18　sysctl_get_reset_status

20.3.18.1　描述

获取复位状态。参见 sysctl*reset*enum*status*t 说明。

20.3.18.2　函数原型

```
sysctl_reset_enum_status_t sysctl_get_reset_status(void)
```

20.3.18.3　参数

无。

20.3.18.4　返回值

复位状态。

20.4　数据类型

相关数据类型、数据结构定义如下：

- sysctl_pll_t: PLL 编号。
- sysctl_threshold_t: 设置分频值时各模块编号。
- sysctl_clock_select_t: 设置时钟源时各模块编号。
- sysctl_clock_t: 各个模块的编号。
- sysctl_reset_t: 复位时各个模块的编号。
- sysctl_dma_channel_t: DMA 通道号。
- sysctl_dma_select_t: DMA 请求源编号。
- sysctl_power_bank_t: 电源域编号。
- sysctl_io_power_mode_t: IO 输出电压值。
- sysctl_reset_enum_status_t: 复位状态。

20.4.1 sysctl_pll_t

20.4.1.1 描述

PLL 编号。

20.4.1.2 定义

```
typedef enum _sysctl_pll_t
{
    SYSCTL_PLL0,
    SYSCTL_PLL1,
    SYSCTL_PLL2,
    SYSCTL_PLL_MAX
} sysctl_pll_t;
```

20.4.1.3 成员

成员名称	描述
SYSCTL_PLL0	PLL0
SYSCTL_PLL1	PLL1
SYSCTL_PLL2	PLL2

20.4.2 sysctl_threshold_t

20.4.2.1 描述

设置分频值各模块编号。

20.4.2.2　定义

```
typedef enum _sysctl_threshold_t
{
    SYSCTL_THRESHOLD_ACLK,
    SYSCTL_THRESHOLD_APB0,
    SYSCTL_THRESHOLD_APB1,
    SYSCTL_THRESHOLD_APB2,
    SYSCTL_THRESHOLD_SRAM0,
    SYSCTL_THRESHOLD_SRAM1,
    SYSCTL_THRESHOLD_AI,
    SYSCTL_THRESHOLD_DVP,
    SYSCTL_THRESHOLD_ROM,
    SYSCTL_THRESHOLD_SPI0,
    SYSCTL_THRESHOLD_SPI1,
    SYSCTL_THRESHOLD_SPI2,
    SYSCTL_THRESHOLD_SPI3,
    SYSCTL_THRESHOLD_TIMER0,
    SYSCTL_THRESHOLD_TIMER1,
    SYSCTL_THRESHOLD_TIMER2,
    SYSCTL_THRESHOLD_I2S0,
    SYSCTL_THRESHOLD_I2S1,
    SYSCTL_THRESHOLD_I2S2,
    SYSCTL_THRESHOLD_I2S0_M,
    SYSCTL_THRESHOLD_I2S1_M,
    SYSCTL_THRESHOLD_I2S2_M,
    SYSCTL_THRESHOLD_I2C0,
    SYSCTL_THRESHOLD_I2C1,
    SYSCTL_THRESHOLD_I2C2,
    SYSCTL_THRESHOLD_WDT0,
    SYSCTL_THRESHOLD_WDT1,
    SYSCTL_THRESHOLD_MAX = 28
} sysctl_threshold_t;
```

20.4.2.3　成员

成员名称	描述
SYSCTL_THRESHOLD_ACLK	ACLK
SYSCTL_THRESHOLD_APB0	APB0
SYSCTL_THRESHOLD_APB1	APB1
SYSCTL_THRESHOLD_APB2	ACLK
SYSCTL_THRESHOLD_SRAM0	SRAM0
SYSCTL_THRESHOLD_SRAM1	SRAM1
SYSCTL_THRESHOLD_AI	AI
SYSCTL_THRESHOLD_DVP	DVP

成员名称	描述
SYSCTL_THRESHOLD_ROM	ROM
SYSCTL_THRESHOLD_SPI0	SPI0
SYSCTL_THRESHOLD_SPI1	SPI1
SYSCTL_THRESHOLD_SPI2	SPI2
SYSCTL_THRESHOLD_SPI3	SPI3
SYSCTL_THRESHOLD_TIMER0	TIMER0
SYSCTL_THRESHOLD_TIMER1	TIMER1
SYSCTL_THRESHOLD_TIMER2	TIMER2
SYSCTL_THRESHOLD_I2S0	I2S0
SYSCTL_THRESHOLD_I2S1	I2S1
SYSCTL_THRESHOLD_I2S2	I2S2
SYSCTL_THRESHOLD_I2S0_M	I2S0 MCLK
SYSCTL_THRESHOLD_I2S1_M	I2S1 MCLK
SYSCTL_THRESHOLD_I2S2_M	I2S2 MCLK
SYSCTL_THRESHOLD_I2C0	I2C0
SYSCTL_THRESHOLD_I2C1	I2C1
SYSCTL_THRESHOLD_I2C2	I2C2
SYSCTL_THRESHOLD_WDT0	WDT0
SYSCTL_THRESHOLD_WDT1	WDT1

20.4.3 sysctl_clock_select_t

20.4.3.1 描述

设置时钟源时各模块编号。

20.4.3.2 定义

```
typedef enum _sysctl_clock_select_t
{
    SYSCTL_CLOCK_SELECT_PLL0_BYPASS,
    SYSCTL_CLOCK_SELECT_PLL1_BYPASS,
    SYSCTL_CLOCK_SELECT_PLL2_BYPASS,
    SYSCTL_CLOCK_SELECT_PLL2,
    SYSCTL_CLOCK_SELECT_ACLK,
    SYSCTL_CLOCK_SELECT_SPI3,
    SYSCTL_CLOCK_SELECT_TIMER0,
    SYSCTL_CLOCK_SELECT_TIMER1,
    SYSCTL_CLOCK_SELECT_TIMER2,
    SYSCTL_CLOCK_SELECT_SPI3_SAMPLE,
```

```
    SYSCTL_CLOCK_SELECT_MAX = 11
} sysctl_clock_select_t;
```

20.4.3.3　成员

成员名称	描述
SYSCTL_CLOCK_SELECT_PLL0_BYPASS	PLL0_BYPASS
SYSCTL_CLOCK_SELECT_PLL1_BYPASS	PLL1_BYPASS
SYSCTL_CLOCK_SELECT_PLL2_BYPASS	PLL2_BYPASS
SYSCTL_CLOCK_SELECT_PLL2	PLL2
SYSCTL_CLOCK_SELECT_ACLK	ACLK
SYSCTL_CLOCK_SELECT_SPI3	SPI3
SYSCTL_CLOCK_SELECT_TIMER0	TIMER0
SYSCTL_CLOCK_SELECT_TIMER1	TIMER1
SYSCTL_CLOCK_SELECT_TIMER2	TIMER2
SYSCTL_CLOCK_SELECT_SPI3_SAMPLE	SPI3 数据采样时钟沿选择

20.4.4　sysctl_clock_t

20.4.4.1　描述

各个模块的编号。

20.4.4.2　定义

```
typedef enum _sysctl_clock_t
{
    SYSCTL_CLOCK_PLL0,
    SYSCTL_CLOCK_PLL1,
    SYSCTL_CLOCK_PLL2,
    SYSCTL_CLOCK_CPU,
    SYSCTL_CLOCK_SRAM0,
    SYSCTL_CLOCK_SRAM1,
    SYSCTL_CLOCK_APB0,
    SYSCTL_CLOCK_APB1,
    SYSCTL_CLOCK_APB2,
    SYSCTL_CLOCK_ROM,
    SYSCTL_CLOCK_DMA,
    SYSCTL_CLOCK_AI,
    SYSCTL_CLOCK_DVP,
    SYSCTL_CLOCK_FFT,
    SYSCTL_CLOCK_GPIO,
    SYSCTL_CLOCK_SPI0,
```

```
    SYSCTL_CLOCK_SPI1,
    SYSCTL_CLOCK_SPI2,
    SYSCTL_CLOCK_SPI3,
    SYSCTL_CLOCK_I2S0,
    SYSCTL_CLOCK_I2S1,
    SYSCTL_CLOCK_I2S2,
    SYSCTL_CLOCK_I2C0,
    SYSCTL_CLOCK_I2C1,
    SYSCTL_CLOCK_I2C2,
    SYSCTL_CLOCK_UART1,
    SYSCTL_CLOCK_UART2,
    SYSCTL_CLOCK_UART3,
    SYSCTL_CLOCK_AES,
    SYSCTL_CLOCK_FPIOA,
    SYSCTL_CLOCK_TIMER0,
    SYSCTL_CLOCK_TIMER1,
    SYSCTL_CLOCK_TIMER2,
    SYSCTL_CLOCK_WDT0,
    SYSCTL_CLOCK_WDT1,
    SYSCTL_CLOCK_SHA,
    SYSCTL_CLOCK_OTP,
    SYSCTL_CLOCK_RTC,
    SYSCTL_CLOCK_ACLK = 40,
    SYSCTL_CLOCK_HCLK,
    SYSCTL_CLOCK_IN0,
    SYSCTL_CLOCK_MAX
} sysctl_clock_t;
```

20.4.4.3　成员

成员名称	描述
SYSCTL_CLOCK_PLL0	PLL0
SYSCTL_CLOCK_PLL1	PLL1
SYSCTL_CLOCK_PLL2	PLL2
SYSCTL_CLOCK_CPU	CPU
SYSCTL_CLOCK_SRAM0	SRAM0
SYSCTL_CLOCK_SRAM1	SRAM1
SYSCTL_CLOCK_APB0	APB0
SYSCTL_CLOCK_APB1	APB1
SYSCTL_CLOCK_APB2	APB2
SYSCTL_CLOCK_ROM	ROM
SYSCTL_CLOCK_DMA	DMA
SYSCTL_CLOCK_AI	AI
SYSCTL_CLOCK_DVP	DVP

成员名称	描述
SYSCTL_CLOCK_FFT	FFT
SYSCTL_CLOCK_GPIO	GPIO
SYSCTL_CLOCK_SPI0	SPI0
SYSCTL_CLOCK_SPI1	SPI1
SYSCTL_CLOCK_SPI2	SPI2
SYSCTL_CLOCK_SPI3	SPI3
SYSCTL_CLOCK_I2S0	I2S0
SYSCTL_CLOCK_I2S1	I2S1
SYSCTL_CLOCK_I2S2	I2S2
SYSCTL_CLOCK_I2C0	I2C0
SYSCTL_CLOCK_I2C1	I2C1
SYSCTL_CLOCK_I2C2	I2C2
SYSCTL_CLOCK_UART1	UART1
SYSCTL_CLOCK_UART2	UART2
SYSCTL_CLOCK_UART3	UART3
SYSCTL_CLOCK_AES	AES
SYSCTL_CLOCK_FPIOA	FPIOA
SYSCTL_CLOCK_TIMER0	TIMER0
SYSCTL_CLOCK_TIMER1	TIMER1
SYSCTL_CLOCK_TIMER2	TIMER2
SYSCTL_CLOCK_WDT0	WDT0
SYSCTL_CLOCK_WDT1	WDT1
SYSCTL_CLOCK_SHA	SHA
SYSCTL_CLOCK_OTP	OTP
SYSCTL_CLOCK_RTC	RTC
SYSCTL_CLOCK_ACLK	ACLK
SYSCTL_CLOCK_HCLK	HCLK
SYSCTL_CLOCK_IN0	外部输入时钟 IN0

20.4.5 sysctl_reset_t

20.4.5.1 描述

复位时各个模块的编号。

20.4.5.2 定义

```
typedef enum _sysctl_reset_t
{
    SYSCTL_RESET_SOC,
    SYSCTL_RESET_ROM,
    SYSCTL_RESET_DMA,
    SYSCTL_RESET_AI,
    SYSCTL_RESET_DVP,
    SYSCTL_RESET_FFT,
    SYSCTL_RESET_GPIO,
    SYSCTL_RESET_SPI0,
    SYSCTL_RESET_SPI1,
    SYSCTL_RESET_SPI2,
    SYSCTL_RESET_SPI3,
    SYSCTL_RESET_I2S0,
    SYSCTL_RESET_I2S1,
    SYSCTL_RESET_I2S2,
    SYSCTL_RESET_I2C0,
    SYSCTL_RESET_I2C1,
    SYSCTL_RESET_I2C2,
    SYSCTL_RESET_UART1,
    SYSCTL_RESET_UART2,
    SYSCTL_RESET_UART3,
    SYSCTL_RESET_AES,
    SYSCTL_RESET_FPIOA,
    SYSCTL_RESET_TIMER0,
    SYSCTL_RESET_TIMER1,
    SYSCTL_RESET_TIMER2,
    SYSCTL_RESET_WDT0,
    SYSCTL_RESET_WDT1,
    SYSCTL_RESET_SHA,
    SYSCTL_RESET_RTC,
    SYSCTL_RESET_MAX = 31
} sysctl_reset_t;
```

20.4.5.3　成员

成员名称	描述
SYSCTL_RESET_SOC	芯片复位
SYSCTL_RESET_ROM	ROM
SYSCTL_RESET_DMA	DMA
SYSCTL_RESET_AI	AI
SYSCTL_RESET_DVP	DVP
SYSCTL_RESET_FFT	FFT
SYSCTL_RESET_GPIO	GPIO
SYSCTL_RESET_SPI0	SPI0
SYSCTL_RESET_SPI1	SPI1

成员名称	描述
SYSCTL_RESET_SPI2	SPI2
SYSCTL_RESET_SPI3	SPI3
SYSCTL_RESET_I2S0	I2S0
SYSCTL_RESET_I2S1	I2S1
SYSCTL_RESET_I2S2	I2S2
SYSCTL_RESET_I2C0	I2C0
SYSCTL_RESET_I2C1	I2C1
SYSCTL_RESET_I2C2	I2C2
SYSCTL_RESET_UART1	UART1
SYSCTL_RESET_UART2	UART2
SYSCTL_RESET_UART3	UART3
SYSCTL_RESET_AES	AES
SYSCTL_RESET_FPIOA	FPIOA
SYSCTL_RESET_TIMER0	TIMER0
SYSCTL_RESET_TIMER1	TIMER1
SYSCTL_RESET_TIMER2	TIMER2
SYSCTL_RESET_WDT0	WDT0
SYSCTL_RESET_WDT1	WDT1
SYSCTL_RESET_SHA	SHA
SYSCTL_RESET_RTC	RTC

20.4.6 sysctl_dma_channel_t

20.4.6.1 描述

DMA 通道号。

20.4.6.2 定义

```
typedef enum _sysctl_dma_channel_t
{
    SYSCTL_DMA_CHANNEL_0,
    SYSCTL_DMA_CHANNEL_1,
    SYSCTL_DMA_CHANNEL_2,
    SYSCTL_DMA_CHANNEL_3,
    SYSCTL_DMA_CHANNEL_4,
    SYSCTL_DMA_CHANNEL_5,
    SYSCTL_DMA_CHANNEL_MAX
} sysctl_dma_channel_t;
```

20.4.6.3　成员

成员名称	描述
SYSCTL_DMA_CHANNEL_0	DMA 通道 0
SYSCTL_DMA_CHANNEL_1	DMA 通道 1
SYSCTL_DMA_CHANNEL_2	DMA 通道 2
SYSCTL_DMA_CHANNEL_3	DMA 通道 3
SYSCTL_DMA_CHANNEL_4	DMA 通道 4
SYSCTL_DMA_CHANNEL_5	DMA 通道 5

20.4.7　sysctl_dma_select_t

20.4.7.1　描述

DMA 请求源编号。

20.4.7.2　定义

```
typedef enum _sysctl_dma_select_t
{
    SYSCTL_DMA_SELECT_SSI0_RX_REQ,
    SYSCTL_DMA_SELECT_SSI0_TX_REQ,
    SYSCTL_DMA_SELECT_SSI1_RX_REQ,
    SYSCTL_DMA_SELECT_SSI1_TX_REQ,
    SYSCTL_DMA_SELECT_SSI2_RX_REQ,
    SYSCTL_DMA_SELECT_SSI2_TX_REQ,
    SYSCTL_DMA_SELECT_SSI3_RX_REQ,
    SYSCTL_DMA_SELECT_SSI3_TX_REQ,
    SYSCTL_DMA_SELECT_I2C0_RX_REQ,
    SYSCTL_DMA_SELECT_I2C0_TX_REQ,
    SYSCTL_DMA_SELECT_I2C1_RX_REQ,
    SYSCTL_DMA_SELECT_I2C1_TX_REQ,
    SYSCTL_DMA_SELECT_I2C2_RX_REQ,
    SYSCTL_DMA_SELECT_I2C2_TX_REQ,
    SYSCTL_DMA_SELECT_UART1_RX_REQ,
    SYSCTL_DMA_SELECT_UART1_TX_REQ,
    SYSCTL_DMA_SELECT_UART2_RX_REQ,
    SYSCTL_DMA_SELECT_UART2_TX_REQ,
    SYSCTL_DMA_SELECT_UART3_RX_REQ,
    SYSCTL_DMA_SELECT_UART3_TX_REQ,
    SYSCTL_DMA_SELECT_AES_REQ,
    SYSCTL_DMA_SELECT_SHA_RX_REQ,
    SYSCTL_DMA_SELECT_AI_RX_REQ,
    SYSCTL_DMA_SELECT_FFT_RX_REQ,
    SYSCTL_DMA_SELECT_FFT_TX_REQ,
```

```
    SYSCTL_DMA_SELECT_I2S0_TX_REQ,
    SYSCTL_DMA_SELECT_I2S0_RX_REQ,
    SYSCTL_DMA_SELECT_I2S1_TX_REQ,
    SYSCTL_DMA_SELECT_I2S1_RX_REQ,
    SYSCTL_DMA_SELECT_I2S2_TX_REQ,
    SYSCTL_DMA_SELECT_I2S2_RX_REQ,
    SYSCTL_DMA_SELECT_MAX
} sysctl_dma_select_t;
```

20.4.7.3　成员

成员名称	描述
SYSCTL_DMA_SELECT_SSI0_RX_REQ	SPI0 接收
SYSCTL_DMA_SELECT_SSI0_TX_REQ	SPI0 发送
SYSCTL_DMA_SELECT_SSI1_RX_REQ	SPI1 接收
SYSCTL_DMA_SELECT_SSI1_TX_REQ	SPI1 发送
SYSCTL_DMA_SELECT_SSI2_RX_REQ	SPI2 接收
SYSCTL_DMA_SELECT_SSI2_TX_REQ	SPI2 发送
SYSCTL_DMA_SELECT_SSI3_RX_REQ	SPI3 接收
SYSCTL_DMA_SELECT_SSI3_TX_REQ	SPI3 发送
SYSCTL_DMA_SELECT_I2C0_RX_REQ	I2C0 接收
SYSCTL_DMA_SELECT_I2C0_TX_REQ	I2C0 发送
SYSCTL_DMA_SELECT_I2C1_RX_REQ	I2C1 接收
SYSCTL_DMA_SELECT_I2C1_TX_REQ	I2C1 发送
SYSCTL_DMA_SELECT_I2C2_RX_REQ	I2C2 接收
SYSCTL_DMA_SELECT_I2C2_TX_REQ	I2C2 发送
SYSCTL_DMA_SELECT_UART1_RX_REQ	UART1 接收
SYSCTL_DMA_SELECT_UART1_TX_REQ	UART1 发送
SYSCTL_DMA_SELECT_UART2_RX_REQ	UART2 接收
SYSCTL_DMA_SELECT_UART2_TX_REQ	UART2 发送
SYSCTL_DMA_SELECT_UART3_RX_REQ	UART3 接收
SYSCTL_DMA_SELECT_UART3_TX_REQ	UART3 发送
SYSCTL_DMA_SELECT_AES_REQ	AES
SYSCTL_DMA_SELECT_SHA_RX_REQ	SHA 接收
SYSCTL_DMA_SELECT_AI_RX_REQ	AI 接收
SYSCTL_DMA_SELECT_FFT_RX_REQ	FFT 接收
SYSCTL_DMA_SELECT_FFT_TX_REQ	FFT 发送
SYSCTL_DMA_SELECT_I2S0_TX_REQ	I2S0 发送
SYSCTL_DMA_SELECT_I2S0_RX_REQ	I2S0 接收

成员名称	描述
SYSCTL_DMA_SELECT_I2S1_TX_REQ	I2S1 发送
SYSCTL_DMA_SELECT_I2S1_RX_REQ	I2S1 接收
SYSCTL_DMA_SELECT_I2S2_TX_REQ	I2S2 发送
SYSCTL_DMA_SELECT_I2S2_RX_REQ	I2S2 接收

20.4.8 sysctl_power_bank_t

20.4.8.1 描述

电源域编号。

20.4.8.2 定义

```
typedef enum _sysctl_power_bank
{
    SYSCTL_POWER_BANK0,
    SYSCTL_POWER_BANK1,
    SYSCTL_POWER_BANK2,
    SYSCTL_POWER_BANK3,
    SYSCTL_POWER_BANK4,
    SYSCTL_POWER_BANK5,
    SYSCTL_POWER_BANK6,
    SYSCTL_POWER_BANK7,
    SYSCTL_POWER_BANK_MAX,
} sysctl_power_bank_t;
```

20.4.8.3 成员

成员名称	描述
SYSCTL_POWER_BANK0	电源域 0，控制 IO0-IO5
SYSCTL_POWER_BANK1	电源域 1，控制 IO6-IO11
SYSCTL_POWER_BANK2	电源域 2，控制 IO12-IO17
SYSCTL_POWER_BANK3	电源域 3，控制 IO18-IO23
SYSCTL_POWER_BANK4	电源域 4，控制 IO24-IO29
SYSCTL_POWER_BANK5	电源域 5，控制 IO30-IO35
SYSCTL_POWER_BANK6	电源域 6，控制 IO36-IO41
SYSCTL_POWER_BANK7	电源域 7，控制 IO42-IO47

20.4.9　sysctl_io_power_mode_t

20.4.9.1　描述

IO 输出电压值。

20.4.9.2　定义

```
typedef enum _sysctl_io_power_mode
{
    SYSCTL_POWER_V33,
    SYSCTL_POWER_V18
} sysctl_io_power_mode_t;
```

20.4.9.3　成员

成员名称	描述
SYSCTL_POWER_V33	设置为 3.3V
SYSCTL_POWER_V18	设置为 1.8V

20.4.10　sysctl_reset_enum_status_t

20.4.10.1　描述

复位状态。

20.4.10.2　定义

```
typedef enum _sysctl_reset_enum_status
{
    SYSCTL_RESET_STATUS_HARD,
    SYSCTL_RESET_STATUS_SOFT,
    SYSCTL_RESET_STATUS_WDT0,
    SYSCTL_RESET_STATUS_WDT1,
    SYSCTL_RESET_STATUS_MAX,
} sysctl_reset_enum_status_t;
```

20.4.10.3　成员

成员名称	描述
SYSCTL_RESET_STATUS_HARD	硬件复位，重新上电或触发 reset 管脚
SYSCTL_RESET_STATUS_SOFT	软件复位

成员名称	描述
SYSCTL_RESET_STATUS_WDT0	看门狗 0 复位
SYSCTL_RESET_STATUS_WDT1	看门狗 1 复位

第 21 章

平台相关（BSP）

21.1 概述

平台相关的通用函数，核之间锁的相关操作。

21.2 功能描述

提供获取当前运行程序的 CPU 核编号的接口以及启动第二个核的入口。

21.3 API 参考

对应的头文件 bsp.h

为用户提供以下接口

- register_core1
- current_coreid
- read_cycle
- corelock_lock
- corelock_trylock
- corelock_unlock

21.3.1 register_core1

21.3.1.1 描述

向 1 核注册函数，并启动 1 核。

21.3.1.2　函数原型

```
int register_core1(core_function func, void *ctx)
```

21.3.1.3　参数

参数名称	描述	输入输出
func	向 1 核注册的函数	输入
ctx	函数的参数，没有设置为 NULL	输入

21.3.1.4　返回值

返回值	描述
0	成功
非 0	失败

21.3.2　current_coreid

21.3.2.1　描述

获取当前 CPU 核编号。

21.3.2.2　函数原型

```
#define current_coreid()        read_csr(mhartid)
```

21.3.2.3　参数

无。

21.3.2.4　返回值

当前所在 CPU 核的编号。

21.3.2.5　read_cycle

21.3.2.6　描述

获取 CPU 开机至今的时钟数。可以用使用这个函数精准的确定程序运行时钟。可以配合 sysctl_clock_get_freq(SYSCTL_CLOCK_CPU) 计算运行的时间。

21.3.2.7　函数原型

```
#define read_cycle()        read_csr(mcycle)
```

21.3.2.8　参数

无。

21.3.2.9　返回值

开机至今的 CPU 时钟数。

21.3.3　corelock_lock

21.3.3.1　描述

获取核间锁，核之间互斥的锁，同核内该锁会嵌套，只有异核之间会阻塞。不建议在中断使用该函数，中断中可以使用 corelock_trylock。

21.3.3.2　函数原型

```
void corelock_lock(corelock_t *lock)
```

21.3.3.3　参数

核间锁，要使用全局变量，参见举例。

21.3.3.4　返回值

无。

21.3.4　corelock_trylock

21.3.4.1　描述

获取核间锁，同核时锁会嵌套，异核时非阻塞。成功获取锁会返回 0，失败返回-1。

21.3.4.2　函数原型

```
corelock_trylock(corelock_t *lock)
```

21.3.4.3　参数

核间锁，要使用全局变量，参见举例。

21.3.4.4　返回值

无。

21.3.5　corelock_unlock

21.3.5.1　描述

核间锁解锁。

21.3.5.2　函数原型

```
void corelock_unlock(corelock_t *lock)
```

21.3.5.3　参数

核间锁，要使用全局变量，参见举例。

21.3.5.4　返回值

无。

21.3.6　举例

```
/* 1核在0核第二次释放锁的时候才会获取到锁，通过读cycle计算时间 */
#include <stdio.h>
#include "bsp.h"
#include <unistd.h>
#include "sysctl.h"

corelock_t lock;

uint64_t get_time(void)
{
```

```
    uint64_t v_cycle = read_cycle();
    return v_cycle * 1000000 / sysctl_clock_get_freq(SYSCTL_CLOCK_CPU);
}

int core1_function(void *ctx)
{
    uint64_t core = current_coreid();
    printf("Core %ld Hello world\n", core);
    while(1)
    {
        uint64_t start = get_time();
        corelock_lock(&lock);
        printf("Core %ld Hello world\n", core);
        sleep(1);
        corelock_unlock(&lock);
        uint64_t stop = get_time();
        printf("Core %ld lock time is %ld us\n",core, stop - start);
        usleep(10);
    }
}

int main(void)
{
    uint64_t core = current_coreid();
    printf("Core %ld Hello world\n", core);
    register_core1(core1_function, NULL);
    while(1)
    {
        corelock_lock(&lock);
        sleep(1);
        printf("1> Core %ld sleep 1\n", core);
        corelock_lock(&lock);
        sleep(2);
        printf("2> Core %ld sleep 2\n", core);
        printf("2> Core unlock\n");
        corelock_unlock(&lock);
        sleep(1);
        printf("1> Core unlock\n");
        corelock_unlock(&lock);
        usleep(10);
    }
}
```

21.4　数据类型

相关数据类型、数据结构定义如下：

- core_function：CPU 核调用的函数。
- spinlock_t：自旋锁。
- corelock_t：核间锁。

21.4.1　core_function

21.4.1.1　描述

CPU 核调用的函数。

21.4.1.2　定义

```
typedef int (*core_function)(void *ctx);
```

21.4.2　spinlock_t

自旋锁。

21.4.2.1　定义

```
typedef struct _spinlock
{
    int lock;
} spinlock_t;
```

21.4.3　corelock_t

核间锁。

21.4.3.1　定义

```
typedef struct _corelock
{
    spinlock_t lock;
    int count;
    int core;
} corelock_t;
```